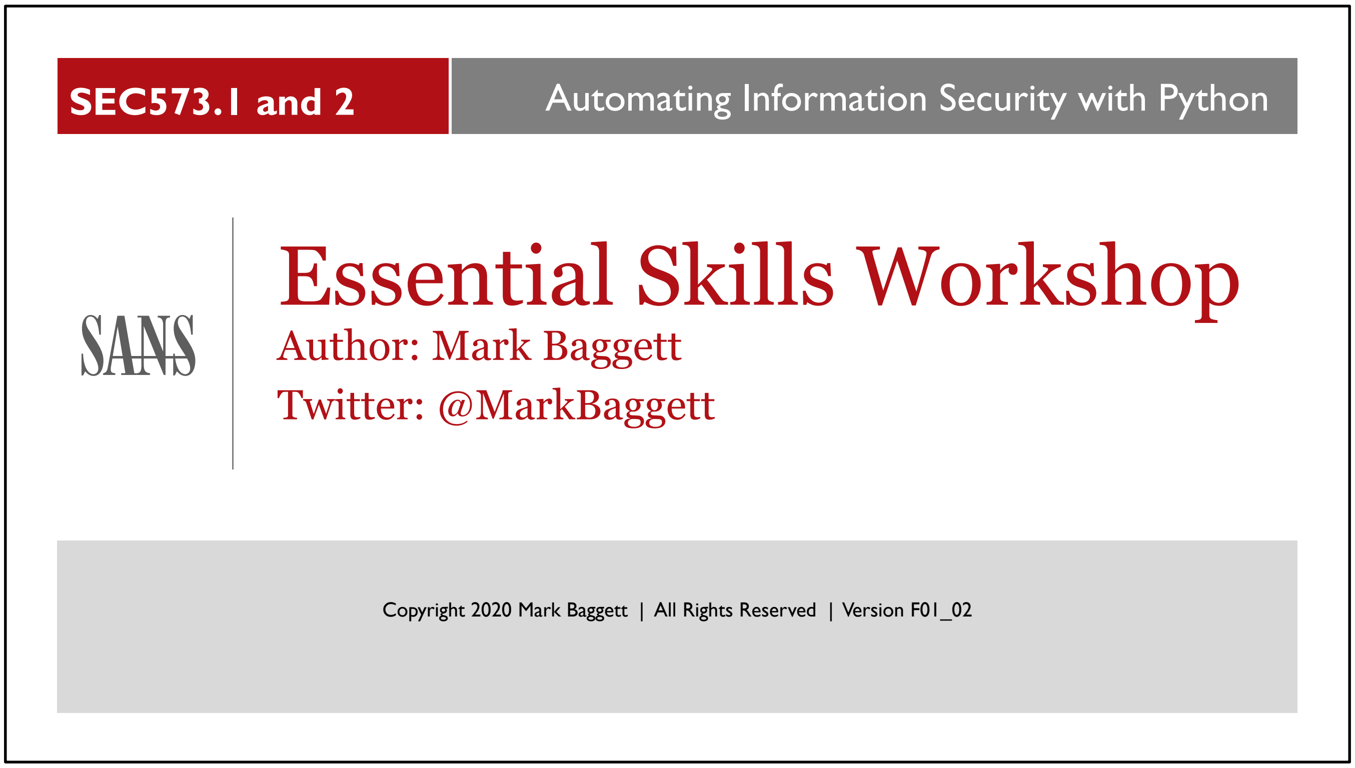
573.1-573.2
Essentials Skills Workshop

SANS



SEC573.1 and 2 | Automating Information Security with Python

SANS

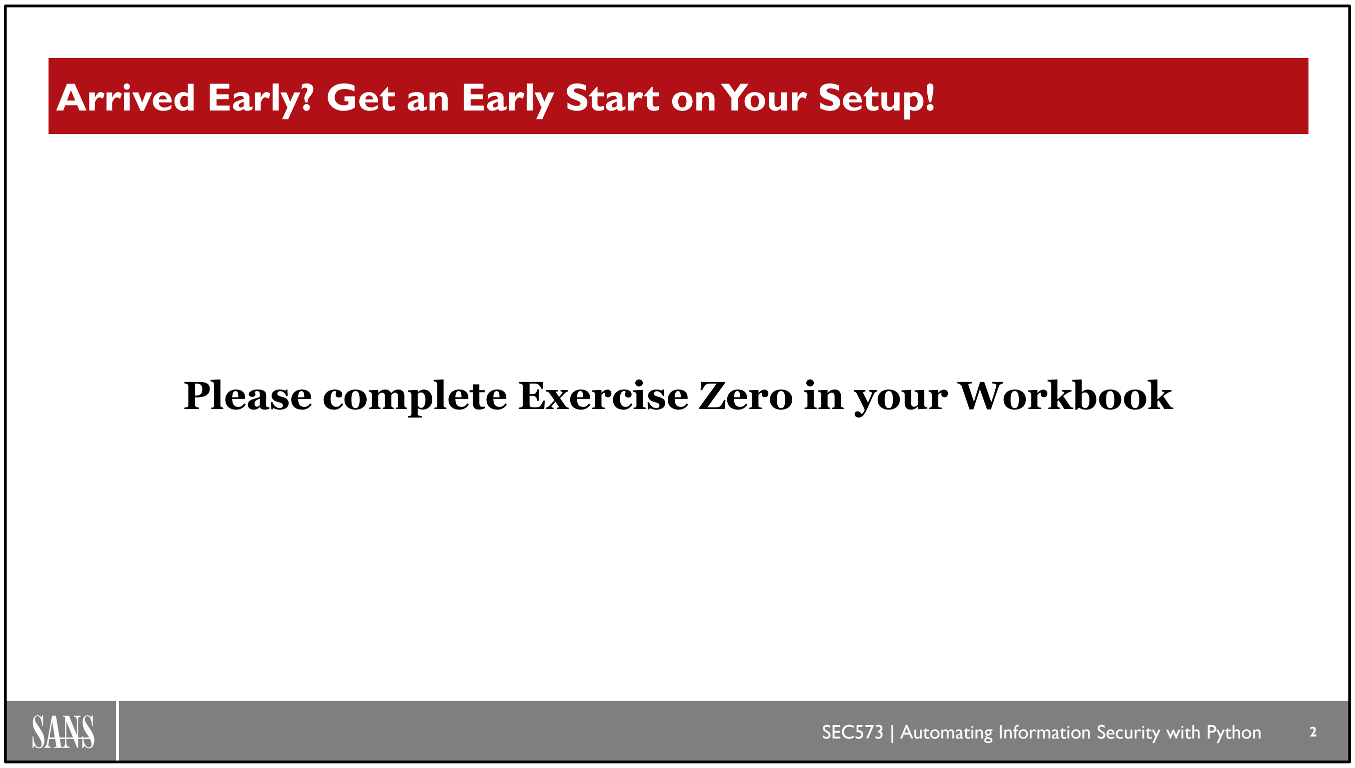
Essential Skills Workshop

Author: Mark Baggett

Twitter: @MarkBaggett

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 Exercise Zero in your Workbook

SANS SEC573 | Automating Information Security with Python 2

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

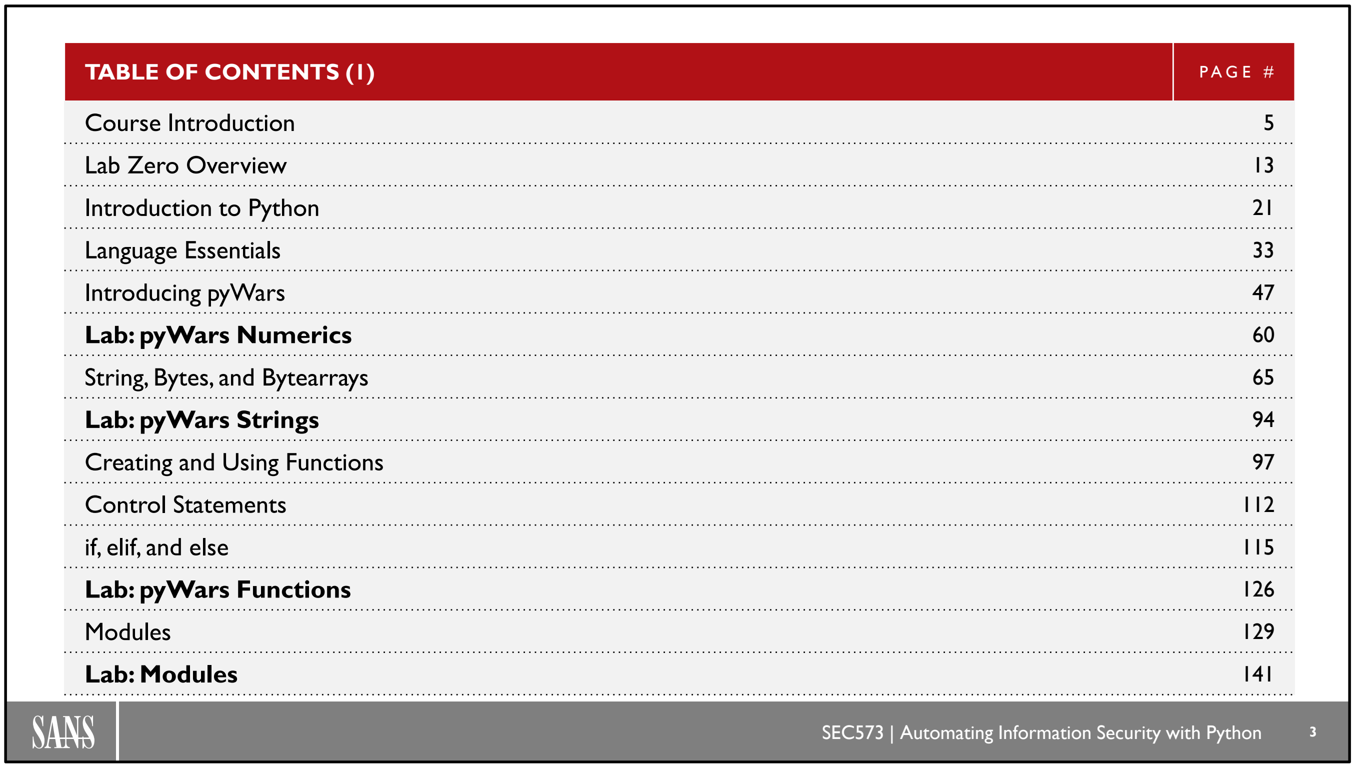


TABLE OF CONTENTS (1) — PAGE

This slide is a table of contents.

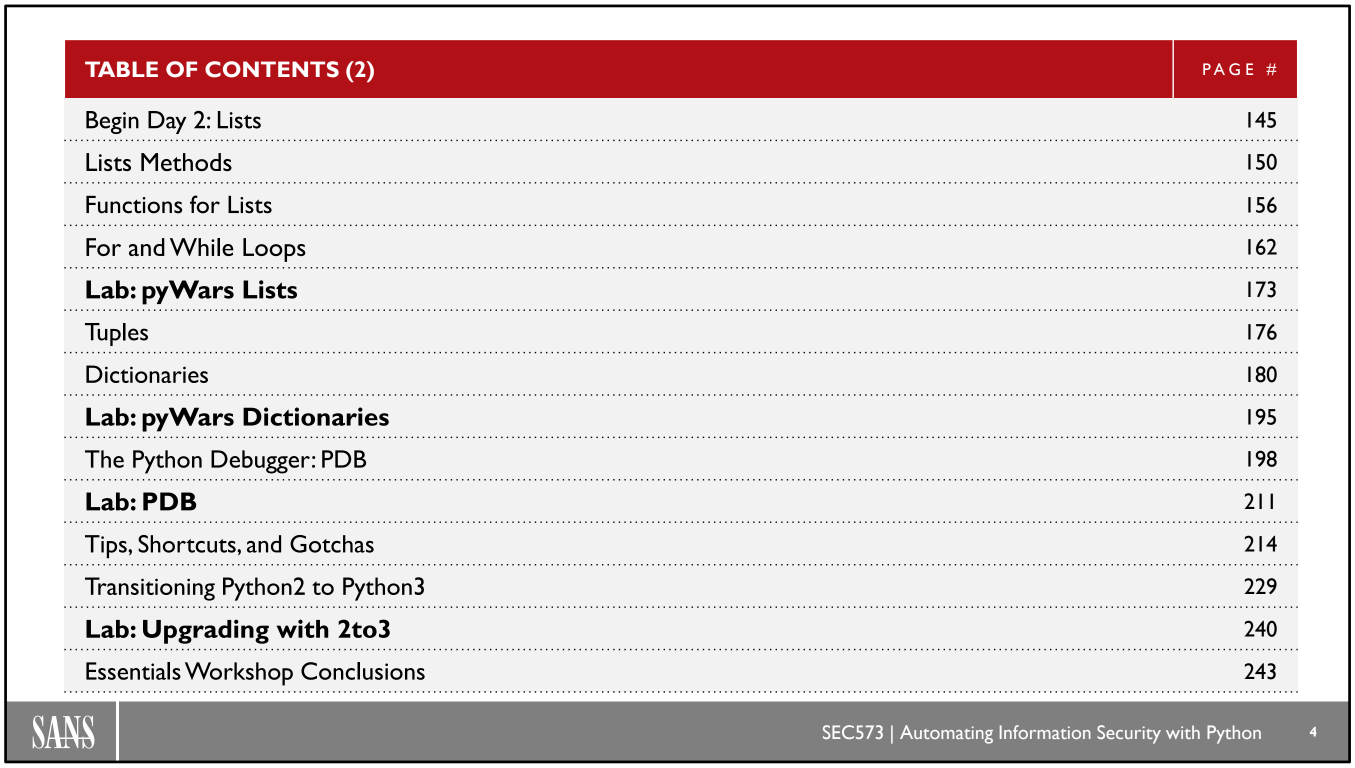
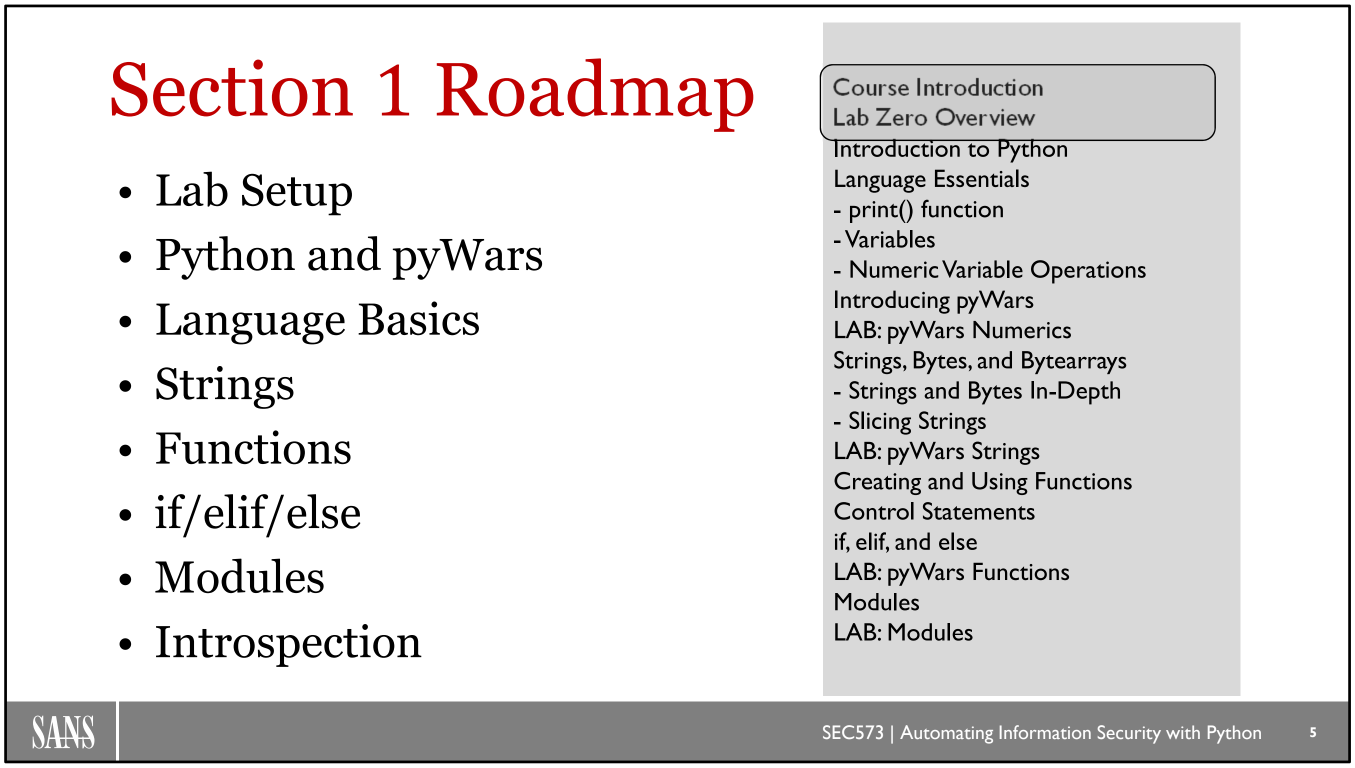


TABLE OF CONTENTS (2)

SANS

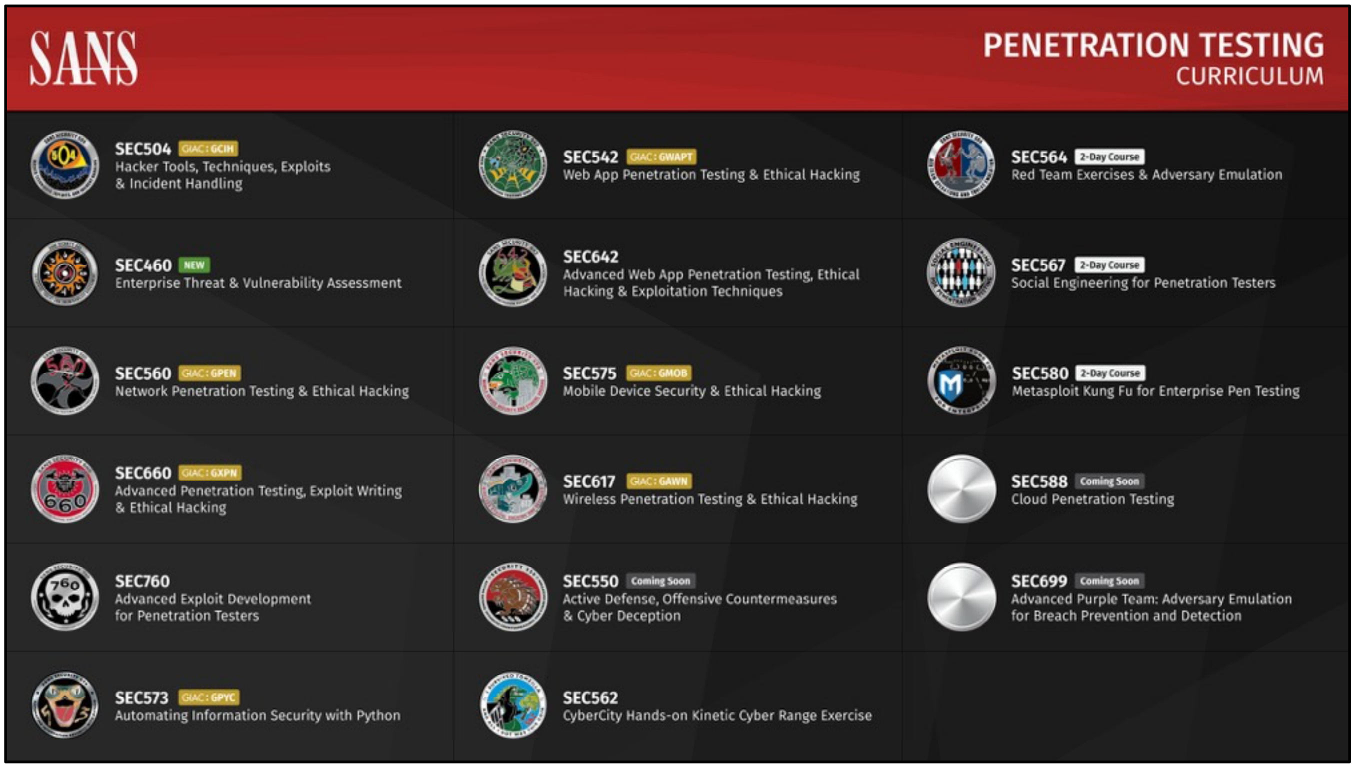
This slide is a table of contents.



Section 1 Roadmap

- Lab Setup
- Python and pyWars
- Language Basics
- Strings
- Functions
- if/elif/else
- Modules
- Introspection

This is a Roadmap slide.

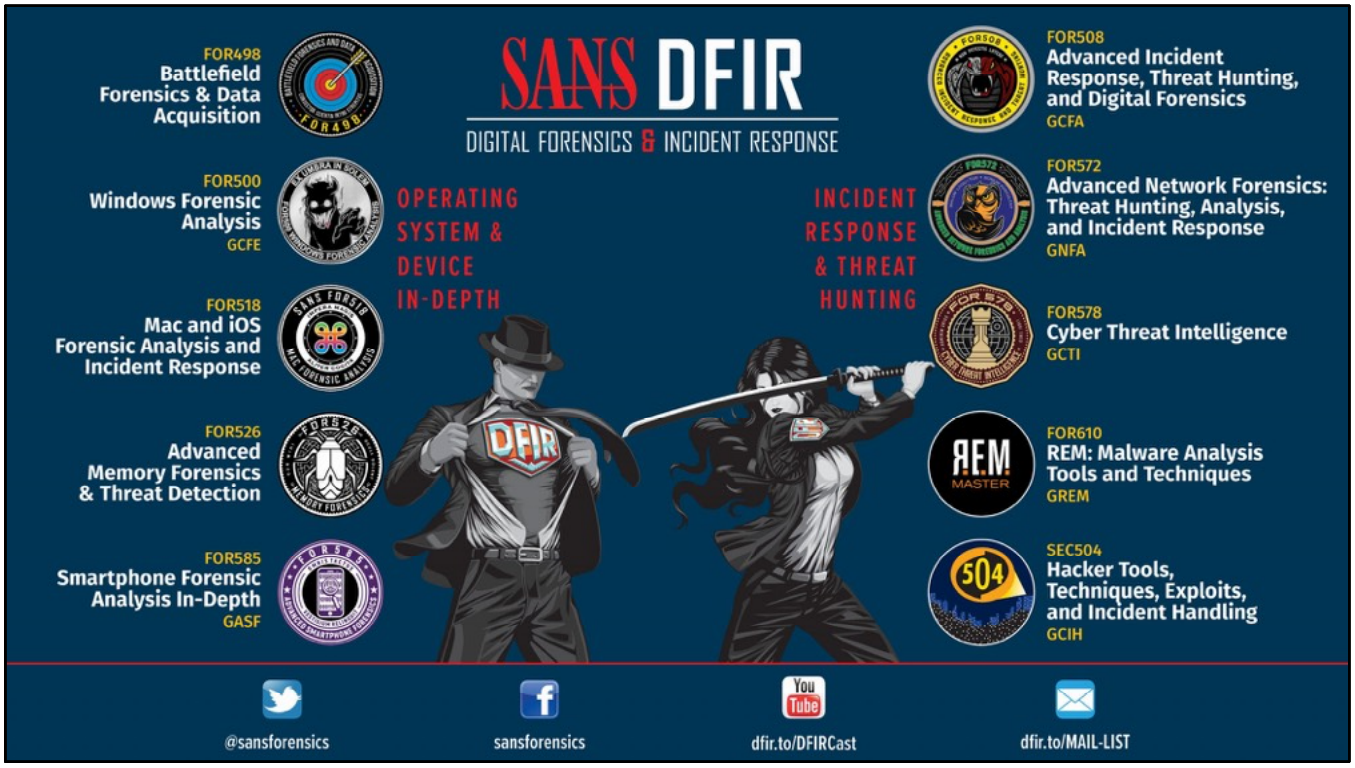


SANS

PENETRATION TESTING CURRICULUM

SEC504 GIAC: GCIH Hacker Tools, Techniques, Exploits & Incident Handling	SEC542 GIAC: GWAPT Web App Penetration Testing & Ethical Hacking	SEC564 2-Day Course Red Team Exercises & Adversary Emulation
SEC460 NEW Enterprise Threat & Vulnerability Assessment	SEC642 Advanced Web App Penetration Testing, Ethical Hacking & Exploitation Techniques	SEC567 2-Day Course Social Engineering for Penetration Testers
SEC560 GIAC: GPEN Network Penetration Testing & Ethical Hacking	SEC575 GIAC: GMOB Mobile Device Security & Ethical Hacking	SEC580 2-Day Course Metasploit Kung Fu for Enterprise Pen Testing
SEC660 GIAC: GXPN Advanced Penetration Testing, Exploit Writing & Ethical Hacking	SEC617 GIAC: GAWN Wireless Penetration Testing & Ethical Hacking	SEC588 Coming Soon Cloud Penetration Testing
SEC760 Advanced Exploit Development for Penetration Testers	SEC550 Coming Soon Active Defense, Offensive Countermeasures & Cyber Deception	SEC699 Coming Soon Advanced Purple Team: Adversary Emulation for Breach Prevention and Detection
SEC573 GIAC: GPYC Automating Information Security with Python	SEC562 CyberCity Hands-on Kinetic Cyber Range Exercise	

In this class, we will be teaching you to automate skills for penetration testers and defensive personnel. SANS Penetration Testing offers a wide variety of other courses that build skills for offensive and defensive personnel.



SANS DFIR

DIGITAL FORENSICS & INCIDENT RESPONSE

OPERATING SYSTEM & DEVICE IN-DEPTH

FOR498 Battlefield Forensics & Data Acquisition

FOR500 Windows Forensic Analysis GCFE

FOR518 Mac and iOS Forensic Analysis and Incident Response

FOR526 Advanced Memory Forensics & Threat Detection

FOR585 Smartphone Forensic Analysis In-Depth GASF

INCIDENT RESPONSE & THREAT HUNTING

FOR508 Advanced Incident Response, Threat Hunting, and Digital Forensics GCFA

FOR572 Advanced Network Forensics: Threat Hunting, Analysis, and Incident Response GNFA

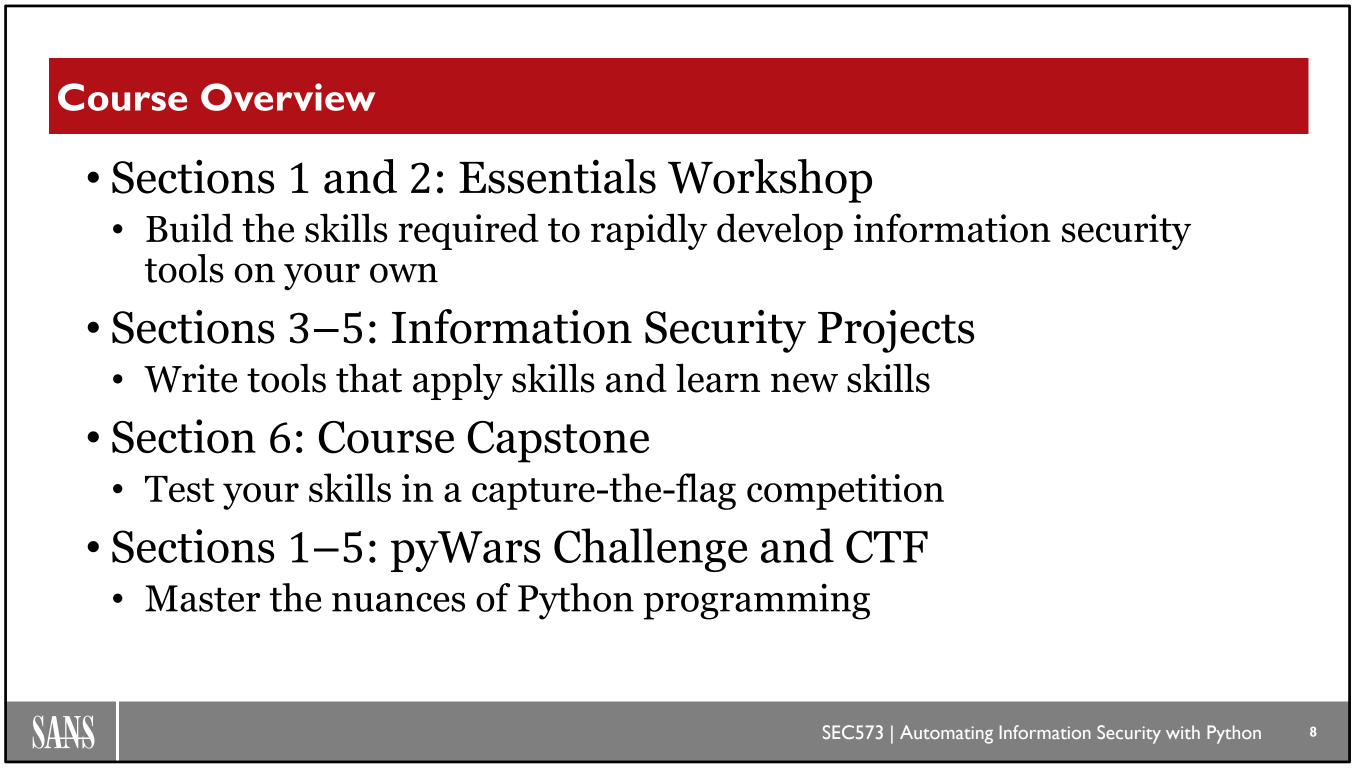
FOR578 Cyber Threat Intelligence GCTI

FOR610 REM: Malware Analysis Tools and Techniques GREM

SEC504 Hacker Tools, Techniques, Exploits, and Incident Handling GCIH

@sansforensics sansforensics dfir.to/DFIRCast dfir.to/MAIL-LIST

We are also teaching skills that will enable you to build new forensics tools and analyze forensics artifacts. SANS DFIR courses can teach you more about forensics techniques.

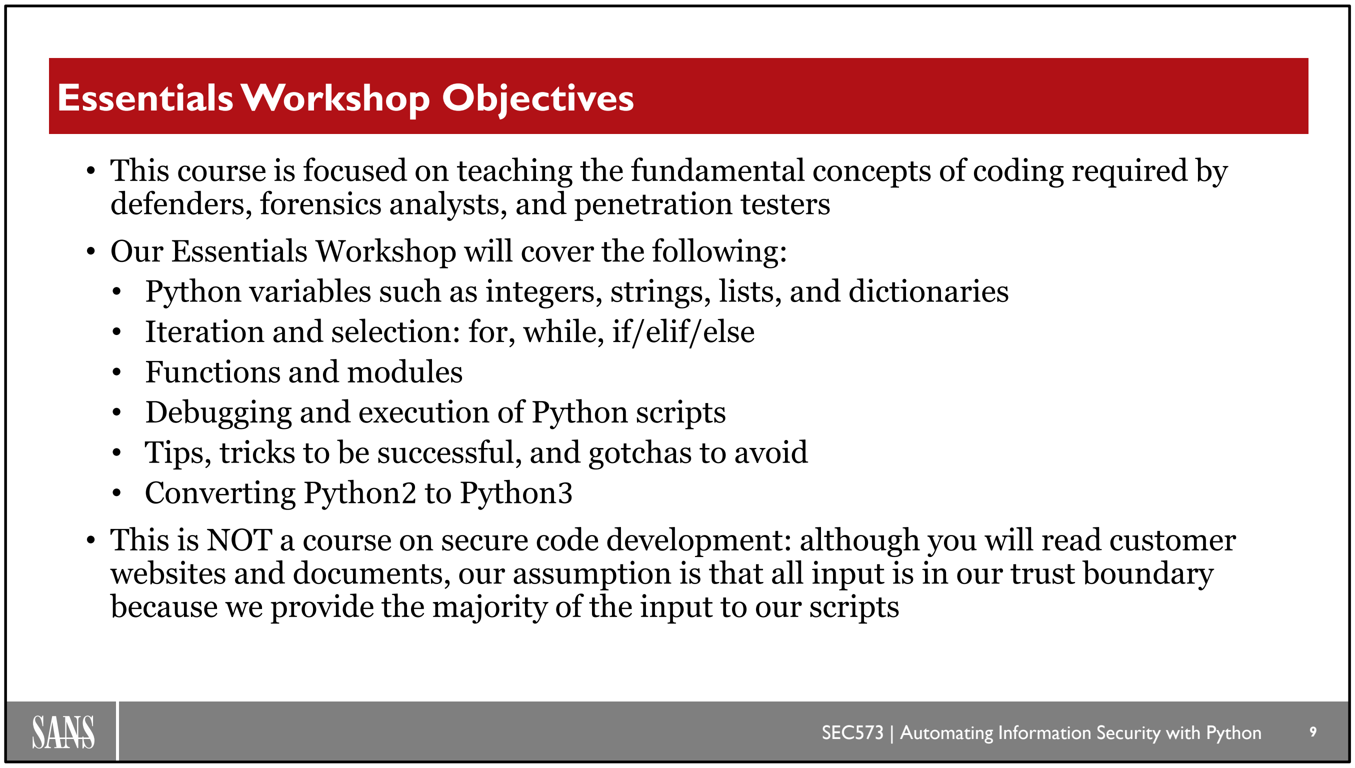


Course Overview

- Sections 1 and 2: Essentials Workshop
  - Build the skills required to rapidly develop information security tools on your own
- Sections 3–5: Information Security Projects
  - Write tools that apply skills and learn new skills
- Section 6: Course Capstone
  - Test your skills in a capture-the-flag competition
- Sections 1–5: pyWars Challenge and CTF
  - Master the nuances of Python programming

SANS SEC573 | Automating Information Security with Python 8

Here is the plan for the next few days. For the first two days, 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 days will reinforce your current knowledge and show you shortcuts and tricks to make your Python code more efficient. The next three days will put all those new skills to good use as we create apps for you to use. The last day 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.



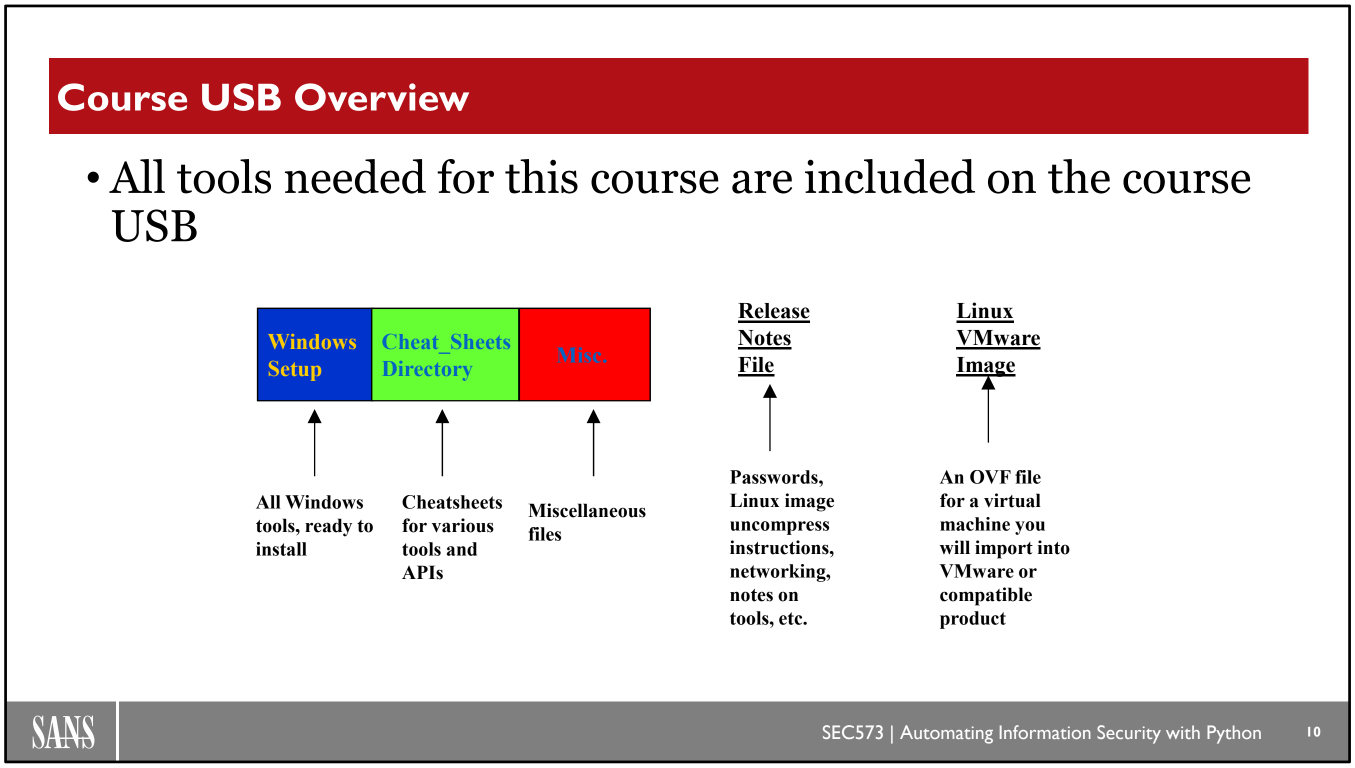
Essentials Workshop Objectives

- 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
  - Converting Python2 to Python3
- 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 the majority of the input to our scripts

SANS SEC573 | Automating Information Security with Python 9

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 Days 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 days 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.



Course USB Overview

- All tools needed for this course are included on the course USB

Windows Setup
Cheat_Sheets Directory
Misc.
Release Notes File
Linux VMware Image
All Windows tools, ready to install
Cheatsheets for various tools and APIs
Miscellaneous files
Passwords, Linux image uncompress instructions, networking, notes on tools, etc.
An OVF file for a virtual machine you will import into VMware or compatible product
SANS
SEC573 | Automating Information Security with Python
10

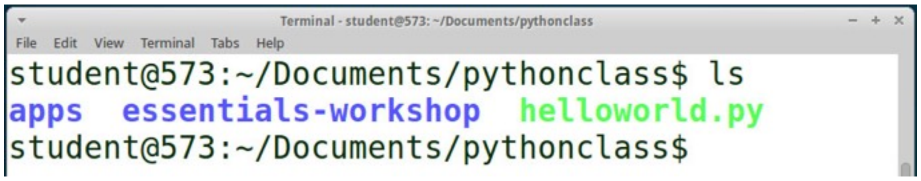
On the course USB, at the top of the directory structure, there are a handful of files and directories. Among the most important of these files are the Release Notes for the course USB. The Release Notes include the usernames and passwords for the VMware image for the course, as well as information about getting that VMware image uncompressed, booted, and networked. They also contain additional notes about some of the individual tools. ***As long as you have the course USB, you will also have the Release Notes, and therefore you will have access to the student password for the course VMware image. You won't be stranded without the password.***

Next, we have the course Windows directory. All Windows tools that you'll need for the course are included here. You will have to install them on your Windows machine for each exercise when the time comes. Please do wait, though, until we start a given exercise so that you understand a tool before you install it.

Another directory, called **Cheat_Sheets**, contains cheatsheets for various tools covered in this class. Feel free to look through the directory. These sheets can be helpful with exercises throughout the course.

The next element of the USB is a large virtual machine in the form of an OVF file and associated VMDK file. You will import this file into VMware Workstation, VMware Player, or VMware Fusion (the Mac OS X product). VMware is not included on the course USB due to redistribution limitations imposed by VMware, so you should download that now if you did not bring it with you.

Directories in the Virtual Machine


Terminal - student@573: ~/Documents/pythonclass
File Edit View Terminal Tabs Help

```
student@573:~/Documents/pythonclass$ ls
apps  essentials-workshop  helloworld.py
student@573:~/Documents/pythonclass$
```

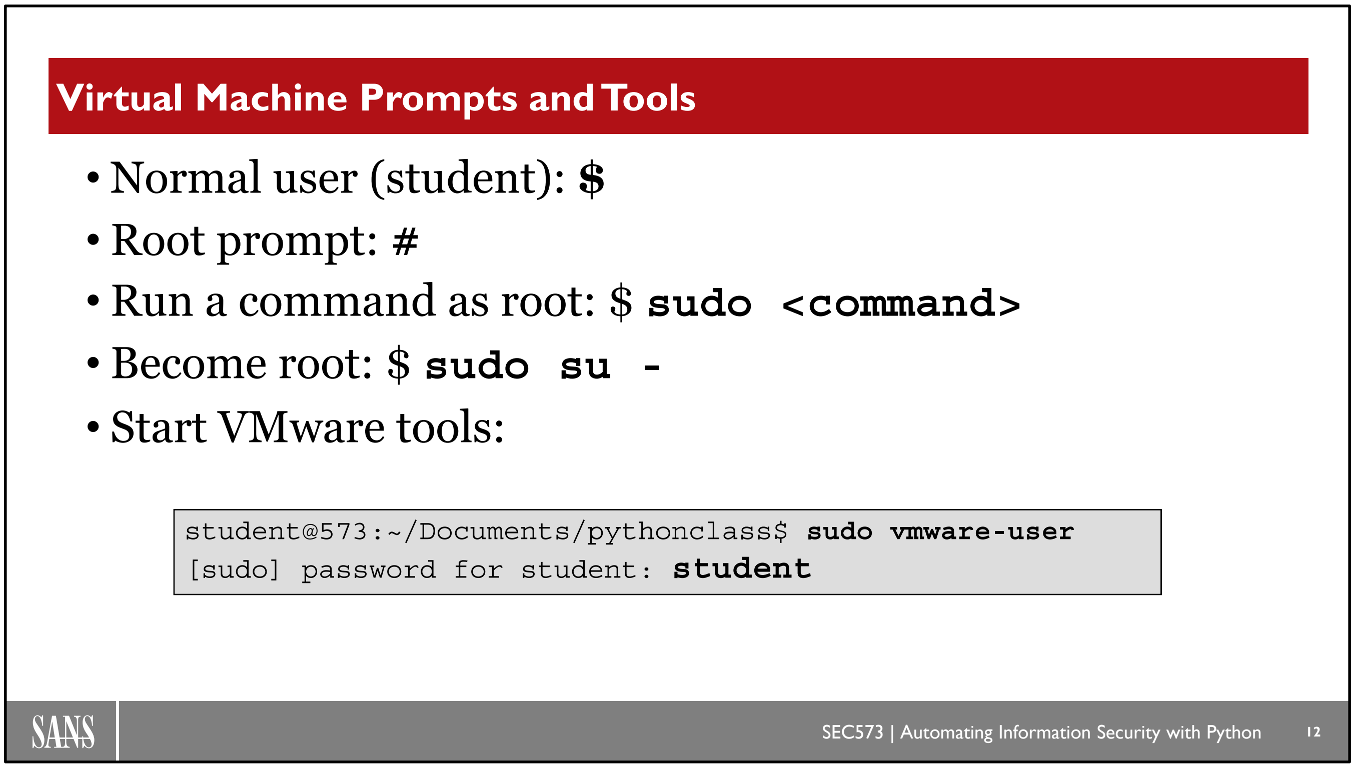
In the **~/Documents/pythonclass/** directory on the virtual machine, you can find the following directories:

- essentials-workshop: Code and labs for Sections 1 and 2
- apps: Different projects that we will develop in class on Sections 3–5

SANS

SEC573 | Automating Information Security with Python 11

On your virtual machine, you will see several directories. In the home directory in the Documents folder is a "pythonclass" directory. Under that directory are some subdirectories. The **~/Documents/pythonclass/essentials-workshop** directory contains all of the code examples and labs that we will work on for the first two sections. There is also an **apps** directory. We'll work on the program in the **~/Documents/pythonclass/apps** directory during Sections 3–5 of the course.



Virtual Machine Prompts and Tools

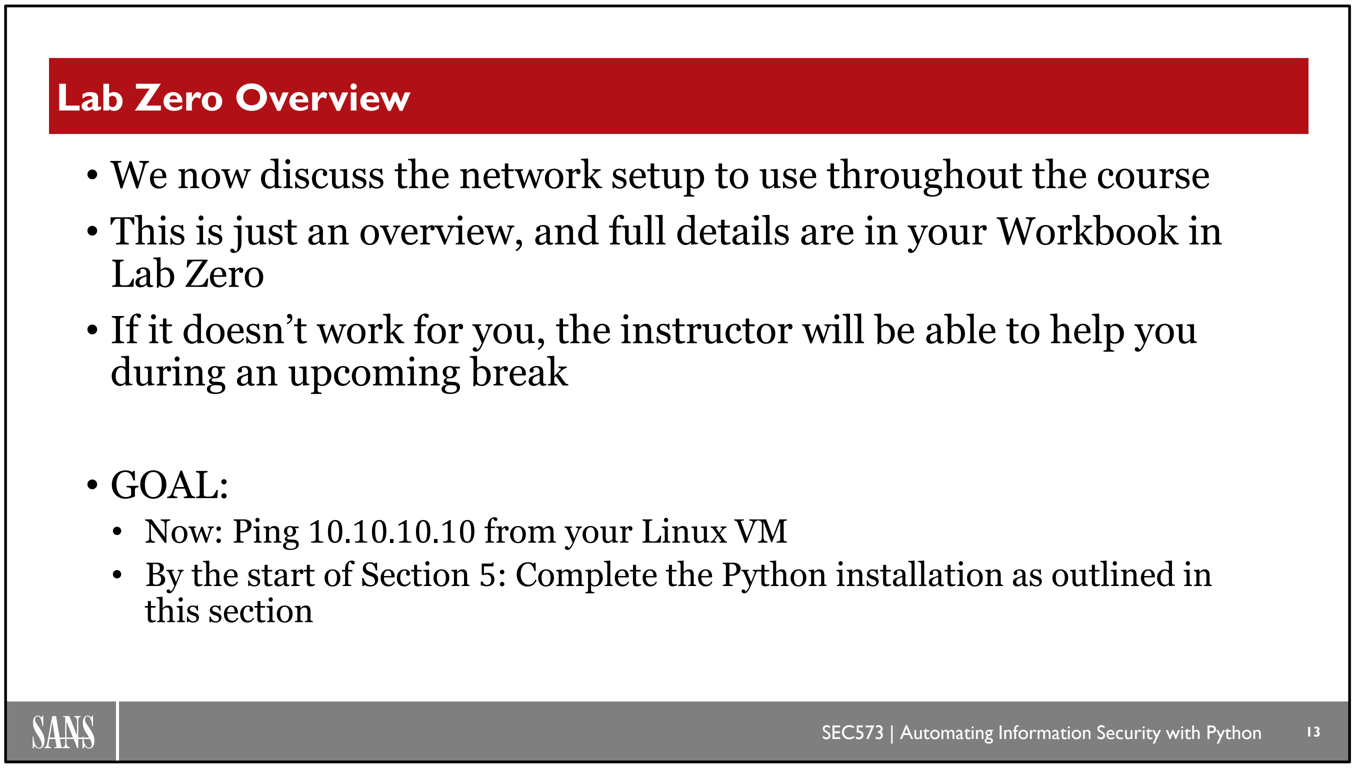
- Normal user (student): **$**
- Root prompt: **#**
- Run a command as root: $ **`sudo <command>`**
- Become root: $ **`sudo su -`**
- Start VMware tools:

```
student@573:~/Documents/pythonclass$ sudo vmware-user
[sudo] password for student: student
```

SANS SEC573 | Automating Information Security with Python 12

Throughout the book, you will see many labs and exercises. Pay close attention to the prompt in the labs. Most everything we do is run under the student account. The **$** (dollar sign) prompt indicates you do not currently have root access. Some of the programs, such as the sniffer you will write, require administrative permissions to run properly. You can become root temporarily or permanently by using the **sudo** command. The student user can run any command as root. The **sudo** command allows you to run a single command using the syntax **`sudo <command to run>`**. To become the root user in your virtual machine, you can use the command **`sudo su -`**. When you do, you see your prompt changes from the **$** (dollar sign) to the # (pound sign), which means you will be running as a root user.

Your virtual machine VMware tools are installed but are not running by default. If the version of VMware tools installed in the virtual machine is compatible with the VMware software installed on your host, you will find it much easier to use by enabling the software. However, if you have an older version of VMware software, then starting the tools may not work properly. To start the VMware tools, run the command **`$ sudo vmware-user`**. When the tools are enabled, the copy and paste functions, enhanced video drivers, and other useful functions will be enabled.



Lab Zero Overview

- We now discuss the network setup to use throughout the course
- This is just an overview, and full details are in your Workbook in Lab Zero
- If it doesn't work for you, the instructor will be able to help you during an upcoming break

- GOAL:
  - Now: Ping 10.10.10.10 from your Linux VM
  - By the start of Section 5: Complete the Python installation as outlined in this section

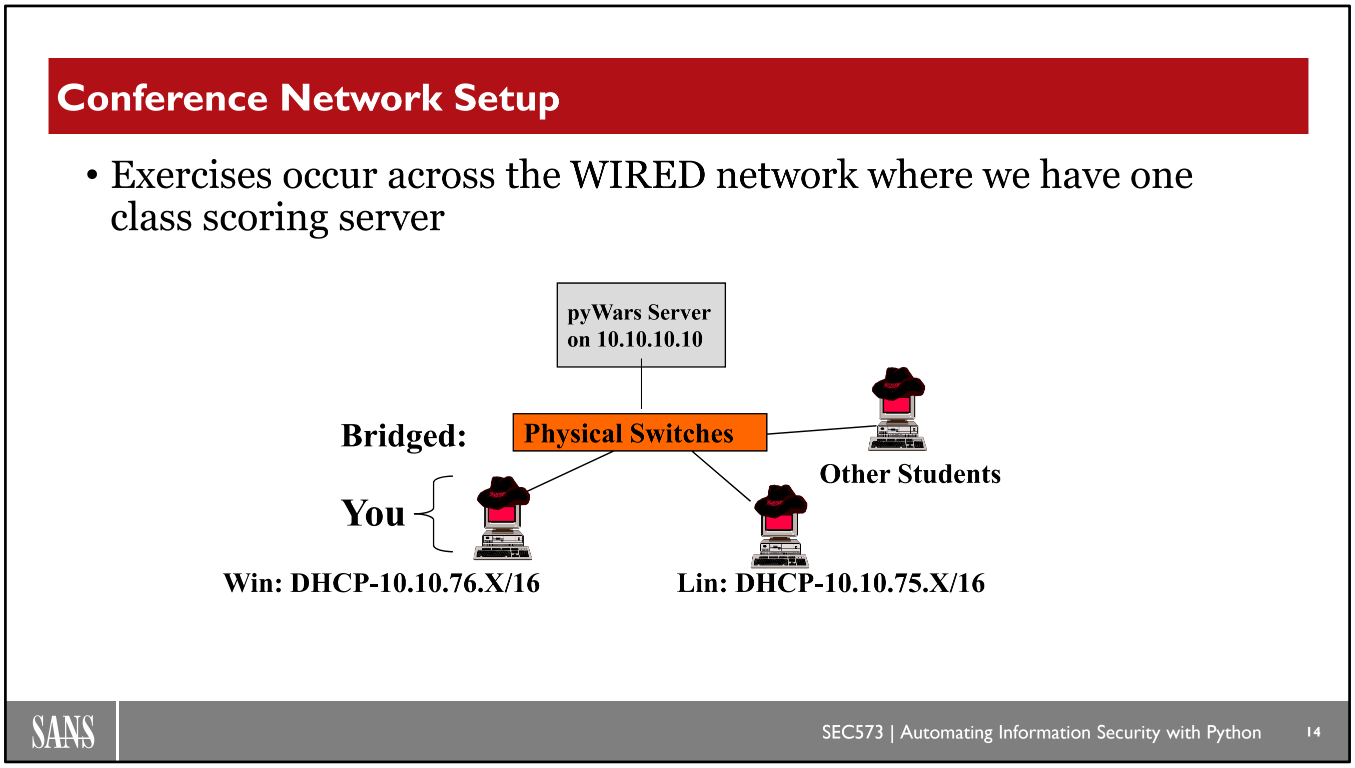
SANS

SEC573 | Automating Information Security with Python 13

We'll now discuss the configuration for the machine that we use for the course. You can follow along and set up the network configuration during this session if you'd like.

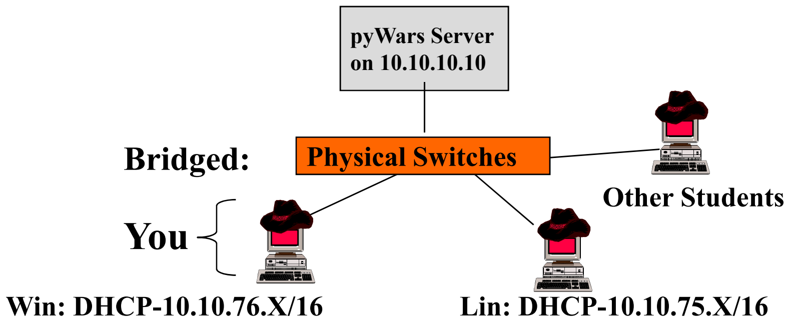
Please note: If your network configuration does not work, the instructor reserves the right to help you get it working during an upcoming break or another appropriate opportunity. Not everyone will get the network configuration functional right now, and the instructor needs to keep the course moving. **Don't worry if the networking doesn't work for you right now**. The instructor will help you get it functioning during the first lab.

The immediate need is to work on a Linux virtual machine and be able to ping 10.10.10.10. When that is done, you are ready for the first four sections. By Section 5, you need to have Python installed on a Windows host, as outlined in workbook Lab Zero.



Conference Network Setup

- Exercises occur across the WIRED network where we have one class scoring server



SANS SEC573 | Automating Information Security with Python 14

In classroom environments, the instructor will provide a centralized pyWars scoring server. In this case, you are given an IP address of the scoring server. You should be aware that you are sharing a network with other students, so take precautions to protect yourself while interacting with the scoring server. The safest approach is to disconnect your network cable when it is not in use.

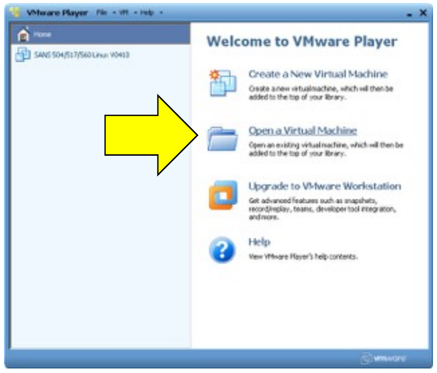
NOTE: ALTHOUGH THIS CLASS TEACHES YOU TO USE PYTHON TO PERFORM EXPLOITATION, EXPLOITATION OF ANY HOST, OTHER THAN YOUR OWN, IS NOT PERMITTED. ATTACKING OTHER MACHINES OR THE SCORING SERVER WILL GET YOU DISMISSED FROM THE CLASS. IT IS A VIOLATION OF THE SANS CODE OF ETHICS TO ATTACK OTHER MACHINES OR THE SCORING SERVER.

Import Your Linux Virtual Machine

- If you haven't already done so, import the OVF virtual machine
  - If VMware is installed properly, you can just double-click on the OVF file on your USB drive
  - If you get an OVF consistency check, just select "Retry"
  - If double-click doesn't work, then do one of the following:
    - select File > Import in VMware
    - Select "Open a Virtual Machine" and select the OVF file
- Run VMware, open the VM, and boot it
- In Linux, log in to the system:
  - Username: **student**
  - Password: **student**
  - Change the student password:

```
$ passwd
```

  - Enter a new password twice and remember it!
- **`Can you PING 10.10.10.10? You're ready for today!`**



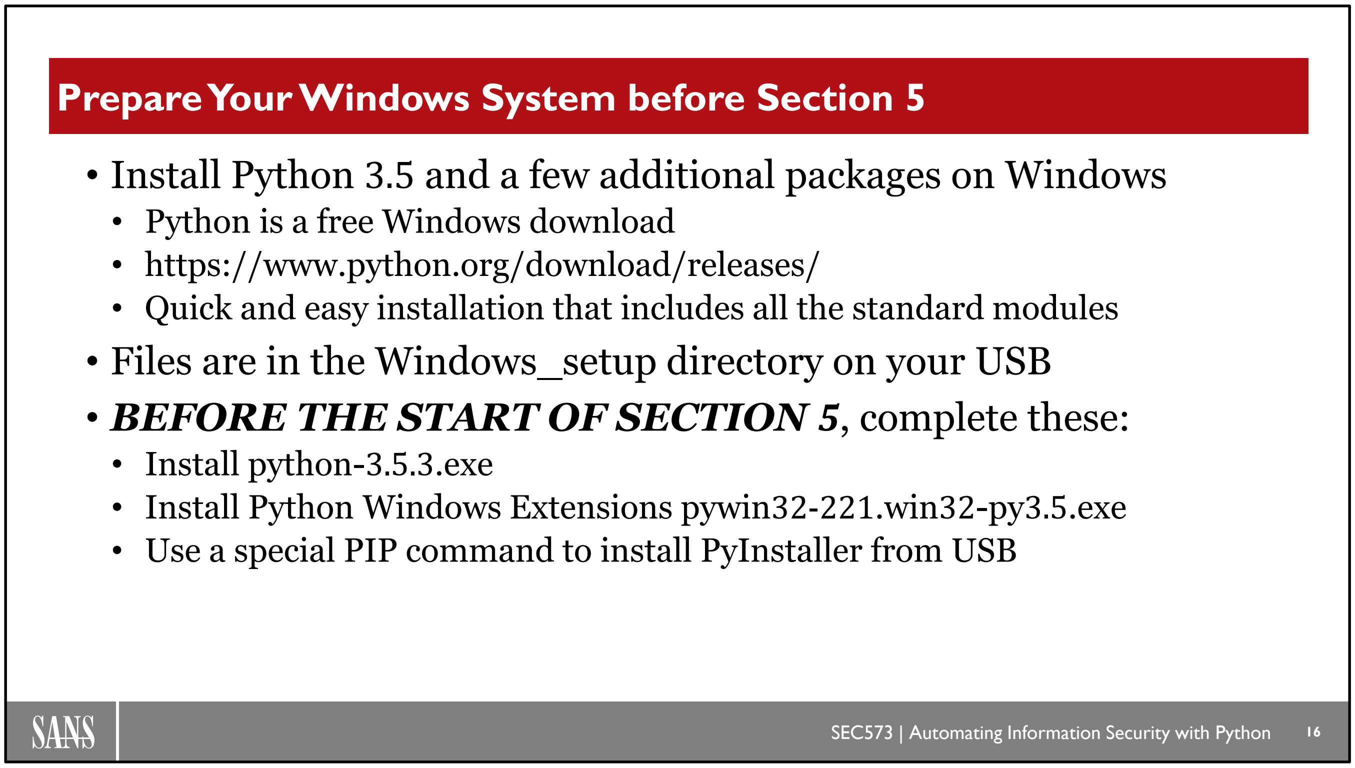
The course virtual machine is distributed as an importable OVF file to maximize compatibility with different virtualization software packages. The easiest way to begin the import process is just to double-click on the OVF file on your USB drive. If that doesn't automatically begin the import, then depending upon your software, you will either click "File -> Import in VMware" or "Open a Virtual Machine" and select the OVF file. Give it a location on your host machine to store the files when prompted. This virtual machine requires approximately 10 GB of space on your hard drive.

After the import is complete, boot your new virtual guest system. When prompted, log in to the guest machine using the following credentials:

Username: **student**
Password: **student**

You can change the student password to a value you'll remember (make sure it isn't easily guessed or cracked). You will be connected to a network with other students in this course, so you do not want them to know the password for your Linux VMware image. To change the student password, use:

```
$ passwd
```



Prepare Your Windows System before Section 5

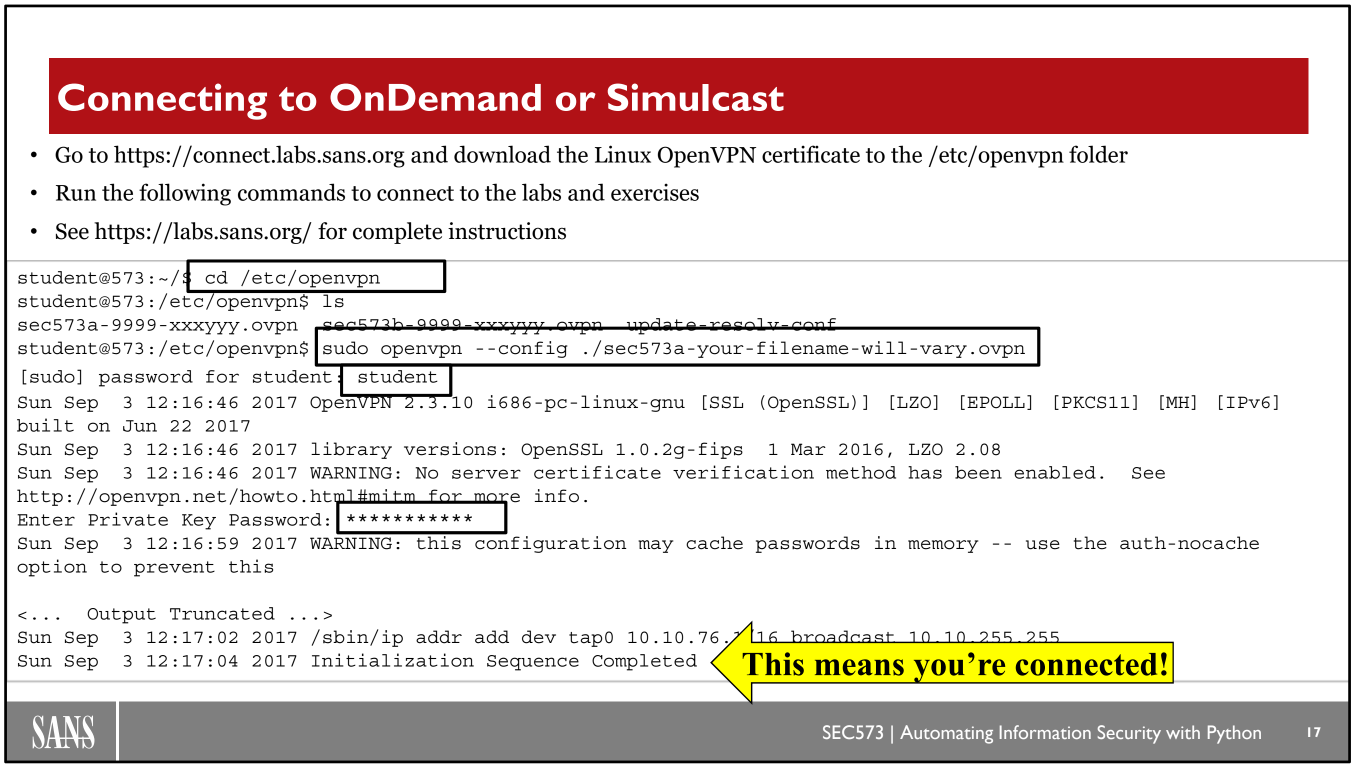
- Install Python 3.5 and a few additional packages on Windows
  - Python is a free Windows download
  - https://www.python.org/download/releases/
  - Quick and easy installation that includes all the standard modules
- Files are in the Windows_setup directory on your USB
- ***BEFORE THE START OF SECTION 5***, complete these:
  - Install python-3.5.3.exe
  - Install Python Windows Extensions pywin32-221.win32-py3.5.exe
  - Use a special PIP command to install PyInstaller from USB

SANS SEC573 | Automating Information Security with Python 16

Let's prepare our Windows environment, run Python, and create a distributable Windows version of our Python programs. As of this writing, the latest version of Python 3 is not compatible with PyInstaller. You can use Python version 3.5 but not 3.6. Your Windows environment will require Python 3 and PyInstaller. Normally, you would download the Python setup files from the internet, but they have been provided for you in the **Windows_setup** directory on your course USB.

By using the versions that are provided on the USB, you will find that you will not run into any known versioning issues associated with packages used in the course and that the syntax will match the book.

To manually download the PyInstaller package, go to http://www.pyinstaller.org. The Python for Windows Extensions is available for download at http://sourceforge.net/projects/pywin32/



Connecting to OnDemand or Simulcast

- Go to https://connect.labs.sans.org and download the Linux OpenVPN certificate to the /etc/openvpn folder
- Run the following commands to connect to the labs and exercises
- See https://labs.sans.org/ for complete instructions

```
student@573:~/$ cd /etc/openvpn
student@573:/etc/openvpn$ ls
sec573a-9999-xxxyyy.ovpn  sec573b-9999-xxxyyy.ovpn  update-resolv-conf
student@573:/etc/openvpn$ sudo openvpn --config ./sec573a-your-filename-will-vary.ovpn
[sudo] password for student: student
Sun Sep  3 12:16:46 2017 OpenVPN 2.3.10 i686-pc-linux-gnu [SSL (OpenSSL)] [LZO] [EPOLL] [PKCS11] [MH] [IPv6]
built on Jun 22 2017
Sun Sep  3 12:16:46 2017 library versions: OpenSSL 1.0.2g-fips  1 Mar 2016, LZO 2.08
Sun Sep  3 12:16:46 2017 WARNING: No server certificate verification method has been enabled.  See
http://openvpn.net/howto.html#mitm for more info.
Enter Private Key Password: ***********
Sun Sep  3 12:16:59 2017 WARNING: this configuration may cache passwords in memory -- use the auth-nocache
option to prevent this

<...  Output Truncated ...>
Sun Sep  3 12:17:02 2017 /sbin/ip addr add dev tap0 10.10.76./16 broadcast 10.10.255.255
Sun Sep  3 12:17:04 2017 Initialization Sequence Completed
```

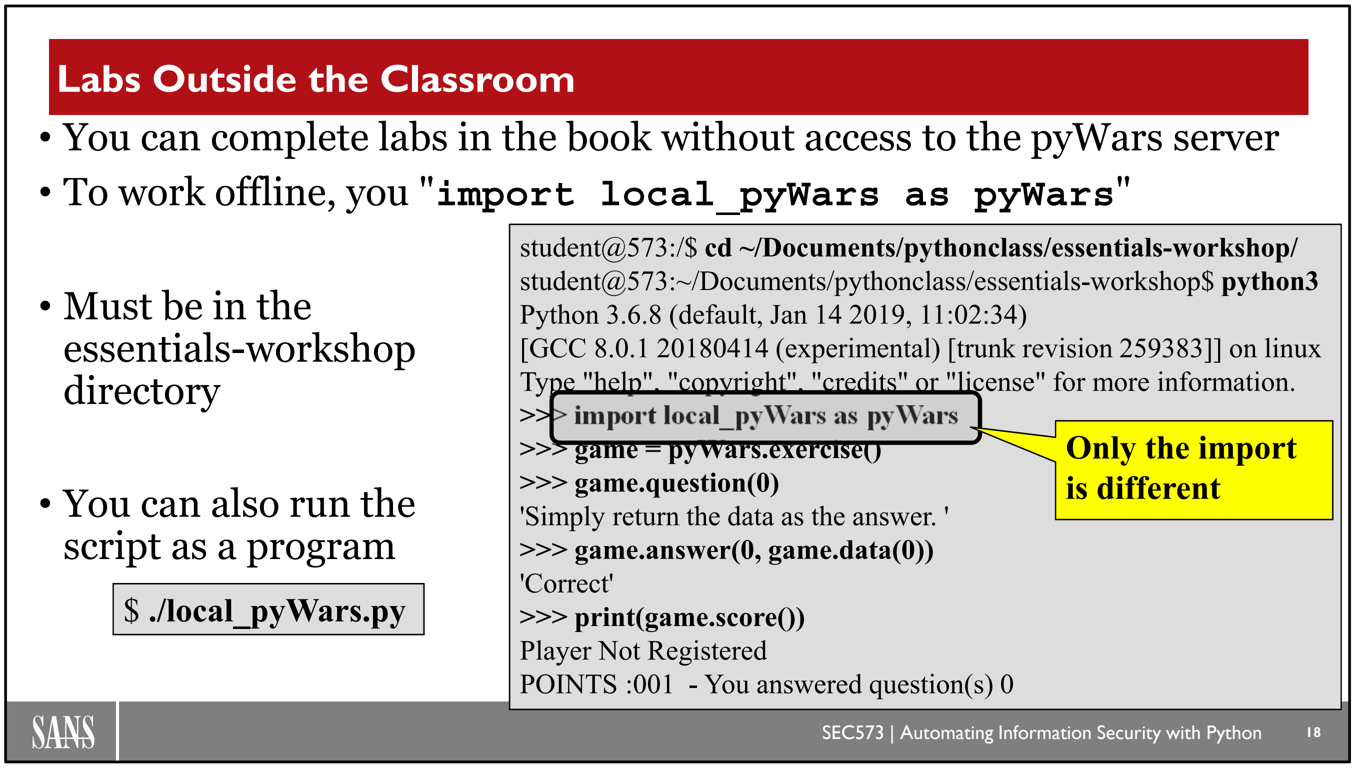
This means you're connected!

SANS

SEC573 | Automating Information Security with Python 17

Those of you who are taking the class OnDemand or have added the OnDemand bundle to your live class will need to connect to the Lab environment with OpenVPN before trying any of the pyWars-based labs. You can retrieve your VPN keys from https://connect.labs.sans.org. That website will have links to two Linux OpenVPN certificates. One of the certificates is used for Sections 1–5 of the course, and the other is used for Section 6. **Download the certificates** to your machine, then use the command **"sudo cp <path to downloaded certificate> /etc/openvpn/"** to move the certificates to the /etc/openvpn directory. Next, open a terminal, change to the /etc/openvpn directory, and **launch openvpn using the commands shown in the slide above**. After launching OpenVPN, you will be prompted for two passwords. The first one is the password to your student account. If you have not changed it, then the password is "**student**". Next, you will be prompted to "Enter Private Key Password". This password is typically "**VpnPassword**", but check the email and web links to confirm your exact password.

Once the connection is completed, the last line displayed will read "`Initialization Sequence Completed`". This is your indication that you are connected. Now you should be able to ping 10.10.10.10 in another terminal to verify your connection. You can minimize this window or otherwise leave the terminal window alone while you are performing your labs. Come back to the window and hit "CTRL-C" when you want to disconnect from the remote servers.



Labs Outside the Classroom

- You can complete labs in the book without access to the pyWars server
- To work offline, you "`import local_pyWars as pyWars`"
- Must be in the essentials-workshop directory
- You can also run the script as a program

```
$ ./local_pyWars.py
```

```
student@573:/$ cd ~/Documents/pythonclass/essentials-workshop/
student@573:~/Documents/pythonclass/essentials-workshop$ python3
Python 3.6.8 (default, Jan 14 2019, 11:02:34)
[GCC 8.0.1 20180414 (experimental) [trunk revision 259383]] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import local_pyWars as pyWars
>>> game = pyWars.exercise()
>>> game.question(0)
'Simply return the data as the answer. '
>>> game.answer(0, game.data(0))
'Correct'
>>> print(game.score())
Player Not Registered
POINTS :001  - You answered question(s) 0
```

Only the import is different

SANS SEC573 | Automating Information Security with Python 18

I wanted to provide you with a means of completing all of the labs in your book when the pyWars server is not around. This enables you to go back through or complete labs during the evenings or weeks after class has finished. To meet that objective, I have created a separate module that provides access to pyWars server-like functions while offline. So even if you do not have access to the pyWars server, you can still complete all of the labs in the book. In your essentials-workshop directory, there is a module called "local_pyWars". This module provides a minimal set of capabilities that allows you to complete the labs without the use of the pyWars server.

To play offline, first you change to the essentials-workshop directory. Then start Python and **import local_pyWars as pyWars**. The rest of your syntax will be exactly the same as when using the in-classroom or OnDemand server. This makes switching back and forth between the two environments very easy. So you can work on labs offline, then change that one line and post them to the server.

```
$ cd ~/Documents/pythonclass/essentials-workshop/
$ python3
>>> import local_pyWars as pyWars
>>> game = pyWars.exercise()
```

You can also just run the local_pyWars.py program as a program, and it will drop you into a Python shell with pyWars loaded. If you pass "victors" as a command line argument, it will start a Python shell with the Hall of Fame questions loaded for you.

```
$ cd ~/Documents/pythonclass/essentials-workshop/
$ ./local_pyWars.py
Welcome to pyWars!
```

pyWars Hall of Fame

- There are three pyWars servers with challenges for you this week: Online Server, Offline Server, and Hall of Fame Server
- Individuals who complete all the pyWars challenges can write a Hall of Fame challenge that bears their name
- Challenges are added to the virtual machine along with course updates
- Pass "victors" as a command line argument: **$ ./local_pyWars.py VICTORS**
- Or pass "VICTORS" to your exercise object:

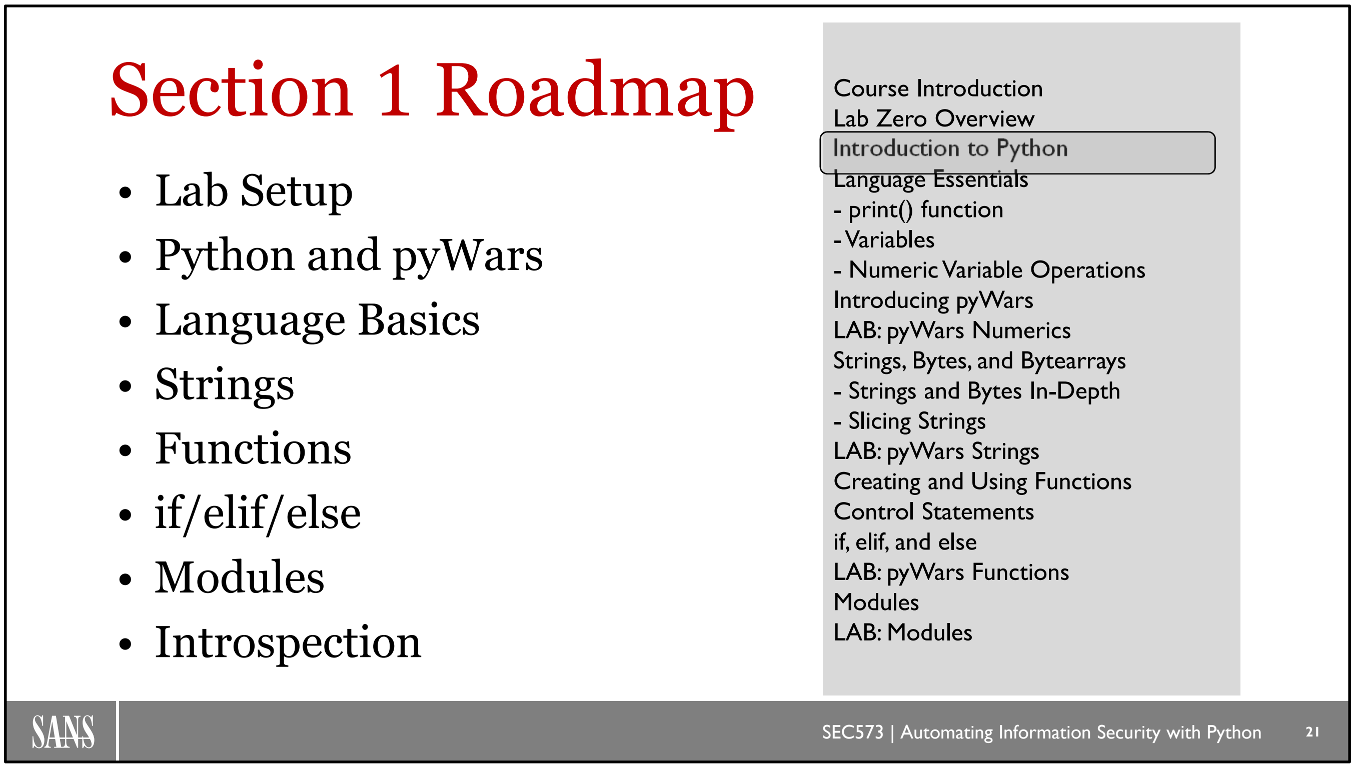
```
>>> import local_pyWars as pyWars
>>> game = pyWars.exercise("VICTORS")
>>> game.question(1)
'NAME: Chris Griffith - Find the string, on the hidden port, running the
fun_server.py, extracted from the zipfile, that is the base64 in .data()'
```

You can play against three servers this week, and you can take two of them with you when you leave. In addition to the offline server, you will also have a copy of the Hall of Fame questions. These are questions created by individuals who are able to complete all of 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, pass the word **VICTORS** to your local_pyWars.exercise() method. Then you can play through challenges created by some of the best Python programmers in the world!

Complete Lab Zero

- Please continue to work through the extraction of the Linux virtual machine
- You may do the installation of Python on Windows anytime between now and Section 5
- While you complete the installation, we will introduce Python
- The goal is still to ping 10.10.10.10 from your Linux host

Please continue to work through the extraction of your virtual machine as we go through the next section. Immediately following this section is an exercise that uses your Linux virtual machine. If you can ping 10.10.10.10 from your Linux virtual machine, you should be ready for that exercise.



Section 1 Roadmap

- Lab Setup
- Python and pyWars
- Language Basics
- Strings
- Functions
- if/elif/else
- Modules
- Introspection

SANS SEC573 | Automating Information Security with Python 21

This is a Roadmap slide.

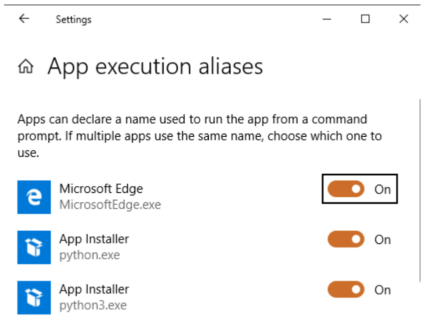
What Is Python?

- 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 over the last five years as a result of its ease of use and rich community support. Another reason for its popularity: One of the stated goals of the Python language development group is to make 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".

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)



App execution aliases

Apps can declare a name used to run the app from a command prompt. If multiple apps use the same name, choose which one to use.

Microsoft Edge MicrosoftEdge.exe	On
App Installer python.exe	On
App Installer python3.exe	On

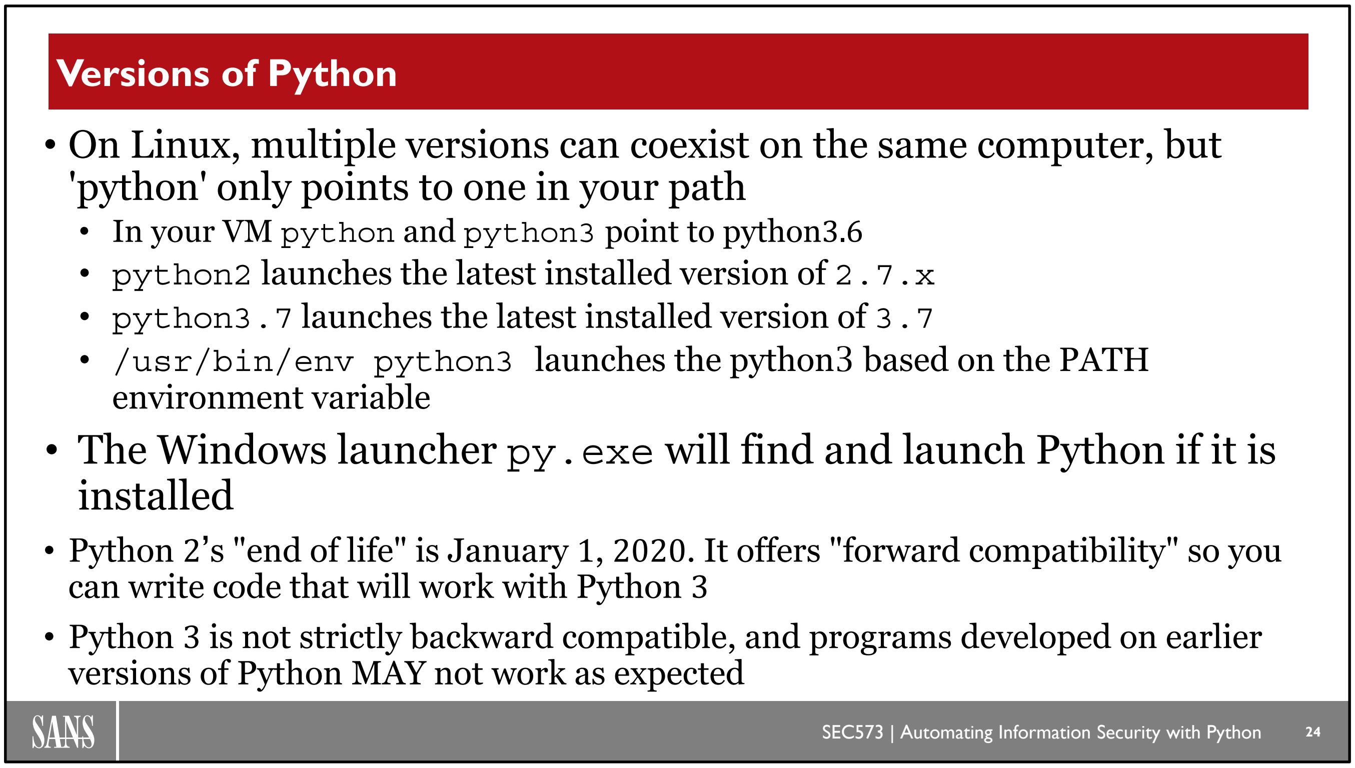
```
C:\> bash -c "python -c 'print(\"Python is on Windows with WSL!\")'"
Python is on Windows with WSL!
```

Python is already installed on most Linux distributions. It even comes preinstalled on your Macintosh, although it is an older version. 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 actually 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.



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 python3.6
  - `python2` launches the latest installed version of `2.7.x`
  - `python3.7` launches the latest installed version of `3.7`
  - `/usr/bin/env python3` launches the python3 based on the PATH environment variable
- The Windows launcher `py.exe` will find and launch Python if it is installed
- 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

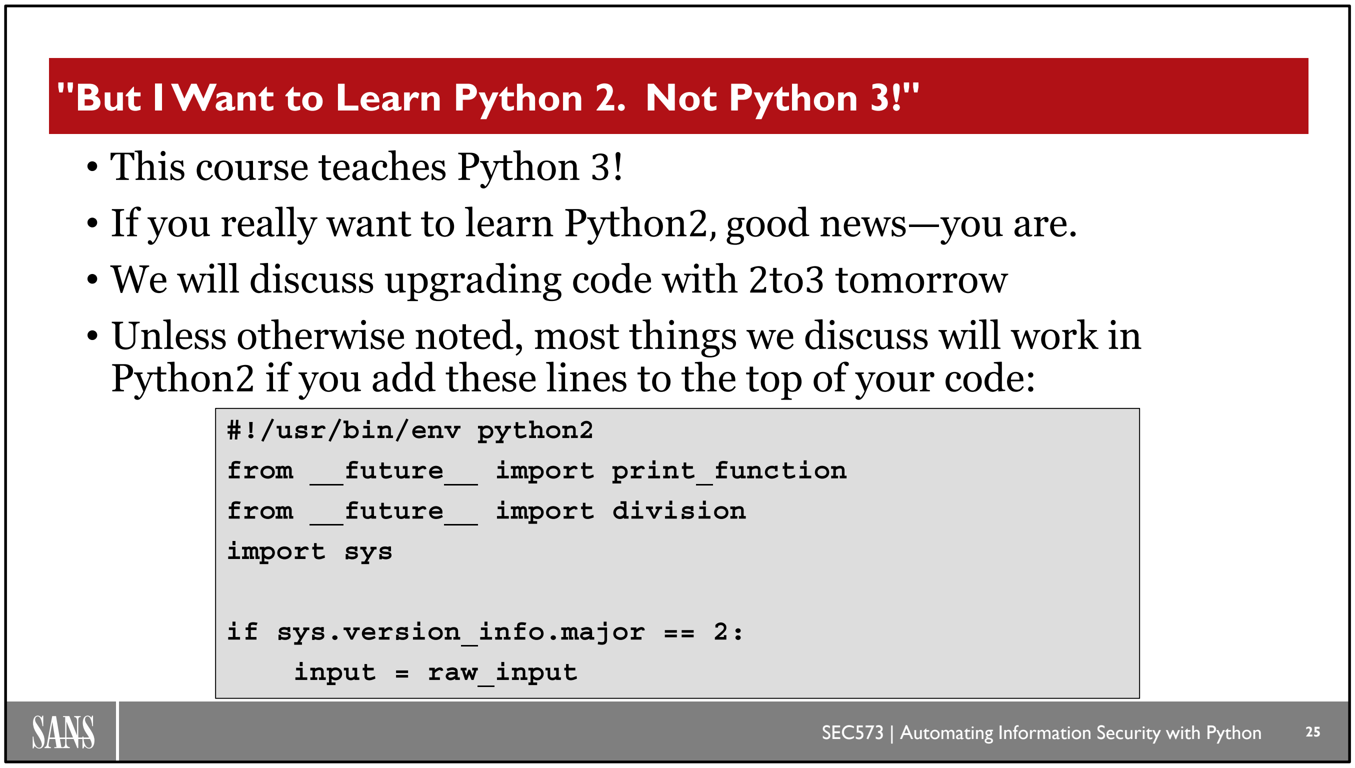
SANS | SEC573 | Automating Information Security with Python 24

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 interpreters prior to version 3.0 are 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 version 3 is different. It doesn't make any promises of backward compatibility. In this course, we will be developing Python 3 programs. Where necessary we will also discuss migrating Python2 to Python3.

You can run different versions of Python by launching different processes. Typing **python** will launch the default version of Python, which is version 2.7 on most systems long after PEP 394 says it should point to Python 3. In your VM, the 'python' command is a symbolic link to the 'python3.6' binary on your system. Ubuntu 18 has a dependency on Python3.6, so this will very likely be the case on every Ubuntu 18 system. If you want to run Python3.7, you use the 'python3.7' command. There is also a 'python3' symbolic link that points to Python 3.6.

Using the command "**/usr/bin/env python3**" is often more reliable than using a hardcoded path to launch Python3 because people may decide to change the installation path. 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.



"But I Want to Learn Python 2. Not Python 3!"

- This course teaches Python 3!
- If you really want to learn Python2, good news—you are.
- We will discuss upgrading code with 2to3 tomorrow
- Unless otherwise noted, most things we discuss will work in Python2 if you add these lines to the top of your code:

```
#!/usr/bin/env python2
from __future__ import print_function
from __future__ import division
import sys

if sys.version_info.major == 2:
    input = raw_input
```

SANS SEC573 | Automating Information Security with Python 25

You may find yourself in a situation where you are forced to support Python2. Perhaps the code has dependencies on libraries (modules) that are not supported on Python3. Or perhaps no one has taken the time to upgrade the code from Python2 to Python3. In Section 2 of this course, we will dig into what it takes to upgrade your code from Python2 to Python3. However, if you must write code that will be run through the Python2 interpreter, you can add a few lines of code to the top of your program to make your life easier. When the lines shown above are added to the top of your Python2 program, it will cause the interpreter to behave more like Python3. With these lines and a few other changes, it is possible to write programs that will work perfectly fine when run through either a Python2 or a Python3 interpreter.

The line "`from __future__ import print_function`" eliminates the Python2 print keyword and replaces it with a print function that is compatible with Python3.

The line "`from __future__ import division`" makes Python2 do division in the same way that Python3 does it. We will discuss the difference in detail at the end of Section 2 of this course.

Last, this code checks to see if we are running Python version 2 and if we are, overwrites the code associated with the input function with the code from the raw_input function.

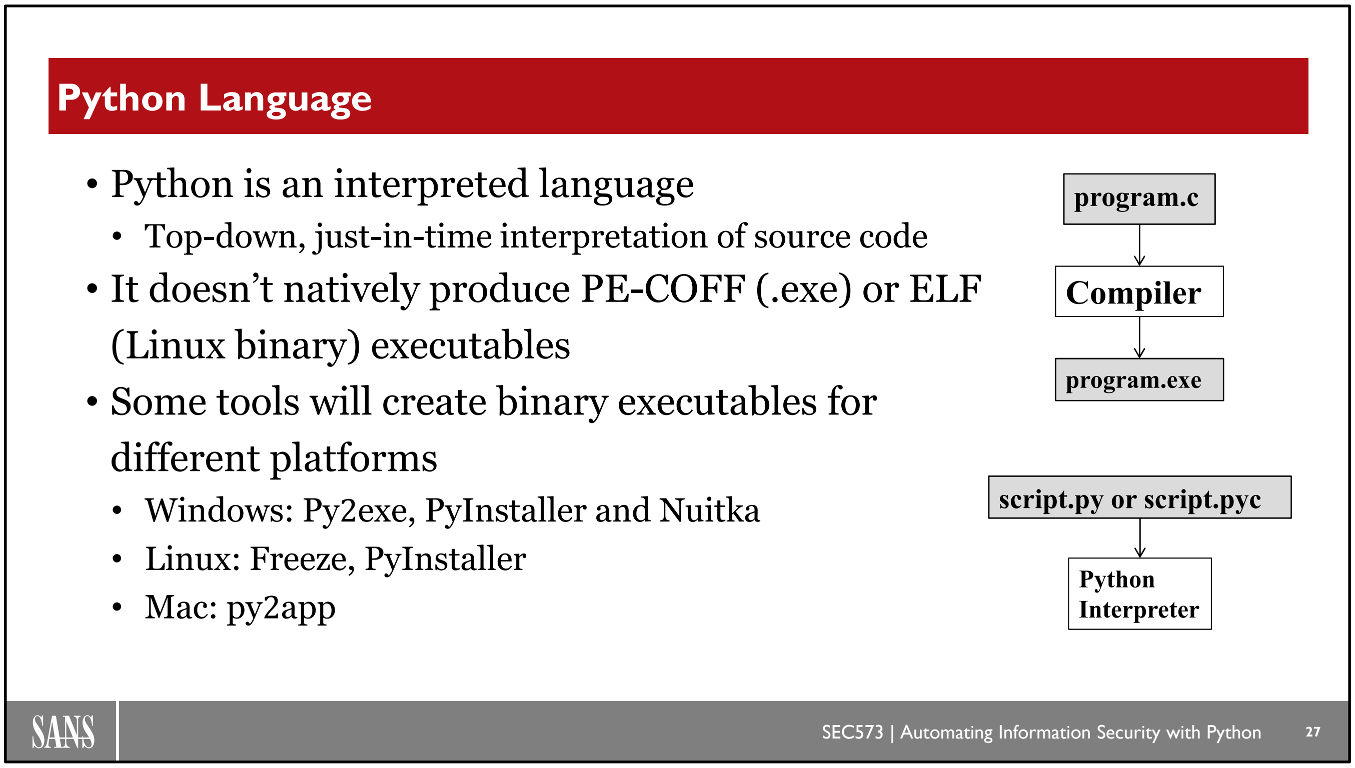
Why we need to make these changes will become more clear when we discuss migrating from Python2 to Python3. For now, just know that if you really want to learn to code in Python2, with these minor changes to your code, the concepts and the majority of syntax in this course can be applied to Python2 programs.

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
    - It changes from pointing to Python2 to Python3 in 2020!
- 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.



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

$$\begin{tikzcd}
\boxed{\text{program.c}} \arrow[d] \\
\boxed{\text{Compiler}} \arrow[d] \\
\boxed{\text{program.exe}}
\end{tikzcd}
\qquad
\begin{tikzcd}
\boxed{\text{script.py or script.pyc}} \arrow[d] \\
\boxed{\text{Python Interpreter}}
\end{tikzcd}$$

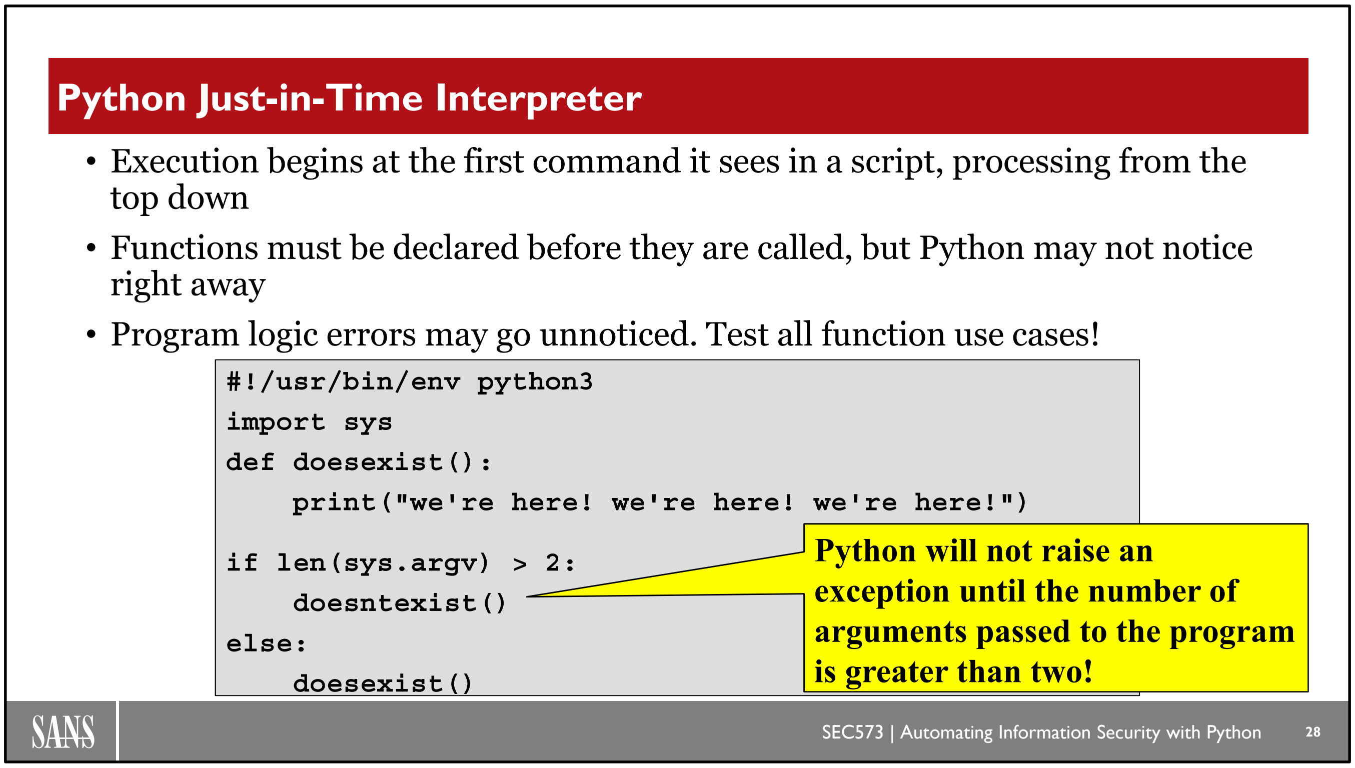
Python is an interpreted language. This is quite different than compiled languages such as C. Compiled languages have a compiler. The compiler will take the source code, read it, and produce a binary executable that is compatible with the operating system. That program can execute natively with the operating 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 http://nuitka.net/.



Python Just-in-Time Interpreter

- Execution begins at the first command it sees in a script, processing from the top down
- Functions must be declared before they are called, but Python may not notice right away
- Program logic errors may go unnoticed. Test all function use cases!

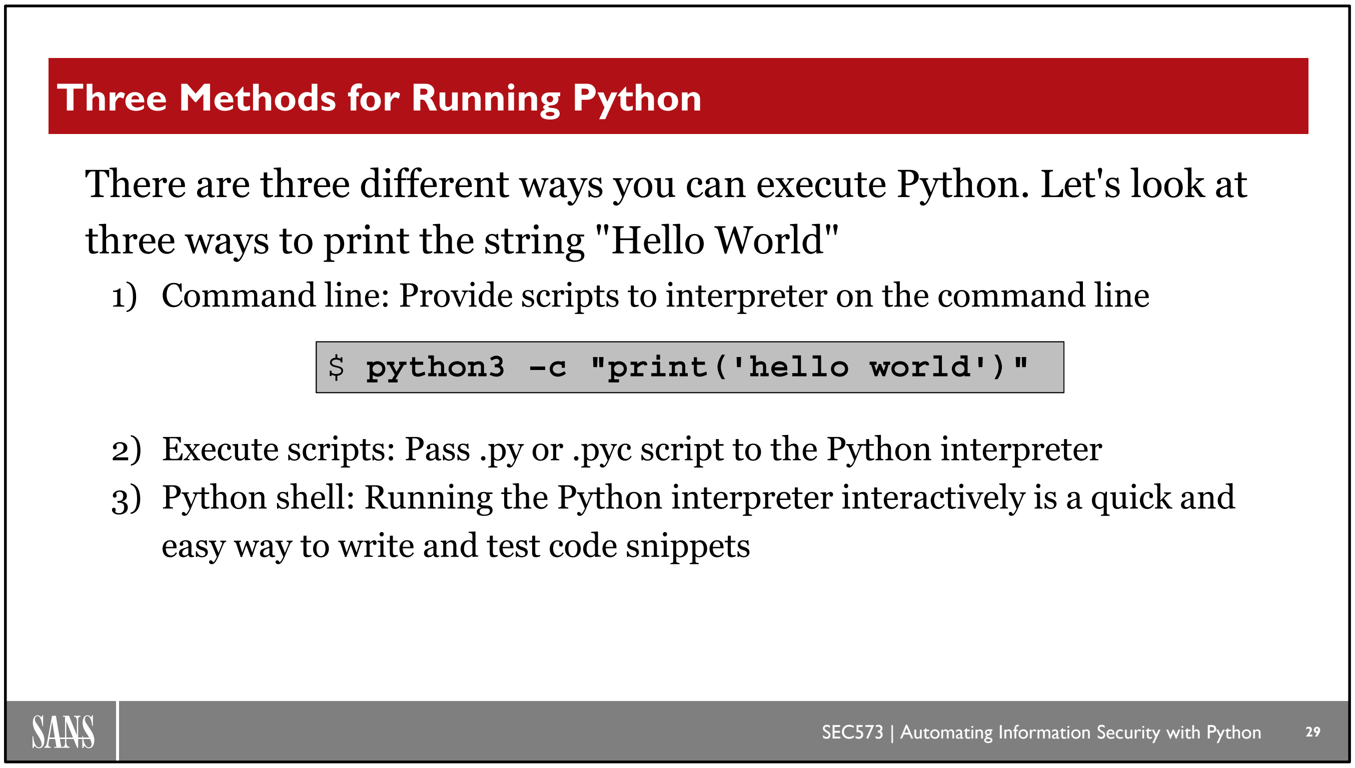
```
#!/usr/bin/env python3
import sys
def doesexist():
    print("we're here! we're here! we're here!")

if len(sys.argv) > 2:
    doesntexist()
else:
    doesexist()
```

Python will not raise an exception until the number of arguments passed to the program is greater than two!

SANS

SEC573 | Automating Information Security with Python 28

Python is a just-in-time interpreter. As a result, it doesn't process the entire script looking for logic errors before it runs. It reads the first line of the script and executes it. It then moves to the second line of the script and executes it. You may have branches in your code that are rarely executed. Errors in the code may go unnoticed by the software developer unless they extensively test their code, being sure to try all the possible test cases.



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

SANS SEC573 | Automating Information Security with Python 29

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.

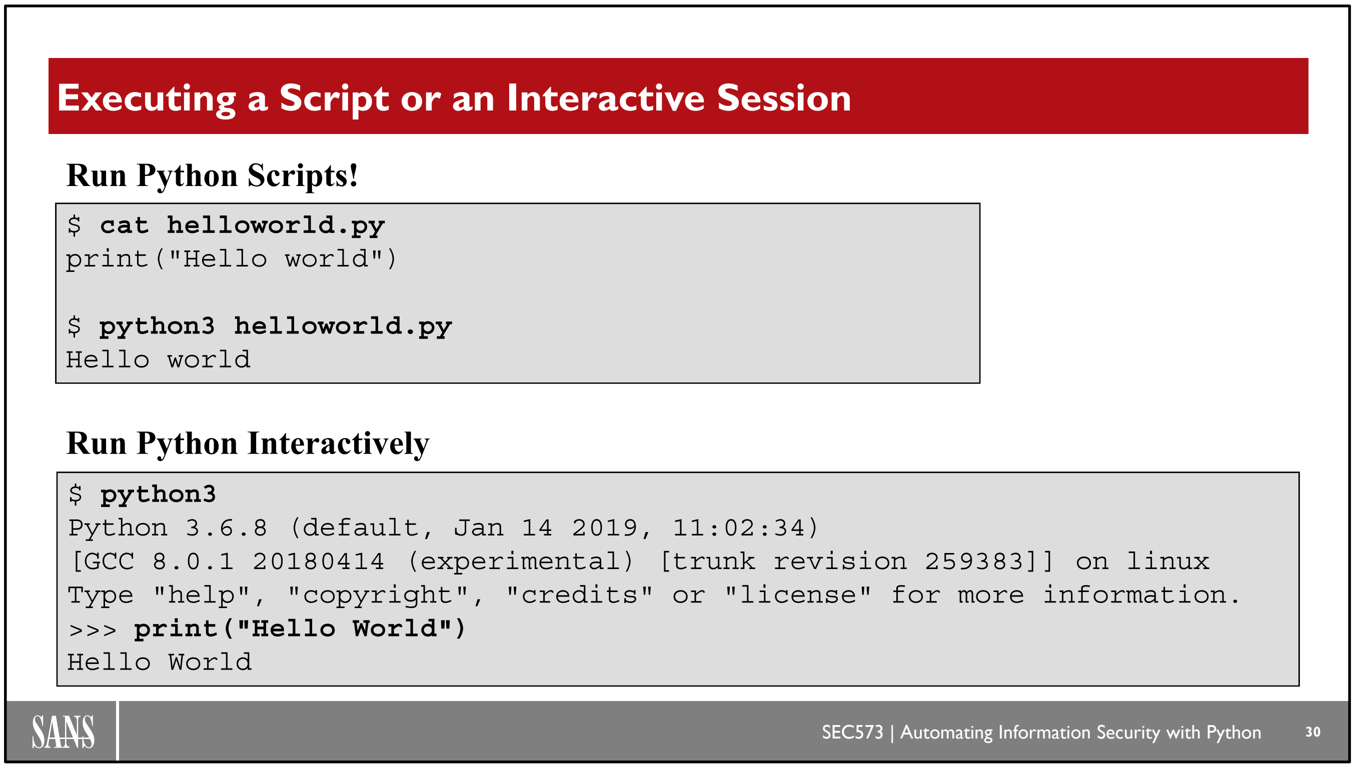
To execute commands from the command line, you put your script commands after the **-c** option on the command line. This method is often used when you want to use Python output and pipe it as input to another command on the Linux command line. For example, you would do this if you wanted to create a 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.



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 3.6.8 (default, Jan 14 2019, 11:02:34)
[GCC 8.0.1 20180414 (experimental) [trunk revision 259383]] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello World")
Hello World
```

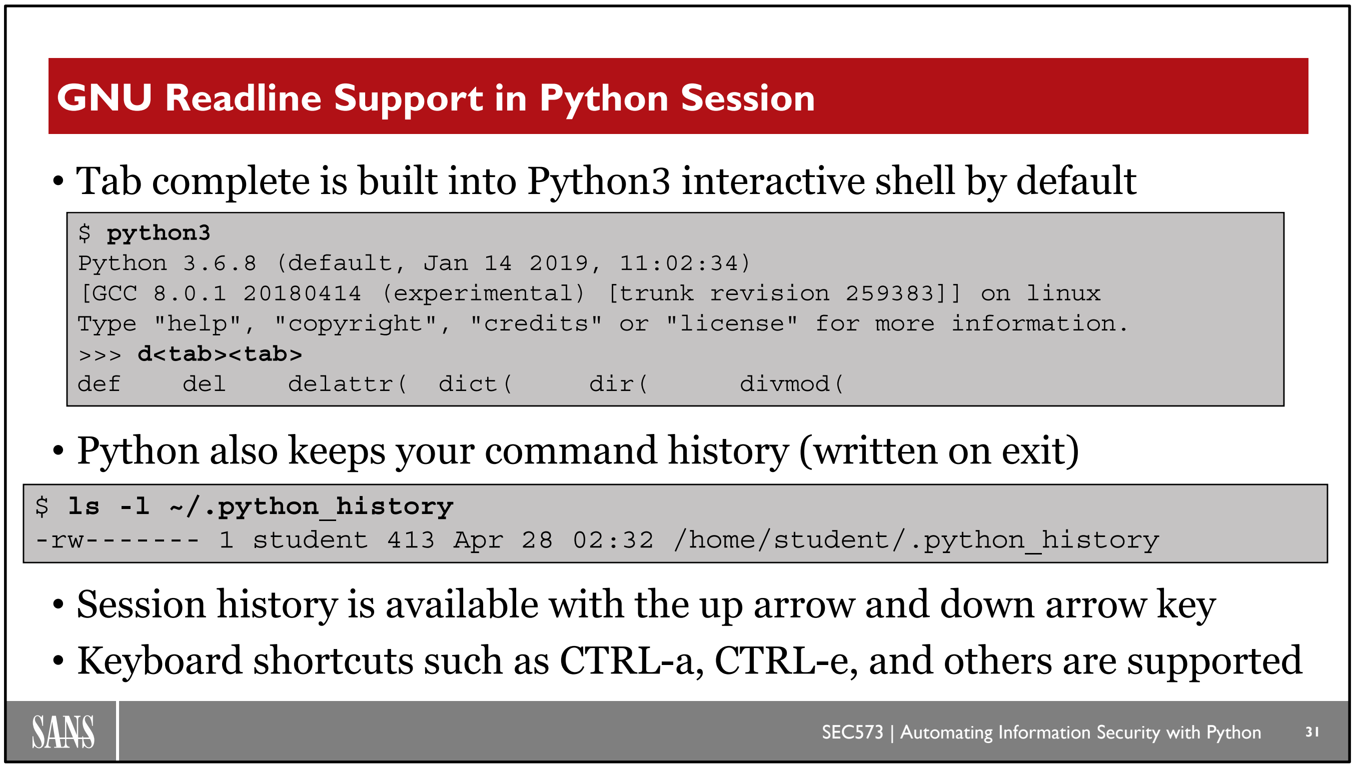
SANS SEC573 | Automating Information Security with Python 30

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!



GNU Readline Support in Python Session

- Tab complete is built into Python3 interactive shell by default

```
$ python3
Python 3.6.8 (default, Jan 14 2019, 11:02:34)
[GCC 8.0.1 20180414 (experimental) [trunk revision 259383]] 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

SANS

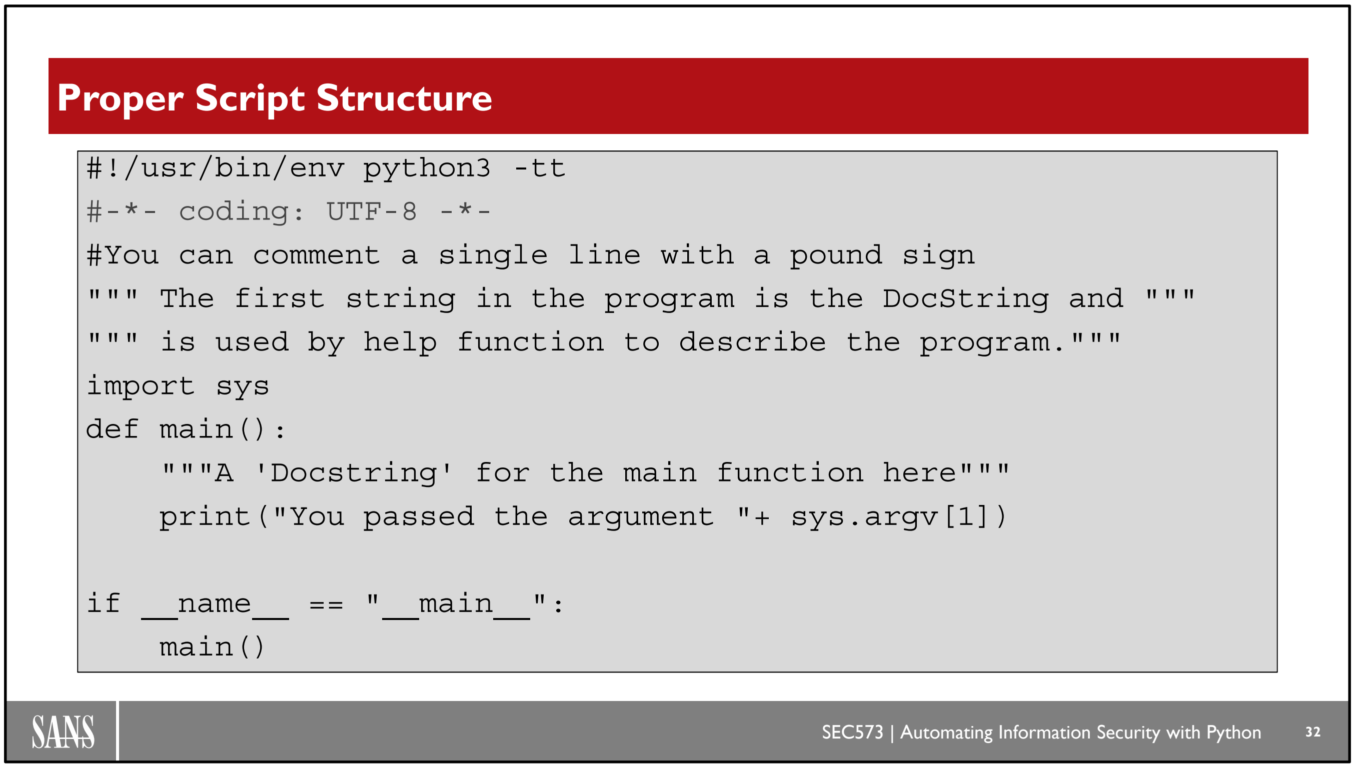
SEC573 | Automating Information Security with Python 31

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.

NOTE: These features are not available in older versions of Python 2 and 3 that you will find commonly deployed in organizations. Importing the 'tab_complete.py' program that is in your 'apps' directory will enable these features on older versions of Python.



Proper Script Structure

```
#!/usr/bin/env python3 -tt
#-*- coding: UTF-8 -*-
#You can comment a single line with a pound sign
""" The first string in the program is the DocString and """
""" is used by help function to describe the program."""
import sys
def main():
    """A 'Docstring' for the main function here"""
    print("You passed the argument "+ sys.argv[1])

if __name__ == "__main__":
    main()
```

SEC573 | Automating Information Security with Python 32

Here you see a basic example of a well-formatted Python program. The #! (shebang) on the first line points Linux to the location of the Python interpreter. The -tt will cause Python to stop with an error if you have a mix of tabs and spaces being used to indent your code. If, instead, you had -t as your option, Python would generate a warning if you mixed tabs and spaces. Stopping if you mix tabs and spaces is the best option. We will talk about the importance of spaces and tabs later. Next is the encoding of the Python script. If don't provide this, Python will assume the script is all ASCII text. If your script will contain any UTF-8 characters, then you need to tell the interpreter to interpret them as UTF-8. The third line is a comment. The interpreter ignores any text on a line after the pound sign.

The first line after the comment is known as the Docstring. Defining a Docstring is optional but is best practice. To define a Docstring, simply include a string as the first line of a module, function, class, or method definition. In this case, we are using triple quotes followed by a description of the program. You use triple quotes anytime you want to define a string that contains freeform text spanning multiple lines. The Docstring is displayed when someone uses the help() function to determine what your code does.

Next, we import any modules that will be required by the program. In this case, we import sys so that we have access to the arguments that were passed to the program. Then we define the main() module. Defining a main() module isn't required. Python will automatically begin executing any Python commands that are defined in the script, but best practice is to put the main body of your script into a function called main. This way, if you later decide to turn your program into a module that you import into another program, you won't need to make a lot of changes to make that happen. Inside the main function here, you can see it simply prints the first element in the list contained in the sys.argv attribute.

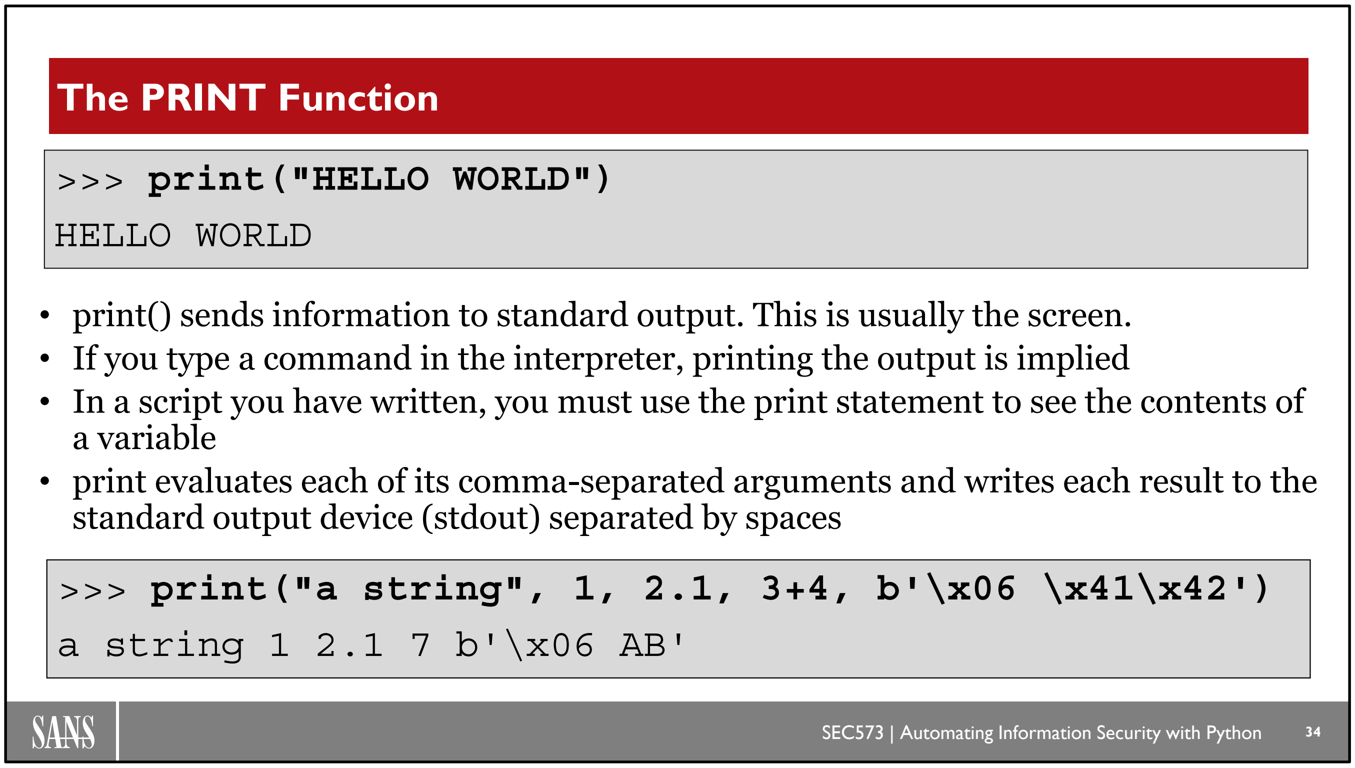
Next, we have the if statement that compares the variable __name__ to the string "__main__". This line of code checks to see if the script is being executed as the main program or if it is being imported by another program. If it is being imported by another program, the main() function is not called. If it is the main program (that is, it was the script that was passed as the first argument to the Python interpreter # python thismodule.py), then the main() function is called and execution begins.

Although not all of your scripts must follow this structure, it is important to know the correct way of doing things.

Section 1 Roadmap

- Lab Setup
- Python and pyWars
- Language Basics
- Strings
- Functions
- if/elif/else
- Modules
- Introspection

This is a Roadmap slide.



The PRINT Function

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

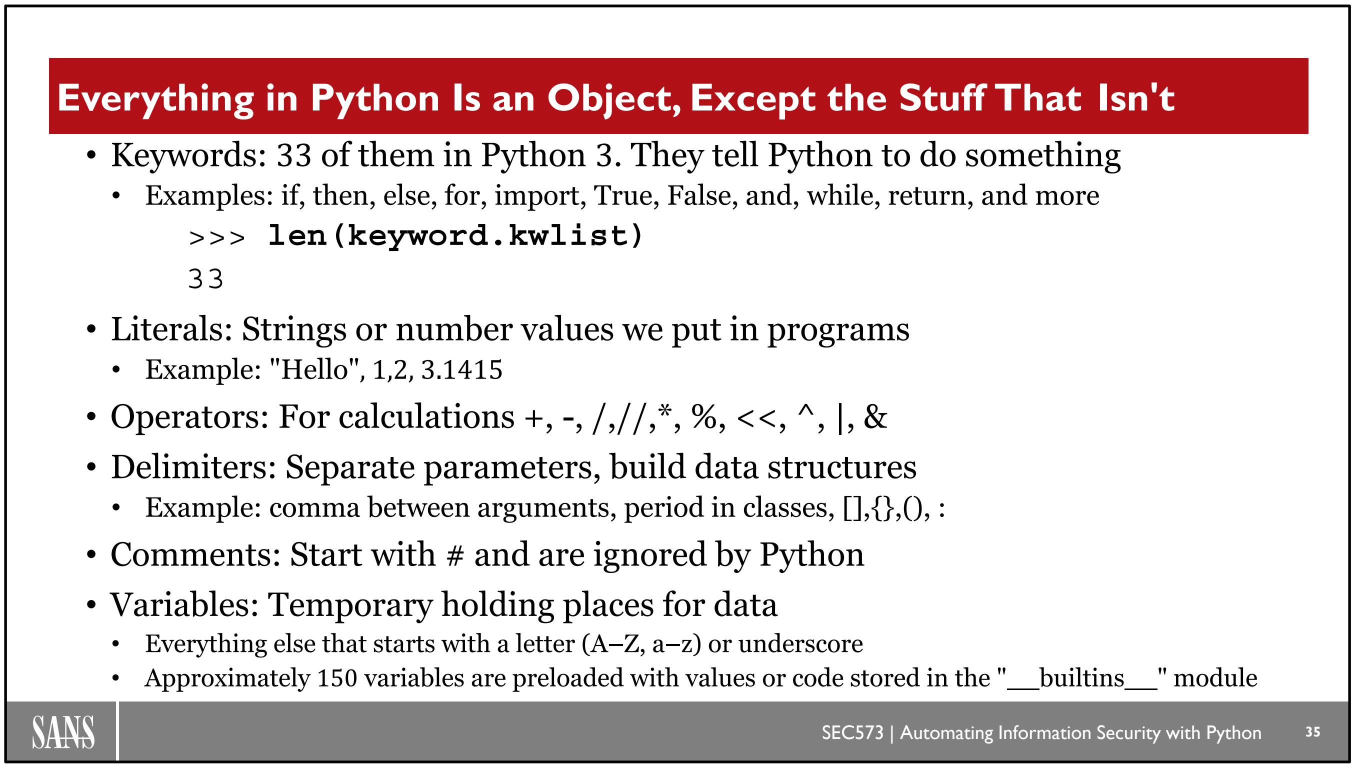
- print() sends information to standard output. This is usually the screen.
- If you type a command in the interpreter, printing the output is implied
- In a script you have written, you must use the print statement to see the contents of a variable
- print evaluates each of its comma-separated arguments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



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

- Keywords: 33 of them in Python 3. They tell Python to do something
  - Examples: if, then, else, for, import, True, False, and, while, return, and more

```
>>> len(keyword.kwlist)
33
```

- 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

SANS SEC573 | Automating Information Security with Python 35

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 understands 33 keywords.

```
>>> 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']
```

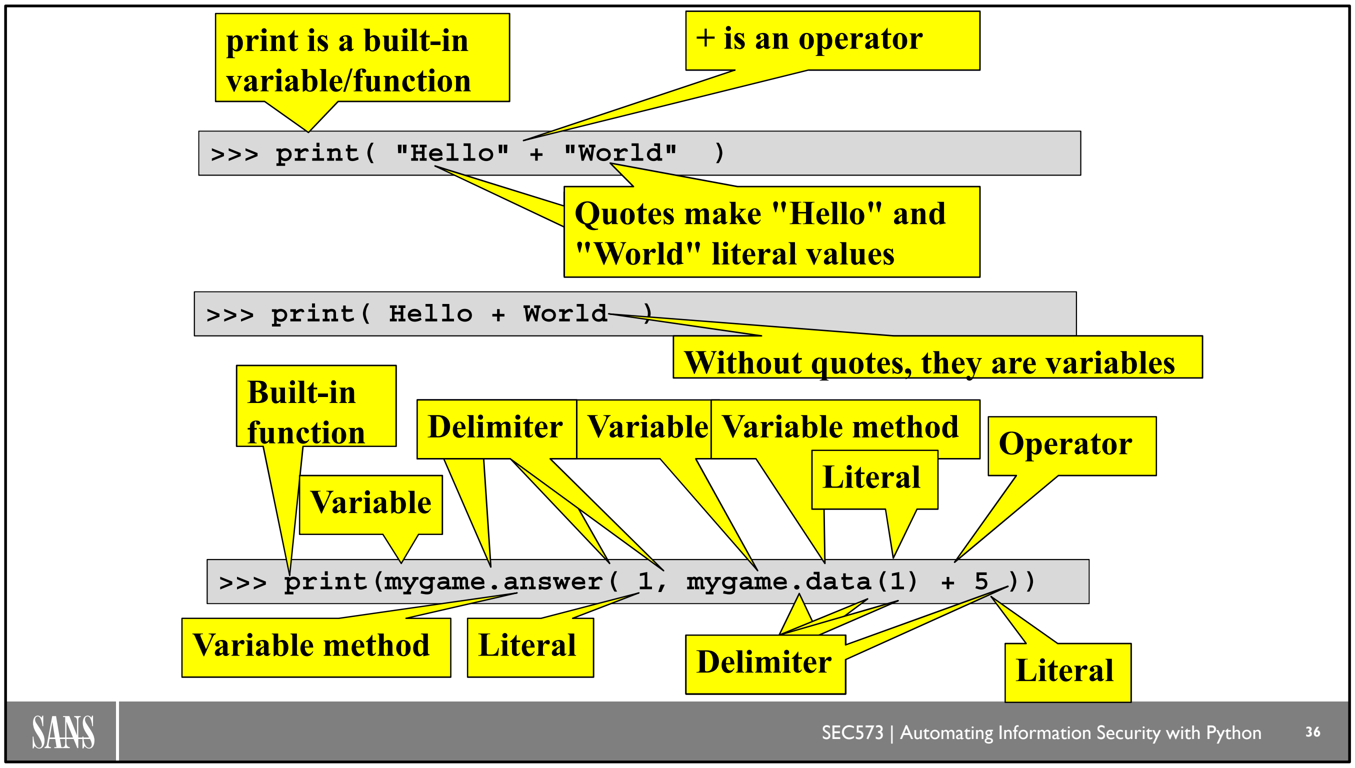
In Python 2.7, there are 31 keywords. 'print' and 'exec' were keywords rather than functions, and 'False', 'None', 'True', and 'nonlocal' were not keywords.

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


print is a built-in variable/function
+ is an operator
>>> print("Hello" + "World")
Quotes make "Hello" and "World" literal values
>>> print(Hello + World)
Without quotes, they are variables
Built-in function
Delimiter
Variable
Variable method
Operator
Literal
Variable
>>> print(mygame.answer(1, mygame.data(1) + 5))
Variable method
Literal
Delimiter
Literal
SANS
SEC573 | Automating Information Security with Python
36

Let's look at how Python interprets these lines:

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

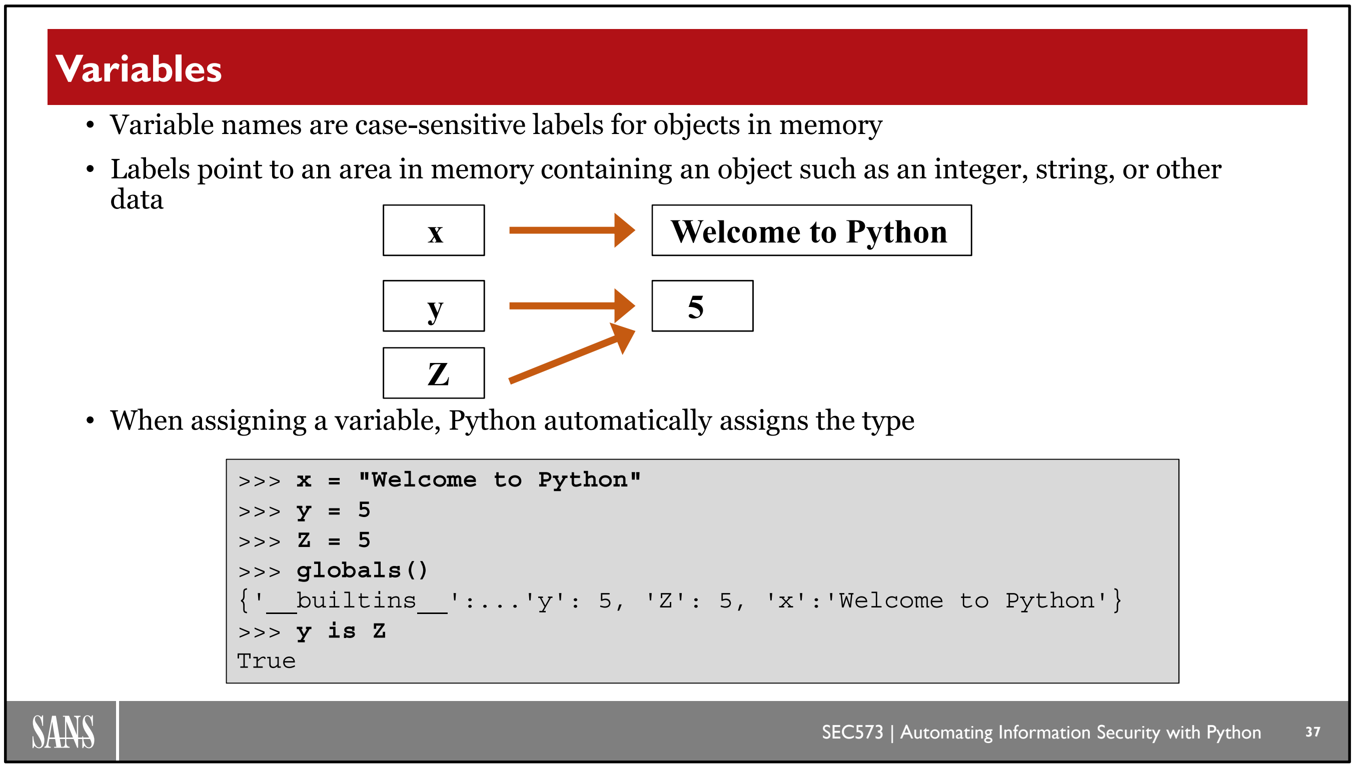
Python sees the first word on this line as a variable named "print". The variable "print" is preloaded with a small bit of code to send output to standard out. These small bits of code are called functions. The open and close parentheses after the variable tell Python we want to run the code for the function stored in that variable. Any input to that function is passed inside the parentheses and are 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.

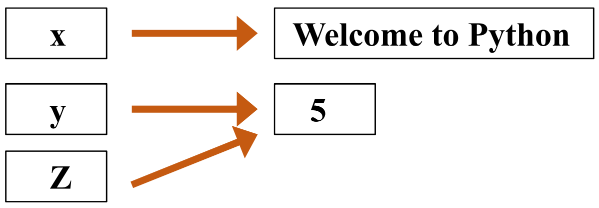
```
>>> print( mygame.answer( 1, mygame.data(1) + 5 ) )
```

With this line, Python will again see the "print" function and send the results of what follows to the screen. Because mygame 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 mygame object. Python then recognizes "answer" as a method that is part of the mygame 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
```

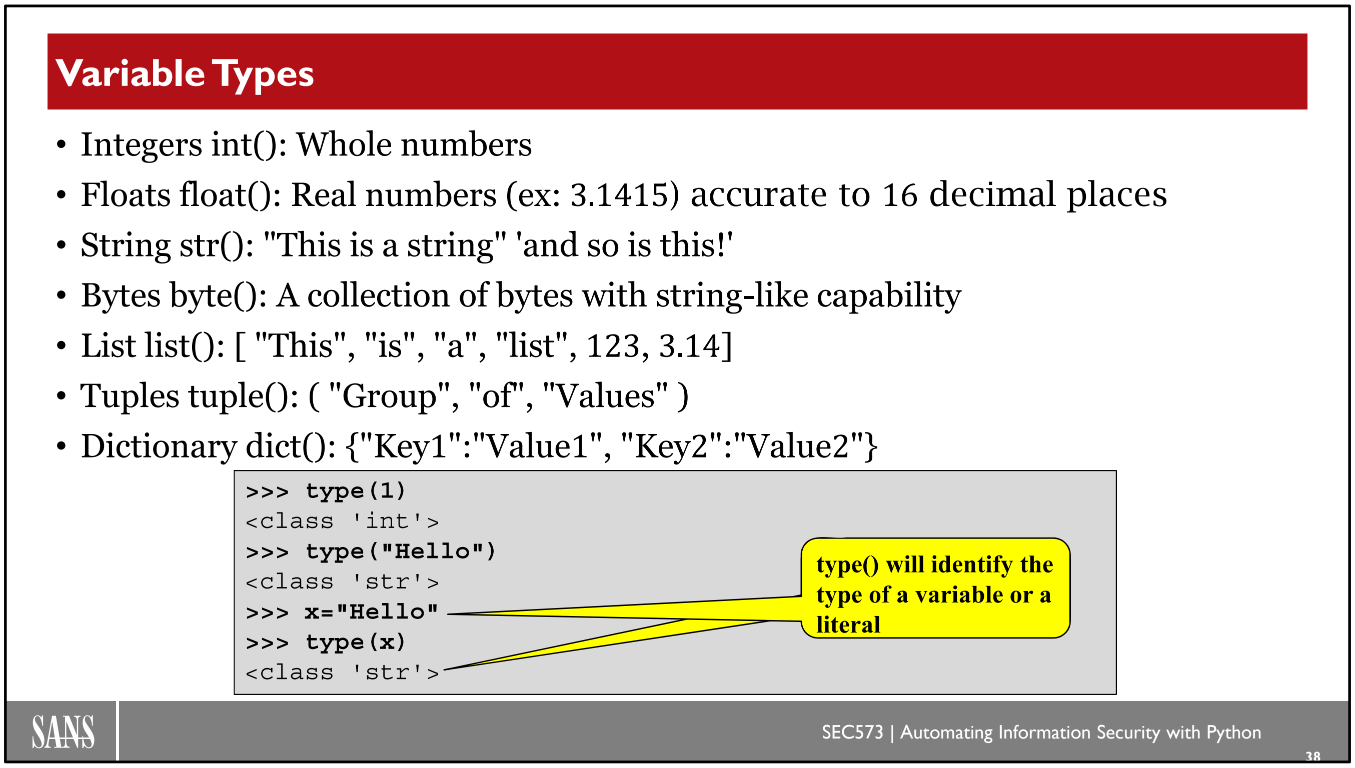
A variable is just a text label we create for a memory address that contains a Python object. When Python executes the command "y = 5", it creates an entry in current namespace that holds variable "y" and points it to a memory location containing an integer object of value 5. Anytime your program refers 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. Because variable names are case sensitive, here 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.



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"}

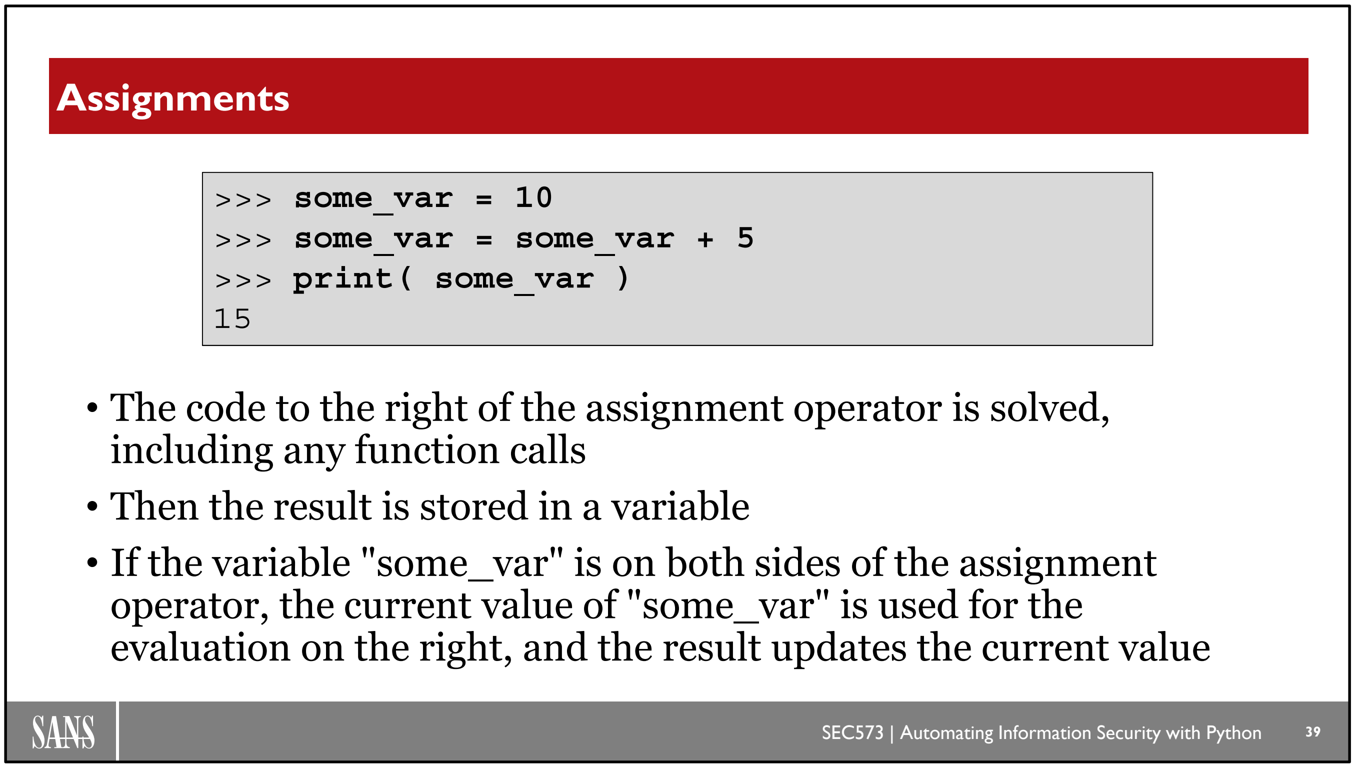
```
>>> type(1)
<class 'int'>
>>> type("Hello")
<class 'str'>
>>> x="Hello"
>>> type(x)
<class 'str'>
```

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.



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

SANS | SEC573 | Automating Information Security with Python 39

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
```

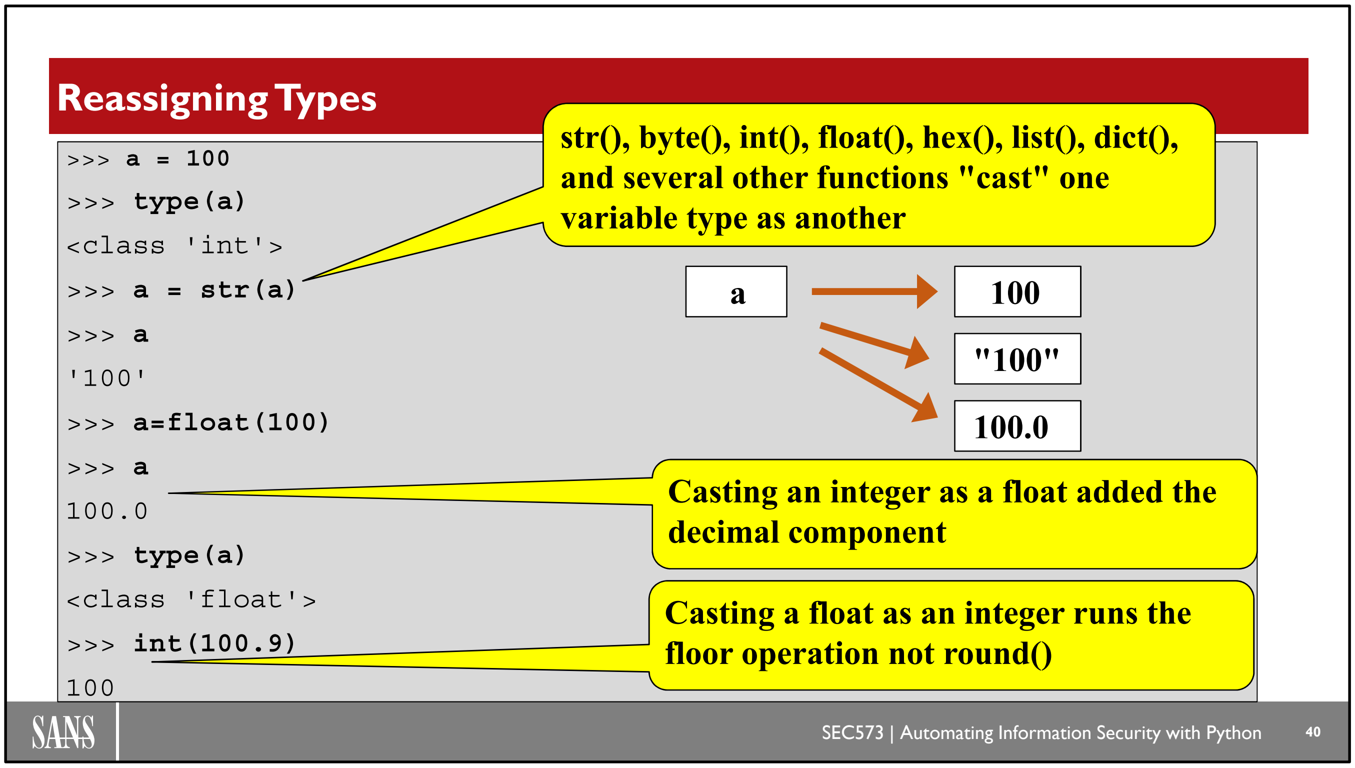
The right side of the equal sign contains a literal and requires no evaluation. This will store the number 10 in a memory location in the global namespace. Then the next line of code is ready to be executed:

```
>>> 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.



Reassigning Types

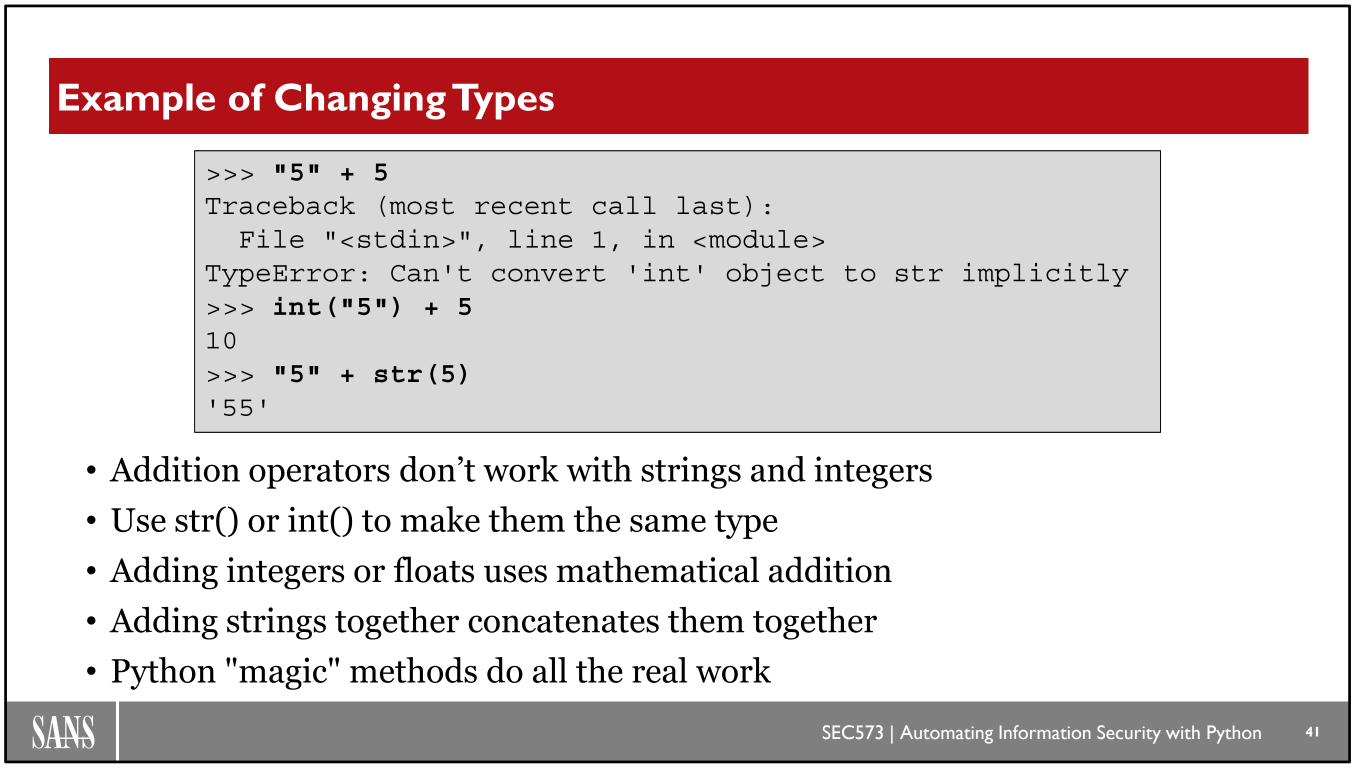
```
>>> a = 100
>>> type(a)
<class 'int'>
>>> a = str(a)
>>> a
'100'
>>> a=float(100)
>>> a
100.0
>>> type(a)
<class 'float'>
>>> int(100.9)
100
```

str(), byte(), int(), float(), hex(), list(), dict(), and several other functions "cast" one variable type as another
a
100
"100"
100.0
Casting an integer as a float added the decimal component
Casting a float as an integer runs the floor operation not round()
SANS
SEC573 | Automating Information Security with Python
40

You can reassign a variable type by passing it as a parameter to a different variable type. This is sometimes referred to as "casting". 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
```



Example of Changing Types

```
>>> "5" + 5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
>>> int("5") + 5
10
>>> "5" + str(5)
'55'
```

- Addition operators don’t work with strings and integers
- Use str() or int() to make them the same type
- Adding integers or floats uses mathematical addition
- Adding strings together concatenates them together
- Python "magic" methods do all the real work

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.

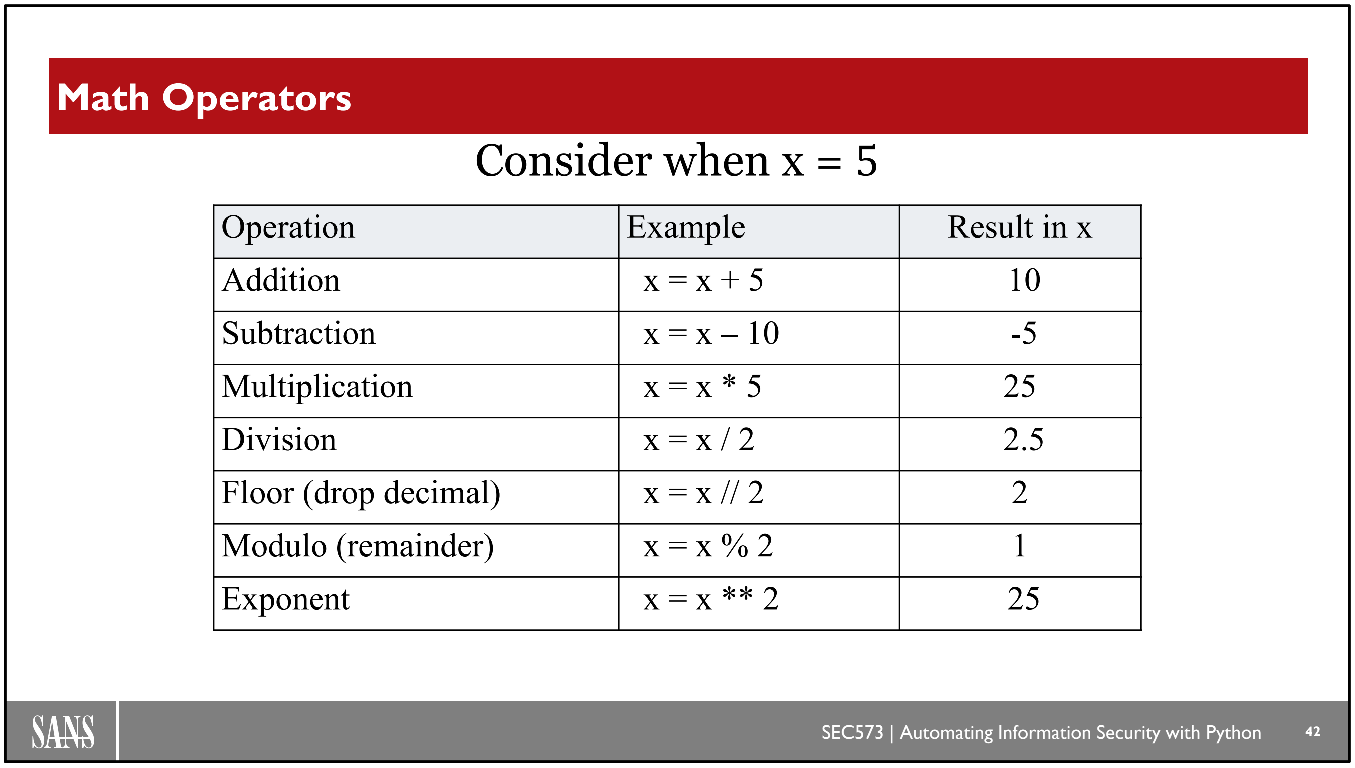
The + operator magically switches between the addition operator and the concatenate operation based on whether you are adding together strings or integers. You might be wondering how that happens. The magic isn't in the + operator. All that the + operator does is call the __add__ method associated with 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.



Math Operators

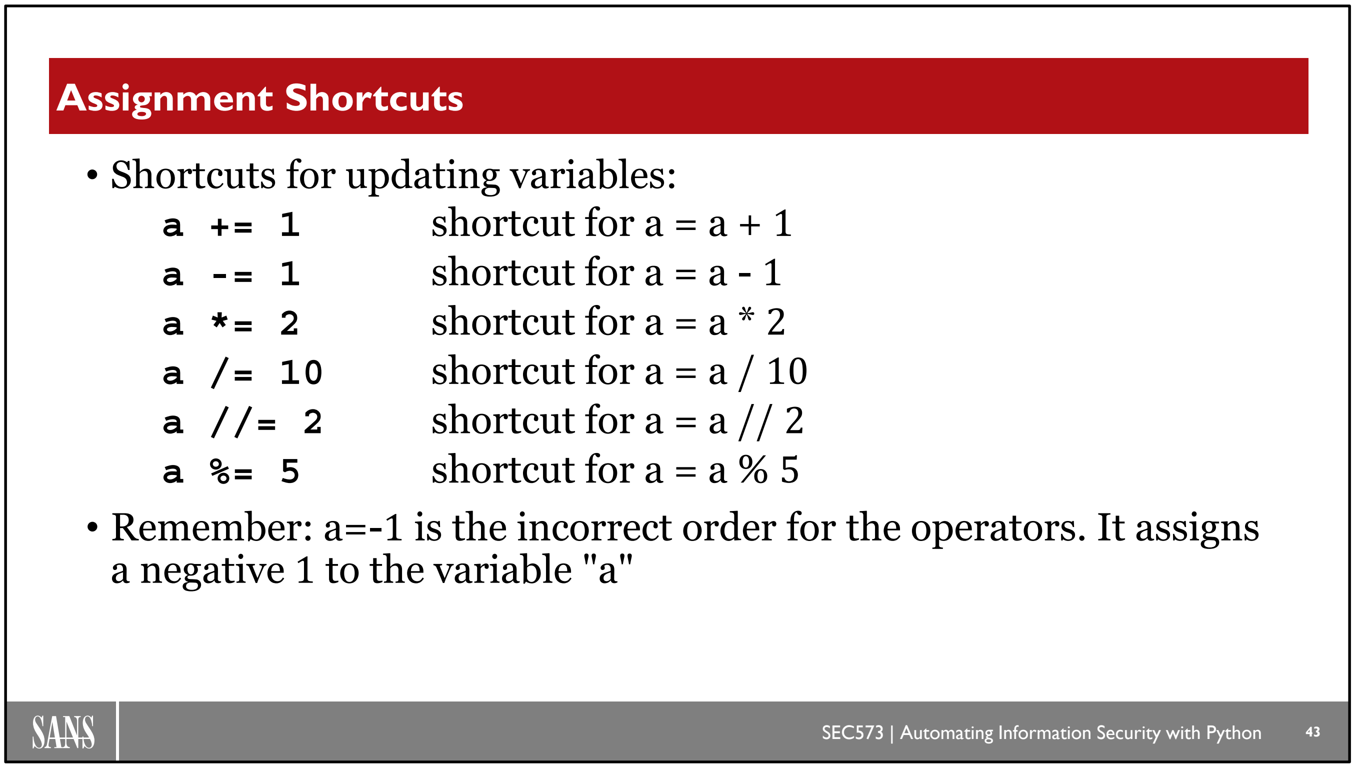
Consider when x = 5

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

SANS

SEC573 | Automating Information Security with Python 42

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.



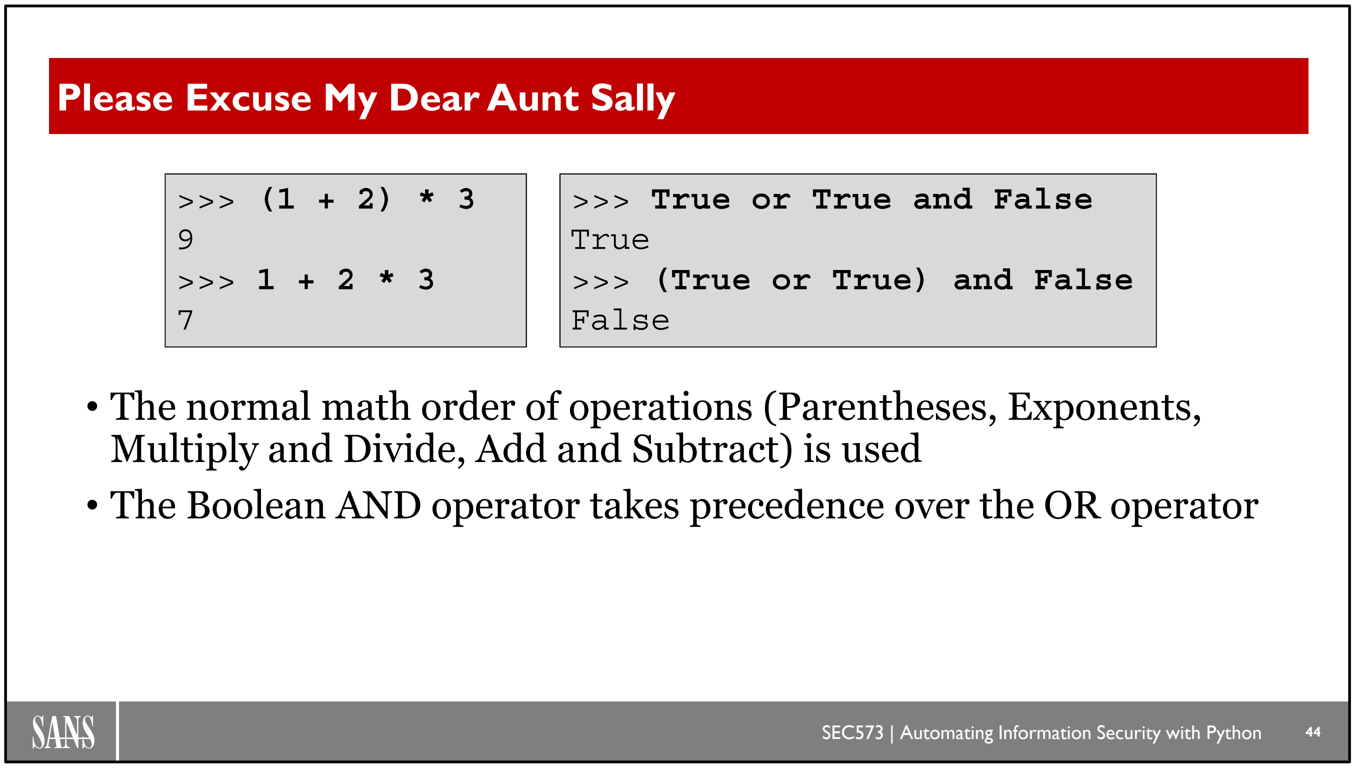
Assignment Shortcuts

- 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 **.



Please Excuse My Dear Aunt Sally

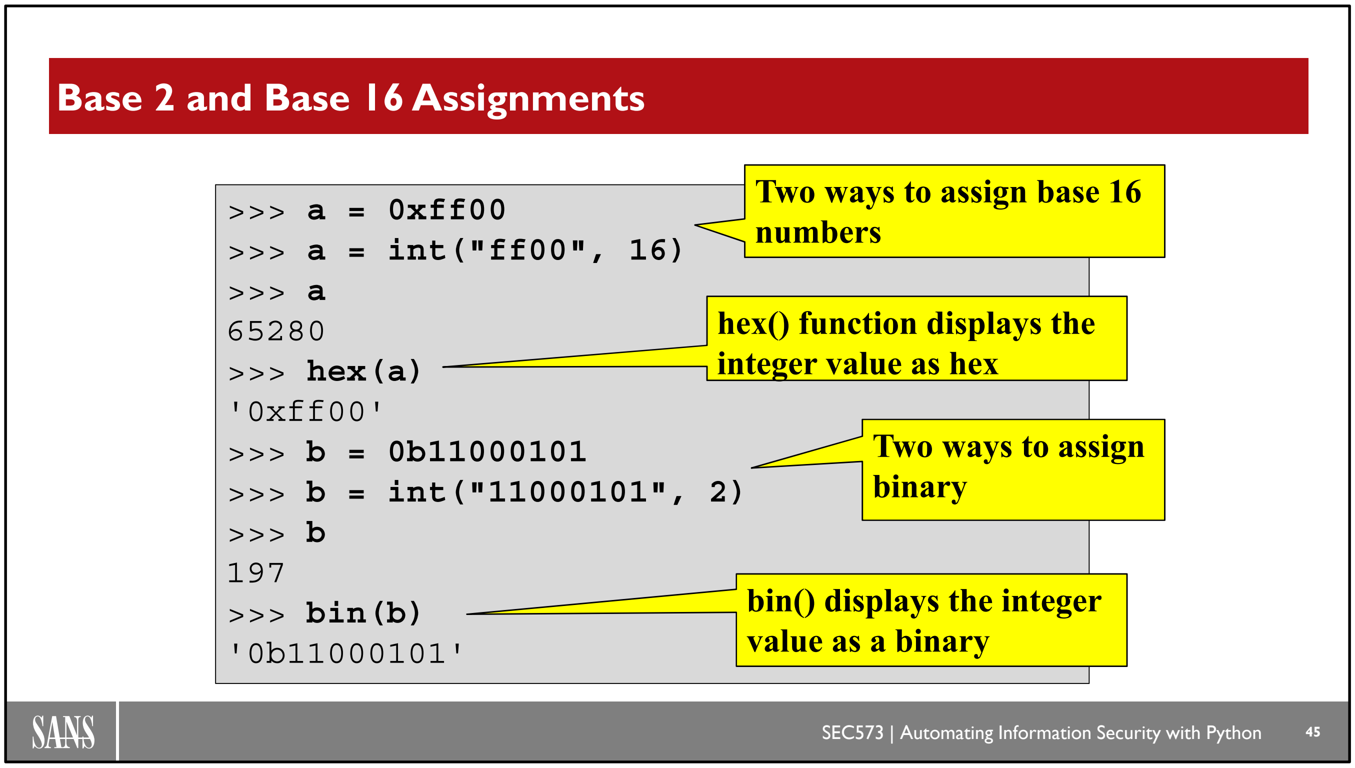
```
>>> (1 + 2) * 3
9
>>> 1 + 2 * 3
7
```

```
>>> 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
- The Boolean AND operator takes precedence over the OR operator

SANS | SEC573 | Automating Information Security with Python 44

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.



Base 2 and Base 16 Assignments

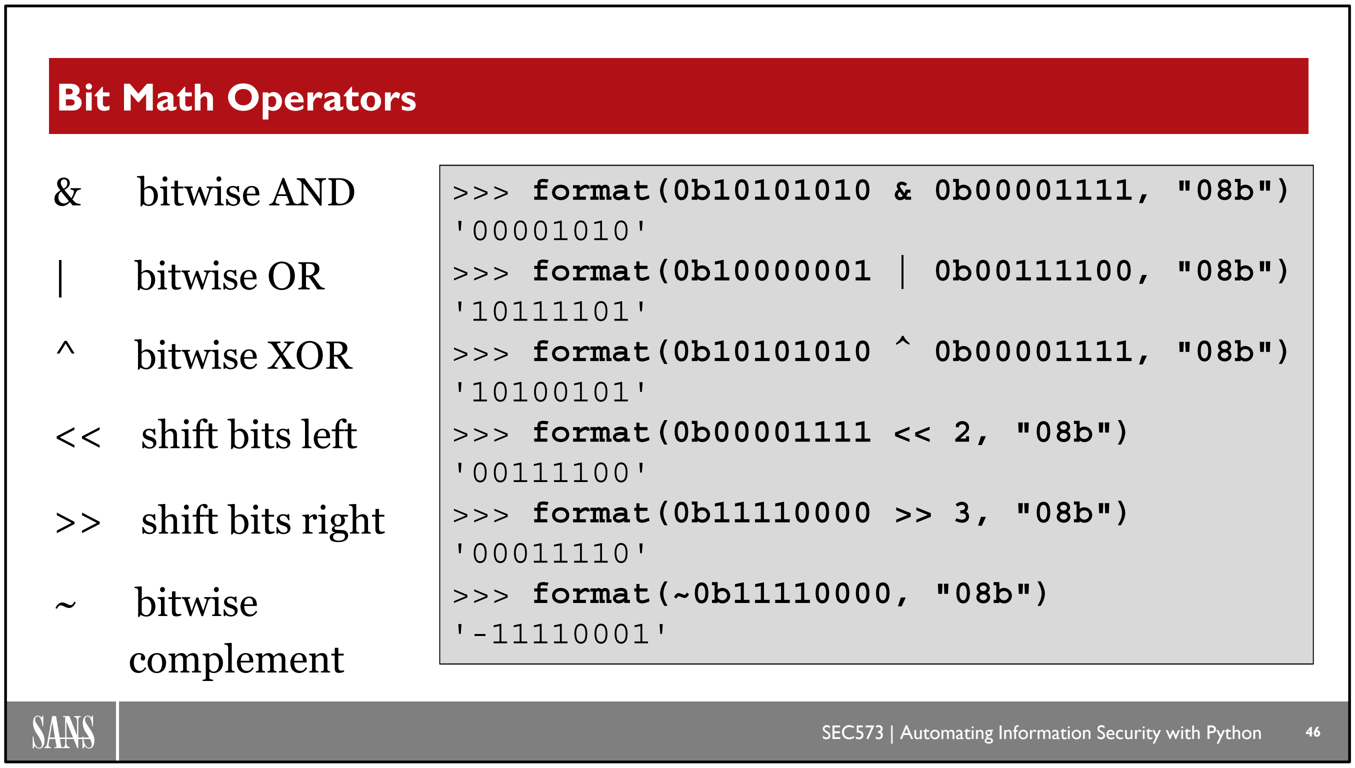
```
>>> a = 0xff00
>>> a = int("ff00", 16)
>>> a
65280
>>> hex(a)
'0xff00'
>>> b = 0b11000101
>>> b = int("11000101", 2)
>>> b
197
>>> bin(b)
'0b11000101'
```

Two ways to assign base 16 numbers
hex() function displays the integer value as hex
Two ways to assign binary
bin() displays the integer value as a binary
SANS
SEC573 | Automating Information Security with Python
45

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 would 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.



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(0b00001111 << 2, "08b")
'00111100'
>>> format(0b11110000 >> 3, "08b")
'00011110'
>>> format(~0b11110000, "08b")
'-11110001'
```

SANS SEC573 | Automating Information Security with Python 46

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. You can perform a logical AND operation of two different values at the bit level with the & (ampersand) operator or an OR operation with the | (pipe) operator:

```
>>> bin(0b11001100 & 0b11110000)
'0b11000000
>>> bin(0b11001100 | 0b11110000)
'0b11111100'
```

An exclusive OR (XOR) is performed with the ^ (caret) operator.

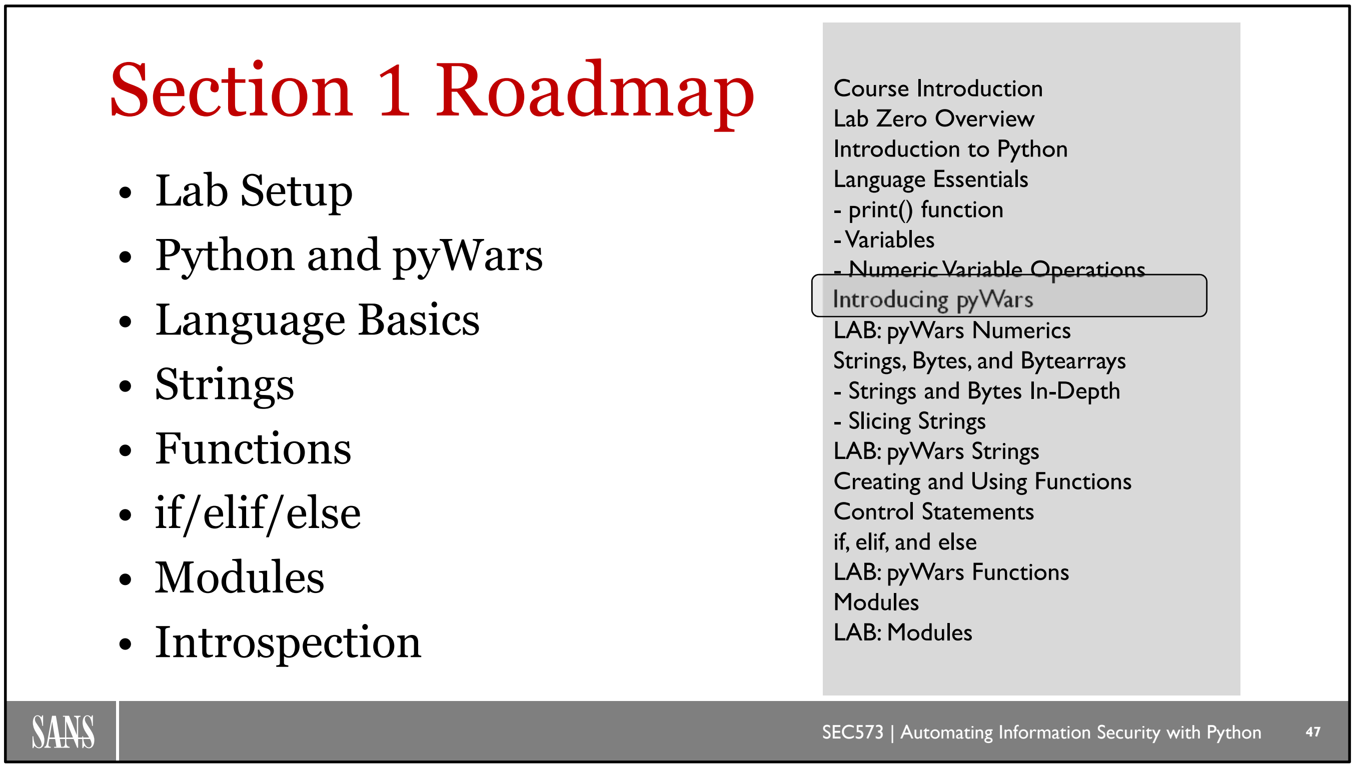
```
>>> bin(0b11001100 ^ 0b11110000)
'0b111100'
```

The ~ (tilde) converts the integer to a negative number with two's complement. This involves reversing the bits and adding one to it. However, Python internally doesn't use two's complement to store negative numbers. Even though the ~ correctly performs a two's complement number operation, displaying the binary of the number does not display a two's complement.

```
>>> bin(~0b1111110)
'-0b1111111'
```

You can also shift the bits left or right. Shifting to the left one place, in effect, multiplies the value by two for each bit that is shifted.

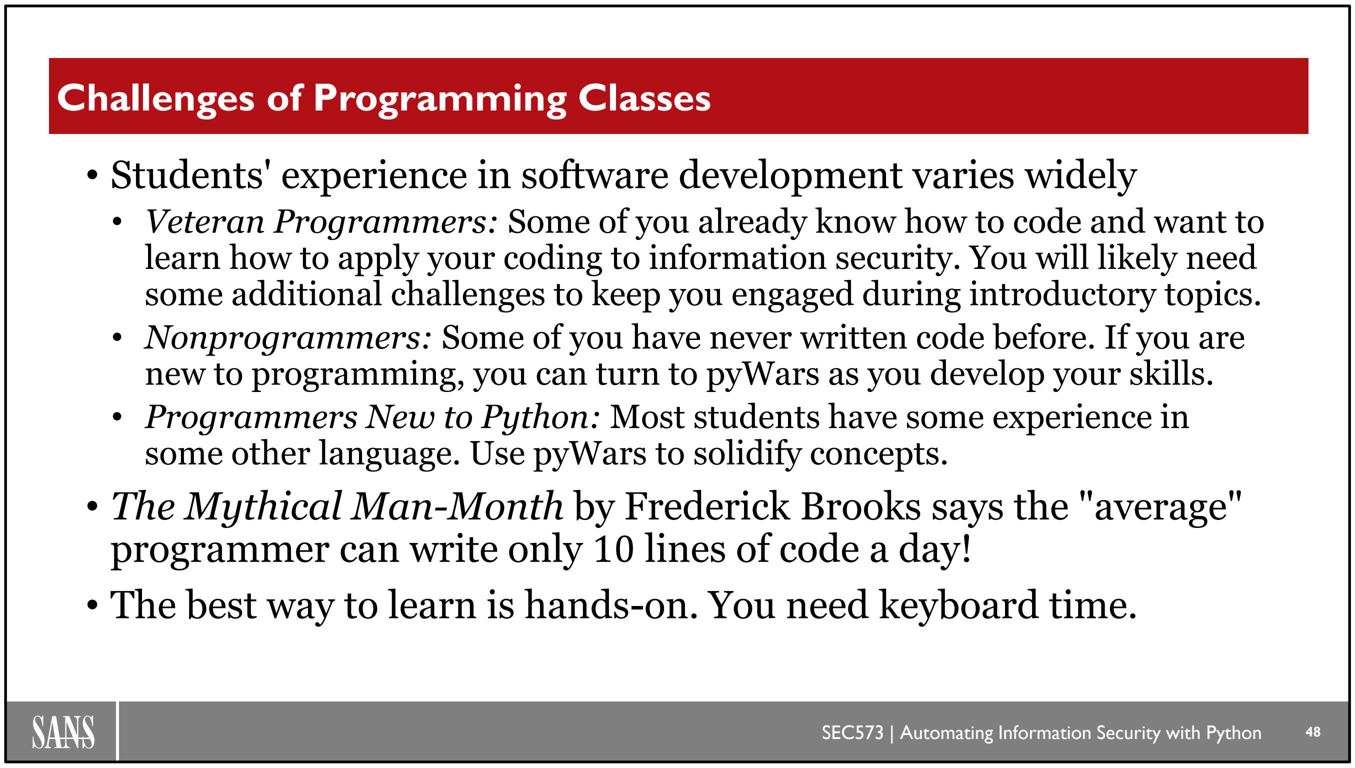
```
>>> bin(0b11001100<<2)
'0b1100110000'
>>> bin(0b11001100>>2)
'0b110011'
```



Section 1 Roadmap

- Lab Setup
- Python and pyWars
- Language Basics
- Strings
- Functions
- if/elif/else
- Modules
- Introspection

This is a Roadmap slide.



Challenges of Programming Classes

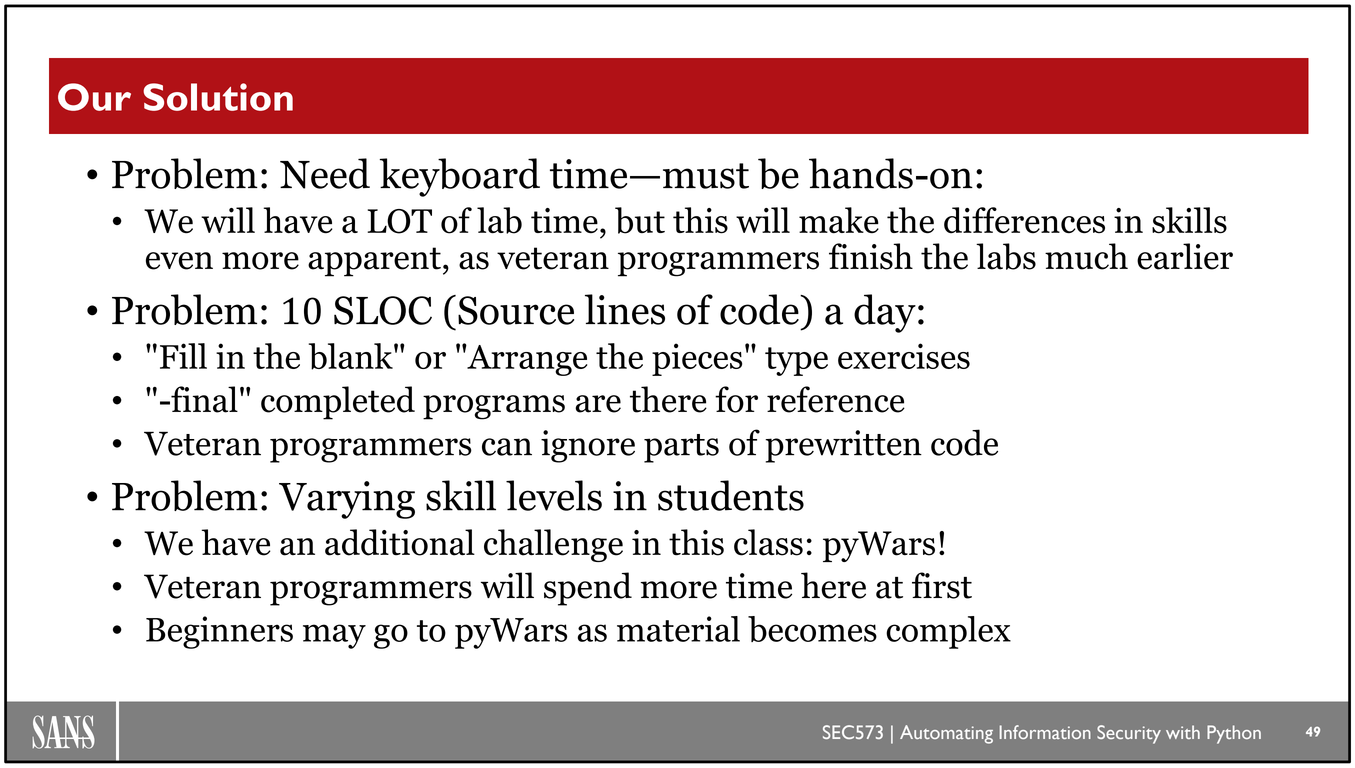
- Students' experience in software development varies widely
  - *Veteran Programmers:* Some of you already know how to code and want to learn how to apply your coding to information security. You will likely need some additional challenges to keep you engaged during introductory topics.
  - *Nonprogrammers:* 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.

SANS | SEC573 | Automating Information Security with Python 48

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.



Our Solution

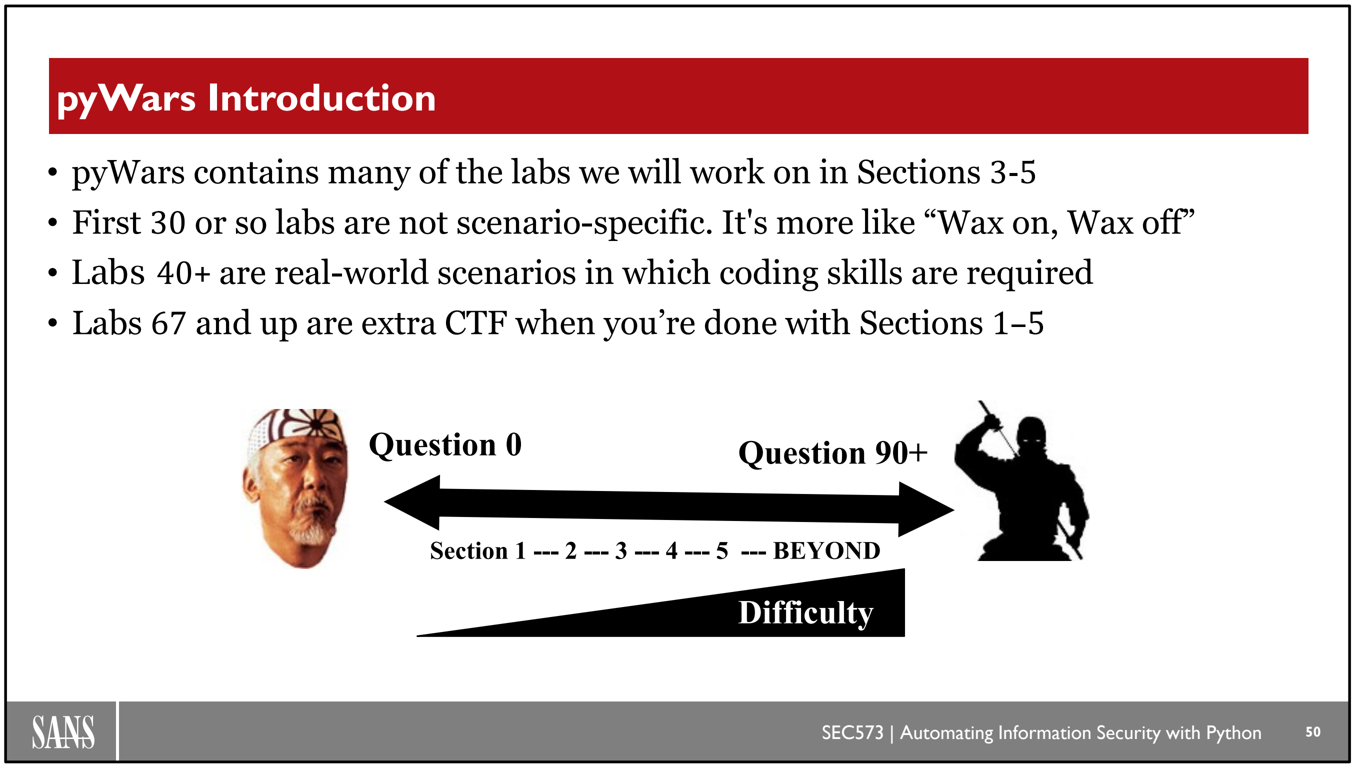
- Problem: Need keyboard time—must be hands-on:
  - We will have a LOT of lab time, but this will make the differences in skills even more apparent, as veteran programmers finish the labs much earlier
- Problem: 10 SLOC (Source lines of code) a day:
  - "Fill in the blank" or "Arrange the pieces" type exercises
  - "-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 exercises 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.

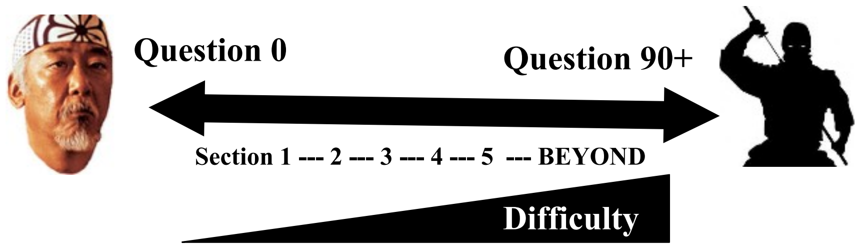
Veteran programmers can use pyWars to challenge themselves as we cover material they may already be familiar with. Beginners will likely ignore pyWars at first, other than exercises, and return to pyWars during Sections 3 through 5 when the material difficulty goes beyond their comfort zone. Most of 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 today, 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.



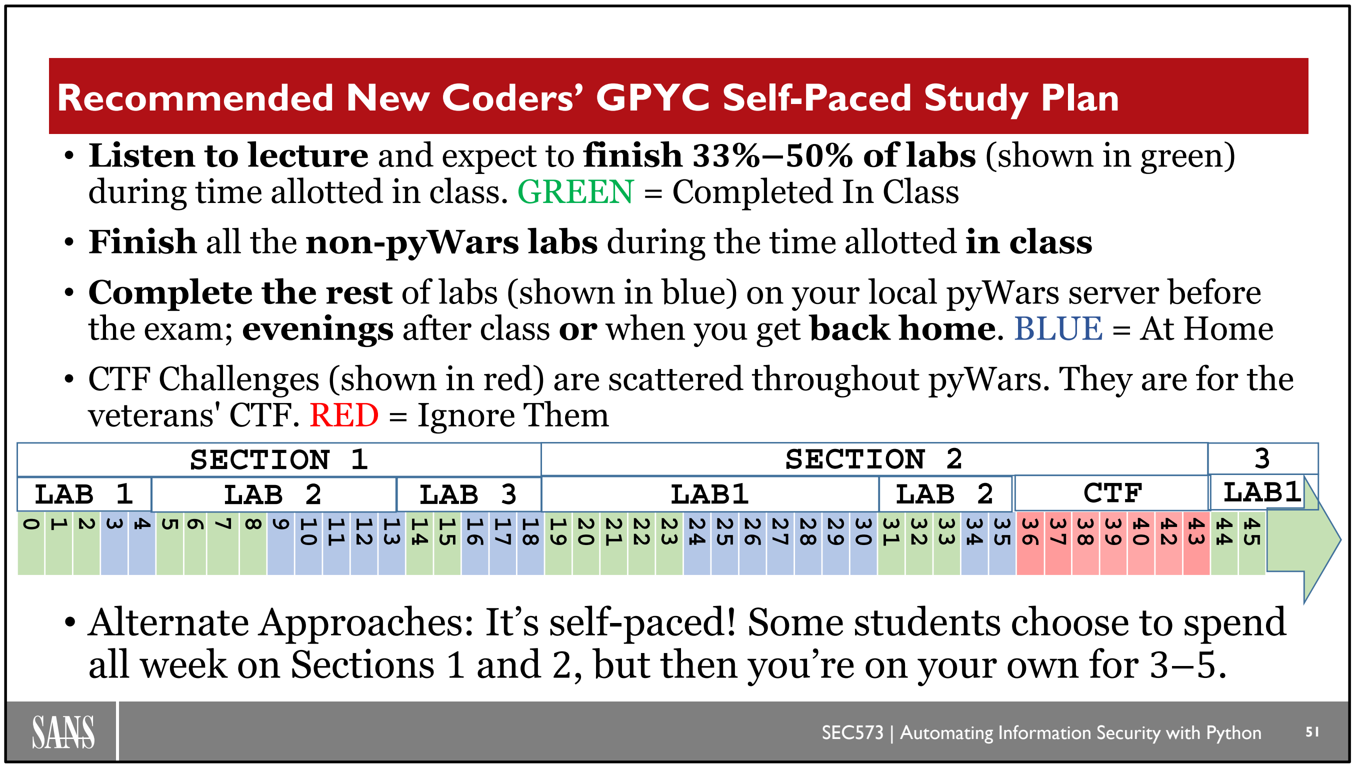
pyWars Introduction

- pyWars contains many of the labs we will work on in Sections 3-5
- First 30 or so labs are not scenario-specific. It's more like “Wax on, Wax off”
- Labs 40+ are real-world scenarios in which coding skills are required
- Labs 67 and up are extra CTF when you're done with Sections 1–5



SANS | SEC573 | Automating Information Security with Python 50

pyWars is an online self-paced lab environment that runs the first five sections of class. The challenges are numbered beginning at 0 and increase to more than 90 challenges. They get increasingly difficult as the question number increases. A little more than half of the challenges are not themed or specific to a real-world problem. Instead, they test or teach essential Python skills and help you develop the essential skills that you will require. For these challenges, think of them as *The Karate Kid*'s Mr. Miyagi having you wax on and wax off to build the muscle strength you need to become a Python Ninja. The second half of the pyWars challenges simulate scenarios that I and other security professionals have faced in the real world. In addition to increasing in difficulty, the pyWars challenges follow the course material through Section 5. So solutions for solving many of these challenges will be presented in class over the next several days.



Recommended New Coders' GPYC Self-Paced Study Plan

- **Listen to lecture** and expect to **finish 33%–50% of labs** (shown in green) during time allotted in class. GREEN = Completed In Class
- **Finish** all the **non-pyWars labs** during the time allotted **in class**
- **Complete the rest** of labs (shown in blue) on your local pyWars server before the exam; **evenings** after class **or** when you get **back home**. BLUE = At Home
- CTF Challenges (shown in red) are scattered throughout pyWars. They are for the veterans' CTF. RED = Ignore Them

SECTION 1	LAB 1	0 🟩, 1 🟩, 2 🟩, 3 🟦, 4 🟦
	LAB 2	5 🟩, 6 🟩, 7 🟩, 8 🟩, 9 🟦, 10 🟦, 11 🟦, 12 🟦, 13 🟦
	LAB 3	14 🟩, 15 🟩, 16 🟦, 17 🟦, 18 🟦
SECTION 2	LAB1	19 🟩, 20 🟩, 21 🟩, 22 🟩, 23 🟩, 24 🟦, 25 🟦, 26 🟦, 27 🟦, 28 🟦, 29 🟦, 30 🟦
	LAB 2	31 🟩, 32 🟩, 33 🟩, 34 🟦, 35 🟦
	CTF	36 🟥, 37 🟥, 38 🟥, 39 🟥, 40 🟥, 42 🟥, 43 🟥
3	LAB1	44 🟩, 45 🟩

- 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.

SANS SEC573 | Automating Information Security with Python 51

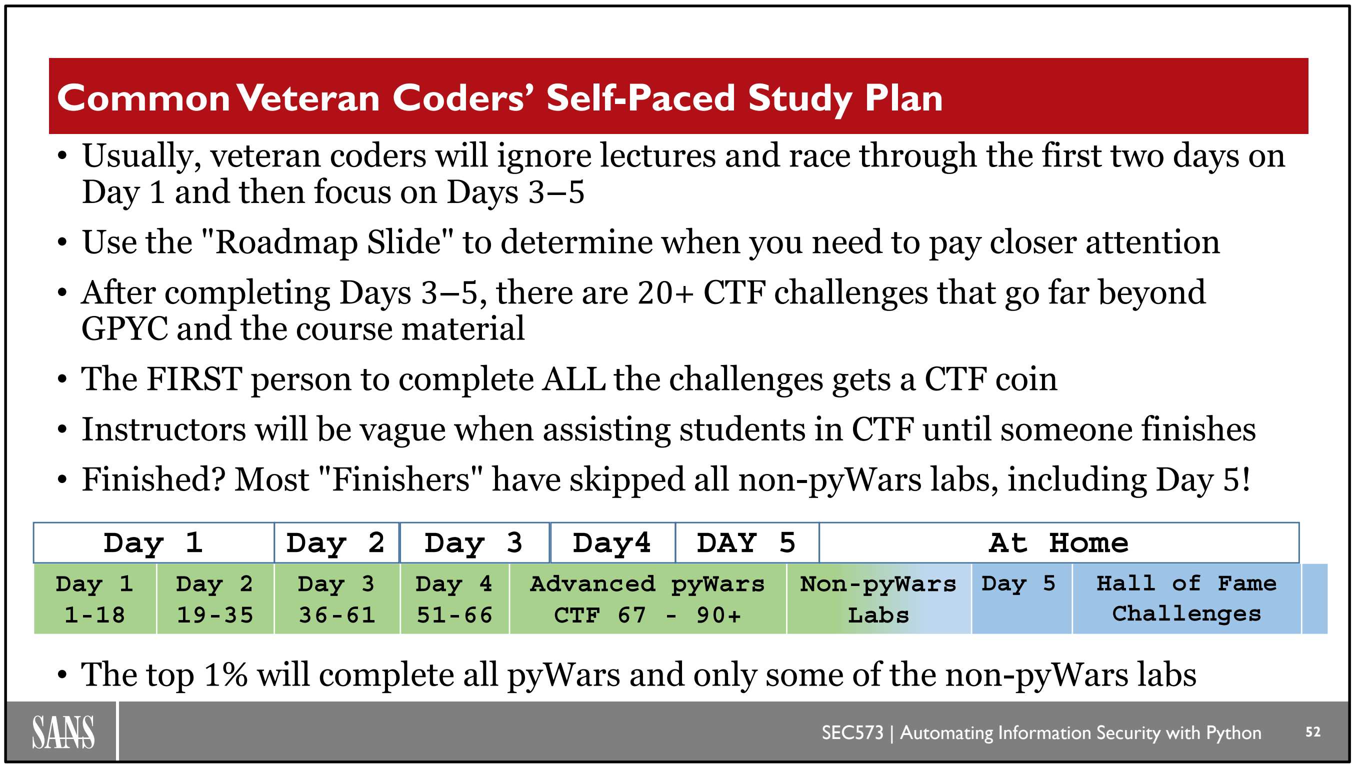
If you're new to coding and you want to pass the GIAC Python Coder Exam (GPYC), let's talk about how to do that. Let's face it—no Python coding certification that someone with no coding experience can pass after five days of lecture is worth having. It will take some work to get there, but the course is structured with you in mind. pyWars will have the advanced users engaged for the first several days 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-third of them. The first third is enough for you to understand the concepts required to move on to the next subject. Then, during the evenings after class or when you get back home next week, go back and finish the other labs. All of the labs that are in the book are in your offline local pyWars server.

There are many labs in your books that are not pyWars labs. 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.

Don't concern yourself with the advanced CTF challenges. You won't need that for the exam. Instead, focus on building a solid foundation of essential coding skills that you can apply to solving security challenges.

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.



Common Veteran Coders' Self-Paced Study Plan

- Usually, veteran coders will ignore lectures and race through the first two days on Day 1 and then focus on Days 3–5
- Use the "Roadmap Slide" to determine when you need to pay closer attention
- After completing Days 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 Day 5!

Day 1		Day 2	Day 3	Day4	DAY 5	At Home		
Day 1 1-18	Day 2 19-35	Day 3 36-61	Day 4 51-66	Advanced pyWars CTF 67 - 90+		Non-pyWars Labs	Day 5	Hall of Fame Challenges

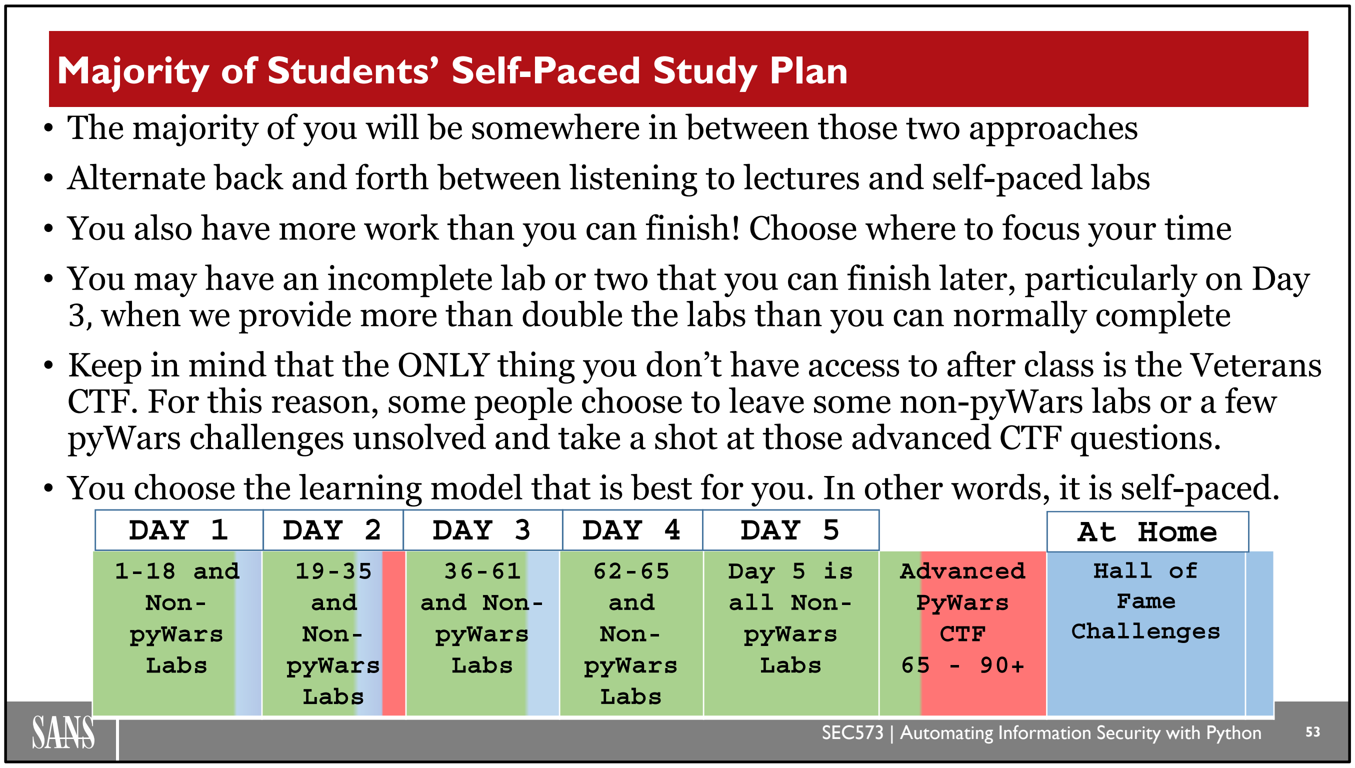
- The top 1% will complete all pyWars and only some of the non-pyWars labs

SANS

SEC573 | Automating Information Security with Python 52

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 day and are working on Day 3 material by the start of Day 2. The most advanced coders may finish all of the pyWars challenges in the course material by the end of Day 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 Day 5's material because it isn't in pyWars. If you finished, you can 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 can 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.



Majority of Students' Self-Paced Study Plan

- The majority of you will be somewhere in between those two approaches
- Alternate back and forth between listening to lectures and self-paced labs
- You also have more work than you can finish! Choose where to focus your time
- You may have an incomplete lab or two that you can finish later, particularly on Day 3, when we provide more than double the labs than you can normally complete
- Keep in mind that the ONLY thing you don't have access to after class is the Veterans CTF. For this reason, some people choose to leave some non-pyWars labs or a few pyWars challenges unsolved and take a shot at those advanced CTF questions.
- 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
1-18 and Non-pyWars Labs	19-35 and Non-pyWars Labs	36-61 and Non-pyWars Labs	62-65 and Non-pyWars Labs	Day 5 is all Non-pyWars Labs	Advanced PyWars CTF 65 - 90+	Hall of Fame Challenges

SANS | SEC573 | Automating Information Security with Python 53

The class is designed to have more material than a veteran coder can handle. So if you haven't been coding for a while, the same will apply to you.

Most of you will be somewhere in between ignoring the lecture part of the time when you get wrapped up in a challenge but tuning the instructor back in to build new skills as you go along. You will still have a few labs that you don't finish, and that's OK. That is work for you to complete when you get back home or in the evening with your offline pyWars server. You can complete every lab that is in this course when you get back home. The only thing you do not have access to afterwards is the CTF challenge for veteran coders. You may want to consider that when deciding where to spend your time in class. Some students decide to leave a few of the book challenges and pyWars labs unfinished to take a shot at a few of the CTF challenges.

Whatever you choose to do is fine. The instructor will answer your questions during labs and breaks and help you wherever you are. This class is, after all, self-paced.

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 raise your hand and ask them questions **AT ANY TIME**!

SANS | SEC573 | Automating Information Security with Python 54

Because pyWars is largely an extra self-guided exercise, 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 day.

pyWars: Getting Started

- Begin anytime you are ready to play
- Start in the **essentials-workshop** directory
- Start a Python shell by typing **python**
- **import pyWars** (case-sensitive)

```
$ cd ~/Documents/pythonclass/essentials-workshop/
$ python3
Python 3.6.8 (default, Jan 14 2019, 11:02:34)
[GCC 8.0.1 20180414 (experimental) [trunk revision 259383]] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import pyWars
>>> game = pyWars.exercise()
```

You can begin playing pyWars whenever you are ready. I'll introduce it to everyone now so that advanced programmers can jump right in. If some of this seems too advanced, that is OK. Just come back to this section when you are ready to begin playing.

Playing pyWars is simple. First, you need to check in to the directory containing the pyWars module:

```
$ cd ~/Documents/pythonclass/essentials-workshop/
```

Then start Python by typing **python** and pressing **Enter**.

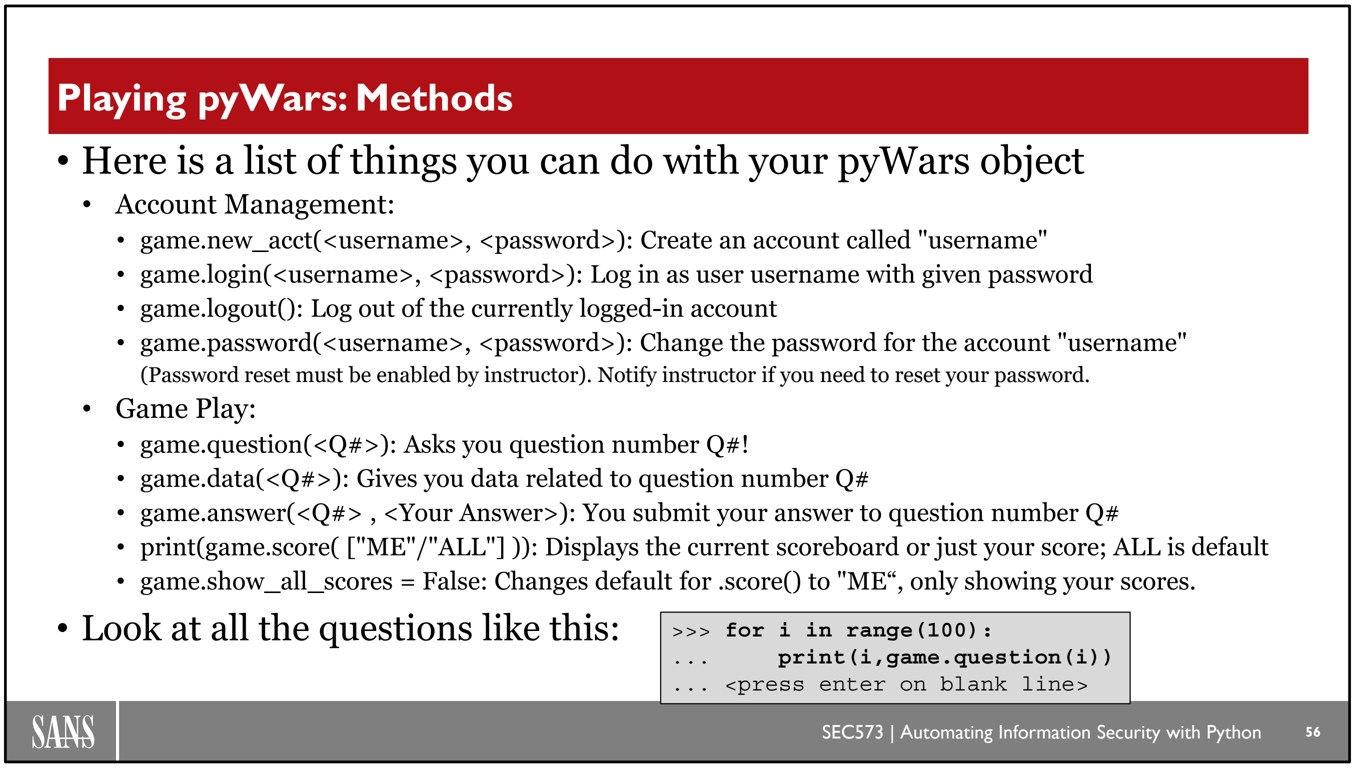
```
$ python3
```

Python starts and you will get the interactive python prompt ">>>". Then, import the pyWars module like this:

```
>>>  import pyWars
```

Last, we create a pyWars object in memory that we can use to interact with the server and store it in a variable. In this case, the variable is named "game", but it can be any name that you like!

```
>>>  game = pyWars.exercise()
```



Playing pyWars: Methods

- Here is a list of things you can do with your pyWars object
  - Account Management:
    - game.new_acct(<username>, <password>): 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(<username>, <password>): Change the password for the account "username" (Password reset must be enabled by instructor). Notify instructor if you need to reset your password.
  - Game Play:
    - game.question(<Q#>): Asks you question number Q#!
    - game.data(<Q#>): Gives you data related to question number Q#
    - game.answer(<Q#> , <Your Answer>): You submit your answer to question number Q#
    - print(game.score(["ME"/"ALL"])): Displays the current scoreboard or just your score; ALL is default
    - game.show_all_scores = False: Changes default for .score() to "ME“, only showing your scores.
- Look at all the questions like this:

```
>>> for i in range(100):
...     print(i,game.question(i))
... <press enter on blank line>
```

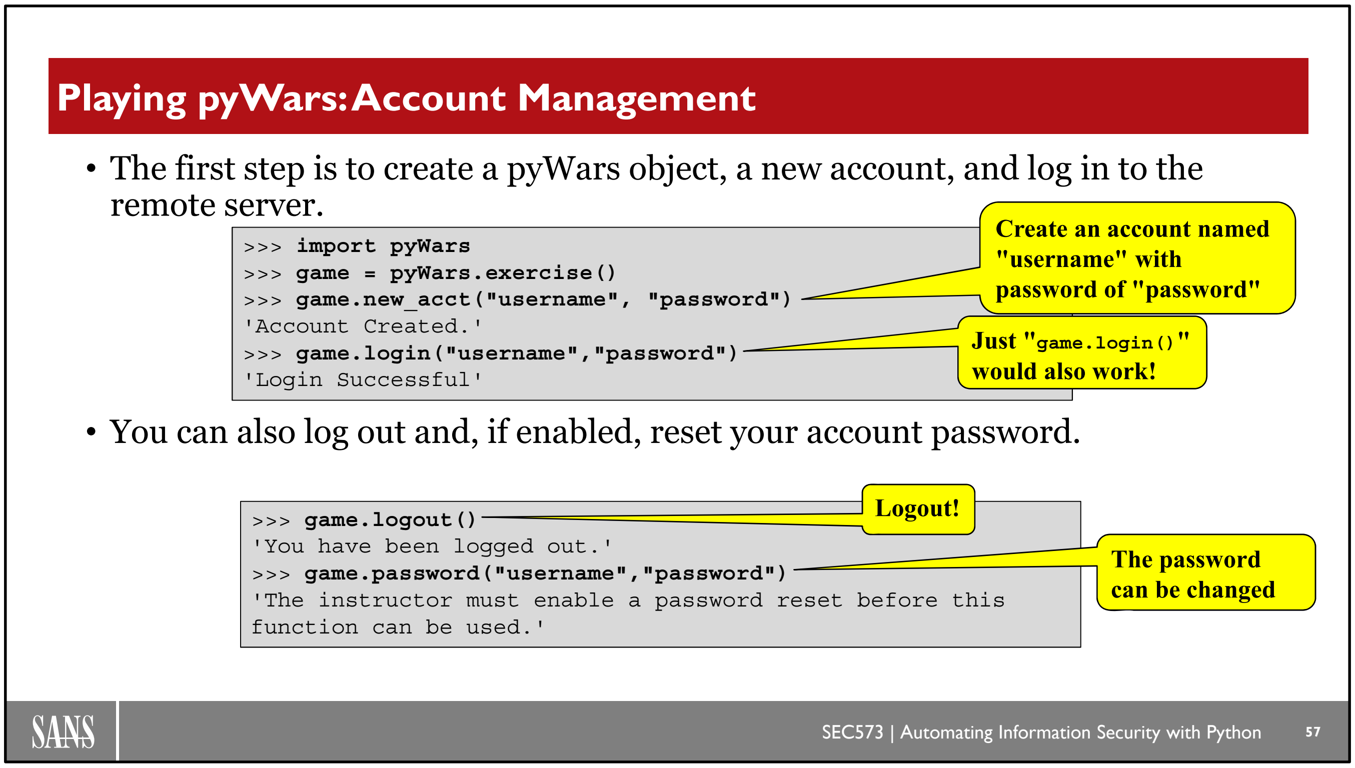
SANS SEC573 | Automating Information Security with Python 56

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 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 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 two arguments separated by a comma. The first is the question number you are submitting an answer to and the second is your answer containing the manipulated data.

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



Playing pyWars: Account Management

- The first step is to create a pyWars object, a new account, and log in to the remote server.

```
>>> import pyWars
>>> game = pyWars.exercise()
>>> game.new_acct("username", "password")
'Account Created.'
>>> game.login("username","password")
'Login Successful'
```

Create an account named "username" with password of "password"

Just "`game.login()`" would also work!

- You can also log out and, if enabled, reset your account password.

```
>>> game.logout()
'You have been logged out.'
>>> game.password("username","password")
'The instructor must enable a password reset before this
function can be used.'
```

Logout!

The password can be changed

SANS

SEC573 | Automating Information Security with Python 57

After importing the module, you will create an instance of a pyWars exercise object. We will cover objects in detail later, so don't worry if you don't quite understand everything yet. But even to the complete novice, this step isn't that difficult. First, you create an instance of the pyWars exercise object by typing the following:

```
>>> game=pyWars.exercise()
```

This creates a new variable named "game" that will hold our pyWars object that we can use to interact with the server. Now we can use that object to create a new account on the server for you to use.

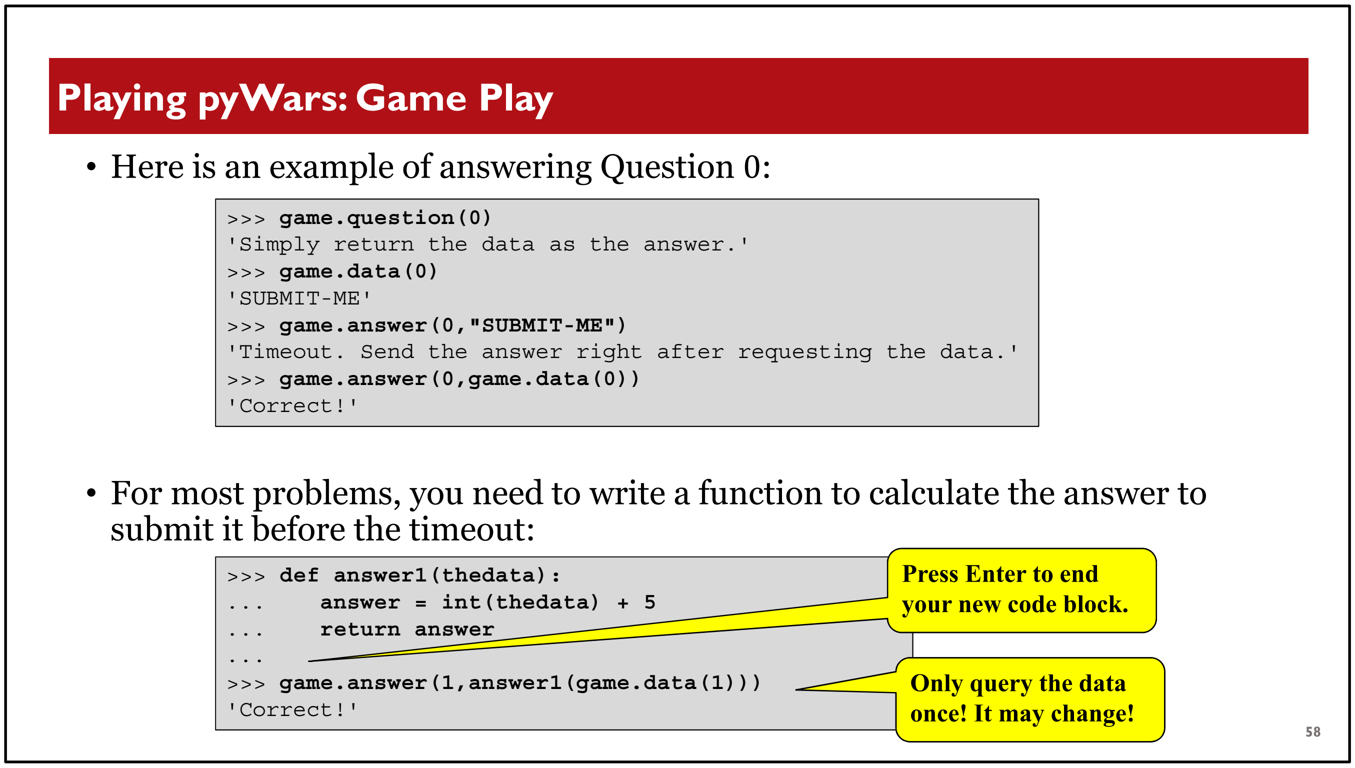
```
>>> game.new_acct("username", "password")
```

Now you can use that username and password to log in to the server and play. To log in, you call the .login() method and pass your username and password.

```
>>> game.login("username","password")
```

It is worth noting that when you call either new_acct() or login(), the client remembers the username and password you used, so you can call login() without passing any username and password in that pyWars window or script. This provides you with a little more protection from shoulder-surfing classmates. It only remembers this information in the current session. If you open a new terminal window or a new script, you will have to log in with the username and password at least once before you can call login() without arguments.

That's it! You are ready to play! You can also logout() of the server when you're not using it. That is always a good security practice. Additionally, if you forget your password, you can use .password() to choose another password, but this feature can only be used after the instructor flags your account for password reset on the server.



Playing pyWars: Game Play

- Here is an example of answering Question 0:

```
>>> game.question(0)
'Simply return the data as the answer.'
>>> game.data(0)
'SUBMIT-ME'
>>> game.answer(0,"SUBMIT-ME")
'Timeout. Send the answer right after requesting the data.'
>>> game.answer(0,game.data(0))
'Correct!'
```

- For most problems, you need to write a function to calculate the answer to submit it before the timeout:

```
>>> def answer1(thedata):
...     answer = int(thedata) + 5
...     return answer
...
>>> game.answer(1,answer1(game.data(1)))
'Correct!'
```

Press Enter to end your new code block.

Only query the data once! It may change!

58

Here is an example of how you would query and answer Question 0. In this case, calling **game.question(0)** tells us that all we have to do to answer the question is to take what is returned by calling **game.data(0)** and submit that as an answer. If we call **game.data(0)**, we see that we must submit the answer "SUBMIT-ME". We call **game.answer(0,'SUBMIT-ME')**. But instead of scoring, we get back a message indicating that we didn't submit the answer in time. We need to automatically submit an answer based on calling **game.data(0)**. Again, Question 0 is very easy. We just have to submit the data. So we call **game.answer(0,game.data(0)),** and we get back the response "Correct". We scored a point!

Sometimes it requires a good bit of processing to calculate the answer. In those cases, you will need to define a function to process the data and return the results. We will talk about declaring and using functions more later, but here is a look at how we would do that:

```
>>> def answer1(thedata):
...     answer=int(thedata)+5
...     return answer
...
>>> game.answer(1,answer1(game.data(1)))
'Correct'
```

This is a great approach for solving most of the challenges. Keep in mind that the .data() function will often return a different value each time you query it. You must submit the answer for the LAST time you queried the .data() method. Using a function and passing the results of calling data() to it ensures that you query the data only once before you answer it.

pyWars Rules

- Only THREE pyWars logins are permitted per IP address. Running a script that executes .login() a fourth time will expire the first login.
- You have only 2 seconds between the time you request data and submit your answer
  - You need to programmatically process .data() as described by .question() to submit .answer() before the timeout!
- Submitting an answer more than once does not score more than one point. You can score on a given question only 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.
- In short, no hacking the scoring server

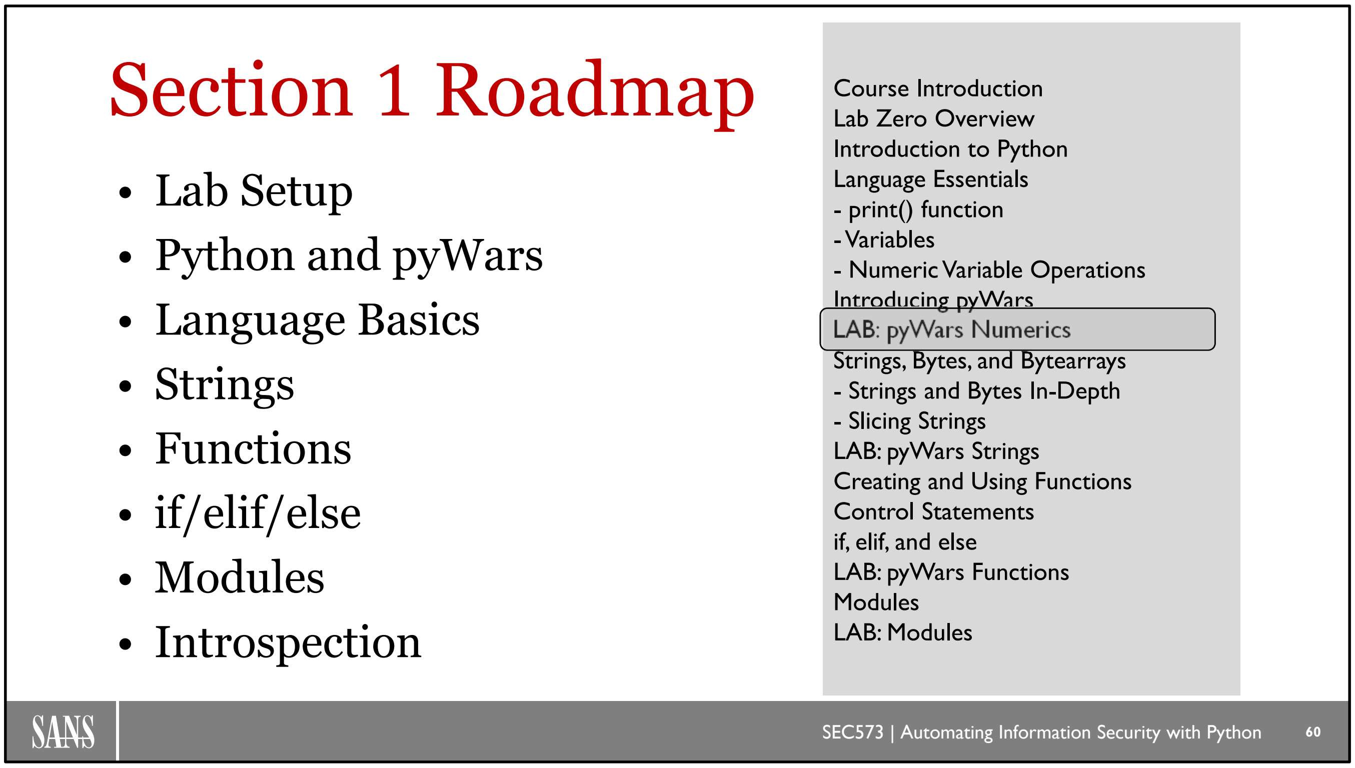
Under normal circumstances, the pyWars server will only allow three logins from any given IP address. This protects your account from session hijacking attacks. Although the instructor can change this setting if the classroom network topology or class size requires it, you will most likely find that you are limited to playing pyWars in, at most, three terminal windows or running scripts at a time. This means that if you are logged in in a terminal window and you run a script that executes login() three times, then you will likely have to log in again in your terminal window.

After you have queried the data associated with a question, by calling games.data(#), you have only 2 seconds to submit the correct answer. 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 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!

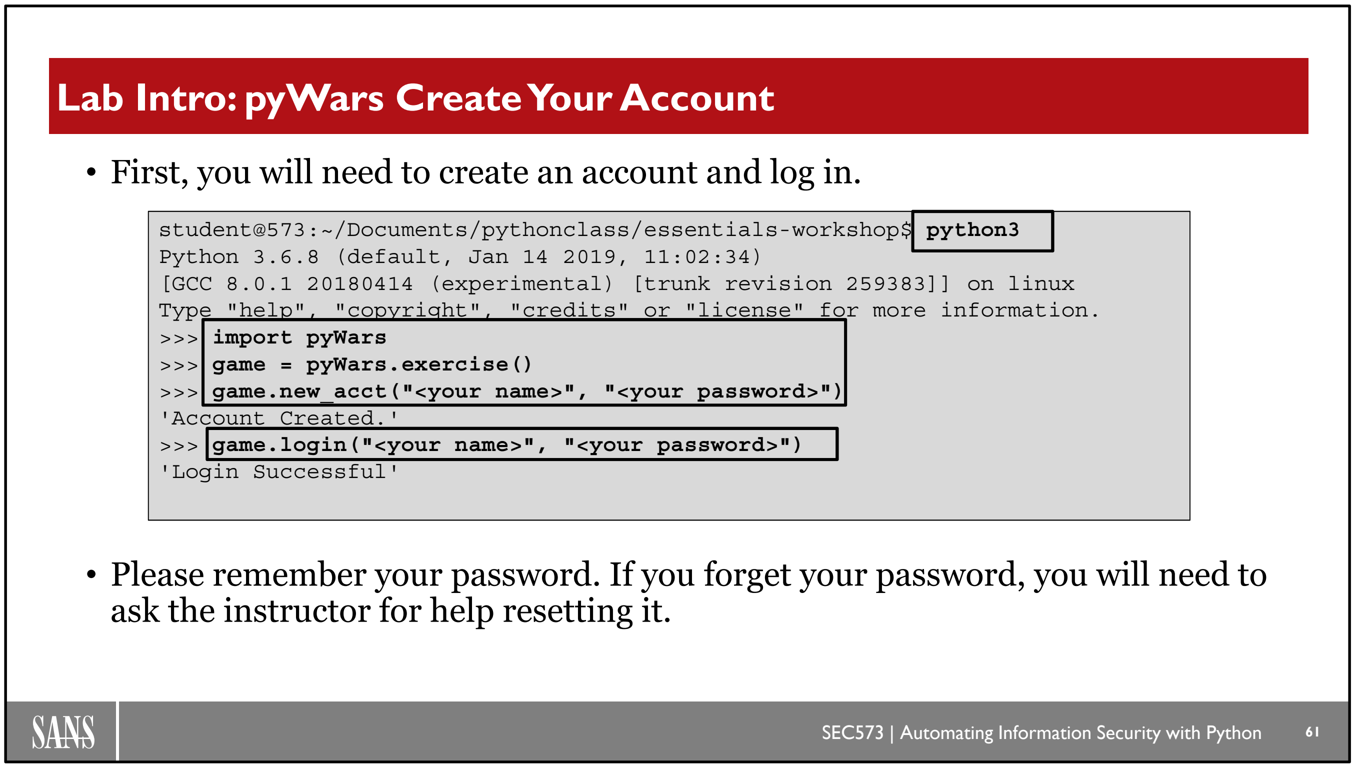
There are several additional rules, but it all comes down to doing the right thing. Don't hack anyone or my servers. Play the game using the tools provided and have fun.



Section 1 Roadmap

- Lab Setup
- Python and pyWars
- Language Basics
- Strings
- Functions
- if/elif/else
- Modules
- Introspection

This is a Roadmap slide.



Lab Intro: pyWars Create Your Account

- First, you will need to create an account and log in.

```
student@573:~/Documents/pythonclass/essentials-workshop$ python3
Python 3.6.8 (default, Jan 14 2019, 11:02:34)
[GCC 8.0.1 20180414 (experimental) [trunk revision 259383]] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import pyWars
>>> game = pyWars.exercise()
>>> game.new_acct("<your name>", "<your password>")
'Account Created.'
>>> game.login("<your name>", "<your password>")
'Login Successful'
```

- Please remember your password. If you forget your password, you will need to ask the instructor for help resetting it.

SEC573 | Automating Information Security with Python 61

The first step is to change to the **essentials-workshop** directory and start the Python interactive shell:

```
$ cd essentials-workshop/
$ python3
Python 3.6.8 (default, Jan 14 2019, 11:02:34)
[GCC 8.0.1 20180414 (experimental) [trunk revision 259383]] on linux
Type "help", "copyright", "credits" or "license" for more information.
```

Next, you import the pyWars module by typing **import pyWars**. Note that this command is case sensitive:

```
>>> import pyWars
```

Then create a variable that will hold your pyWars exercise object. There is nothing special about the name of the variable we are using. Here we use "game", but this can be anything we like. You will see in your Workbook I often use the variable "d" because it is shorter and easier to type. Choose whatever variable name you like, but it is best to choose a variable name that is intuitive and explains what type of data it holds.

```
>>> game=pyWars.exercise()
```

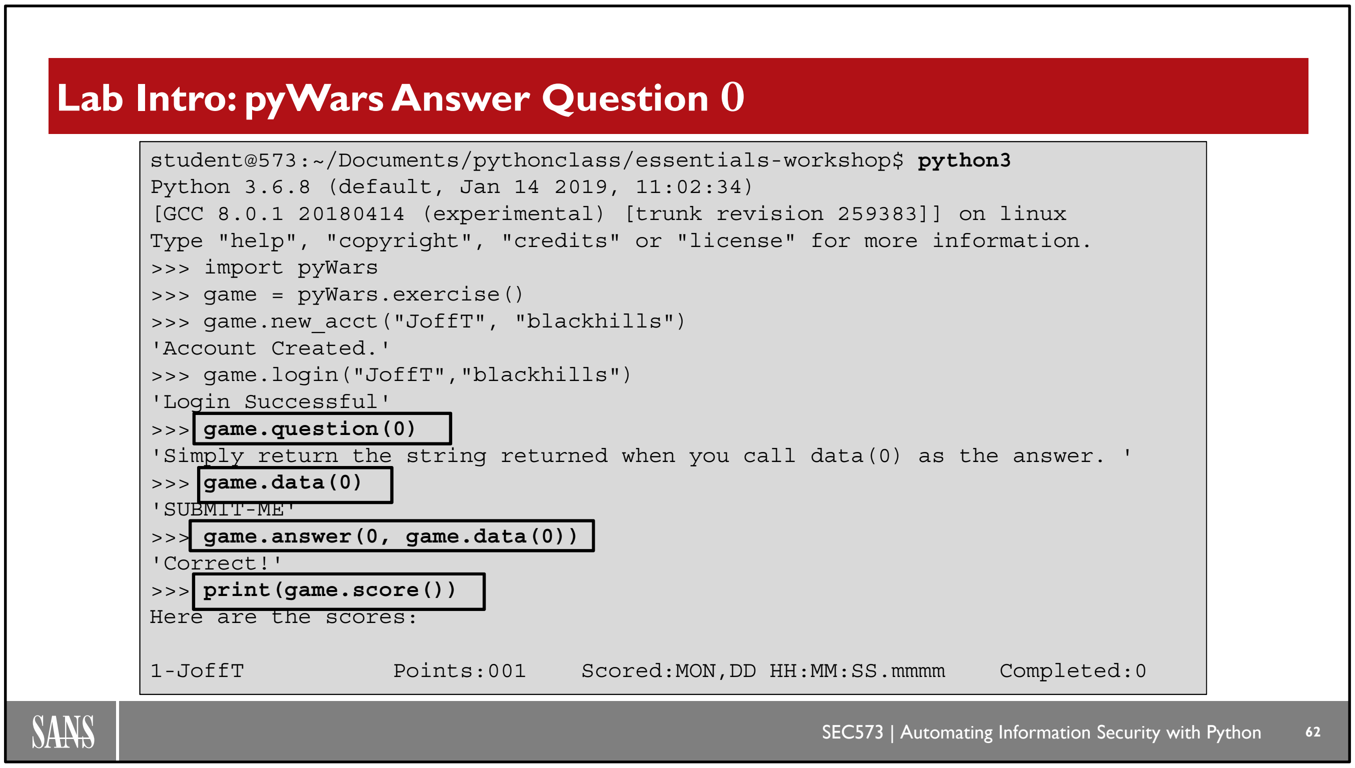
Next, you may create your account on the server with a username and password of your choosing:

```
 >>> game.new_acct("<your name>", "<your password>")
'Account Created.'
```

Now that the account exists, you can log in to the server. The pyWars client will remember the username and password that are passed to either the new_acct() or login() method. Now you can log in with just login() or provide the username and password as shown in the slide above :

```
>>> game.login()
'Login Successful'
```

Your pyWars object can now perform authenticated actions against the pyWars server, such as asking for challenges and submitting answers.



Lab Intro: pyWars Answer Question 0

```
student@573:~/Documents/pythonclass/essentials-workshop$ python3
Python 3.6.8 (default, Jan 14 2019, 11:02:34)
[GCC 8.0.1 20180414 (experimental) [trunk revision 259383]] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import pyWars
>>> game = pyWars.exercise()
>>> game.new_acct("JoffT", "blackhills")
'Account Created.'
>>> game.login("JoffT","blackhills")
'Login Successful'
>>> game.question(0)
'Simply return the string returned when you call data(0) as the answer. '
>>> game.data(0)
'SUBMIT-ME'
>>> game.answer(0, game.data(0))
'Correct!'
>>> print(game.score())
Here are the scores:

1-JoffT             Points:001     Scored:MON,DD HH:MM:SS.mmmm     Completed:0
```

SANS

SEC573 | Automating Information Security with Python 62

Now look at the question and data associated with Question 0 by passing a zero to game.question():

```
>>> game.question(0)
'Simply return the data as the answer.'
>>> game.data(0)
'SUBMIT-ME'
```

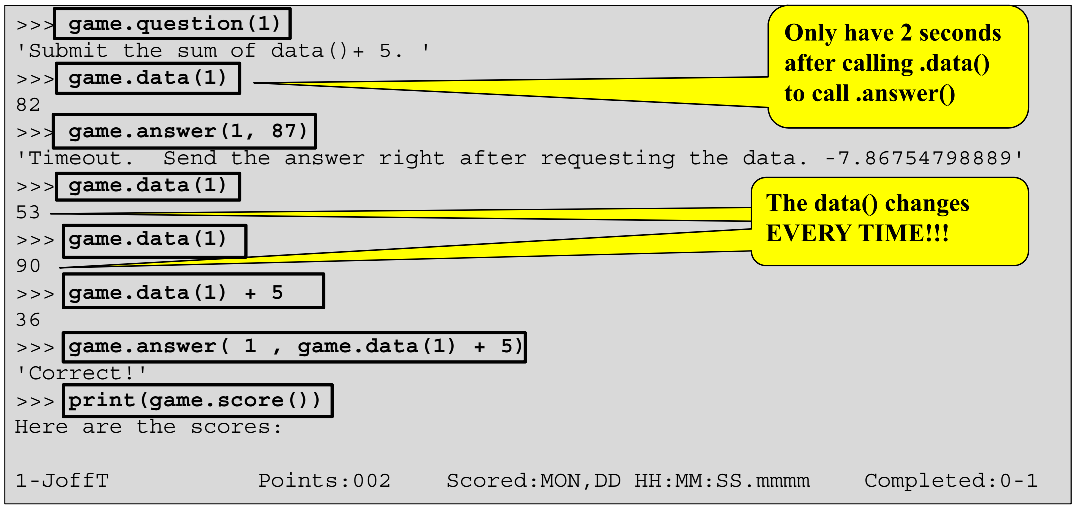
You can see that Question 0 requires no manipulation of the data at all. All you have to do is read the data and then submit it as the answer. You can do that like this:

```
>>> game.answer( 0, game.data(0) )
'Correct'
```

Last, print the score to see how people are doing:

```
>>> print(game.score())
Here are the scores:

1-JoffT             Points:001     Scored:MON,DD HH:MM:SS.mmmm     Completed:0
>>>
```

Lab Intro: pyWars Answer Question 1



```
>>> game.question(1)
'Submit the sum of data()+ 5. '
>>> game.data(1)
82
>>> game.answer(1, 87)
'Timeout.  Send the answer right after requesting the data. -7.86754798889'
>>> game.data(1)
53
>>> game.data(1)
90
>>> game.data(1) + 5
36
>>> game.answer( 1 , game.data(1) + 5)
'Correct!'
>>> print(game.score())
Here are the scores:

1-JoffT           Points:002    Scored:MON,DD HH:MM:SS.mmmm    Completed:0-1
```

Only have 2 seconds after calling .data() to call .answer()
The data() changes EVERY TIME!!!

Now let's try Question 1 together. Call the game variable's **question()** method and pass it a **1** indicating that you want to see question number 1:

```
>>> game.question(1)
```

It says, "Submit the sum of data()+5." Remember, all pyWars challenges will ask you to do something with the corresponding data and return an answer. So let's look at the .data() values for Question 1:

```
>>> game.data(1)
```

It gives back a number. In the example above, it gave 82. Notice that if you call .data() again, it will give you back a different number.

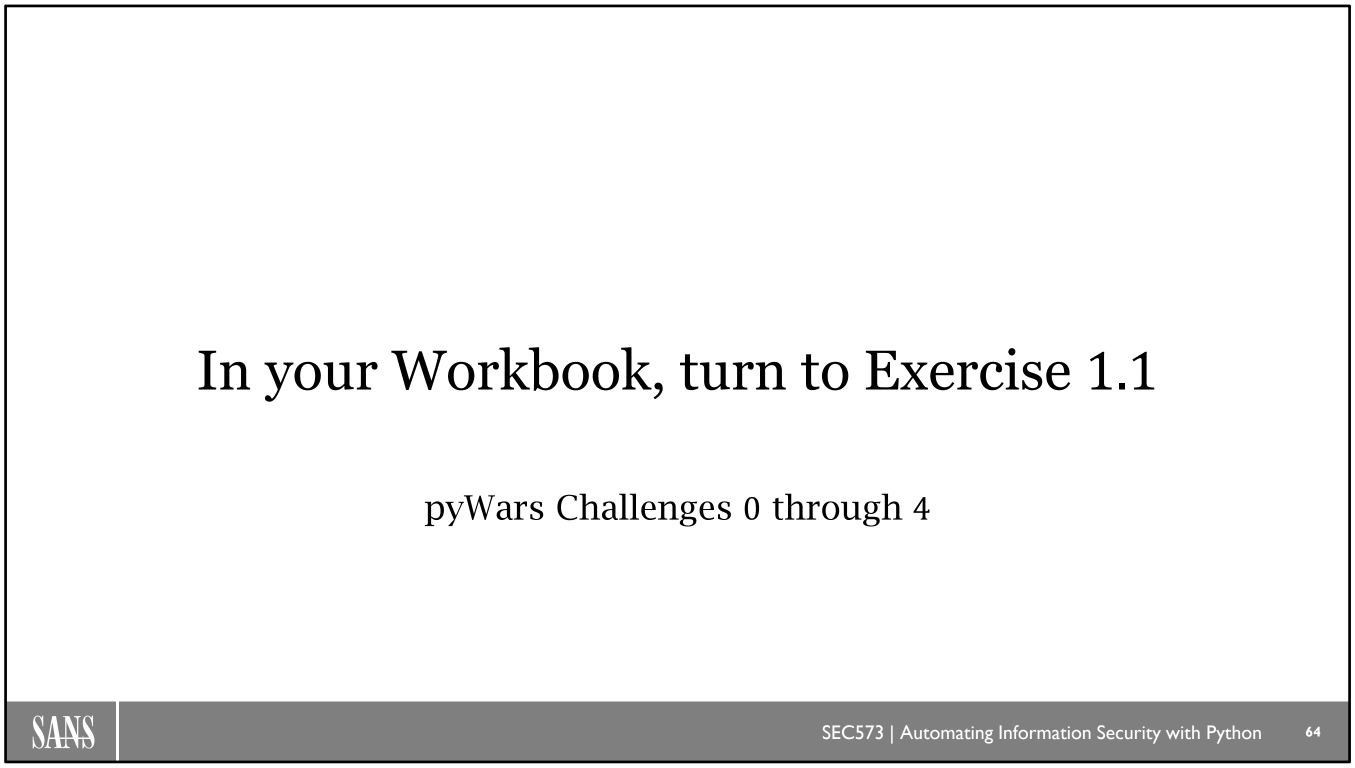
```
>>> game.data(1)
```

The second time it gave us the number 53. And remember, you have only 2 seconds to answer. Notice that, in the preceding example, when we try to submit an answer of 87, it tells us that it timed out, indicating that we took more than 2 seconds. So you will have to write Python code to call the .data() method, capture the number it gives you, and submit an answer in less than 2 seconds. Fortunately, computers are pretty fast at math, and Python can add two numbers together using a plus sign. When you call **game.data(1)**, it returns a number. So, if you want to add 5 to that number, all you have to do is add **+5** to the end of the call to game.data(). It will call the function and add 5 to the result:

```
>>> game.data(1)+5
```

In the example above, this resulted in "36". Therefore, .data(1) must have returned a 31. Python then dutifully added 5 to that, giving an answer of 36. Let's submit that as our answer to number 1:

```
>>> game.answer( 1 ,  game.data(1) + 5 )
```



In your Workbook, turn to Exercise 1.1

pyWars Challenges 0 through 4

SANS

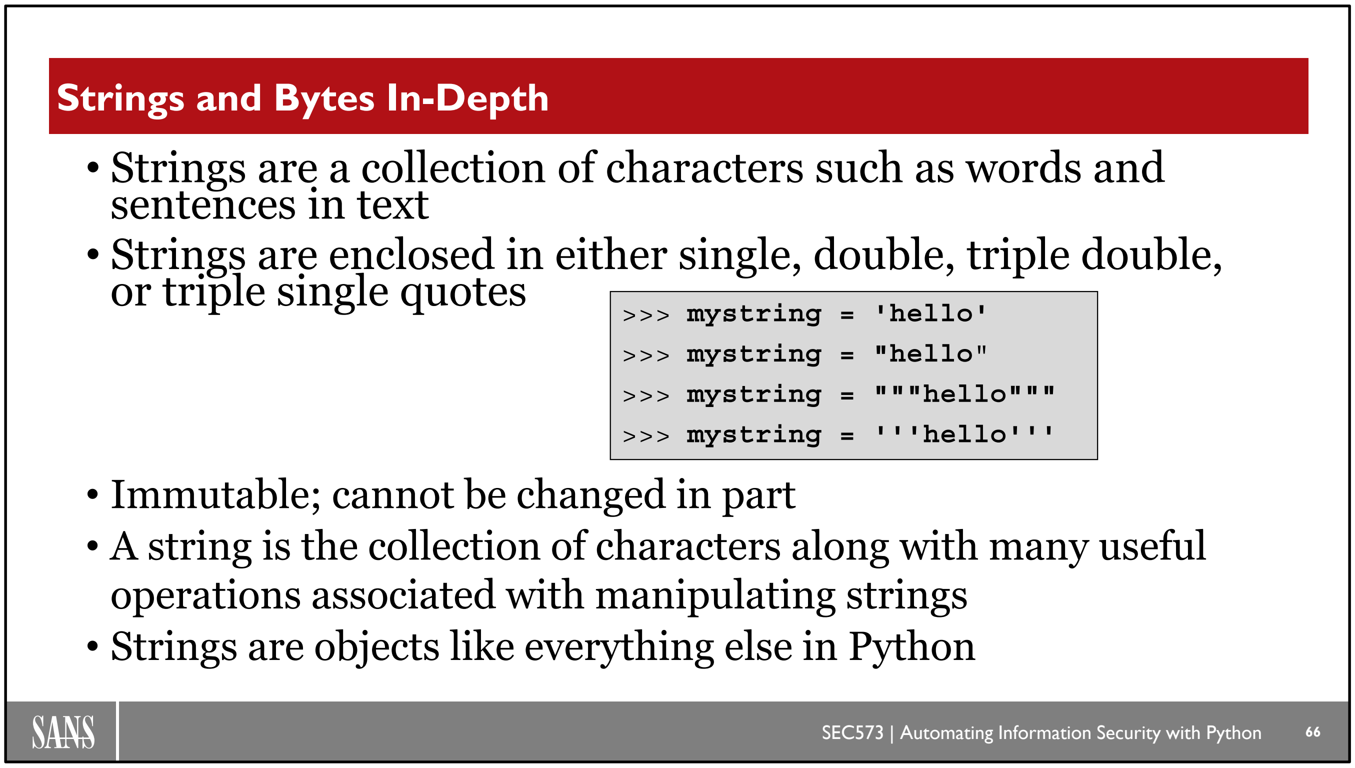
SEC573 | Automating Information Security with Python 64

Please complete the exercise in your Workbook.

Section 1 Roadmap

- Lab Setup
- Python and pyWars
- Language Basics
- Strings
- Functions
- if/elif/else
- Modules
- Introspection

This is a Roadmap slide.



Strings and Bytes In-Depth

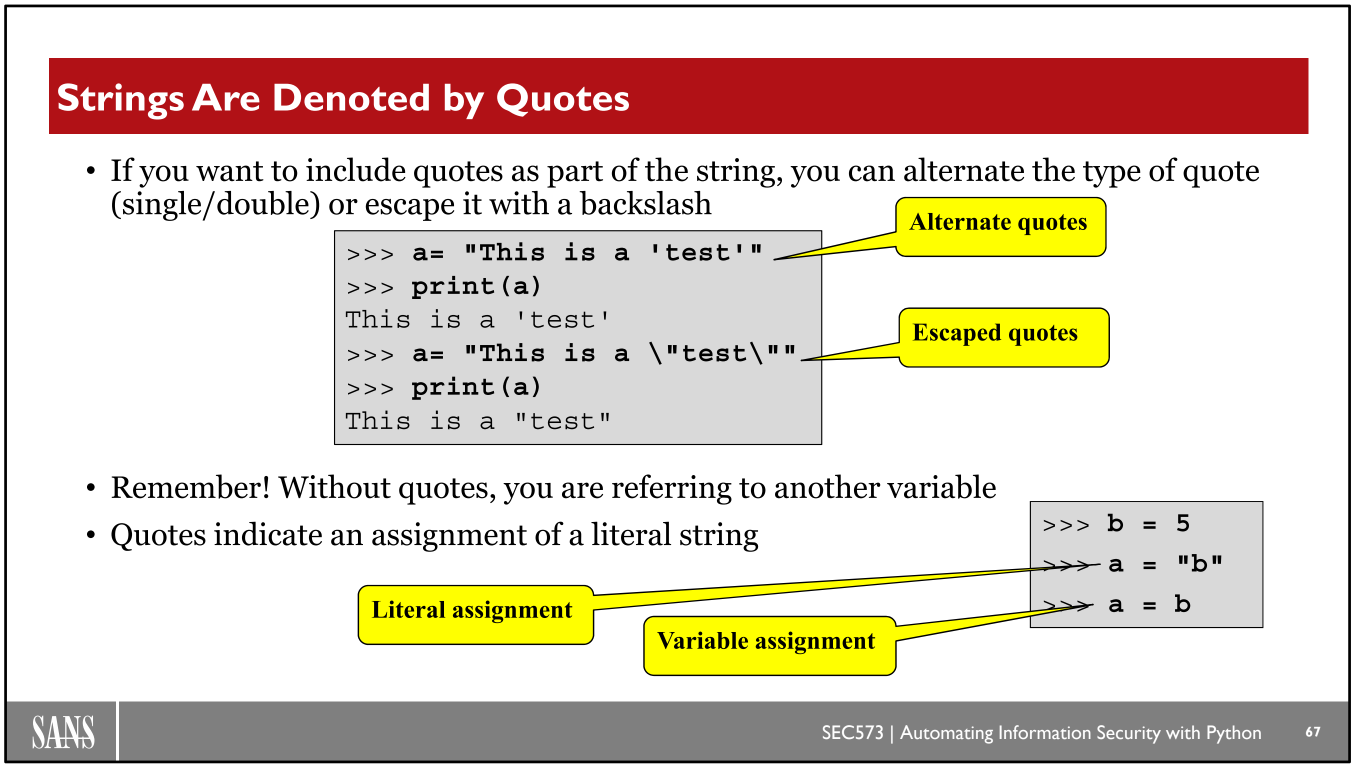
- Strings are a collection of characters such as words and sentences in text
- Strings are enclosed in either single, double, triple double, or triple single quotes

```
>>> 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

SANS SEC573 | Automating Information Security with Python 66

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.



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

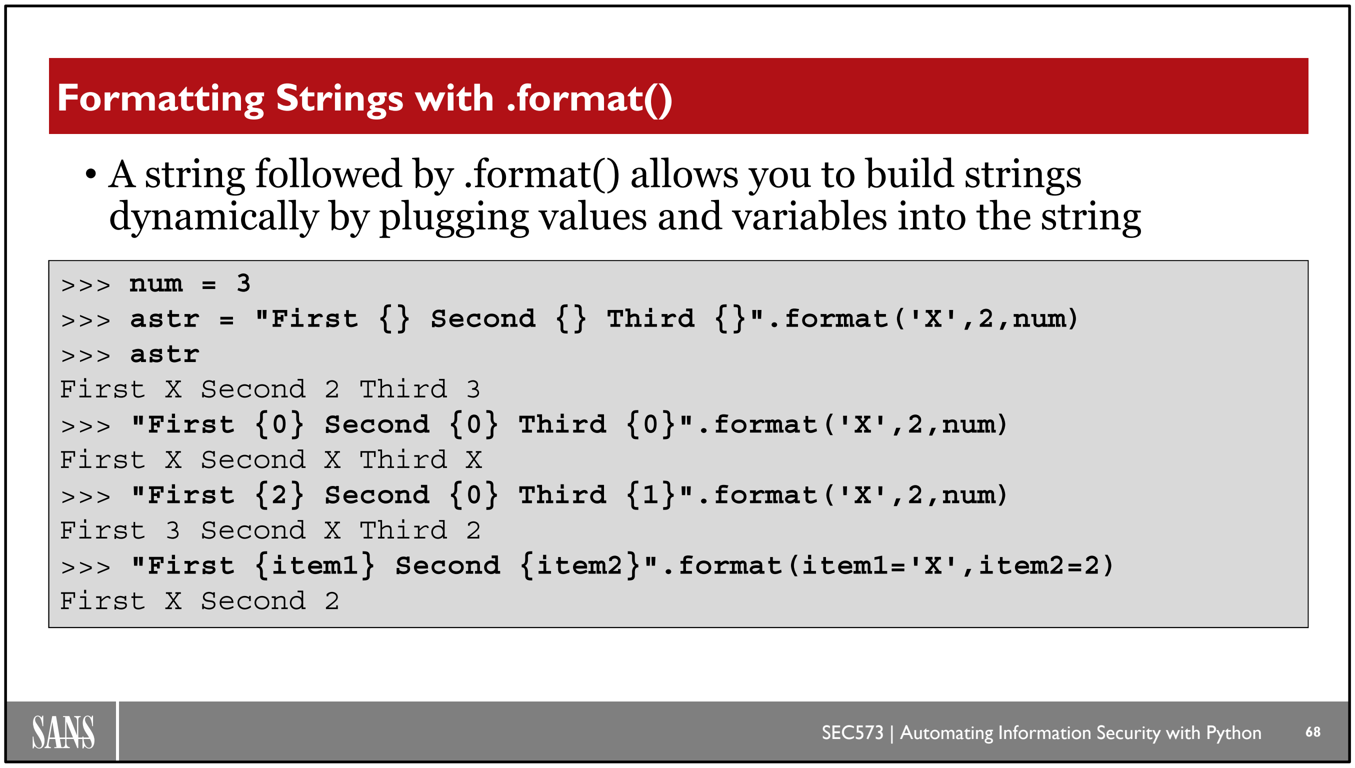
```
>>> b = 5
>>> a = "b"
>>> a = b
```

Literal assignment

Variable assignment

SANS SEC573 | Automating Information Security with Python 67

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".



Formatting Strings with .format()

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

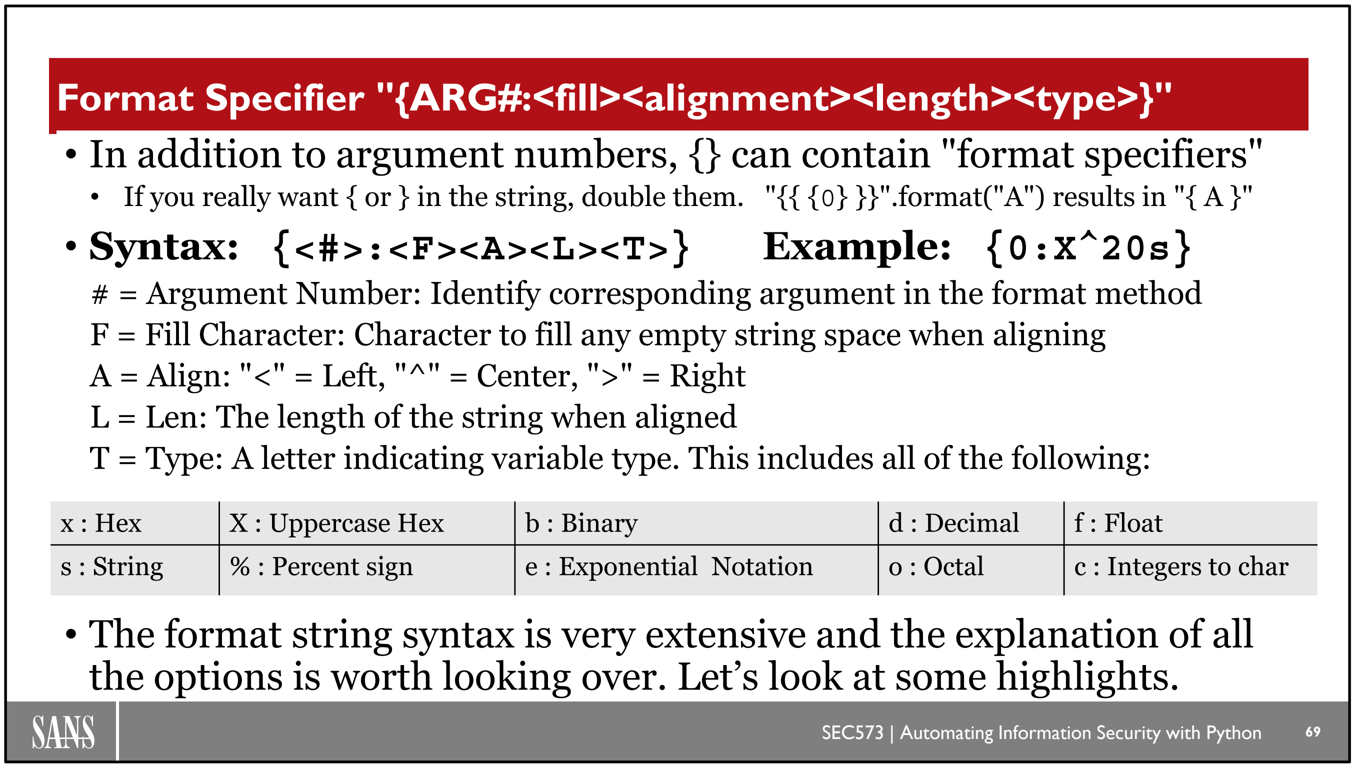
```
>>> num = 3
>>> astr = "First {} Second {} Third {}".format('X',2,num)
>>> astr
First X Second 2 Third 3
>>> "First {0} Second {0} Third {0}".format('X',2,num)
First X Second X Third X
>>> "First {2} Second {0} Third {1}".format('X',2,num)
First 3 Second X Third 2
>>> "First {item1} Second {item2}".format(item1='X',item2=2)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

SANS SEC573 | Automating Information Security with Python 68

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.



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.

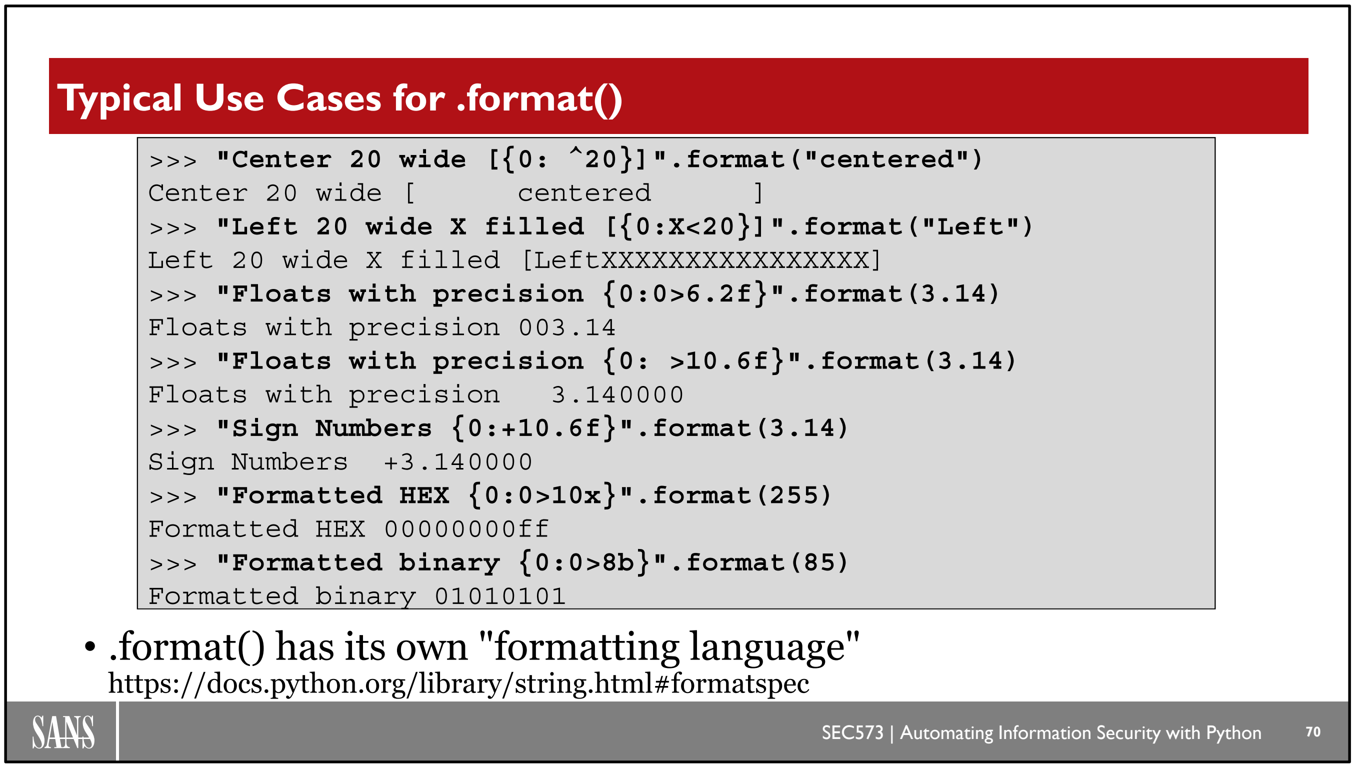
SANS SEC573 | Automating Information Security with Python 69

Between the brackets, you put 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.



Typical Use Cases for .format()

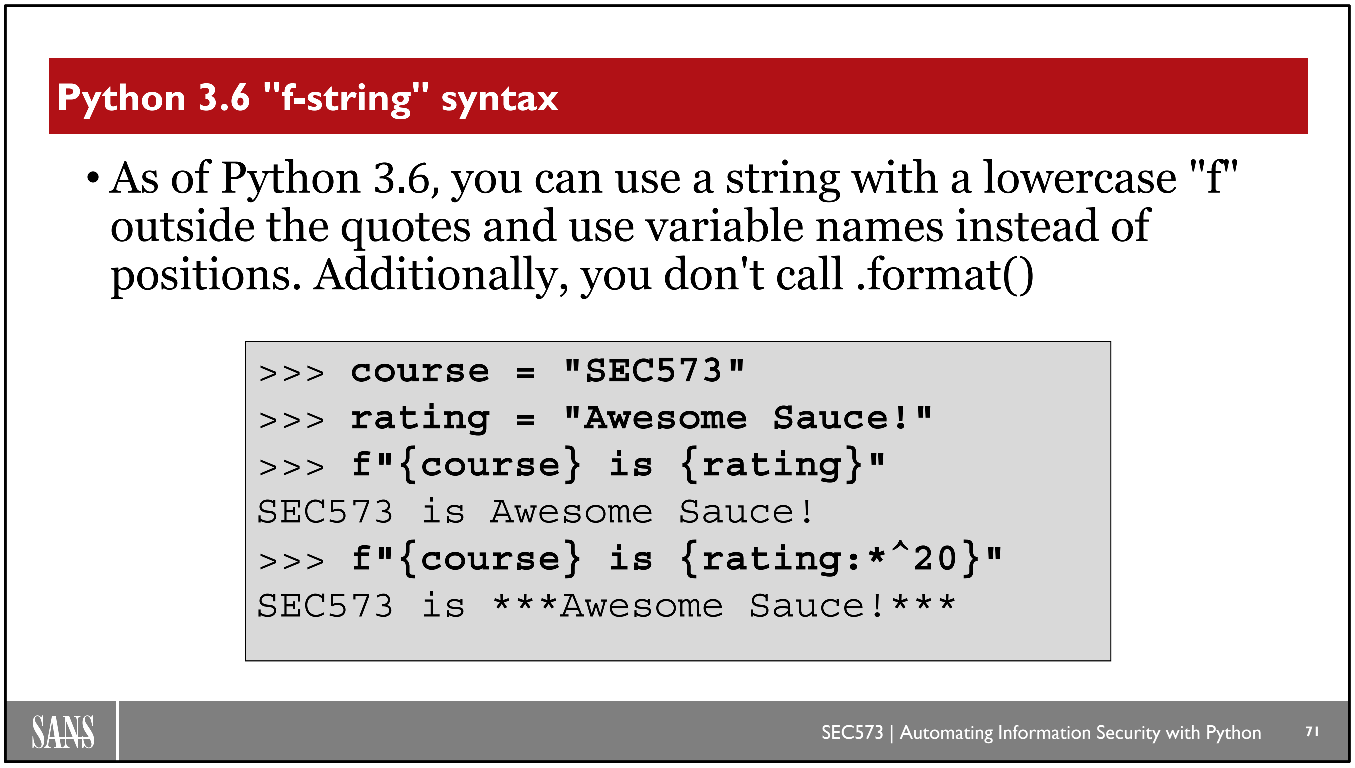
```
>>> "Center 20 wide [{0: ^20}]".format("centered")
Center 20 wide [      centered      ]
>>> "Left 20 wide X filled [{0:X<20}]".format("Left")
Left 20 wide X filled [LeftXXXXXXXXXXXXXXXX]
>>> "Floats with precision {0:0>6.2f}".format(3.14)
Floats with precision 003.14
>>> "Floats with precision {0: >10.6f}".format(3.14)
Floats with precision   3.140000
>>> "Sign Numbers {0:+10.6f}".format(3.14)
Sign 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

SANS

SEC573 | Automating Information Security with Python 70

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 fill 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.



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!***
```

SANS SEC573 | Automating Information Security with Python 71

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 of 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.

C Style Format Strings

- Python also supports format strings typically used in C and other languages.
- You will see these used frequently because of broad support
- Format string contains "%" followed by type specifier
- % (percent sign) and parentheses containing variable follows the string

```
>>> "String %s Decimal %d" % ("HI!",100)
String HI! Decimal 100
>>> "Formatted Decimal %010d" % (100)
Formatted Decimal 0000000100
>>> "Formatted Float %10.4f" % (3.14)
Formatted Float     3.1400
>>> "Centered String [%10s]" % ("HI".center(10))
Centered String [    HI    ]
```

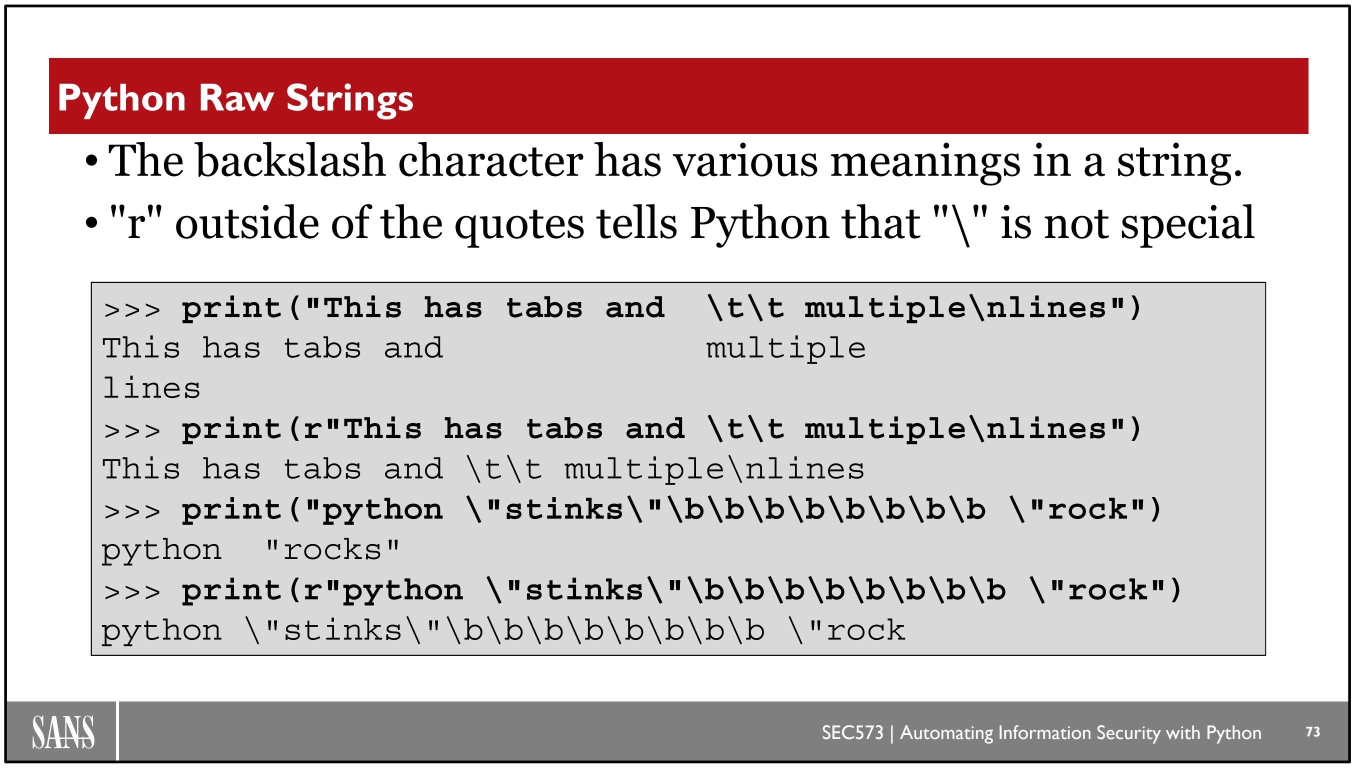
There is another style of format strings that are typically associated with C and other languages. At one time, this was the only style of format supported in Python2. These are still supported in Python3 and still widely used, so you should be familiar with them.

With this type for format strings, you use a percent sign followed by a specifier in the string instead of braces. Then instead of .format() after the string, you put another percent sign and the argument to plug into the string inside of parentheses. The types are largely the same as they are in the new style of format strings. In the example above, "String %s Decimal %d" will put the word "HI!" where the %s is and the decimal number 100 where the %d is. Additional modifiers like the "0" to fill with leading zeros and a width can be specified between the percent sign and the type. For example, "%010d" prints a decimal number 10 characters wide with leading zeros. "%10.4f" prints a floating point number 10 characters wide with 4 places after the decimal point.

Notably absent from these older format strings are the alignments. Centering, left of right justifying of strings has to be done in conjunction with additional parts of Python, such as the .center() method. Right aligning is the default, so "%10s" without the use of center would print it right aligned. To left align things, you would use a negative size specifier.

```
>>> "Hi Left aligned [%-10s]" % ("HI")
'Hi Left aligned [HI        ]'
```

The arguments are not as flexible as in the new style, but they are compatible with format strings from other languages such as C. As a result, you will find that some developers who already know format strings in other languages will use them in favor of the new style.



Python Raw Strings

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

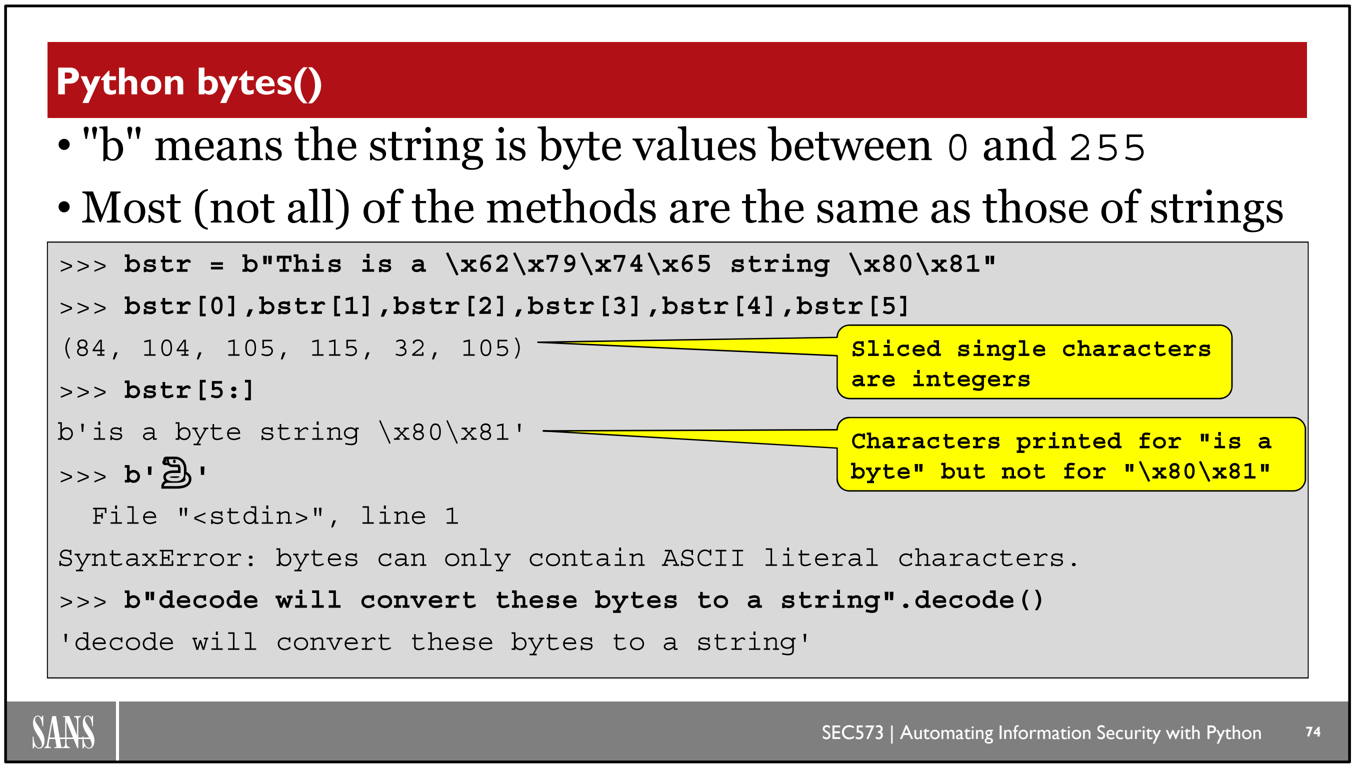
```
>>> print("This has tabs and  \t\t multiple\nlines")
This has tabs and             multiple
lines
>>> print(r"This has tabs and \t\t multiple\nlines")
This has tabs and \t\t multiple\nlines
>>> print("python \"stinks\"\b\b\b\b\b\b\b\b \"rock")
python  "rocks"
>>> print(r"python \"stinks\"\b\b\b\b\b\b\b\b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



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"

SANS SEC573 | Automating Information Security with Python 74

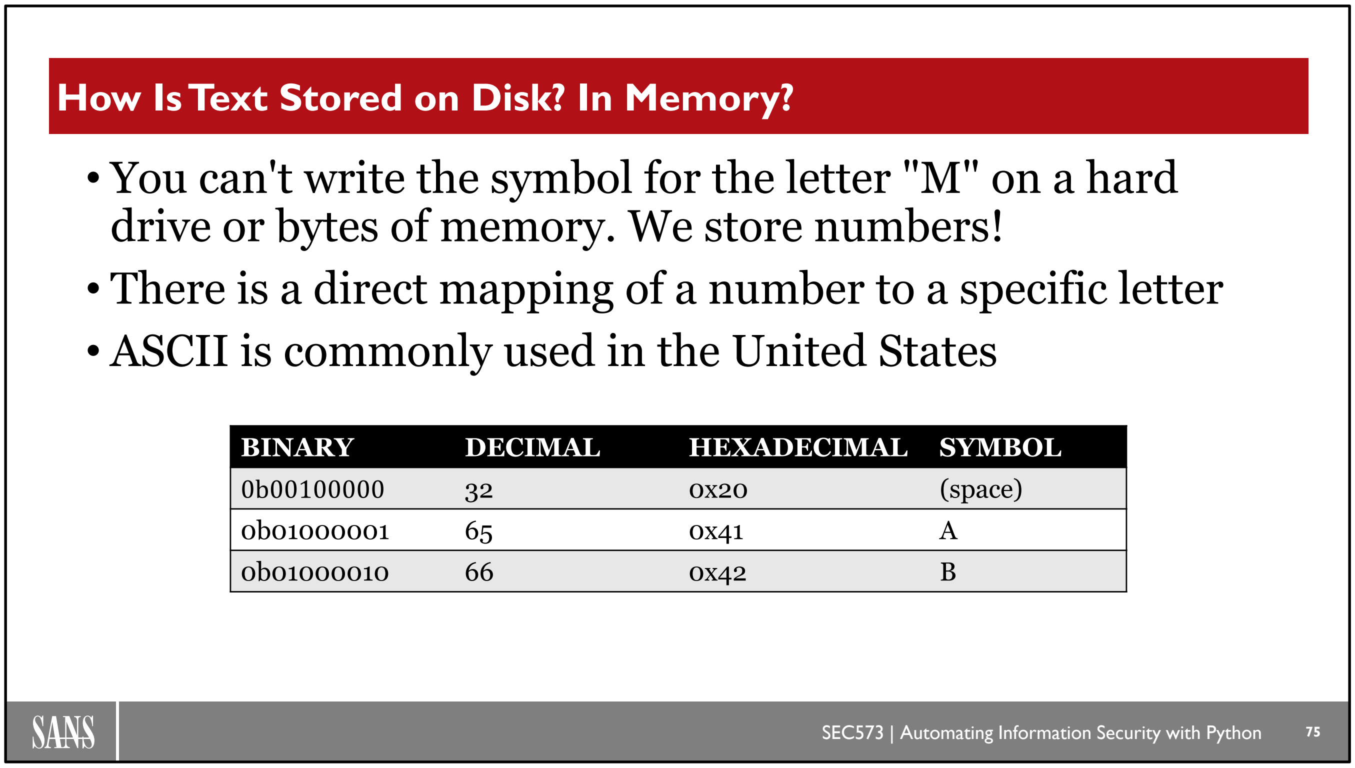
Another kind of string is the byte string. These are very similar to what strings were in Python 2. The values in the string are treated as individual bytes and characters are interpreted as ASCII values. Many of the methods that can be used to change strings can also be used to change byte strings.

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 bytes 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.

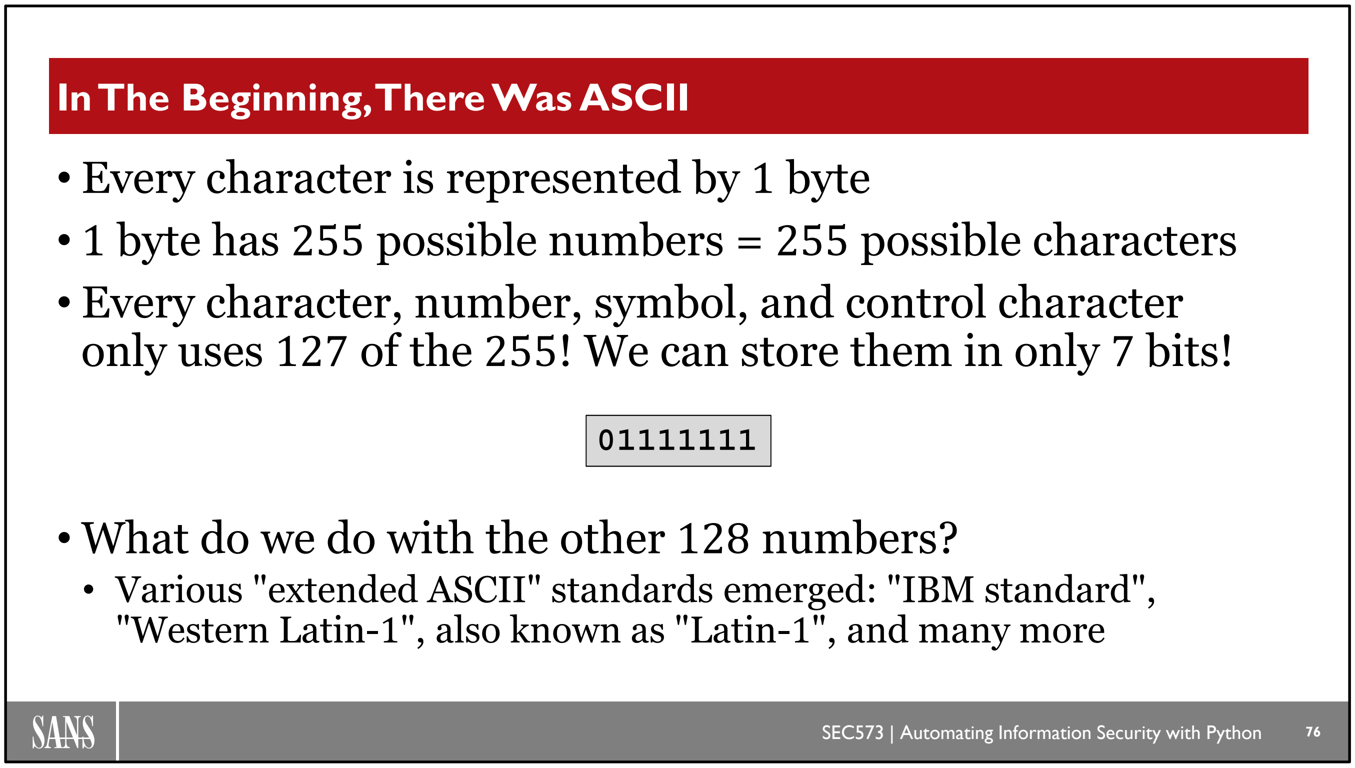


How Is Text Stored on Disk? In Memory?

- 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.



In The Beginning, There Was ASCII

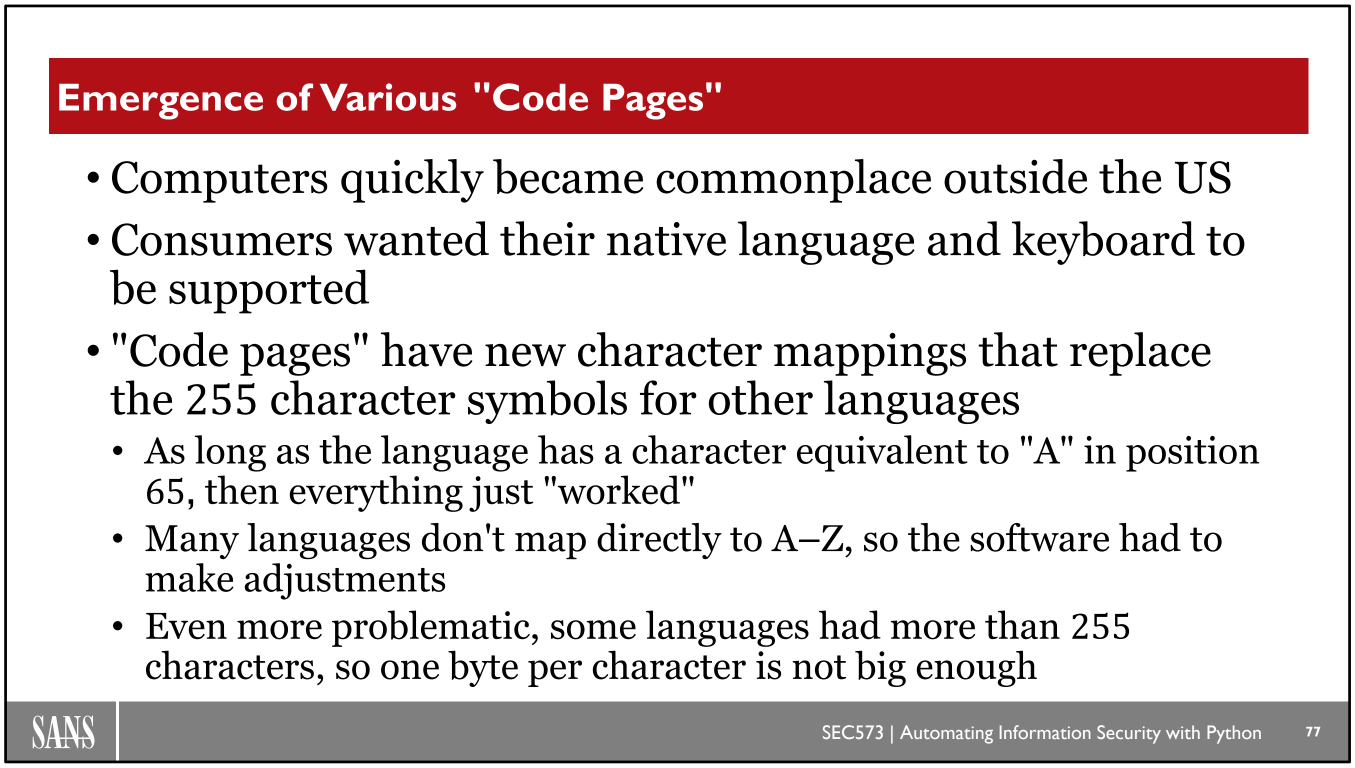
- Every character is represented by 1 byte
- 1 byte has 255 possible numbers = 255 possible characters
- Every character, number, symbol, and control character only uses 127 of the 255! We can store them in only 7 bits!

```
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

SANS | SEC573 | Automating Information Security with Python 76

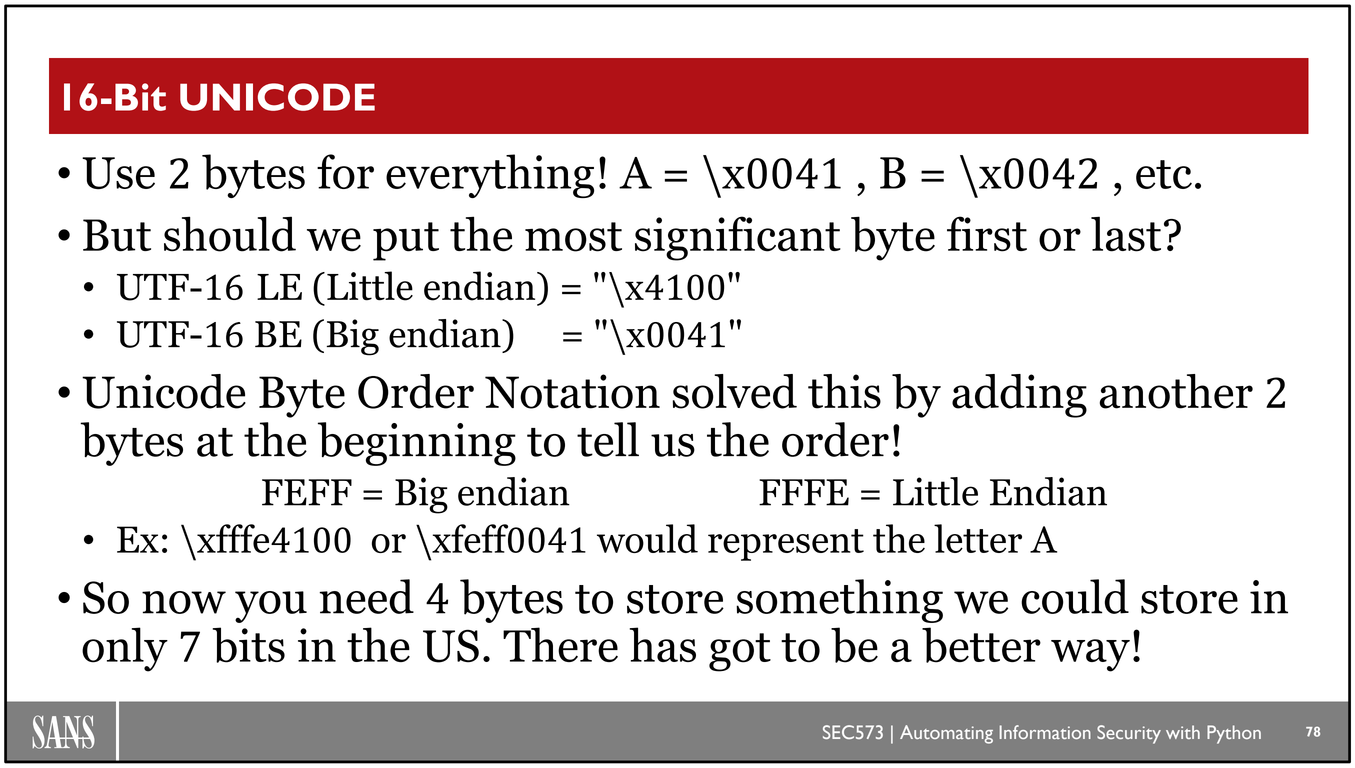
One of the first standards for these mappings of numbers to symbols was ASCII. Yes... I know. There was EBCDIC, but let's skip the history lesson. ASCII is the American Standard Code for Information Interchange. It represented all the letters and digits as numbers between 0 and 127. A single byte of 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".



Emergence of Various "Code Pages"

- Computers quickly became commonplace outside the US
- Consumers wanted their native language and keyboard to be supported
- "Code pages" have new character mappings that replace the 255 character symbols for other languages
  - As long as the language has a character equivalent to "A" in position 65, then everything just "worked"
  - Many languages don't map directly to A–Z, so the software had to make adjustments
  - Even more problematic, some languages had more than 255 characters, so one byte per character is not big enough

As computers became commonplace in locations outside the US, we needed different mappings for different characters. Customers wanted to use the pound sign for their currency instead of the dollar sign or put accents on vowels such as è or ê. New "code pages" were introduced that could be loaded into memory, replacing the ASCII table with the characters that matched that country. If the new code page characters could be mapped directly to a US equivalent character, then everything was fine. If not, then the program would have to be changed to adapt to the new language. Even more problematic was the fact that some languages contained more than 255 symbols, and they would not fit in a code page that only had one byte per character. So Unicode characters were born.



16-Bit UNICODE

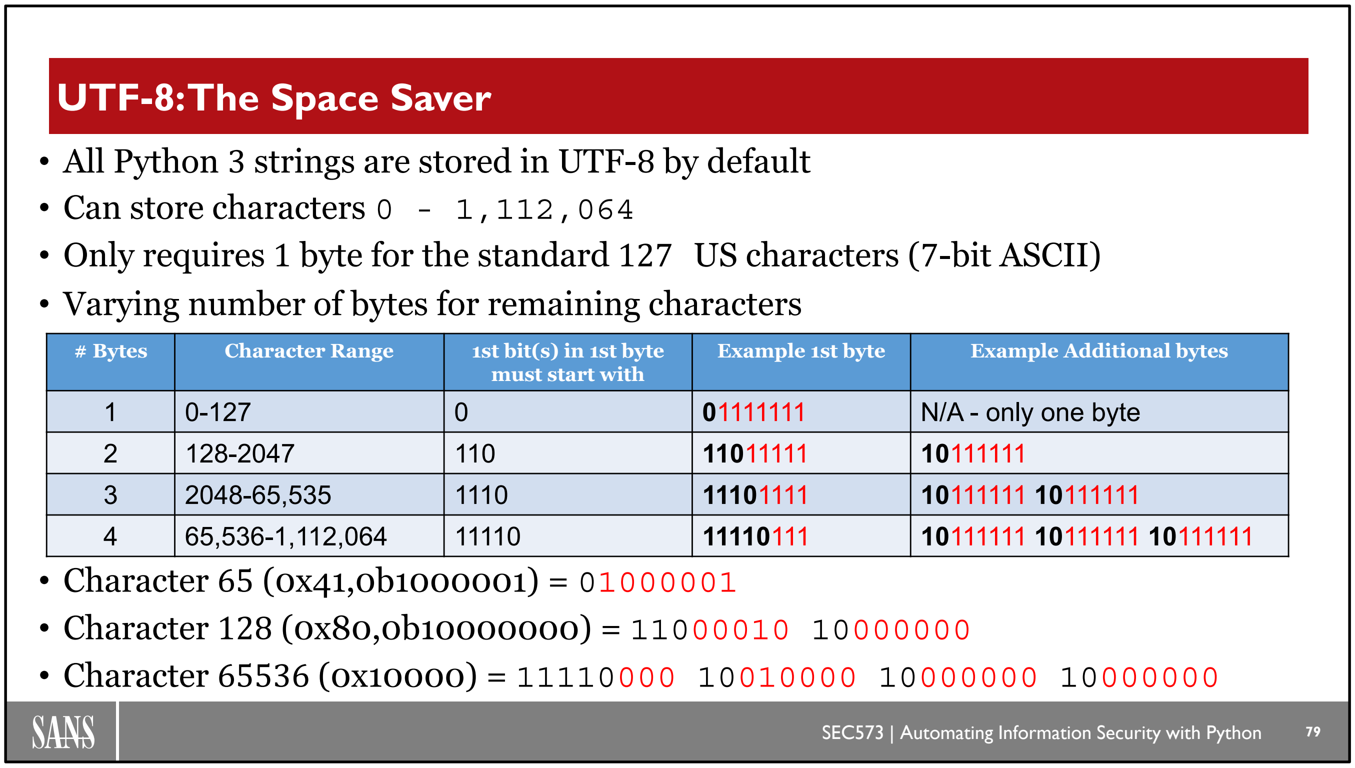
- Use 2 bytes for everything! A = \x0041 , B = \x0042 , etc.
- But should we put the most significant byte first or last?
  - UTF-16 LE (Little endian) = "\x4100"
  - UTF-16 BE (Big endian) = "\x0041"
- Unicode Byte Order Notation solved this by adding another 2 bytes at the beginning to tell us the 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!

SANS SEC573 | Automating Information Security with Python 78

Unicode character 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'
```



UTF-8:The Space Saver

- All Python 3 strings are stored in UTF-8 by default
- Can store characters `0 - 1,112,064`
- Only requires 1 byte for the standard 127 US characters (7-bit ASCII)
- Varying number of bytes for remaining characters

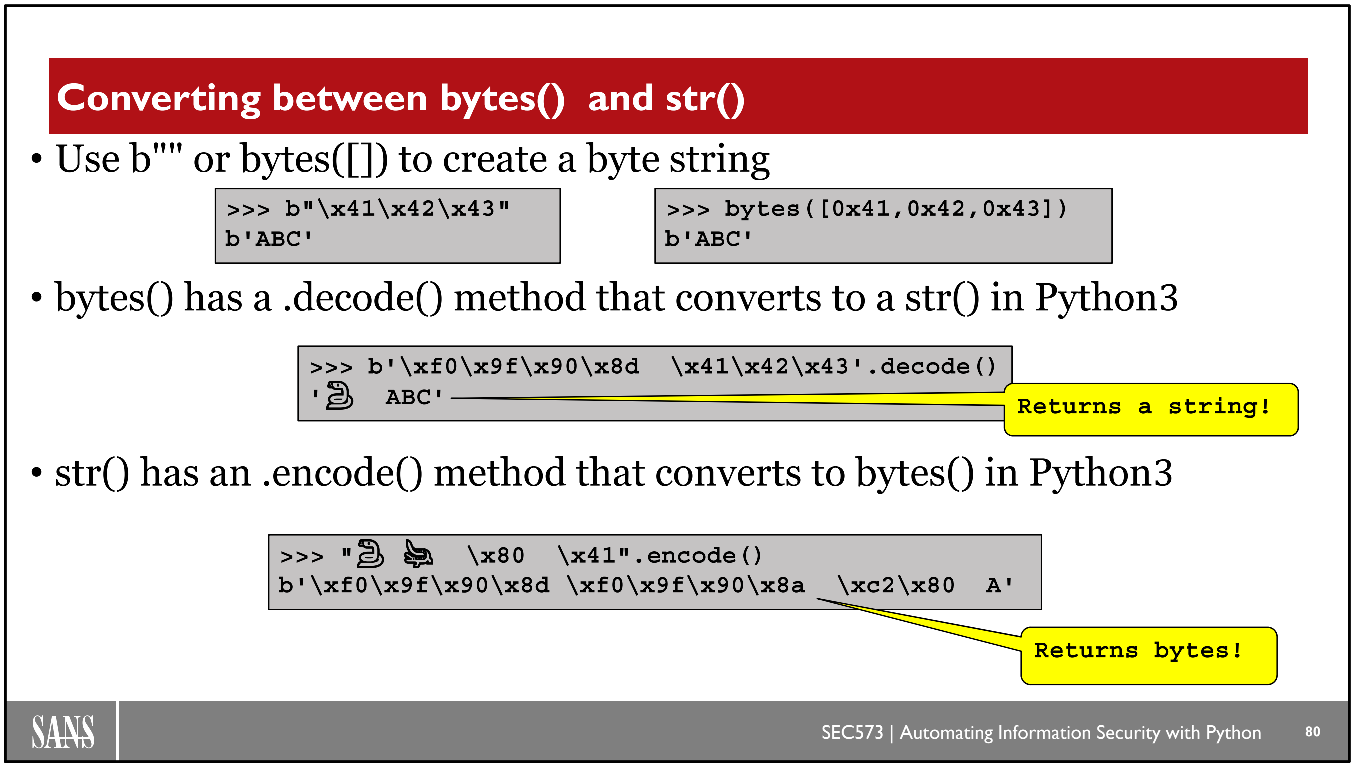
# Bytes	Character Range	1st bit(s) in 1st byte must start with	Example 1st byte	Example Additional bytes
1	0-127	0	**0**1111111	N/A - only one byte
2	128-2047	110	**110**11111	**10**111111
3	2048-65,535	1110	**1110**1111	**10**111111 **10**111111
4	65,536-1,112,064	11110	**11110**111	**10**111111 **10**111111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

- Character 65 (0x41,0b1000001) = `01000001`
- Character 128 (0x80,0b10000000) = `11000010 10000000`
- Character 65536 (0x10000) = `11110000 10010000 10000000 10000000`

SANS

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']
```



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!

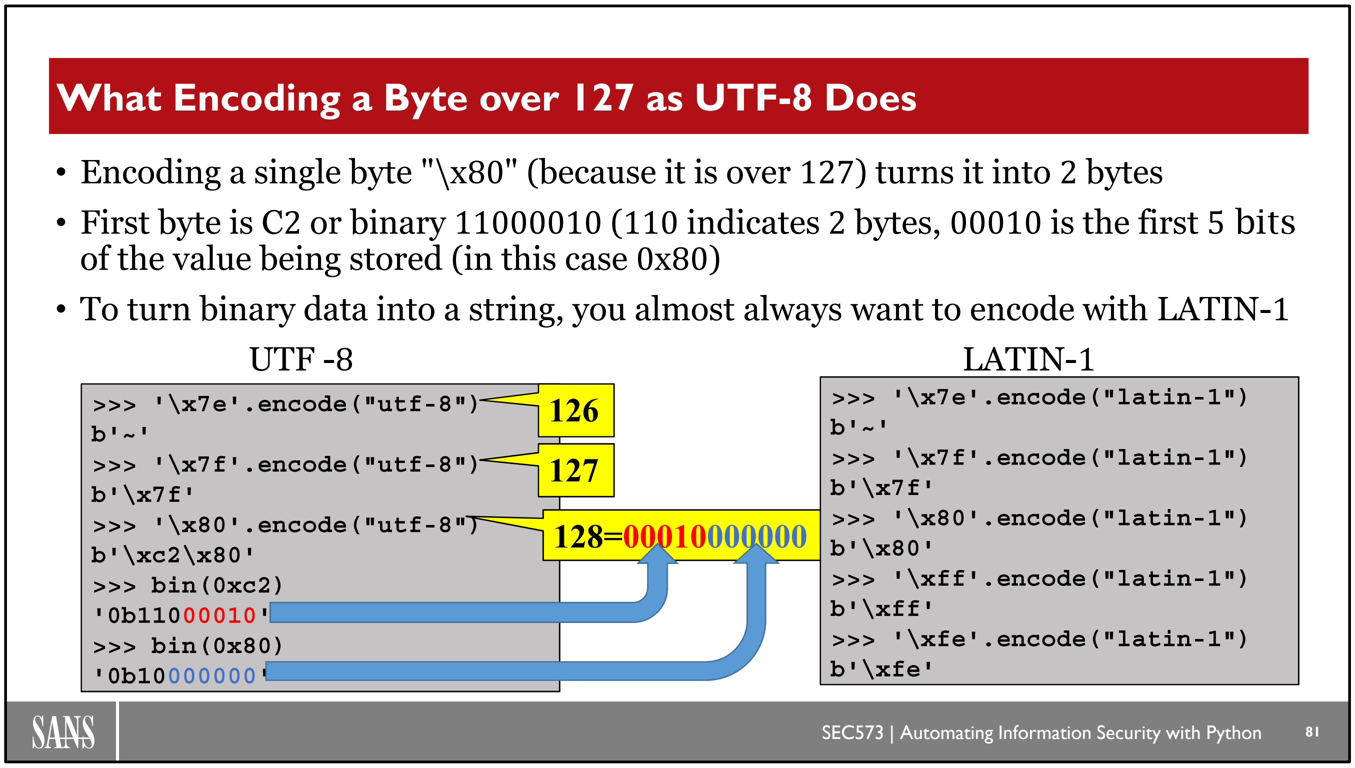
- str() has an .encode() method that converts to bytes() in Python3

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

Returns bytes!

SANS | SEC573 | Automating Information Security with Python 80

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.



What Encoding a Byte over 127 as UTF-8 Does

- Encoding a single byte "\x80" (because it is over 127) turns it into 2 bytes
- First byte is C2 or binary 11000010 (110 indicates 2 bytes, 00010 is the first 5 bits of the value being stored (in this case 0x80)
- To turn binary data into a string, you almost always want to encode with LATIN-1

UTF -8

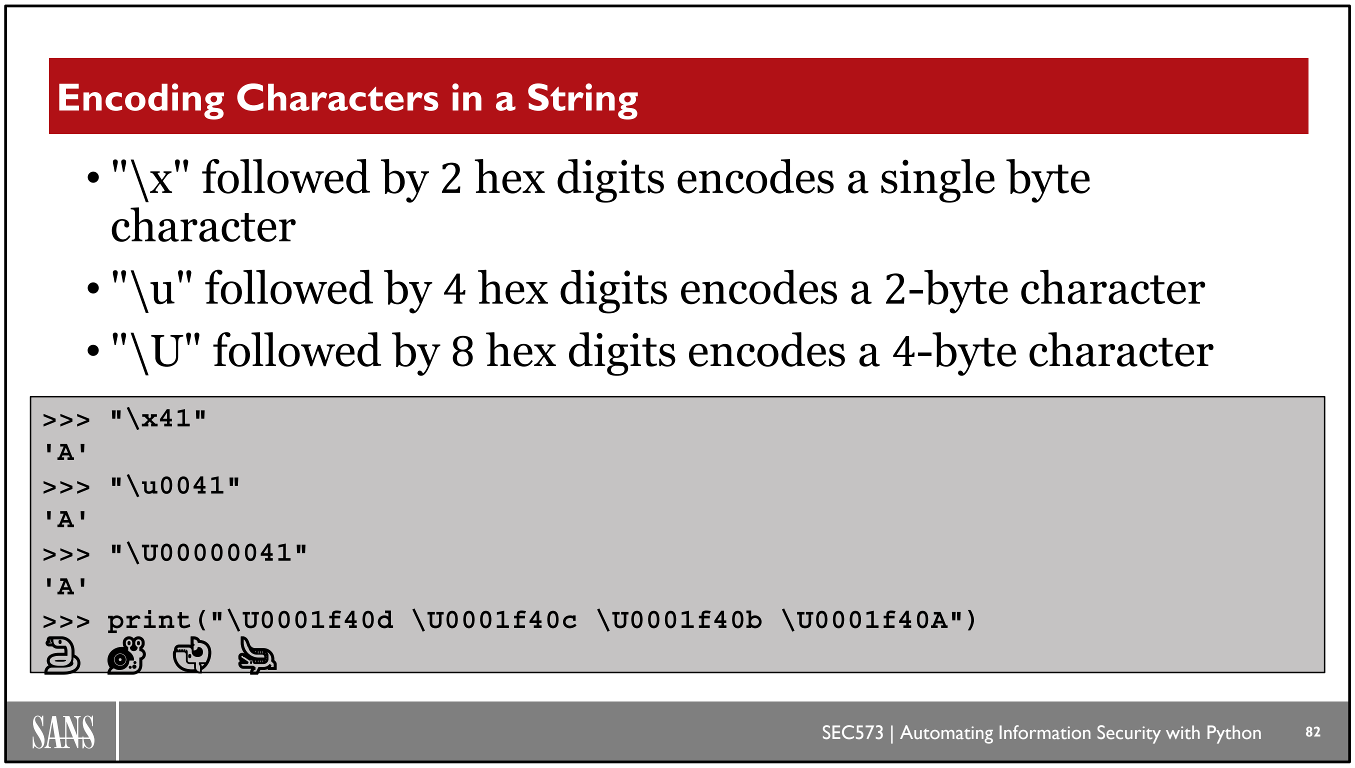
```
>>> '\x7e'.encode("utf-8")
b'~'
>>> '\x7f'.encode("utf-8")
b'\x7f'
>>> '\x80'.encode("utf-8")
b'\xc2\x80'
>>> bin(0xc2)
'0b11000010'
>>> bin(0x80)
'0b10000000'
```

LATIN-1

```
>>> '\x7e'.encode("latin-1")
b'~'
>>> '\x7f'.encode("latin-1")
b'\x7f'
>>> '\x80'.encode("latin-1")
b'\x80'
>>> '\xff'.encode("latin-1")
b'\xff'
>>> '\xfe'.encode("latin-1")
b'\xfe'
```

SANS SEC573 | Automating Information Security with Python 81

With UTF-8, if we are storing standard ASCII characters, then it only requires 1 byte of data. However, once we store a character with an ordinal value higher than 127, it requires 2 bytes of data. This can cause problems for us if we are dealing with binary data. If I want to store eight consecutive 1 bits and I store \xFF in UTF-8, it will create 2 bytes rather than just storing 1 byte. Furthermore, when I am reading a binary stream and treating it as UTF-8, anytime I read a byte that is larger than 128, the next byte (or possibly several bytes depending upon the number) must begin with a binary 10xxxxxx (continuation marker). If it doesn't, then it isn't a valid UTF-8 and Python will generate an exception. So when dealing with binary data, we often leave the data in byte strings. If it must be stored in a regular string, then use the "LATIN-1" encoder because it has a one-for-one mapping of characters between 0 and 255.



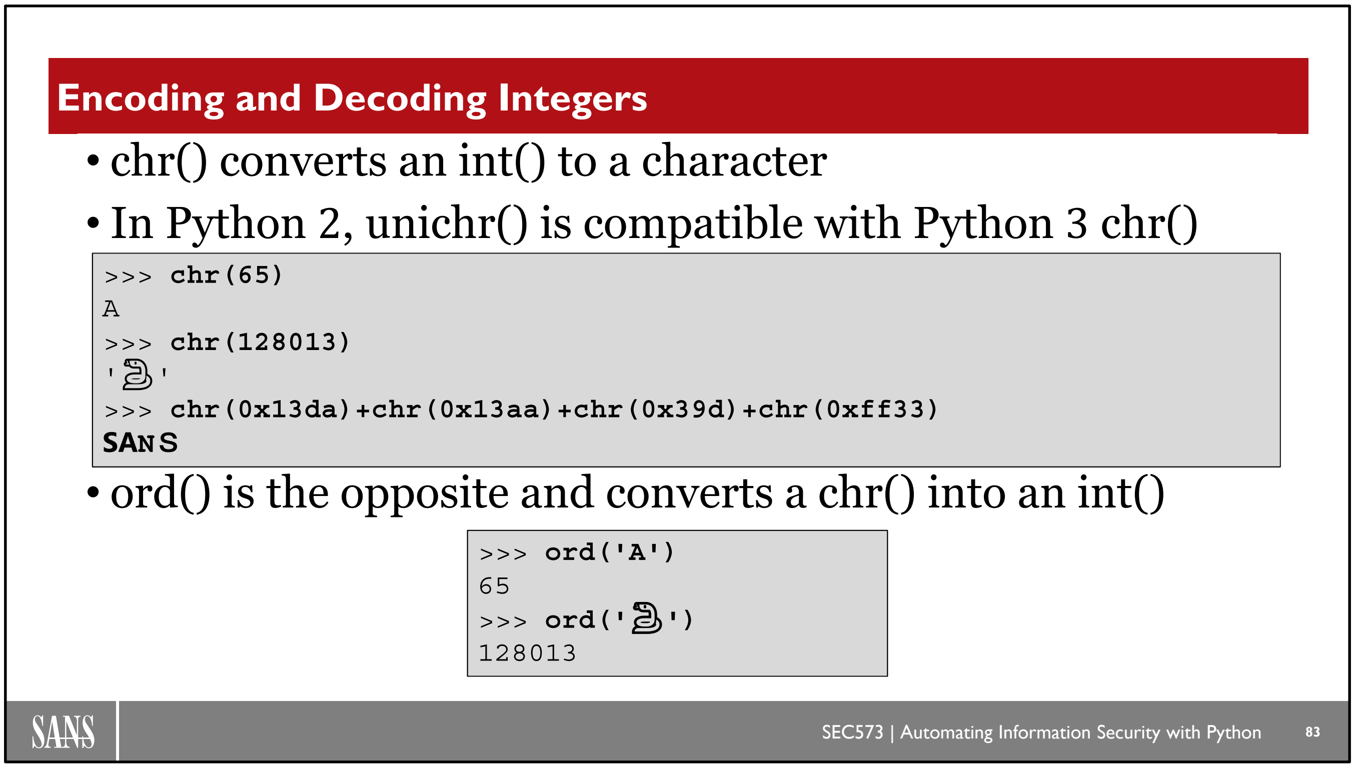
Encoding Characters in a String

- "\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")
🐍 🐌 🐋 🐊
```

SANS | SEC573 | Automating Information Security with Python 82

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.



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)
ᏚᎪΝＳ
```

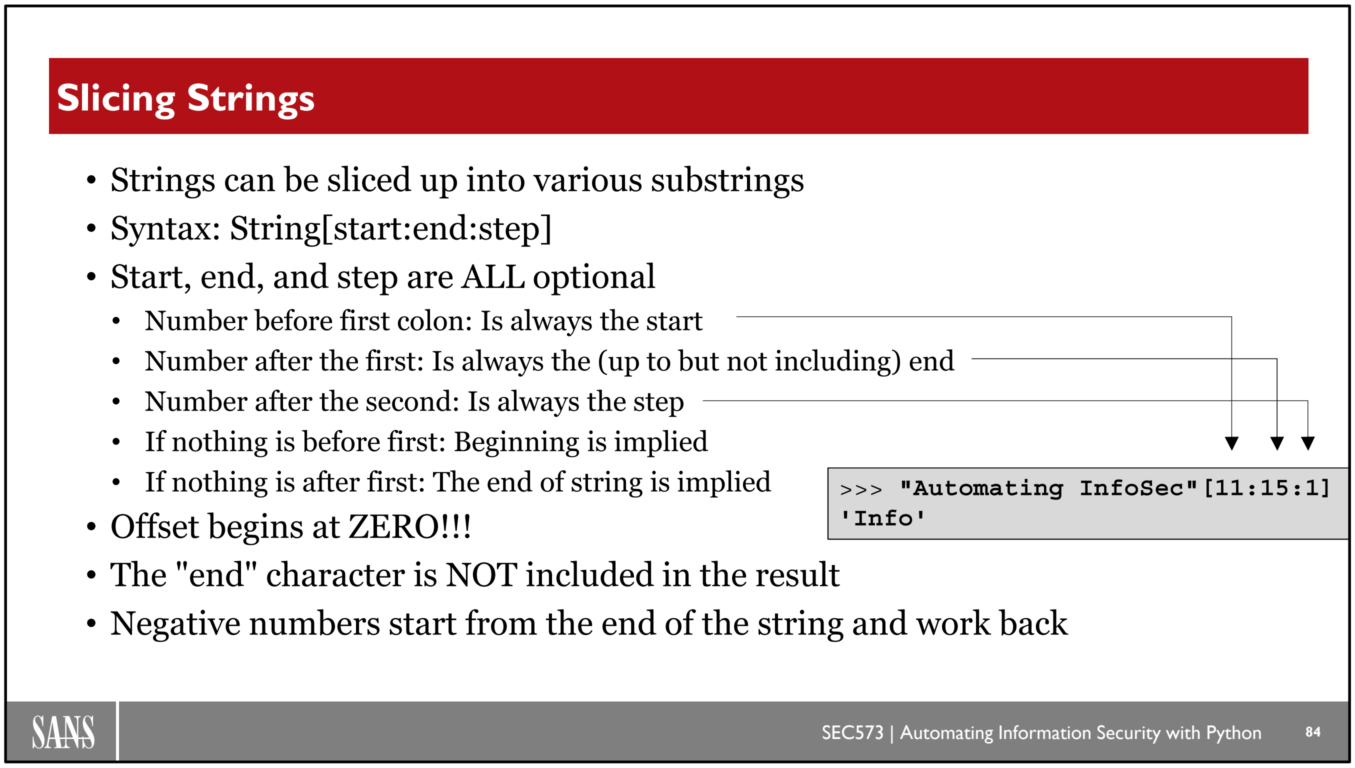
- ord() is the opposite and converts a chr() into an int()

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

SANS

SEC573 | Automating Information Security with Python 83

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.



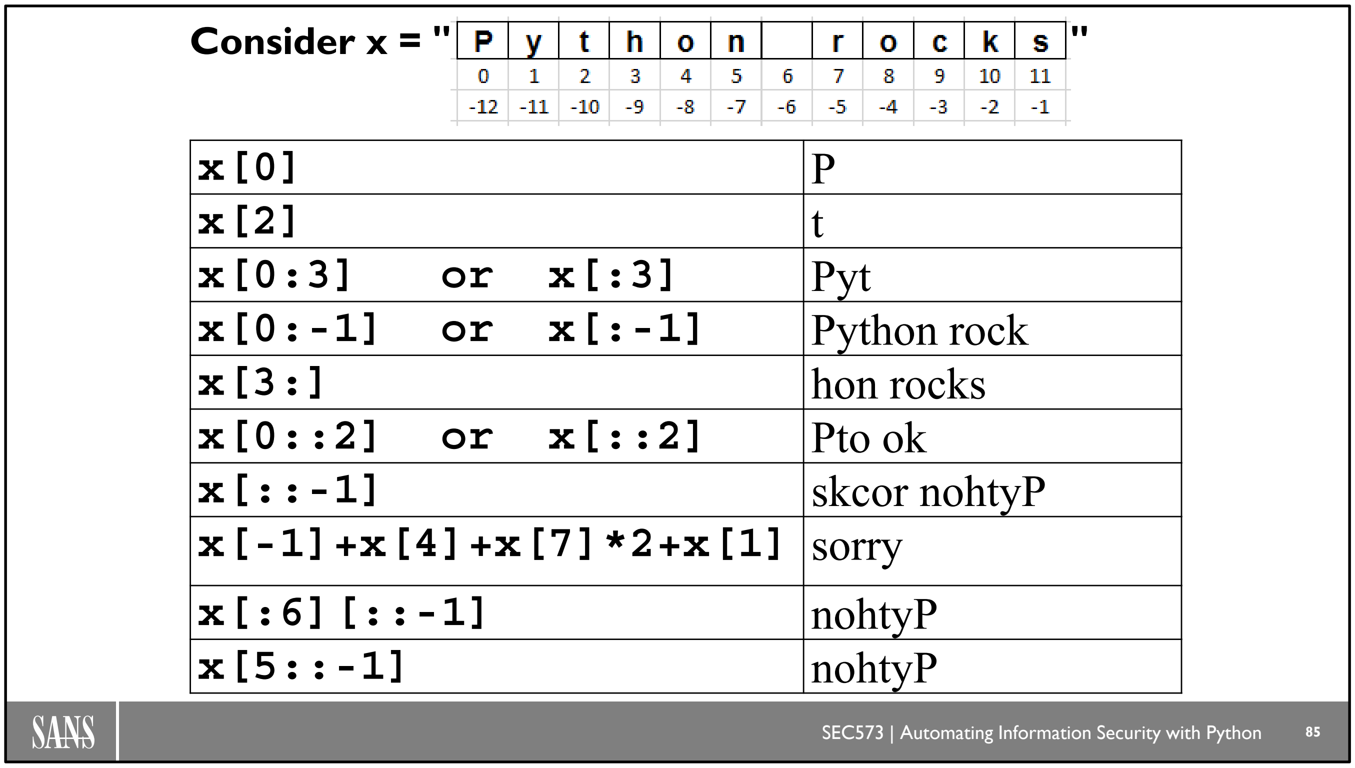
Slicing Strings

- 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
- 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

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

SANS SEC573 | Automating Information Security with Python 84

You can "slice" up strings based on character indexes. You do this by following a string with open and close 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.



Consider x = "

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]`	P
`x[2]`	t
`x[0:3]   or   x[:3]`	Pyt
`x[0:-1]   or   x[:-1]`	Python rock
`x[3:]`	hon rocks
`x[0::2]   or   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

SANS

SEC573 | Automating Information Security with Python 85

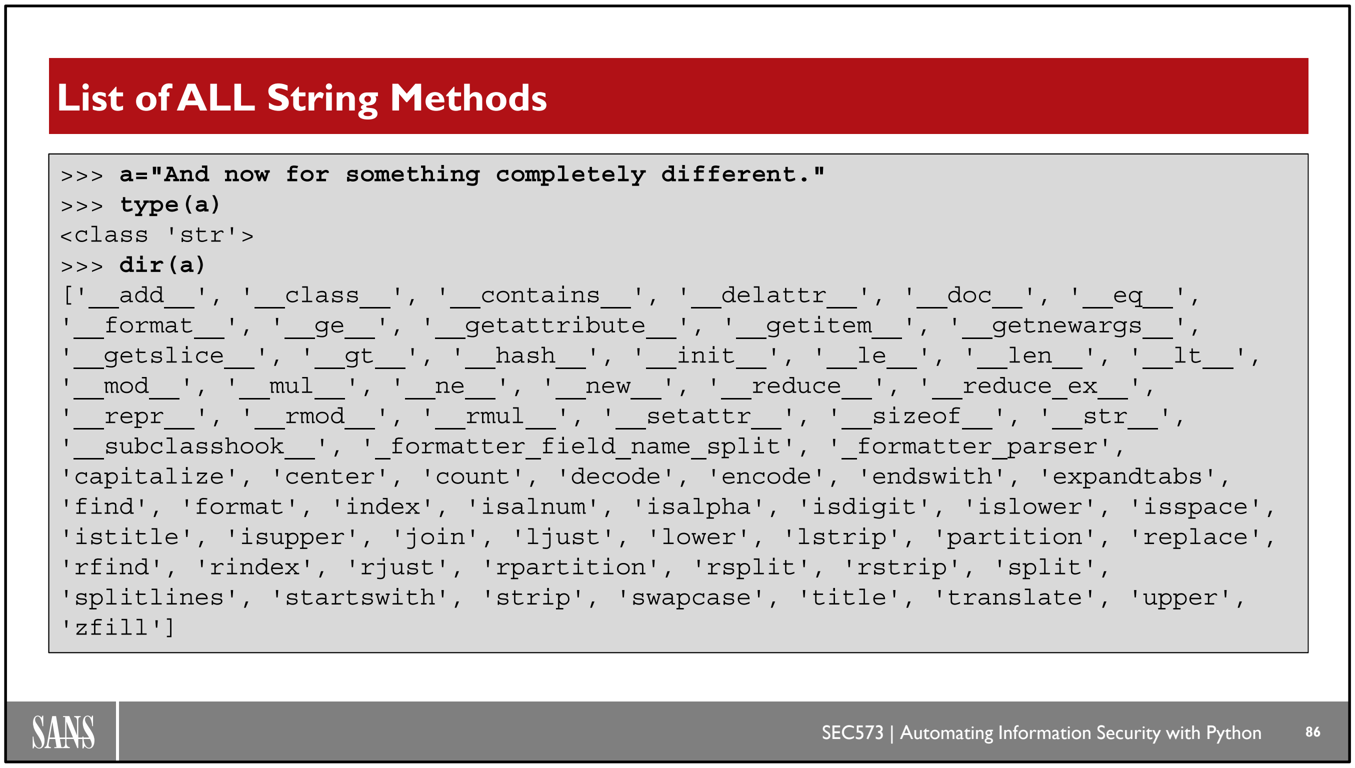
Here are some examples of string slicing. If you just provide an index such as x[0] or x[2], you will get the character at that offset. Remember that indexes begin at zero.

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'
```

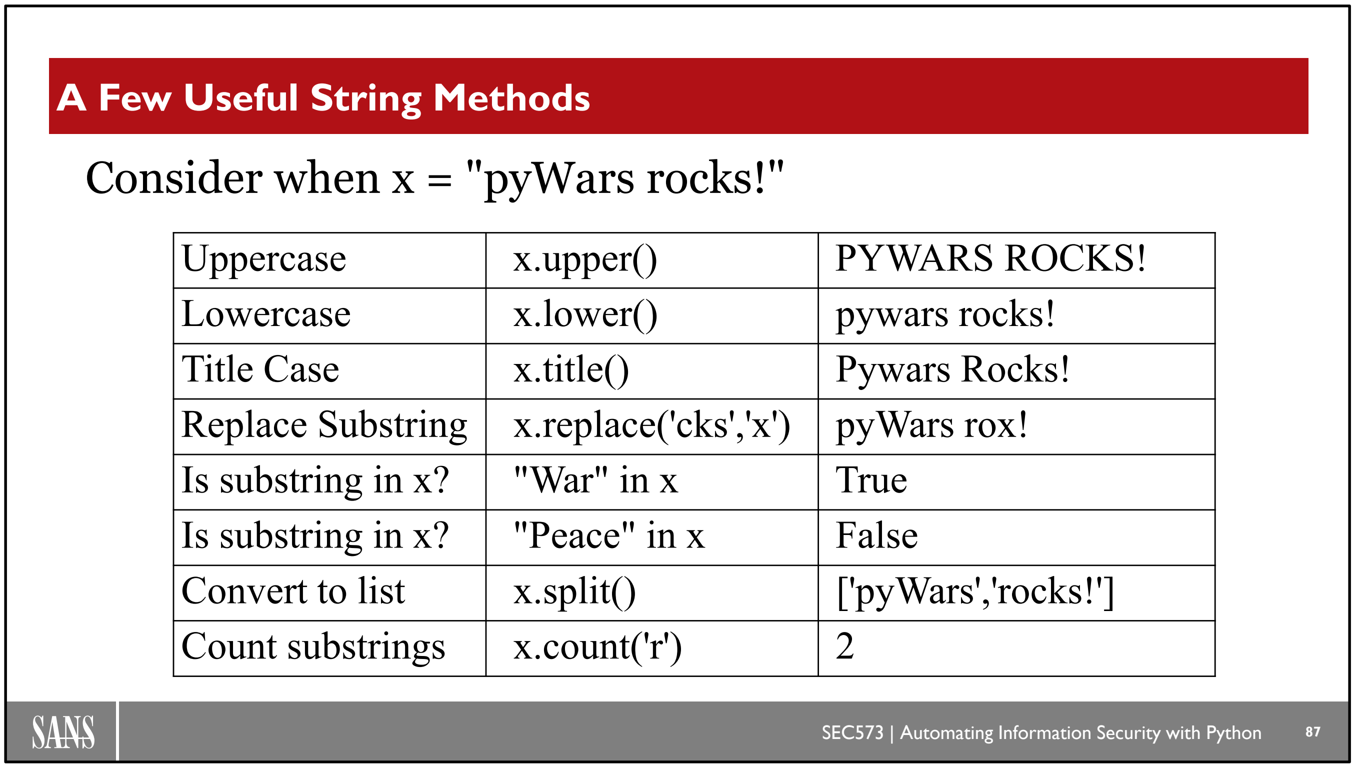


List of ALL String Methods

```
>>> 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 of 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.



A Few Useful String Methods

Consider when x = "pyWars rocks!"

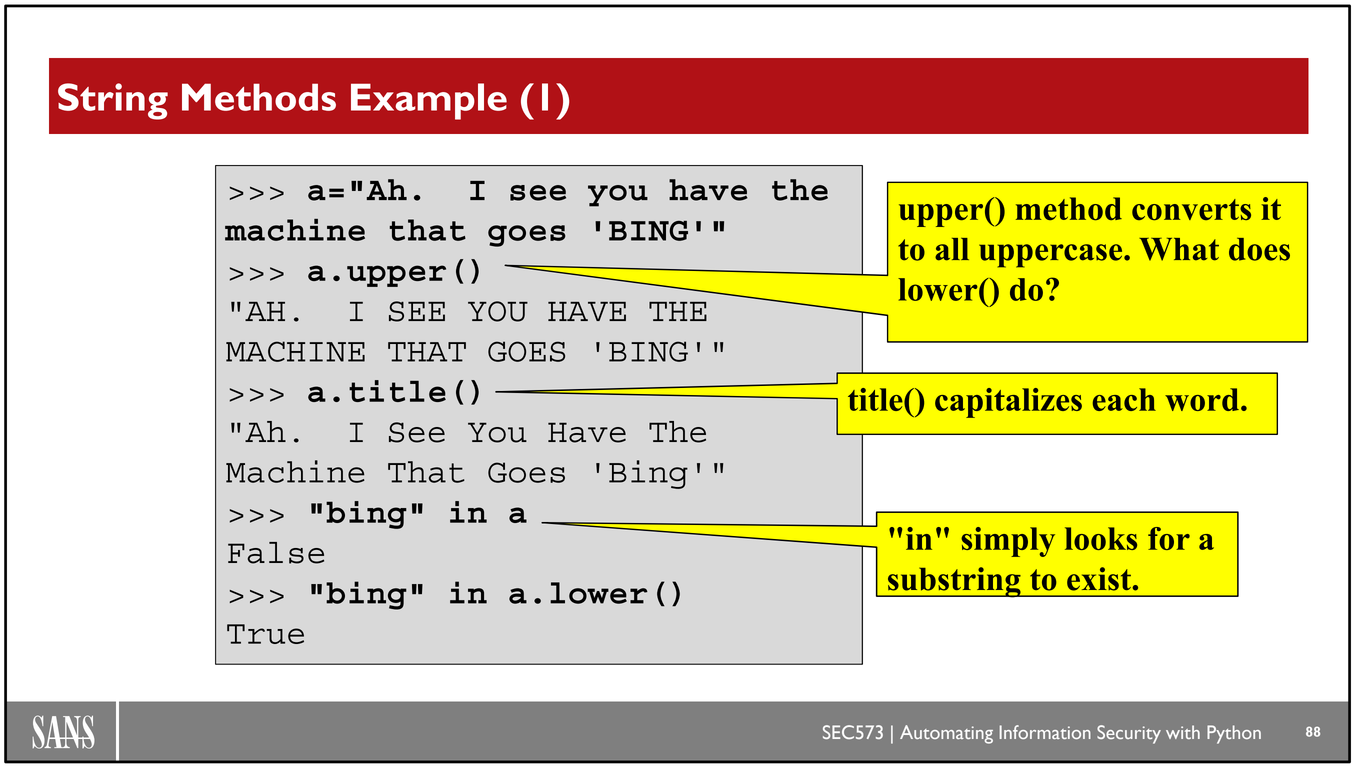
Uppercase	x.upper()	PYWARS ROCKS!
Lowercase	x.lower()	pywars rocks!
Title Case	x.title()	Pywars Rocks!
Replace Substring	x.replace('cks','x')	pyWars rox!
Is substring in x?	"War" in x	True
Is substring in x?	"Peace" in x	False
Convert to list	x.split()	['pyWars','rocks!']
Count substrings	x.count('r')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

SANS

SEC573 | Automating Information Security with Python 87

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
```



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.

SEC573 | Automating Information Security with Python 88

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'"
```

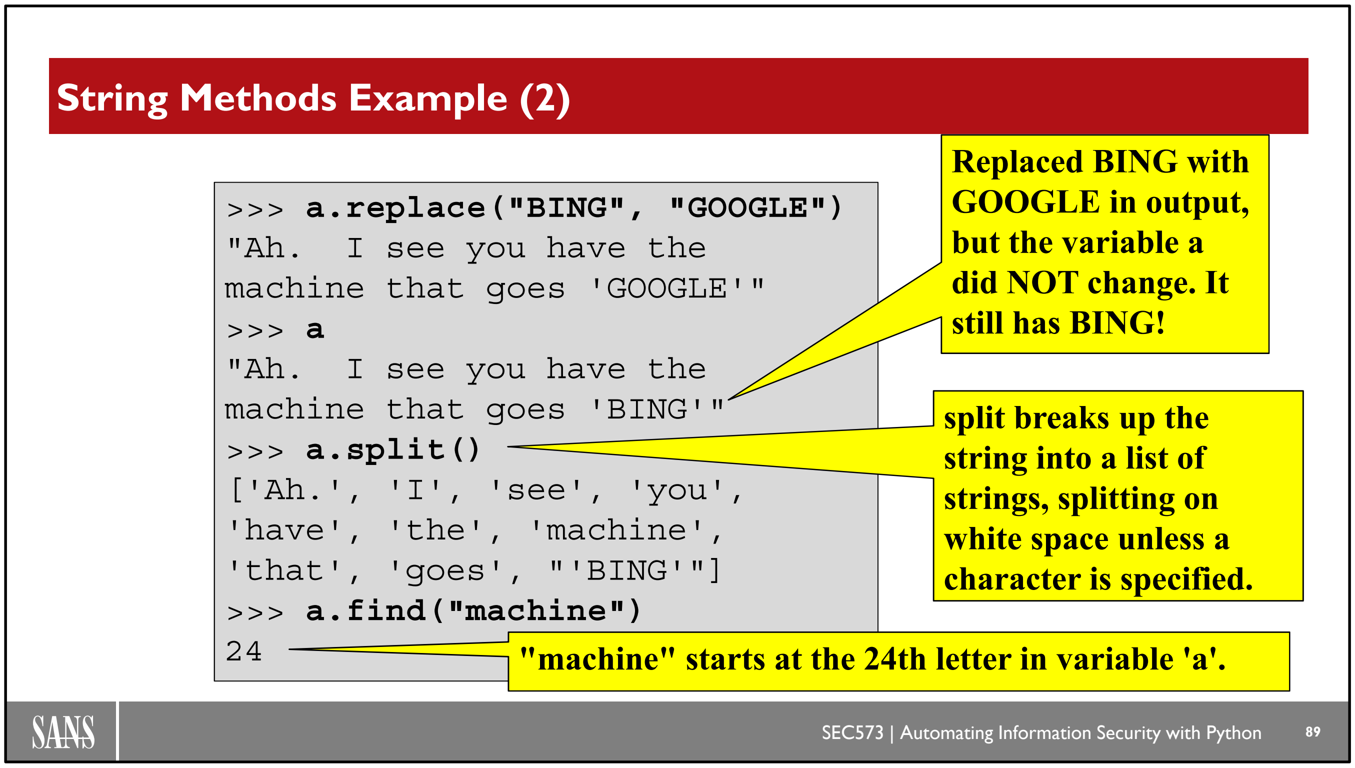
We can use the keyword "in" to test for the presence of a substring in another string. To see if the string "bing" exists inside variable a, we could do this:

```
>>> "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.



String Methods Example (2)

```
>>> a.replace("BING", "GOOGLE")
"Ah.  I see you have the
machine that goes 'GOOGLE'"
>>> a
"Ah.  I see you have the
machine that goes 'BING'"
>>> a.split()
['Ah.', 'I', 'see', 'you',
'have', 'the', 'machine',
'that', 'goes', "'BING'"]
>>> a.find("machine")
24
```

Replaced BING with GOOGLE in output, but the variable a did NOT change. It still has BING!

split breaks up the string into a list of strings, splitting on white space unless a character is specified.

"machine" starts at the 24th letter in variable 'a'.

SANS

SEC573 | Automating Information Security with Python 89

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'?

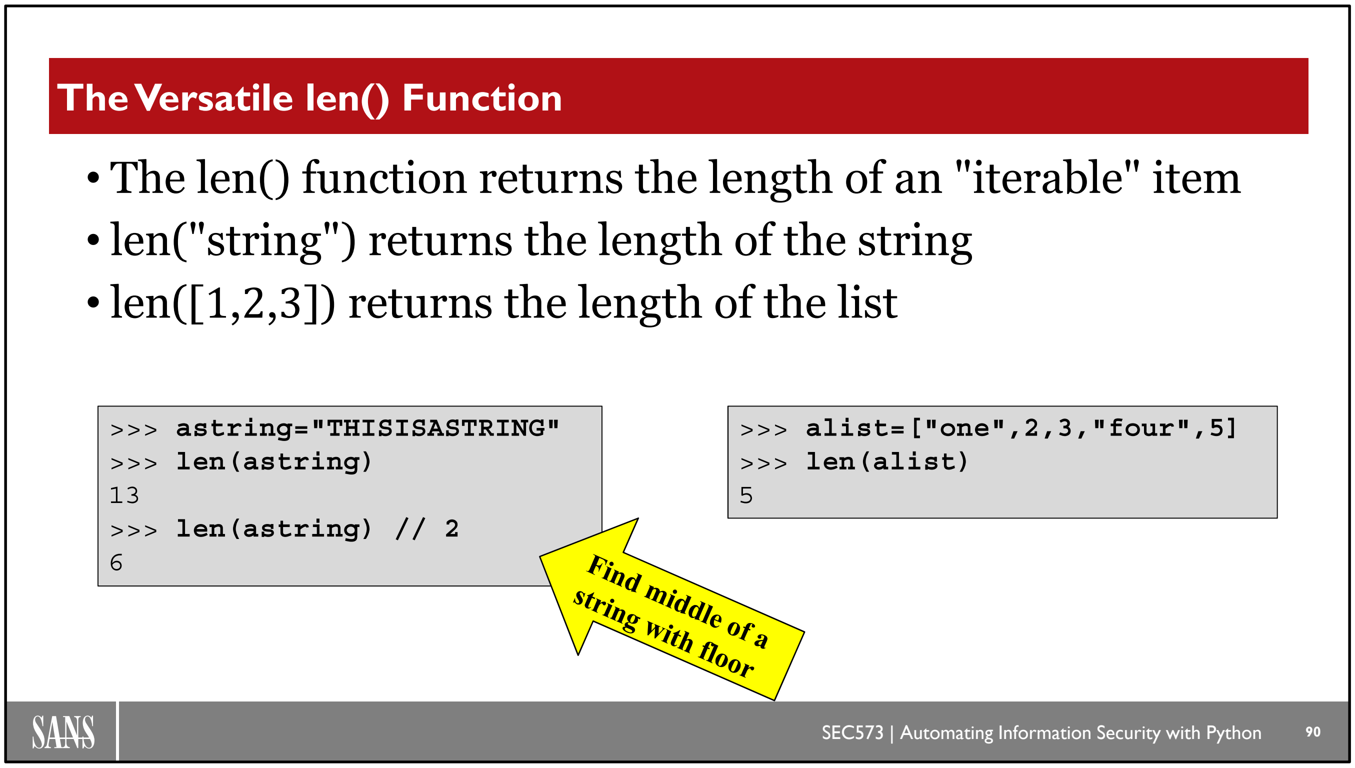
```
>>> a
"Ah.  I see you have the machine that goes 'BING'"
```
Notice, it didn't change 'a'.

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
```



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

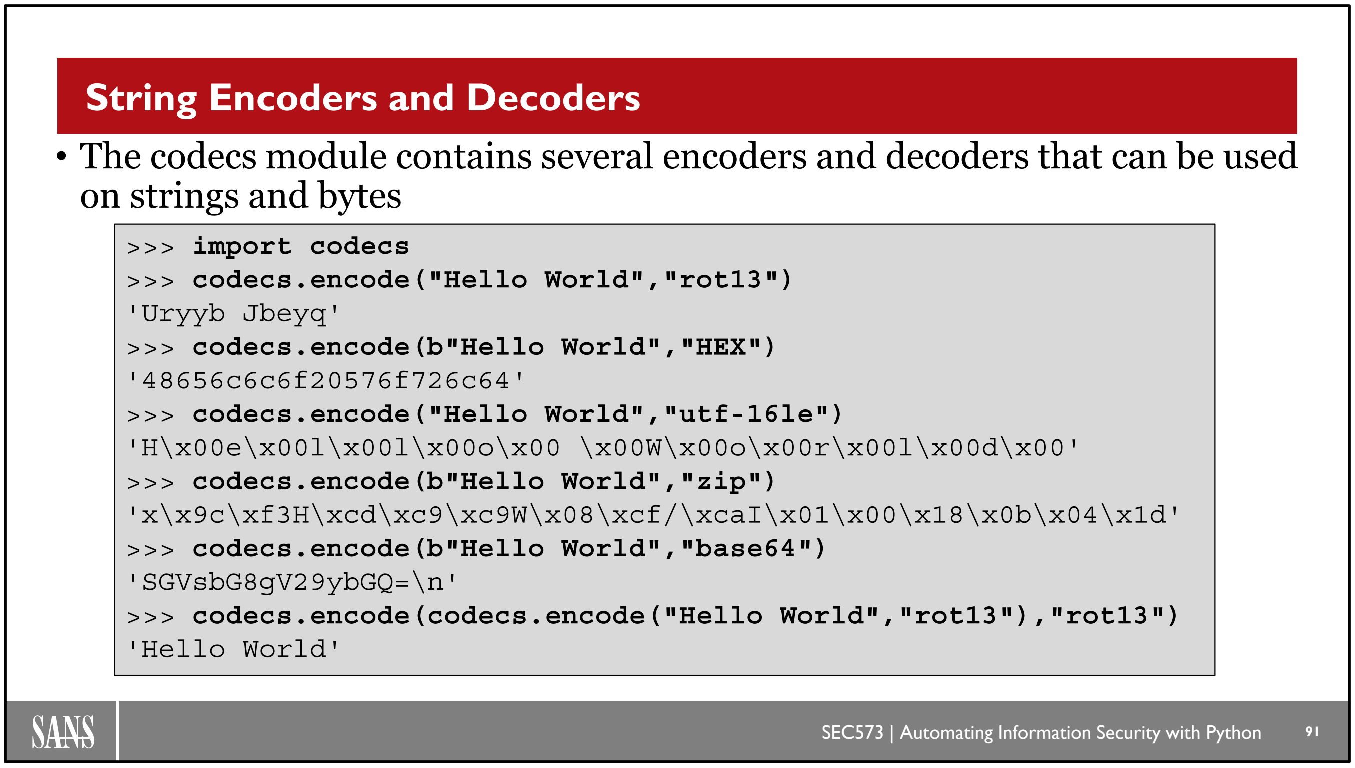
```
>>> astring="THISISASTRING"
>>> len(astring)
13
>>> len(astring) // 2
6
```

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

Find middle of a string with floor

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.



String Encoders and Decoders

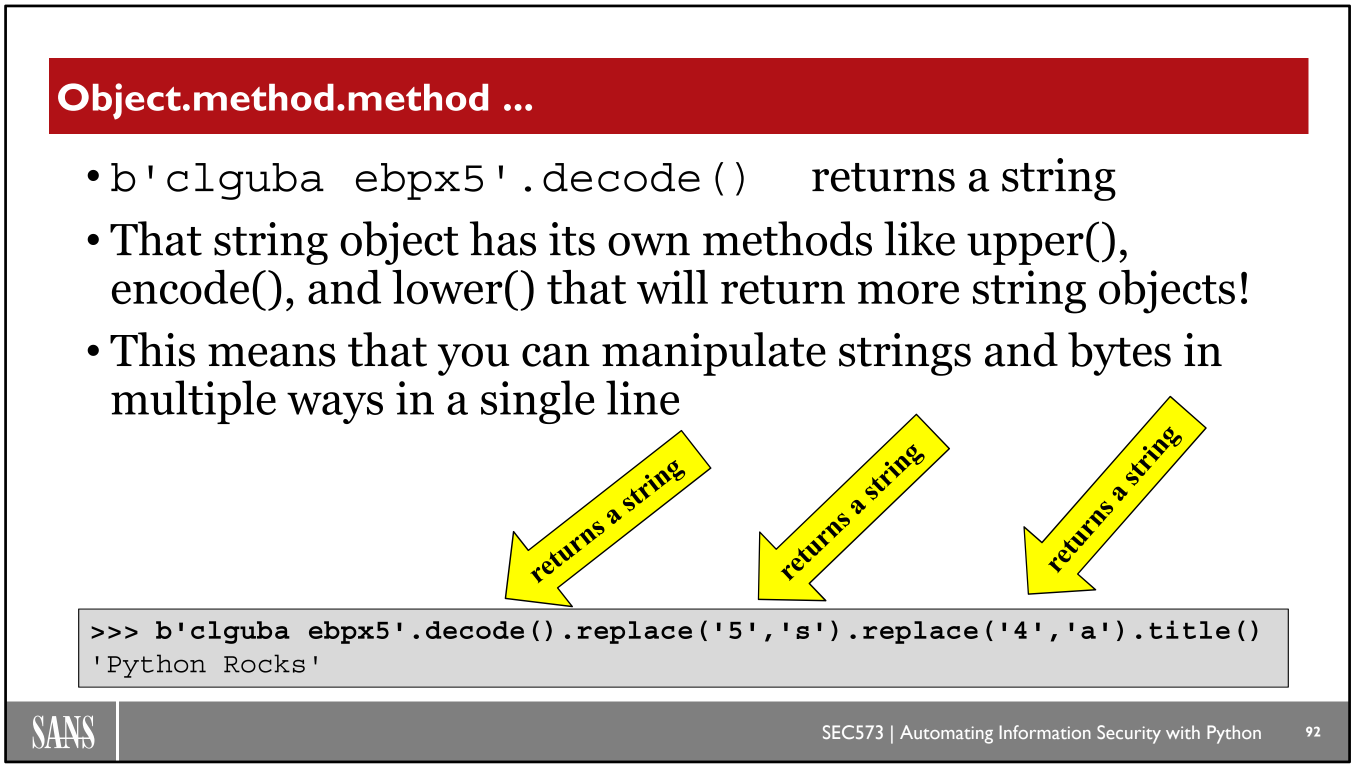
- The codecs module contains several encoders and decoders that can be used on strings and bytes

```
>>> 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.



Object.method.method ...

- `b'clguba ebpx5'.decode()` 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

returns a string
returns a string
returns a string

```
>>> b'clguba ebpx5'.decode().replace('5','s').replace('4','a').title()
'Python Rocks'
```

SANS | SEC573 | Automating Information Security with Python 92

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")
```

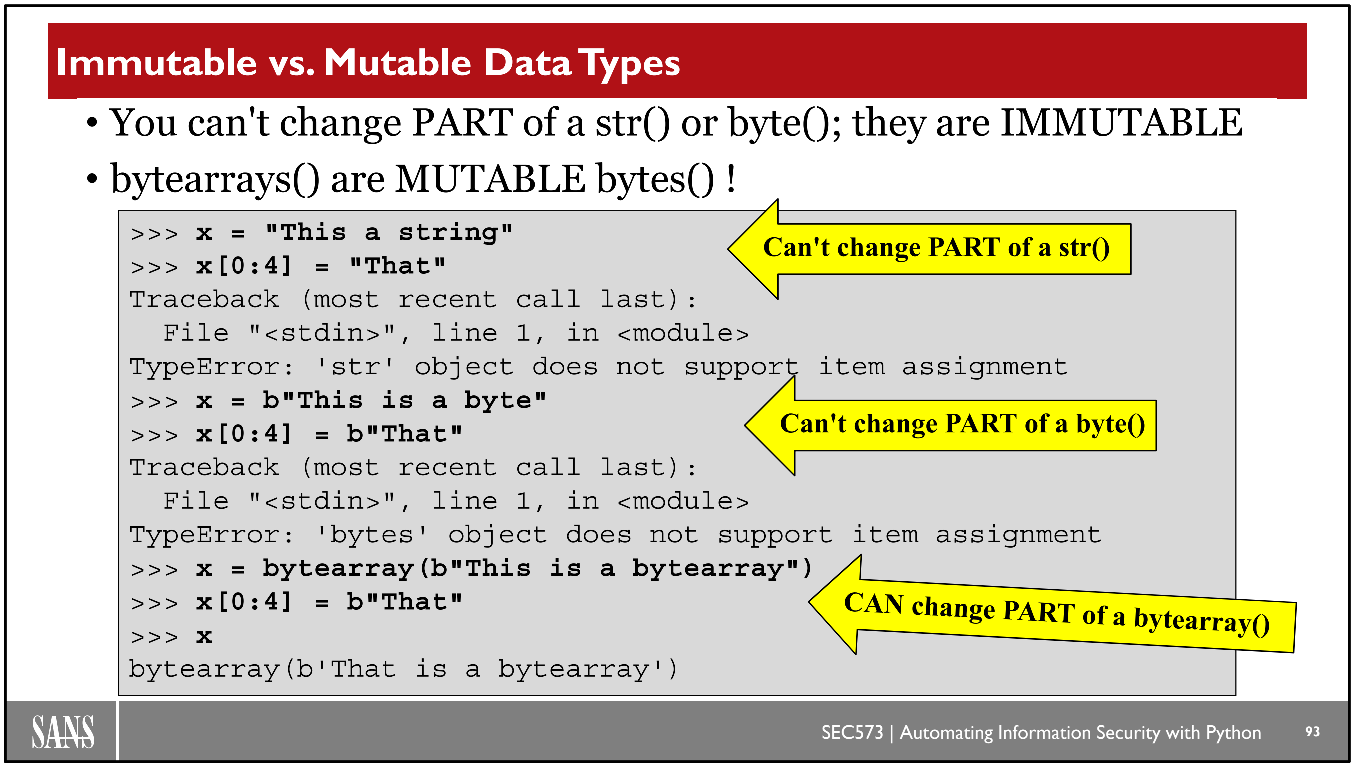
Or you could put it all on one line:

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

Or consider this example:

```
>>> b'clguba ebpx5'.decode().replace('5','s').replace('4','a').title()
'Python Rocks'
```

This example starts with encoded bytes, decodes it to a string, does some character replacements, and prints it as a title, resulting in the string "Python Rocks".



Immutable vs. Mutable Data Types

- You can't change PART of a str() or byte(); they are IMMUTABLE
- bytearrays() are MUTABLE bytes() !

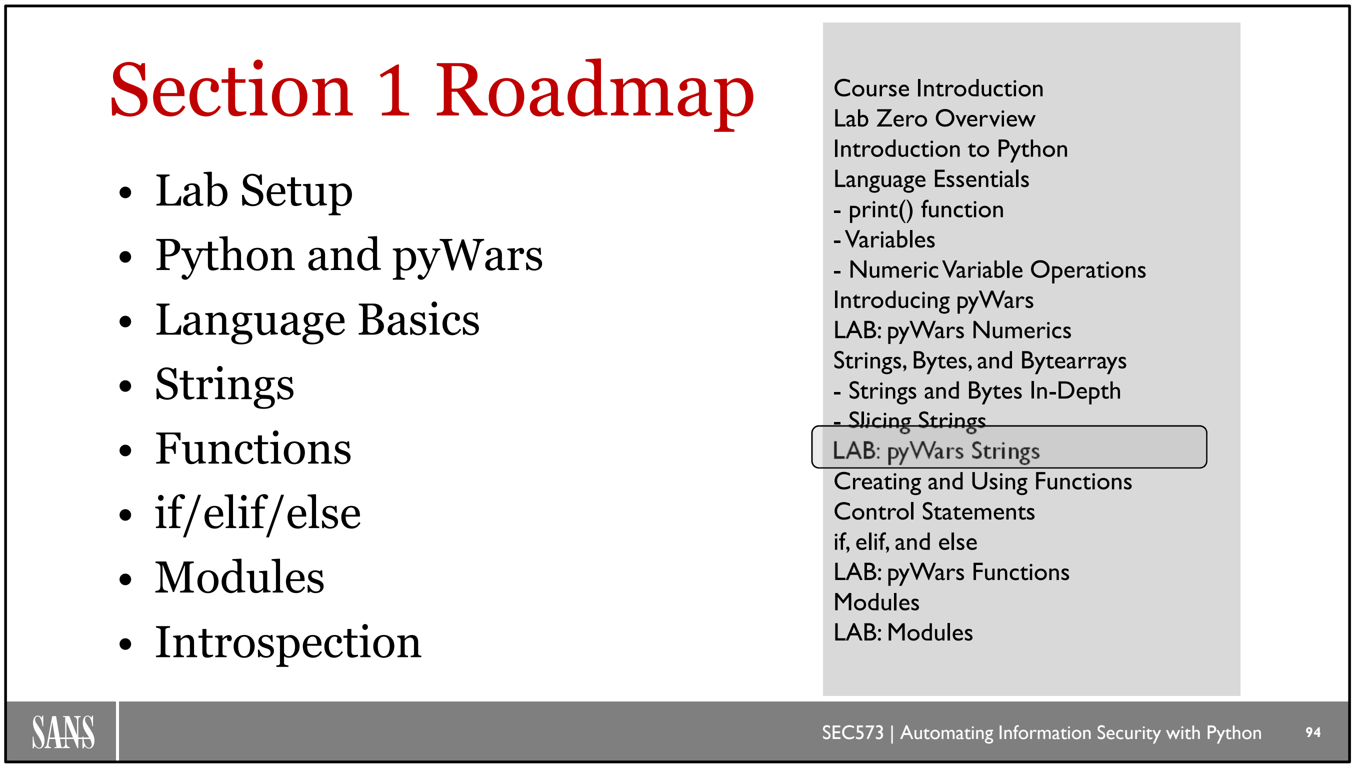
```
>>> x = "This a string"
>>> x[0:4] = "That"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> x = b"This is a byte"
>>> x[0:4] = b"That"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: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

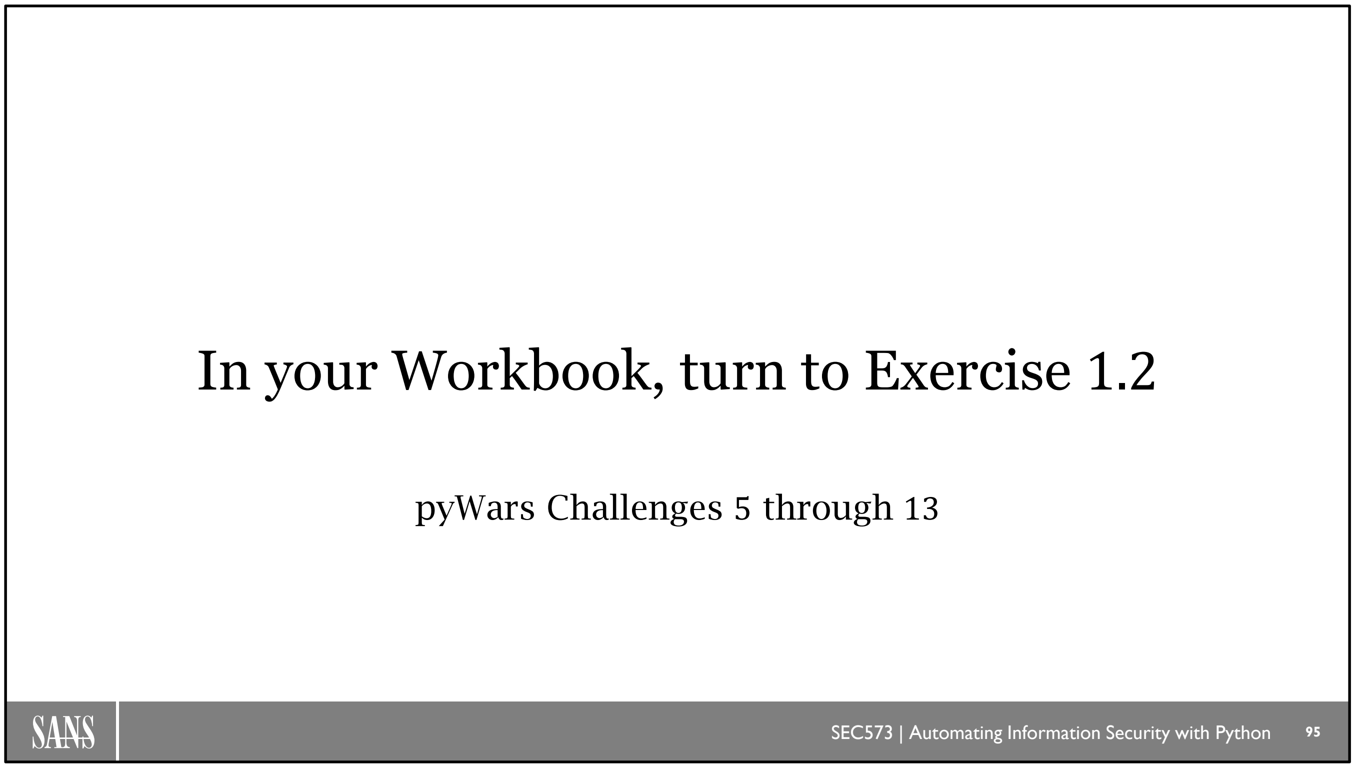


Section 1 Roadmap

- Lab Setup
- Python and pyWars
- Language Basics
- Strings
- Functions
- if/elif/else
- Modules
- Introspection

SANS

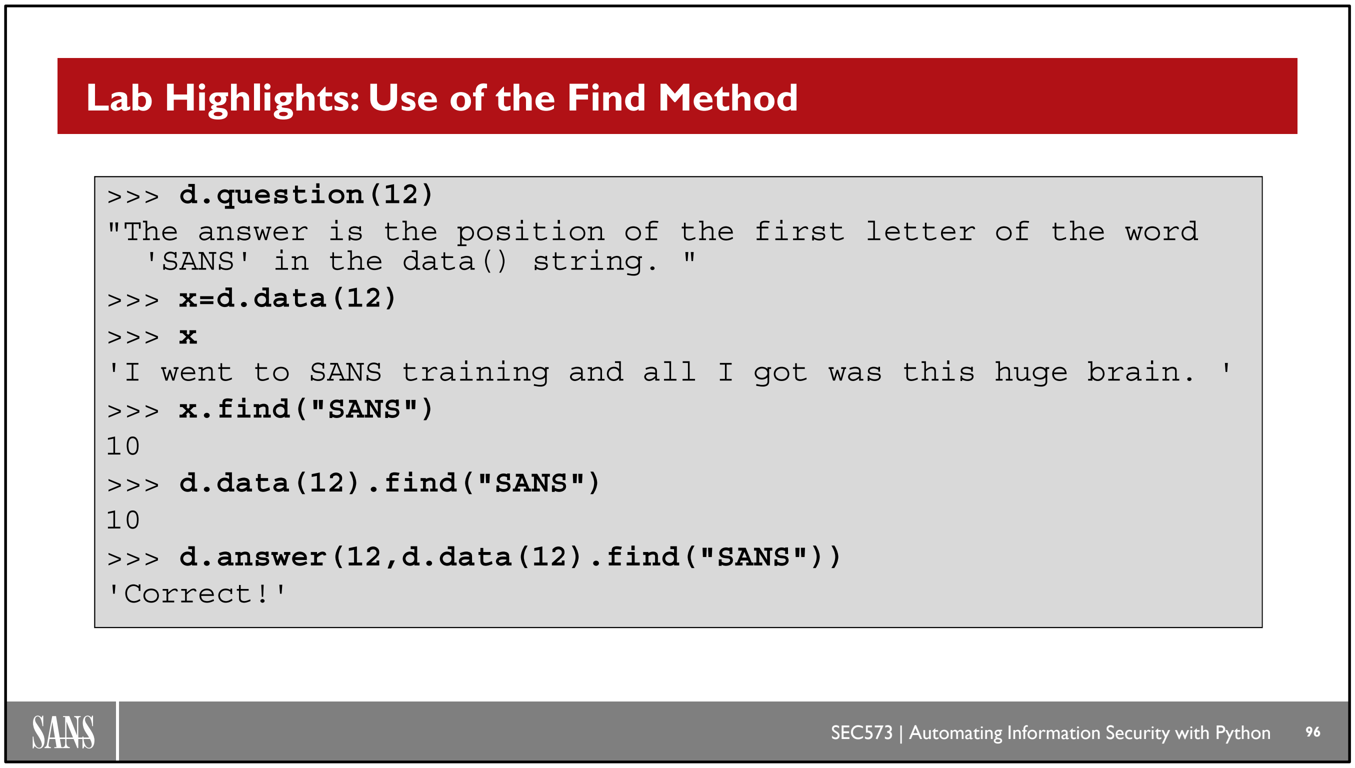
This is a Roadmap slide.



In your Workbook, turn to Exercise 1.2

pyWars Challenges 5 through 13

Please complete the exercise in your Workbook.

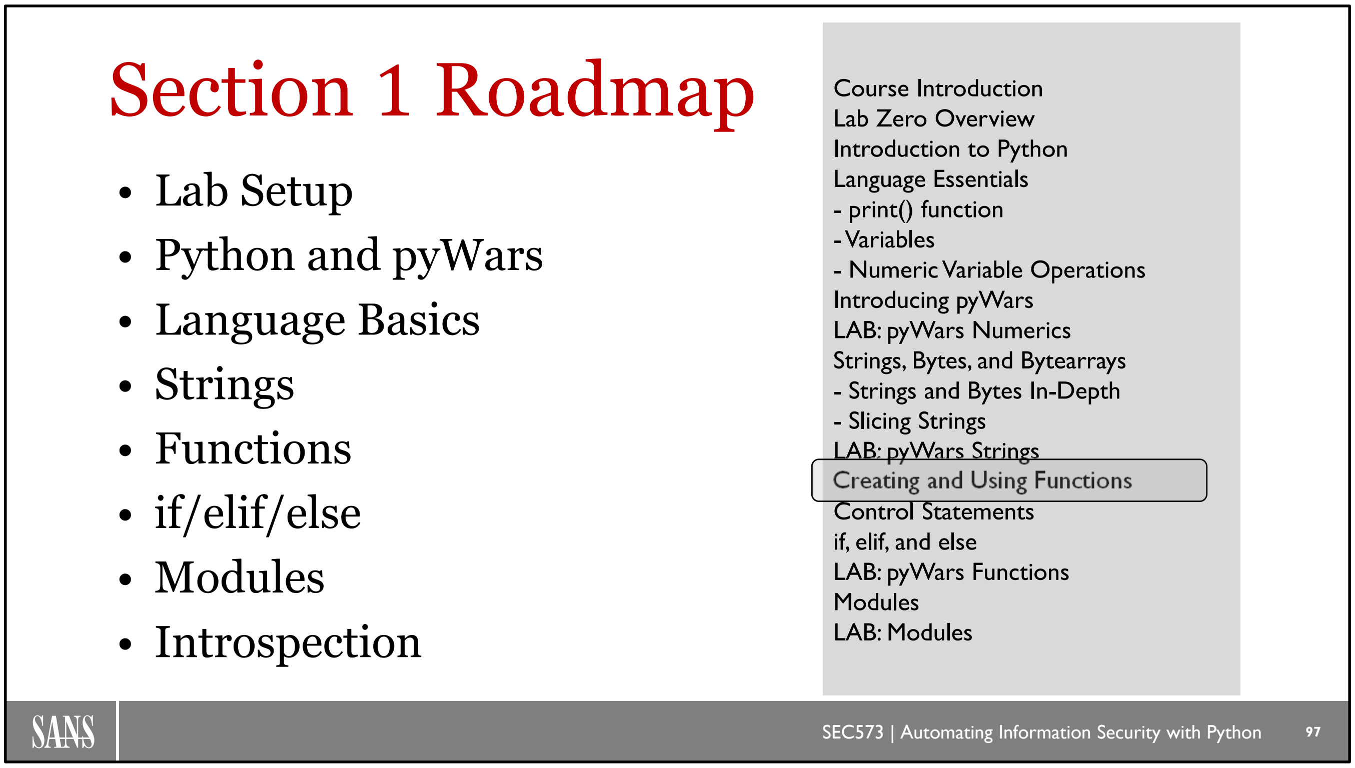


Lab Highlights: Use of the Find Method

```
>>> d.question(12)
"The answer is the position of the first letter of the word
  'SANS' in the data() string. "
>>> x=d.data(12)
>>> x
'I went to SANS training and all I got was this huge brain. '
>>> x.find("SANS")
10
>>> d.data(12).find("SANS")
10
>>> d.answer(12,d.data(12).find("SANS"))
'Correct!'
```

SANS | SEC573 | Automating Information Security with Python 96

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(12), we are asking it to find the letter "S" from the beginning of the word "SANS" inside of d.data(12). 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.

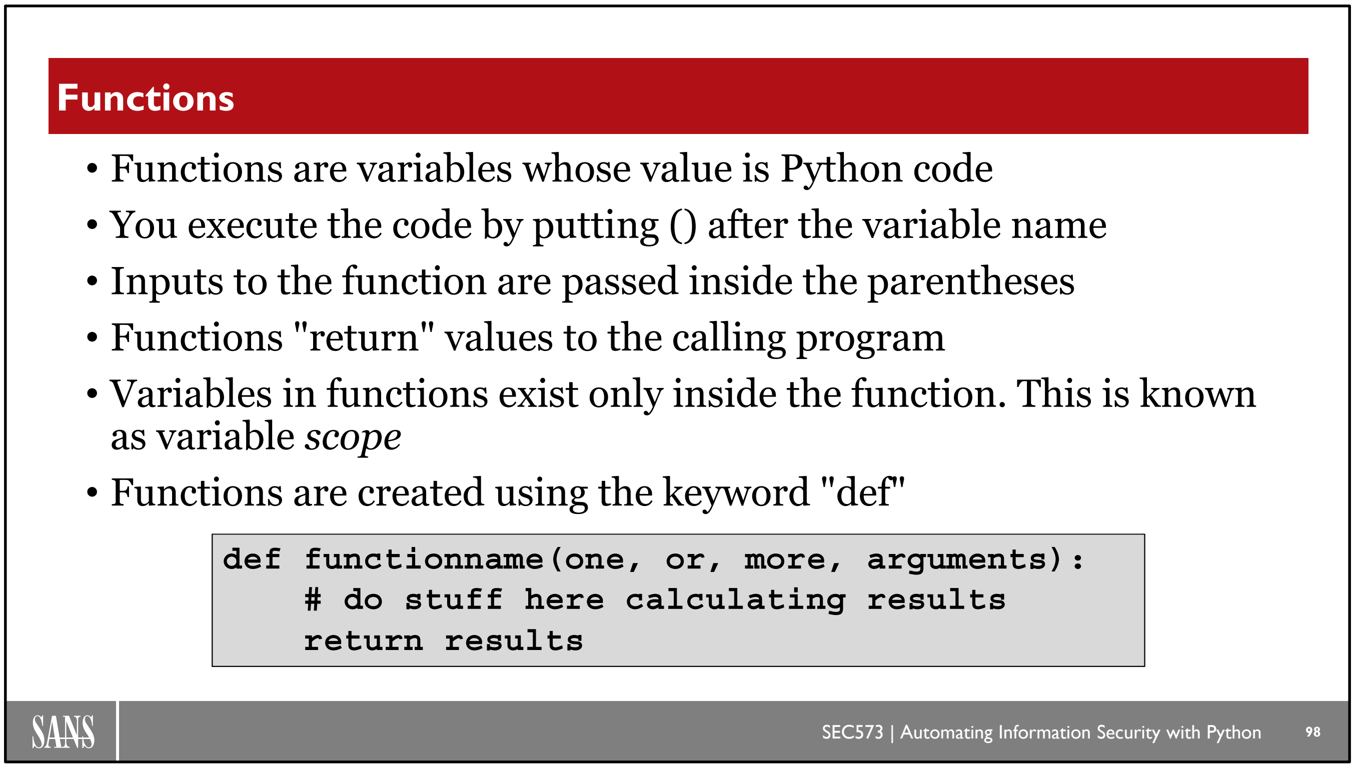


Section 1 Roadmap

- Lab Setup
- Python and pyWars
- Language Basics
- Strings
- Functions
- if/elif/else
- Modules
- Introspection

SANS

This is a Roadmap slide.



Functions

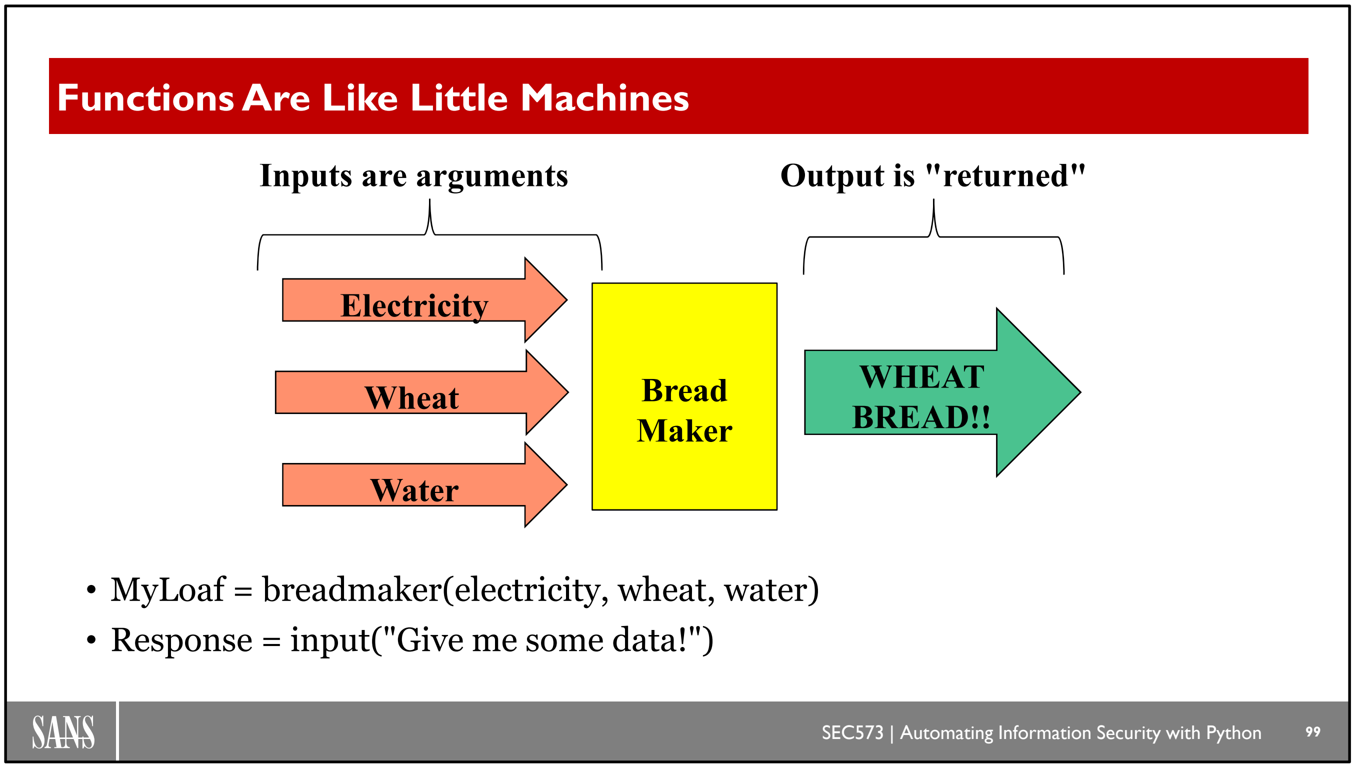
- Functions are variables whose value is Python code
- You execute the code by putting () after the variable name
- Inputs to the function are passed inside the parentheses
- Functions "return" values to the calling program
- Variables in functions exist only inside the function. This is known as variable *scope*
- Functions are created using the keyword "def"

```
def functionname(one, or, more, arguments):
    # do stuff here calculating results
    return results
```

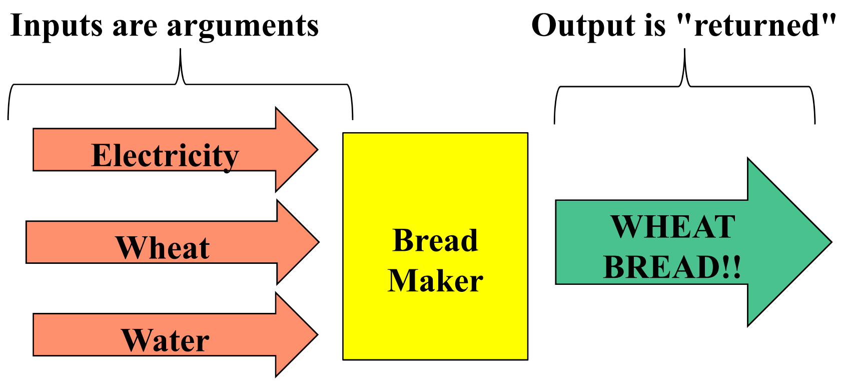
SANS SEC573 | Automating Information Security with Python 98

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 an open and close parenthesis 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.



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 corn bread 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 exercise. 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.

Defining Functions

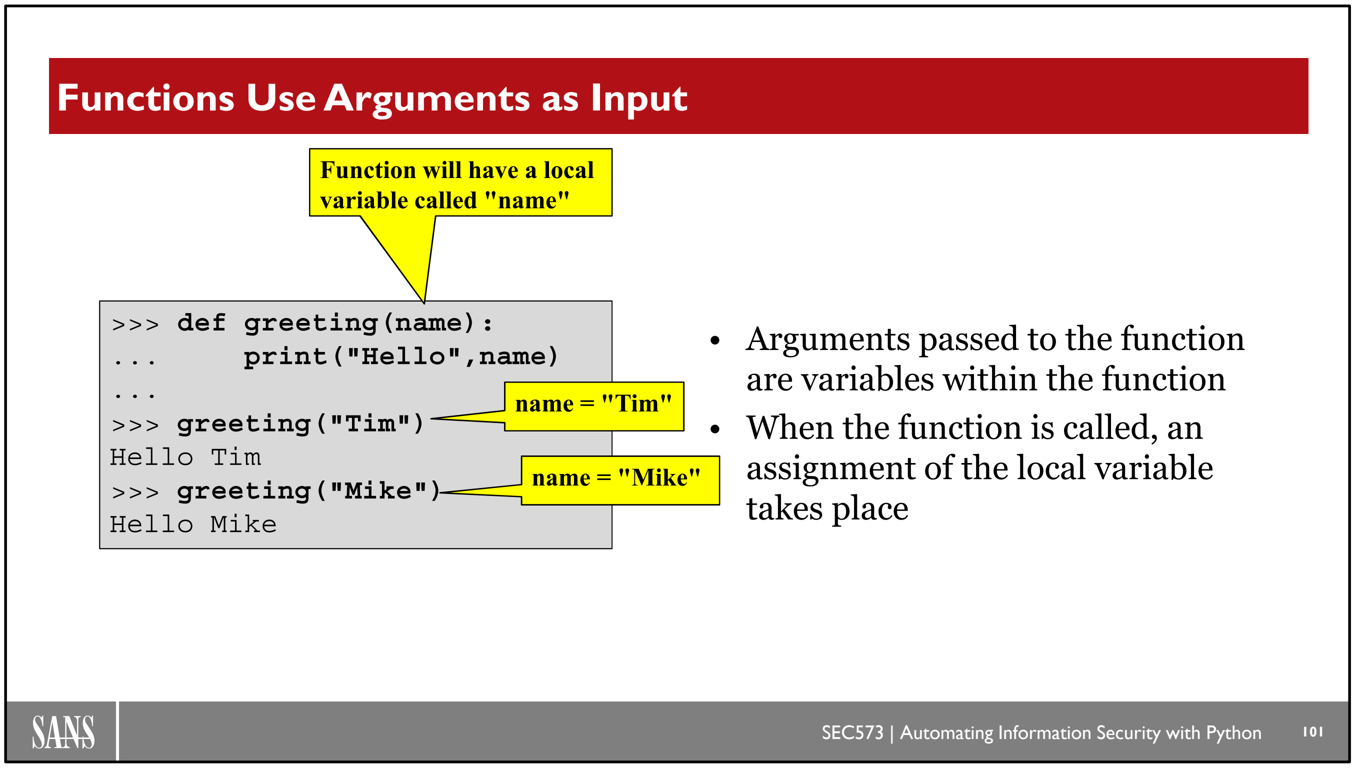
- Functions begin with the keyword "def" and a colon
- Accepts "Arguments" to be processed
- Followed by a "code block" that is typically indented with 4 spaces
- Code block will end when it is no longer indented
- Values are returned to the calling program by "return"

```
def function_name(argument,arguments...):
    """An optional DocString for help()"""
    # - Code beneath it is executed when function
    # is called
    # - Arguments can be given default values by
    # assigning them a value.
    # - The code can return 1 or more values
    return "Returns this string"
```

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 close 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".



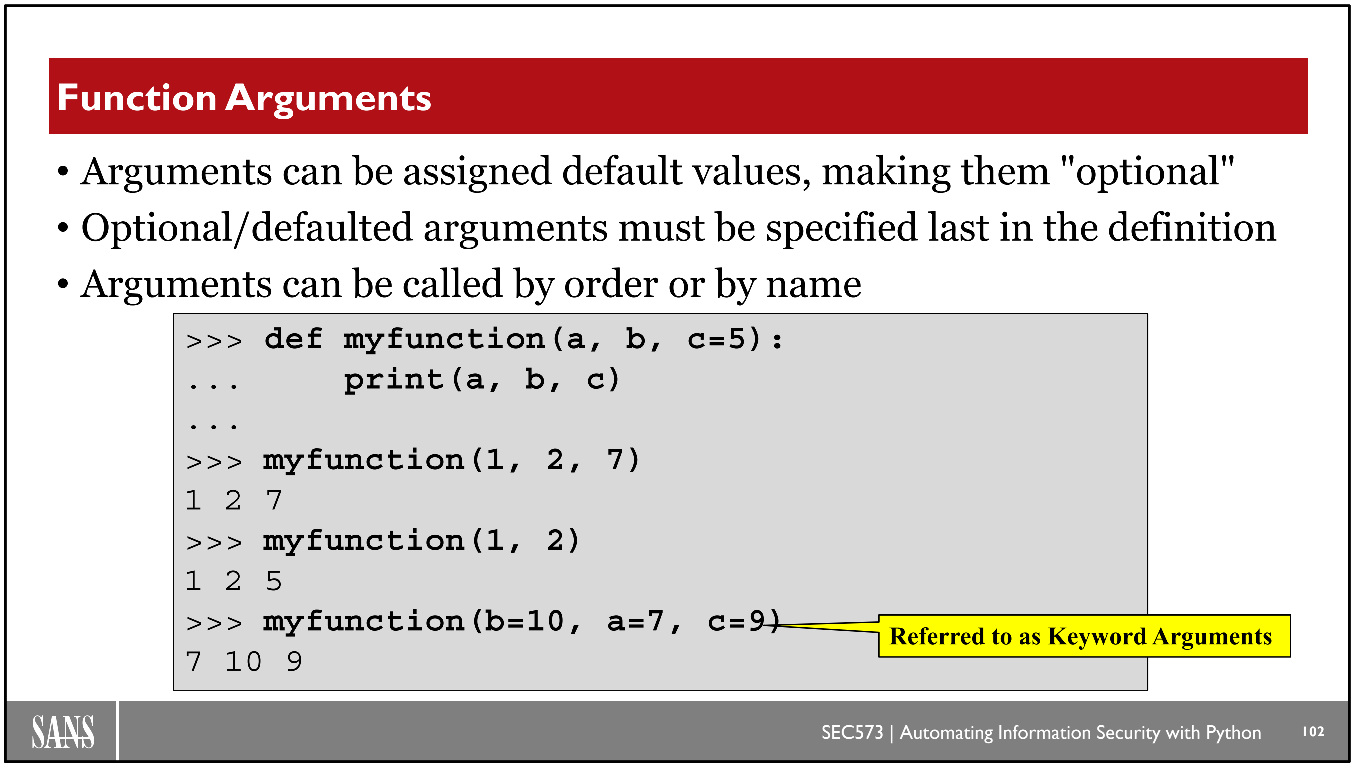
Functions Use Arguments as Input

```
>>> def greeting(name):
...     print("Hello",name)
...
>>> greeting("Tim")
Hello Tim
>>> greeting("Mike")
Hello Mike
```

Function will have a local variable called "name"
name = "Tim"
name = "Mike"

- Arguments passed to the function are variables within the function
- When the function is called, an assignment of the local variable takes place

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.



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

SANS

SEC573 | Automating Information Security with Python 102

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

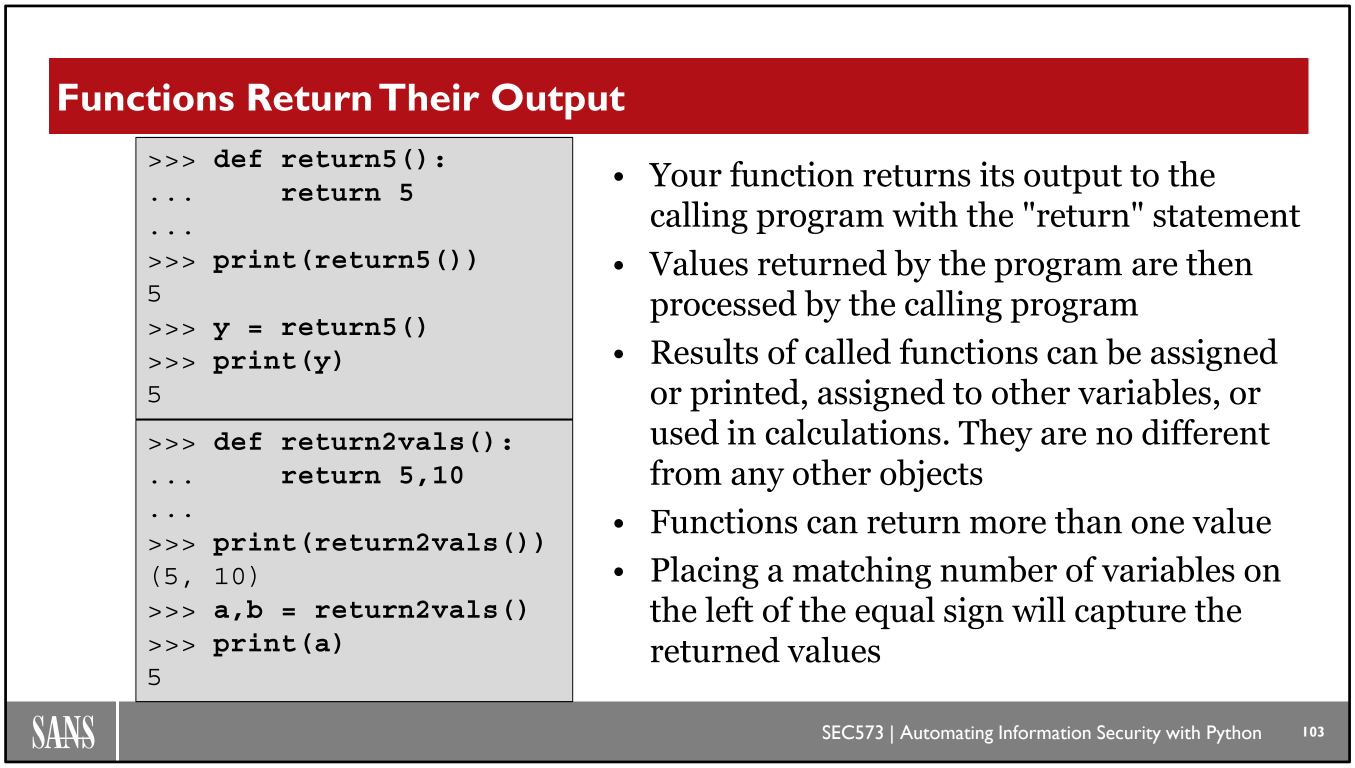
```
>>> myfunction(1, 2)
1 2 5
```

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
```



Functions Return Their Output

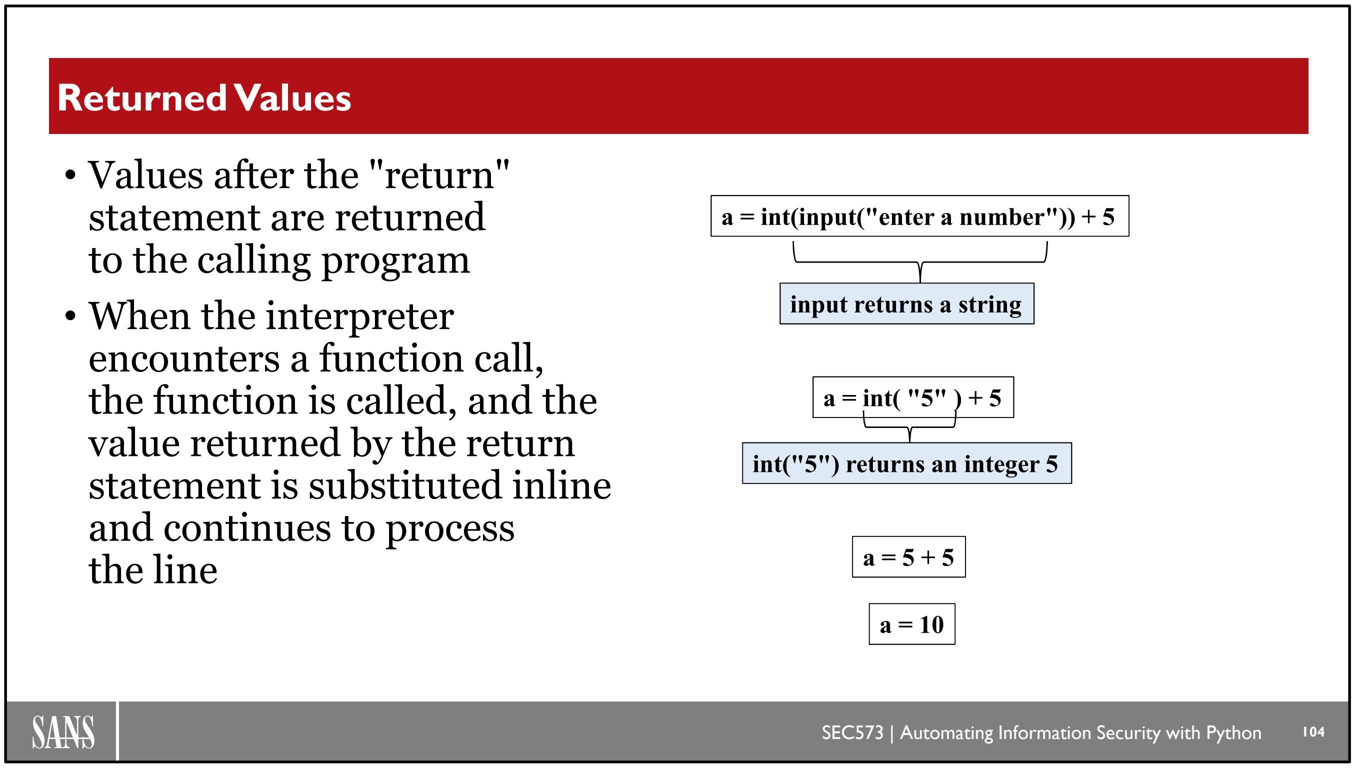
```
>>> def return5():
...     return 5
...
>>> print(return5())
5
>>> y = return5()
>>> print(y)
5
```

```
>>> 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.



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

$$a = \underbrace{\text{int(input("enter a number"))}}_{\text{input returns a string}} + 5$$

$$a = \text{int(}\underbrace{\text{ "5" }}_{\text{int("5") returns an integer 5}}\text{)} + 5$$

a = 5 + 5

a = 10

SANS SEC573 | Automating Information Security with Python 104

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
```

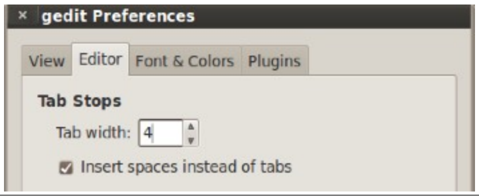
Now Python calls the function int("5"), which will convert the string containing a 5 into an integer. Python now has

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

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

Syntax and Spacing

- In Python, spacing is important. It tells the interpreter what code is in a block
- Never mix tabs and spaces! Text may line up visually on screen but be interpreted as different code blocks or not line up and be interpreted as the same
- #!/usr/bin/python –tt prints an error if tabs are used
- According to PEP-0008, you should use 4 spaces for each indentation level
  - https://www.python.org/dev/peps/pep-0008/
- If you use gedit in this class, you can make your life easier by changing preferences
- In gedit, click the "hamburger" cog in the upper-right corner and click **Preferences**
- Click the **Editor** tab
- Change the Tab Stops to this:



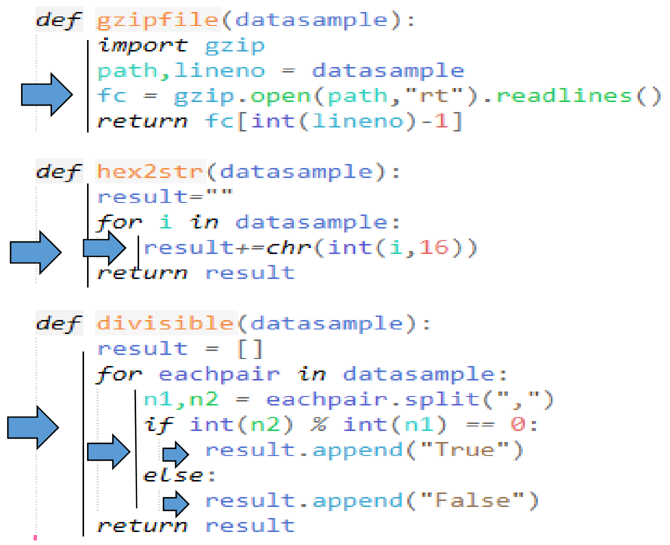
Tab Stops

Tab width: 4

☑ Insert spaces instead of tabs

Because spaces are used to tell Python what portions of code are part of the same code block, spacing becomes very important to the execution of your program. If your program mixes tabs and spaces, the lines might look as if they are part of the same code block when they are not because they visually line up on the screen. There are a couple of ways you can deal with this problem. One way is to place a "#!/usr/bin/python -tt" shebang line at the beginning of your program. This tells Python to generate an error anytime it sees a mix of spaces and tabs in your program. You can also make your life easier by configuring gedit to type 4 spaces anytime you press the Tab key. You do this by selecting the hamburger cog in the upper-right corner and clicking **Preferences**. Then, on the **Editor** tab, select **Insert spaces instead of tabs** and select 4 spaces for **Tab width**.

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 (:)
- Code in the block all shares the same indentation levels
- Block ends when indentation stops

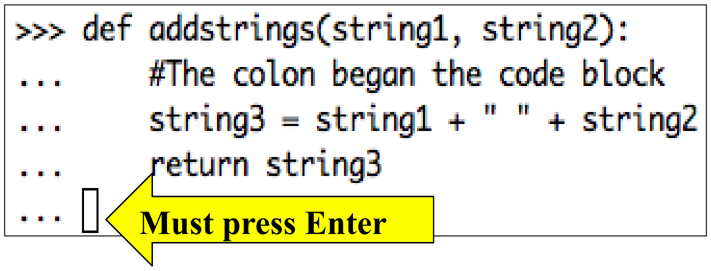
When you begin declaring a new function with the keyword "def", how does Python know which lines that follow are part of the new function and which are part of the next function? In other words, how does Python mark the beginning and end of a code block? Other languages, such as C, use specific characters,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

Defining a Function in the Shell

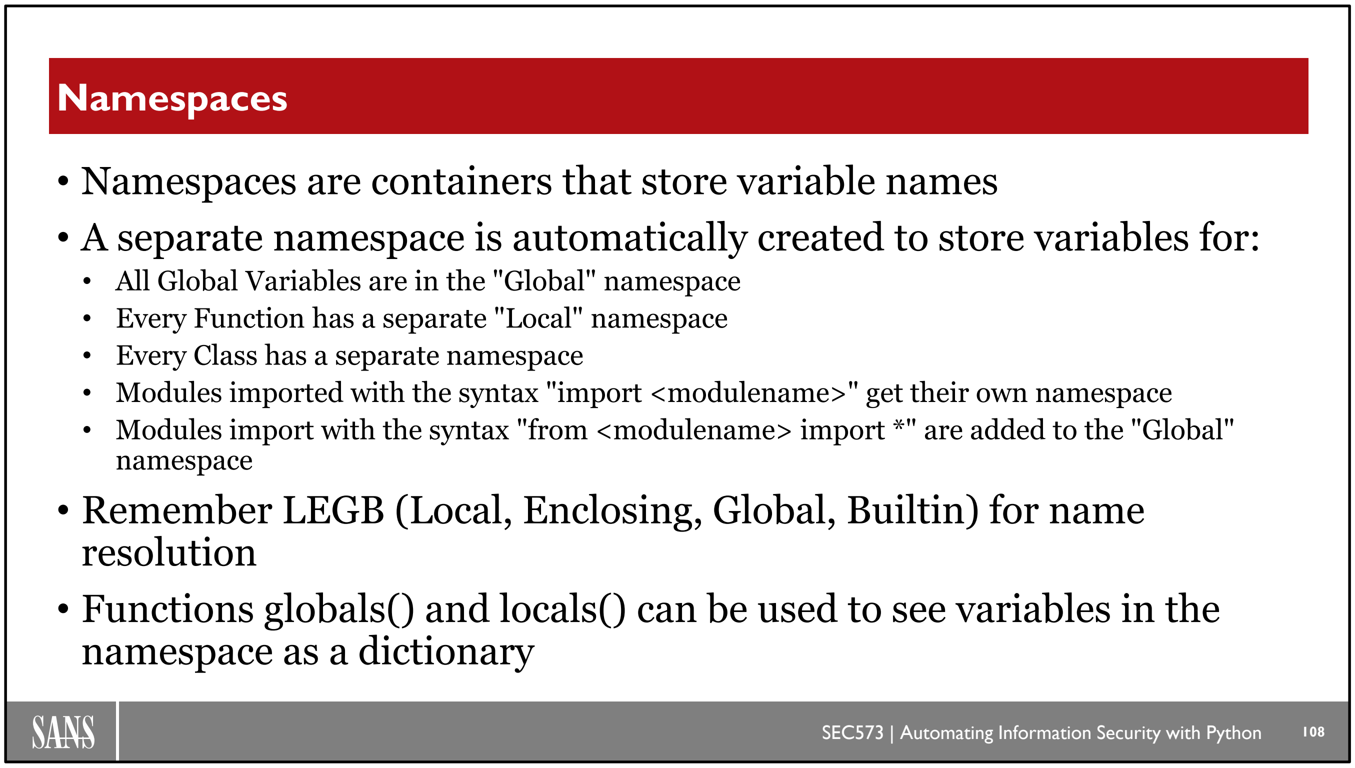
- From within a shell, the prompt changes from ">>>" to "..." when you are within a code block
- When defining a function in the Python shell, you must press Enter to change the indentation back to the beginning of the line
- When copying and pasting from a program into a shell, you must add blank lines between functions



```
>>> def addstrings(string1, string2):
...     #The colon began the code block
...     string3 = string1 + " " + string2
...     return string3
...
```

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.



Namespaces

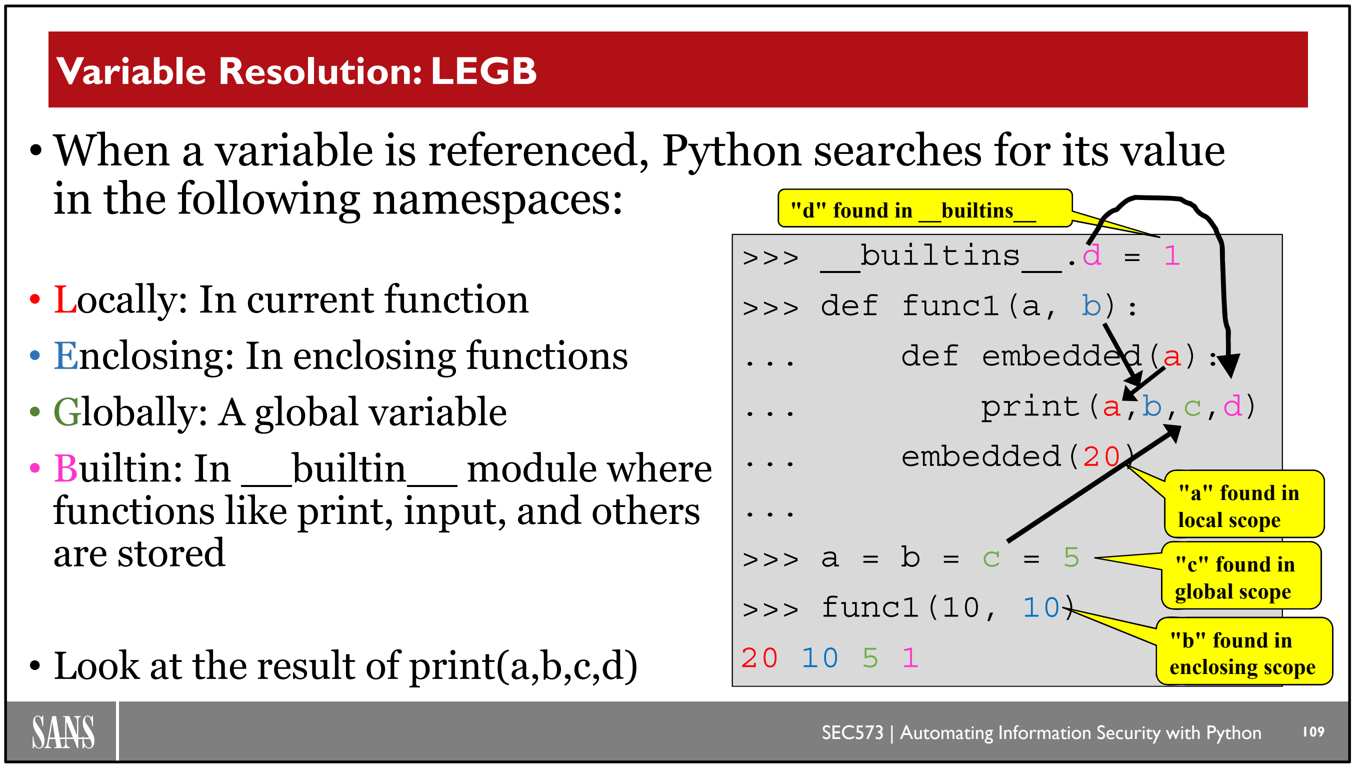
- Namespaces are containers that store variable names
- A separate namespace is automatically created to store variables for:
  - All Global Variables are in the "Global" namespace
  - Every Function has a separate "Local" namespace
  - Every Class has a separate namespace
  - Modules imported with the 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

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. Within them, you can use the locals() function to view the content of their 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 functions. 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)]
```



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)

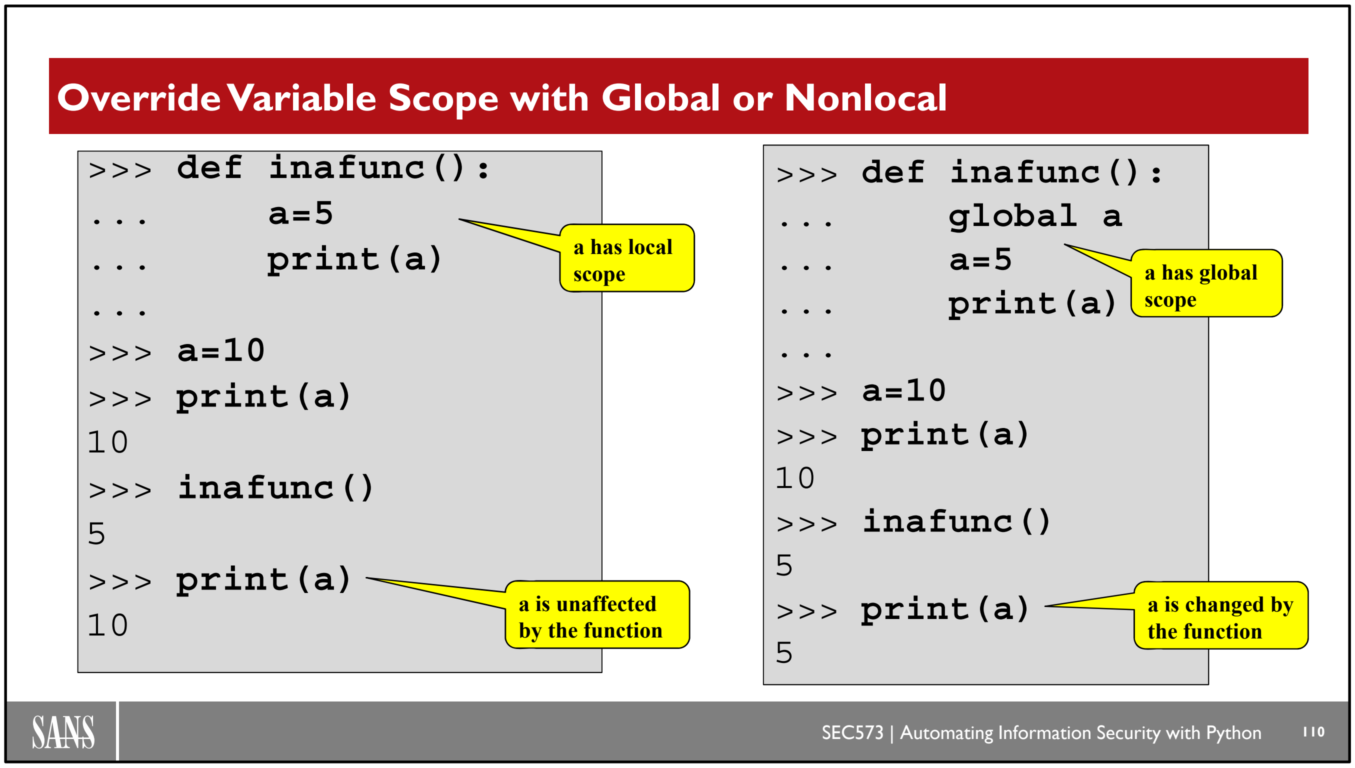
```
>>> __builtins__.d = 1
>>> def func1(a, b):
...     def embedded(a):
...         print(a,b,c,d)
...     embedded(20)
...
>>> a = b = c = 5
>>> func1(10, 10)
20 10 5 1
```

"d" found in __builtins__
"a" found in local scope
"c" found in global scope
"b" found in enclosing scope

SANS SEC573 | Automating Information Security with Python 109

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.



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

SEC573 | Automating Information Security with Python 110

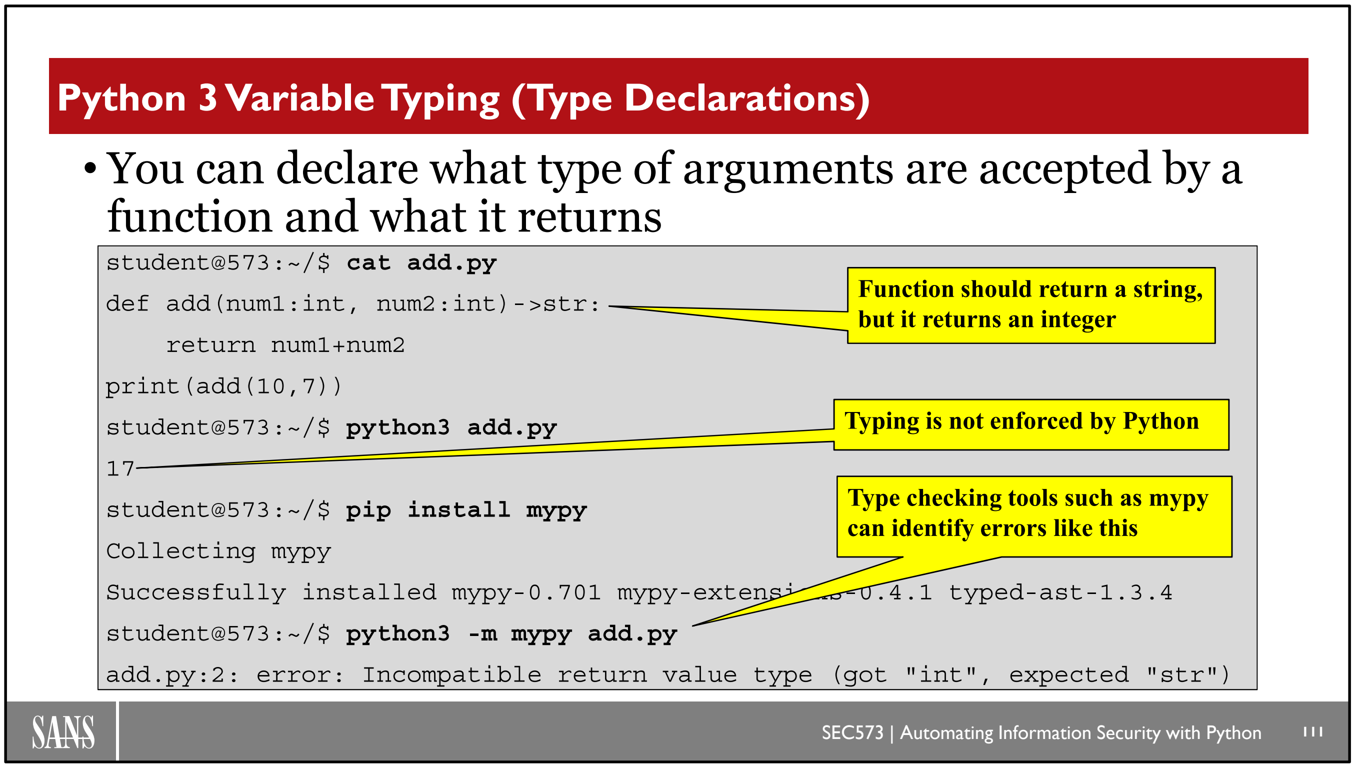
When a function is executing, it can read and write to any variable in its local namespace. It can also read from the global namespace. However, functions do not update the global namespace. The reason is that the assignment of any variable inside the function results in the creation of a new local variable. 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 left, 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']
```



Python 3 Variable Typing (Type Declarations)

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

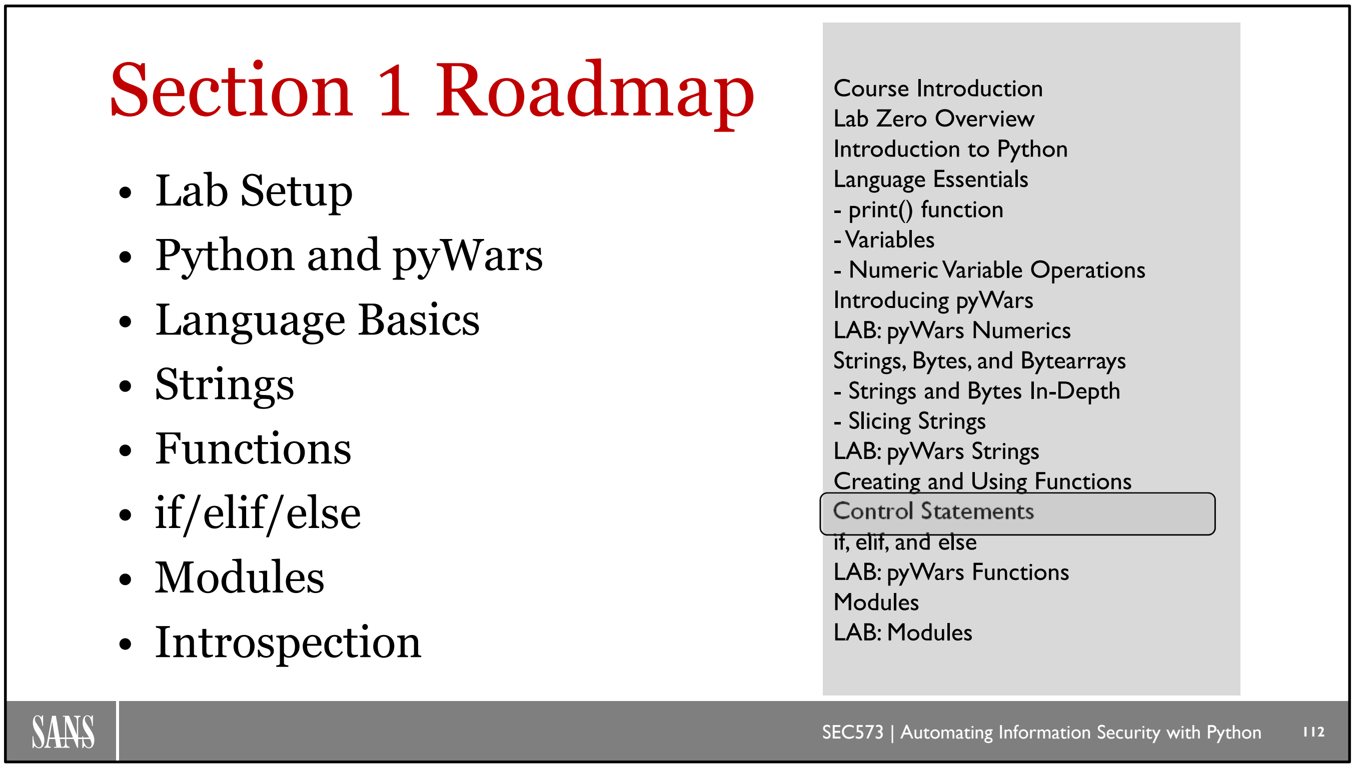
```
student@573:~/$ cat add.py
def add(num1:int, num2:int)->str:
    return num1+num2
print(add(10,7))
student@573:~/$ python3 add.py
17
student@573:~/$ pip install mypy
Collecting mypy
Successfully installed mypy-0.701 mypy-extensions-0.4.1 typed-ast-1.3.4
student@573:~/$ python3 -m mypy add.py
add.py:2: error: Incompatible return value type (got "int", expected "str")
```

Function should return a string, but it returns an integer
Typing is not enforced by Python
Type checking tools such as mypy can identify errors like this

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.

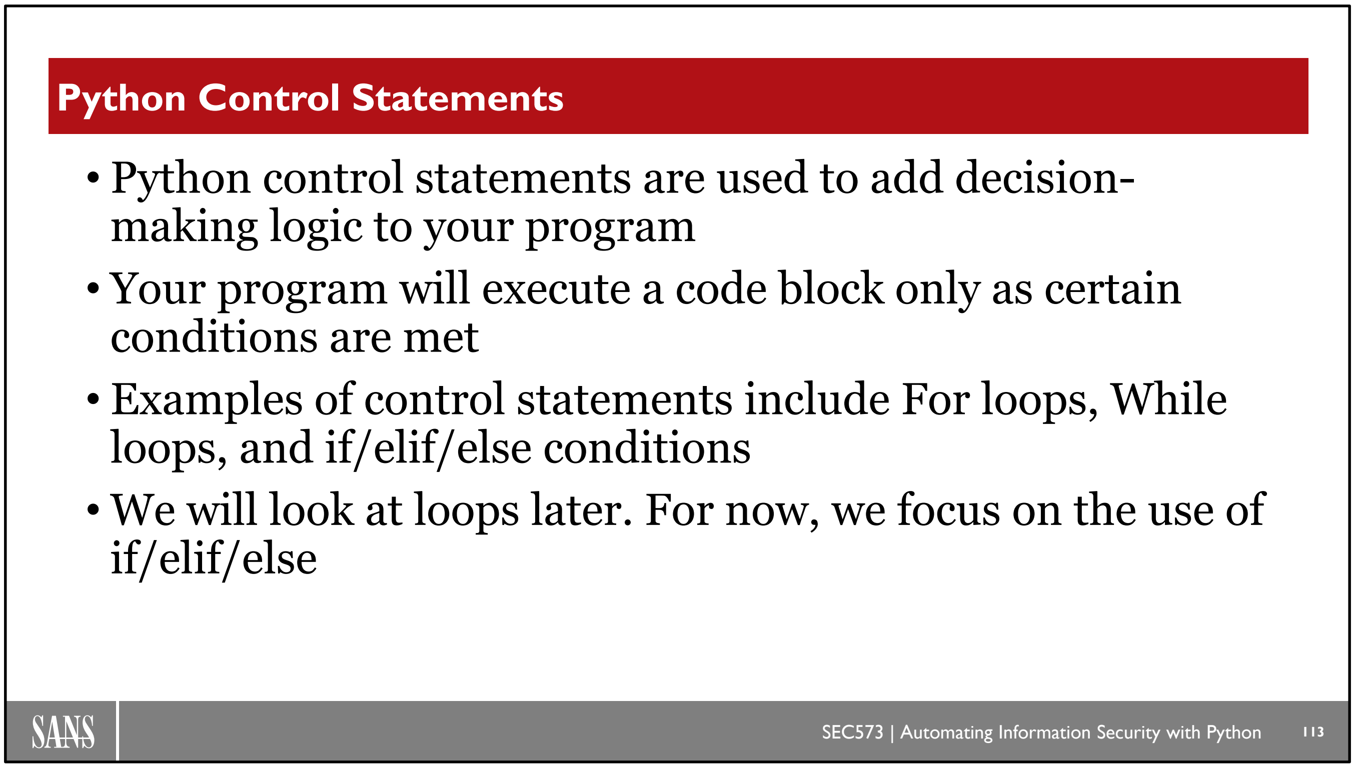
Because some Python interpreter other than CPython such as iPython, IronPython, Jython, etc. may decide to enforce this standard in a future version, I recommend if you do use this feature, you always scan your code with mypy and fix any issues it identifies. Otherwise, you may have code that you think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



Section 1 Roadmap

- Lab Setup
- Python and pyWars
- Language Basics
- Strings
- Functions
- if/elif/else
- Modules
- Introspection

This is a Roadmap slide.

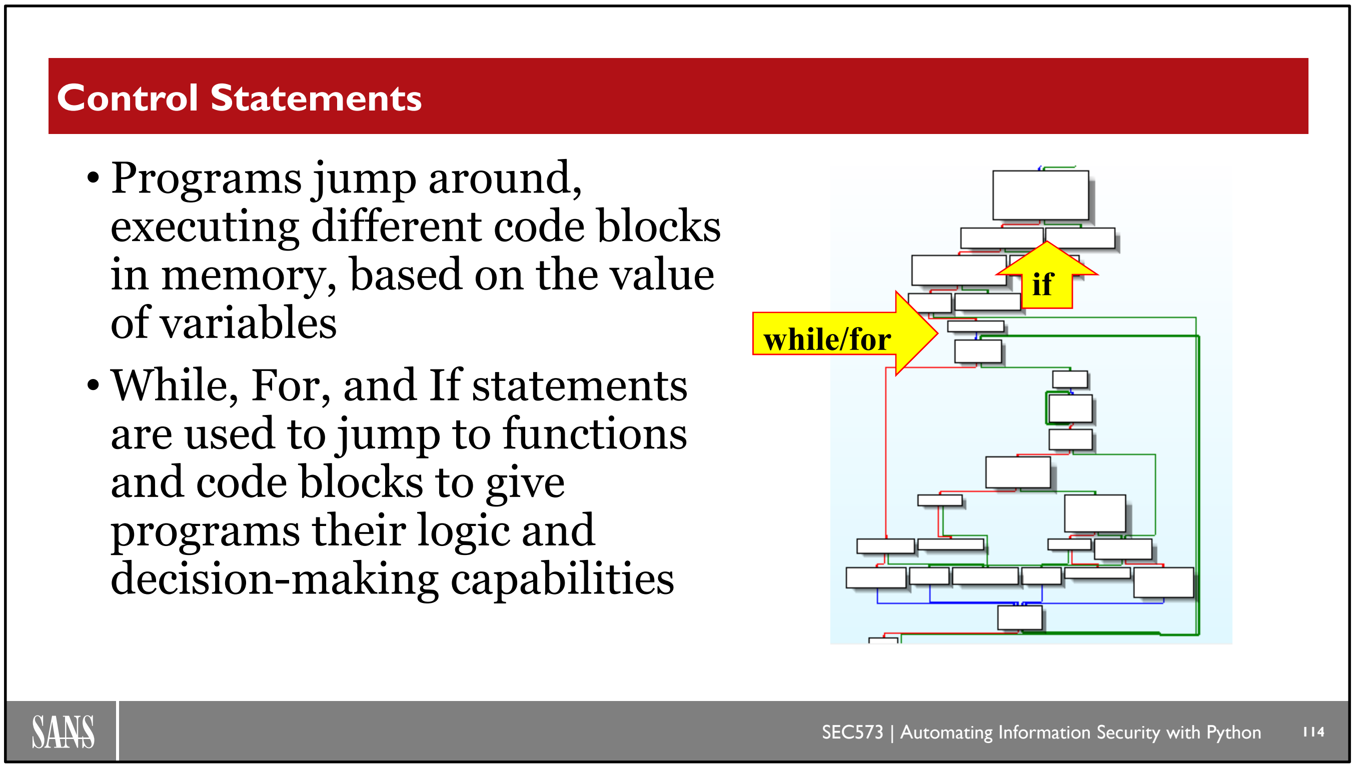


Python Control Statements

- 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

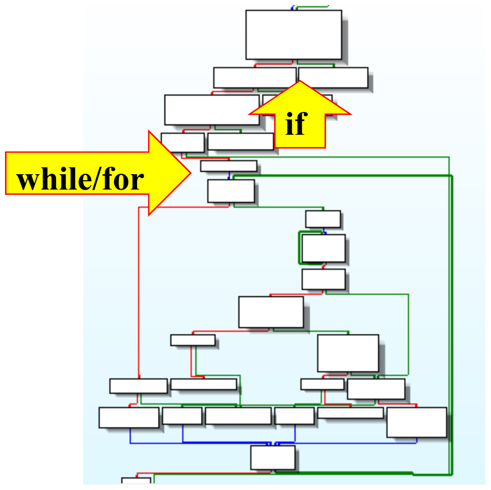
SANS SEC573 | Automating Information Security with Python 113

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.



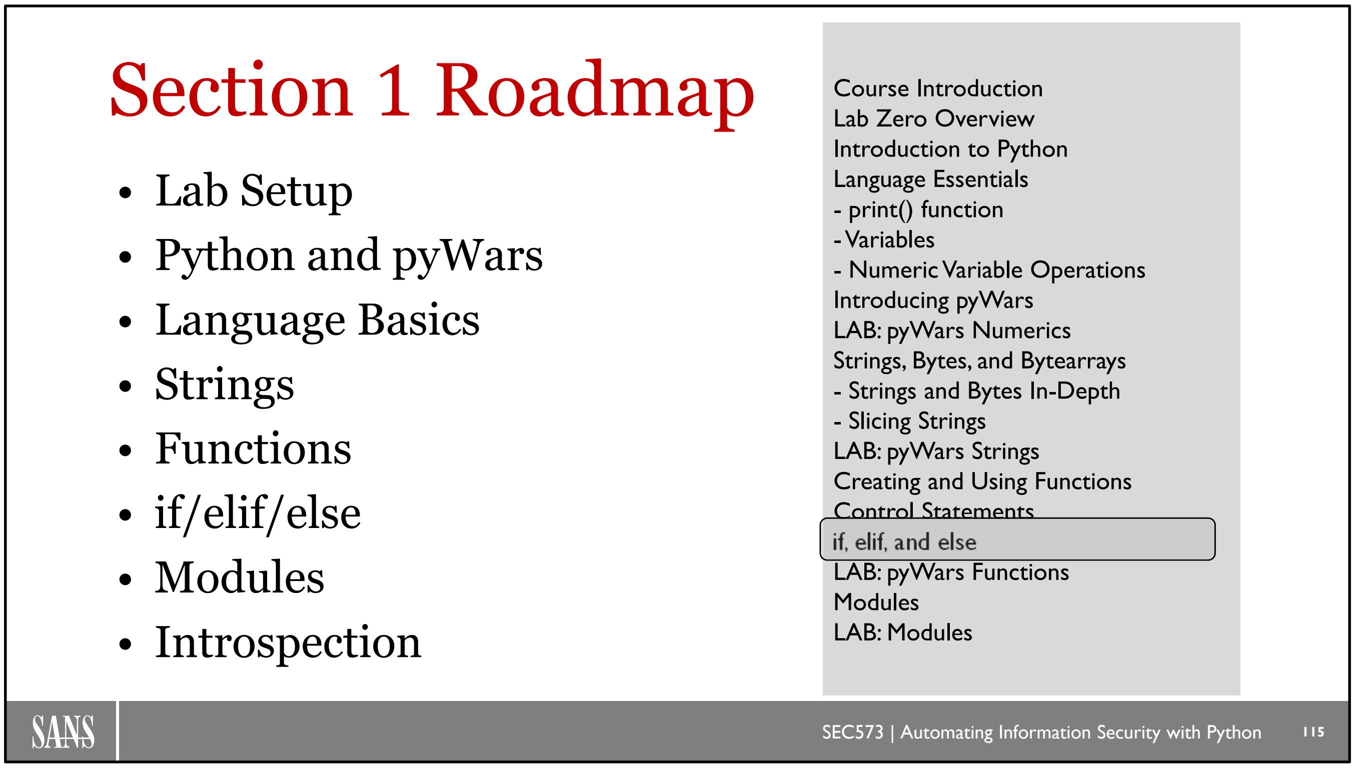
Control Statements

- 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



SANS | SEC573 | Automating Information Security with Python 114

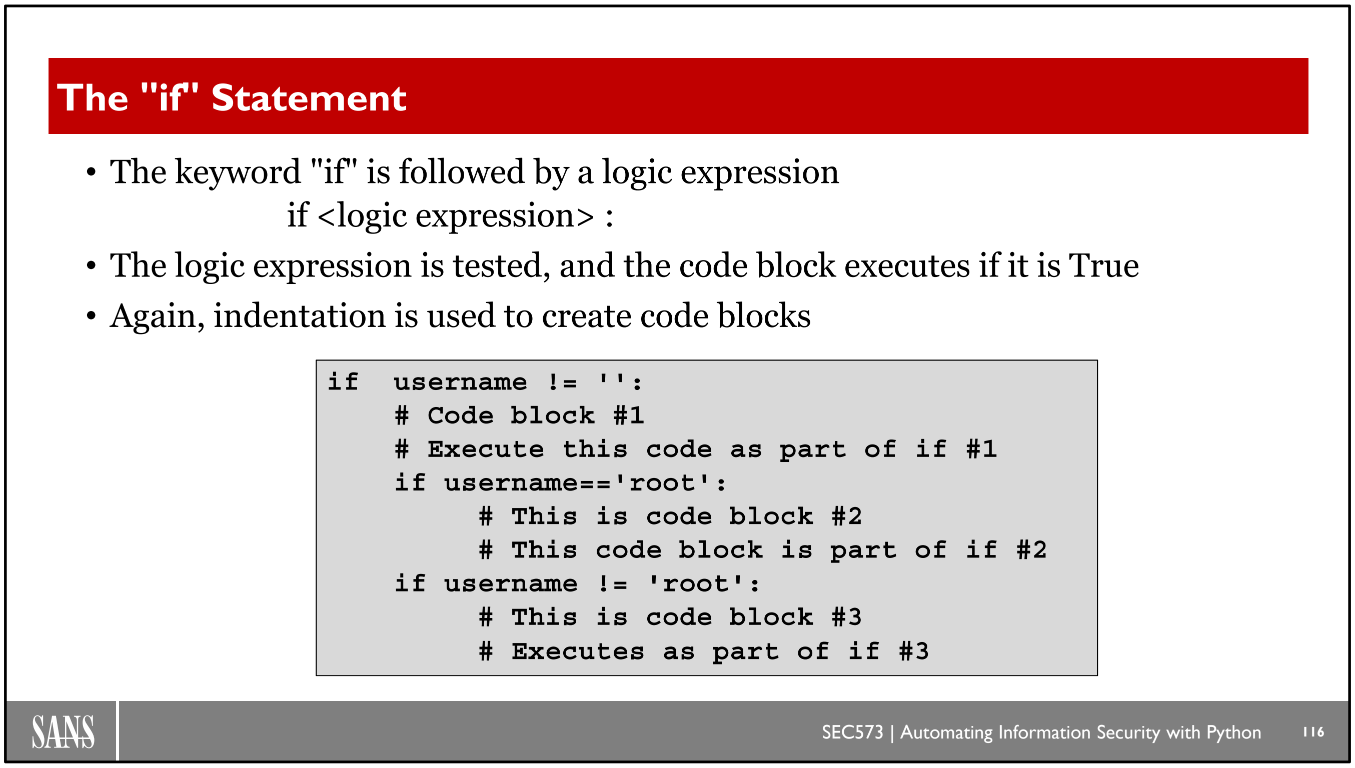
Very few programs execute sequentially from top to bottom without any "branches" in code execution. Most programs jump around in memory, based on the value of variables in the program. The program hits a control statement such as an "IF" or a “FOR” loop, which then jumps to different functions and code 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.



Section 1 Roadmap

- Lab Setup
- Python and pyWars
- Language Basics
- Strings
- Functions
- if/elif/else
- Modules
- Introspection

This is a Roadmap slide.



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.

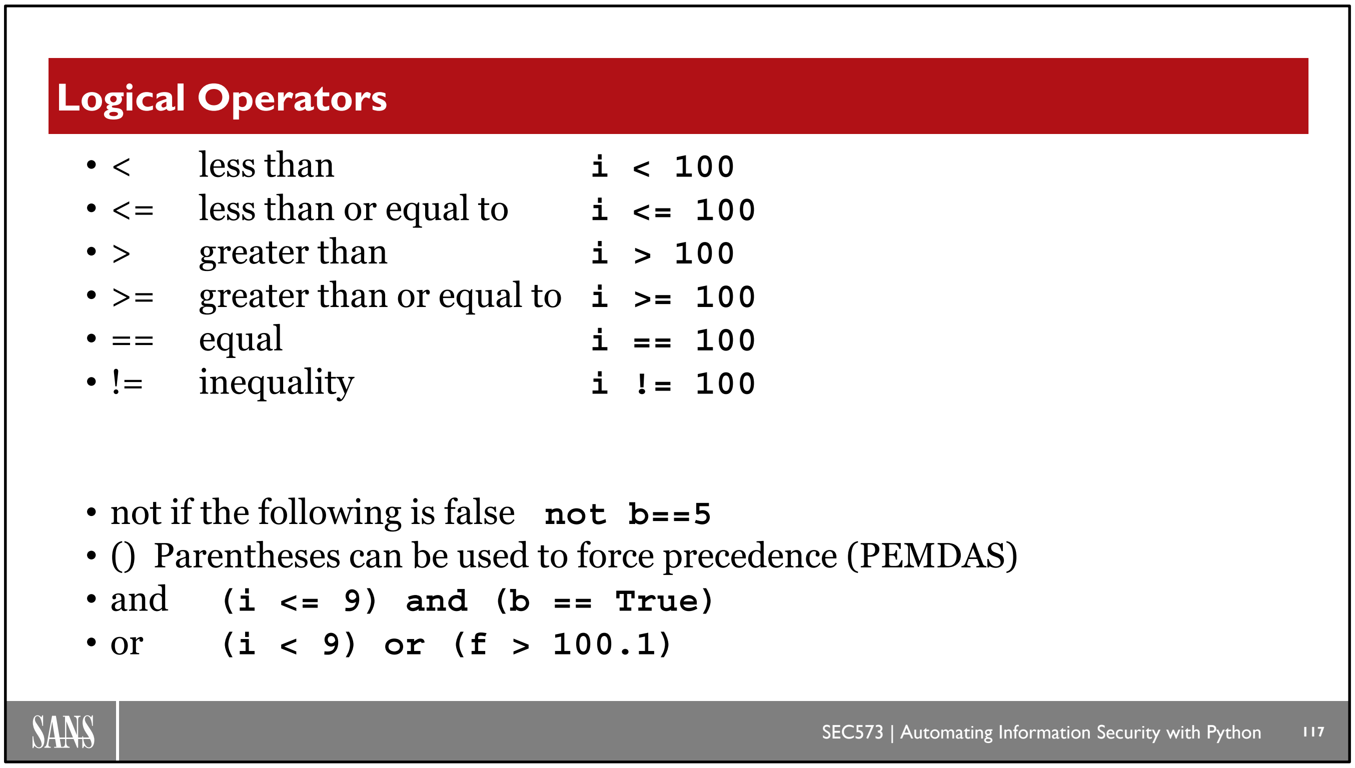
Suppose we want our program to take specific action depending on whether or not the username is "root". We might create a code block that executes only if the username is root by starting a code block with

```
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.



Logical Operators

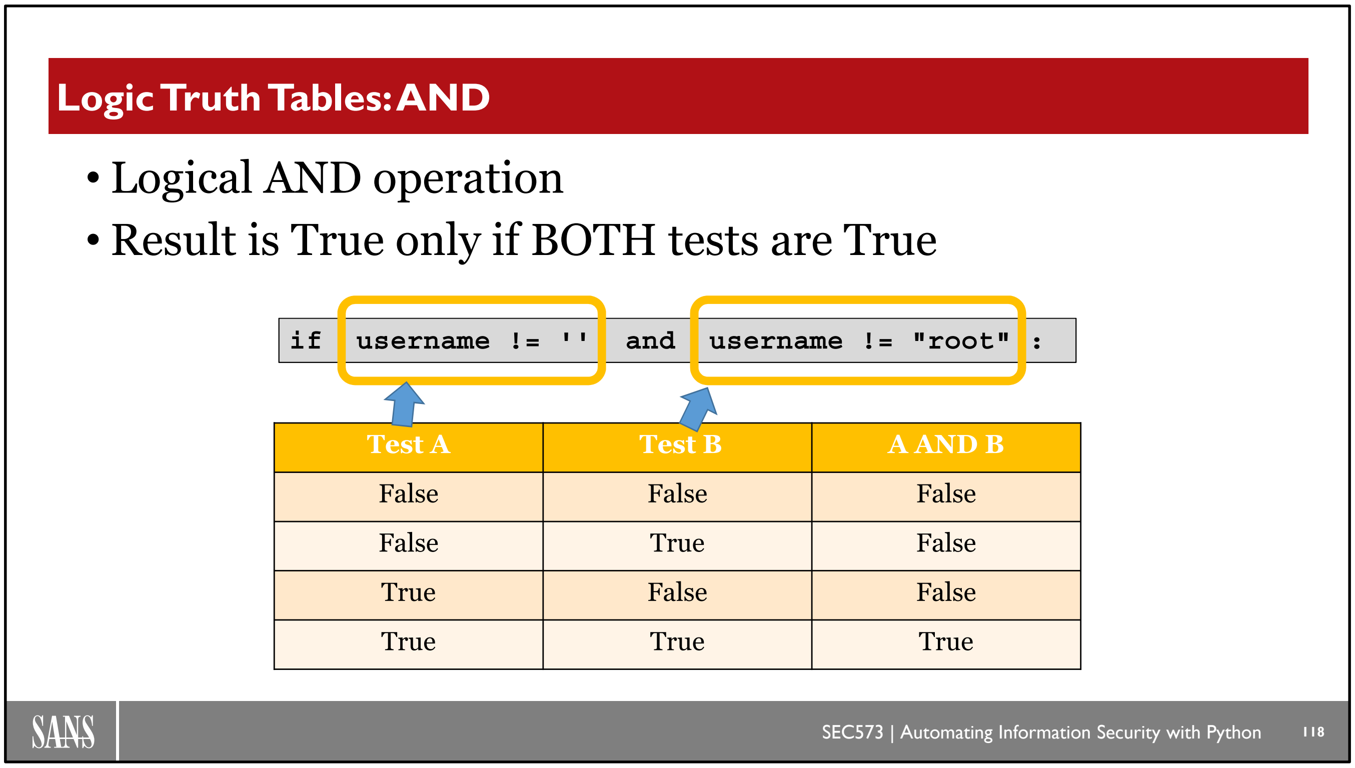
• <	less than	`i < 100`
• <=	less than or equal to	`i <= 100`
• >	greater than	`i > 100`
• >=	greater than or equal to	`i >= 100`
• ==	equal	`i == 100`
• !=	inequality	`i != 100`

- 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)`**

SANS SEC573 | Automating Information Security with Python 117

Here is a list of the logical operators you can use in your logic expression. Most of these are pretty 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 parenthesis, 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
```



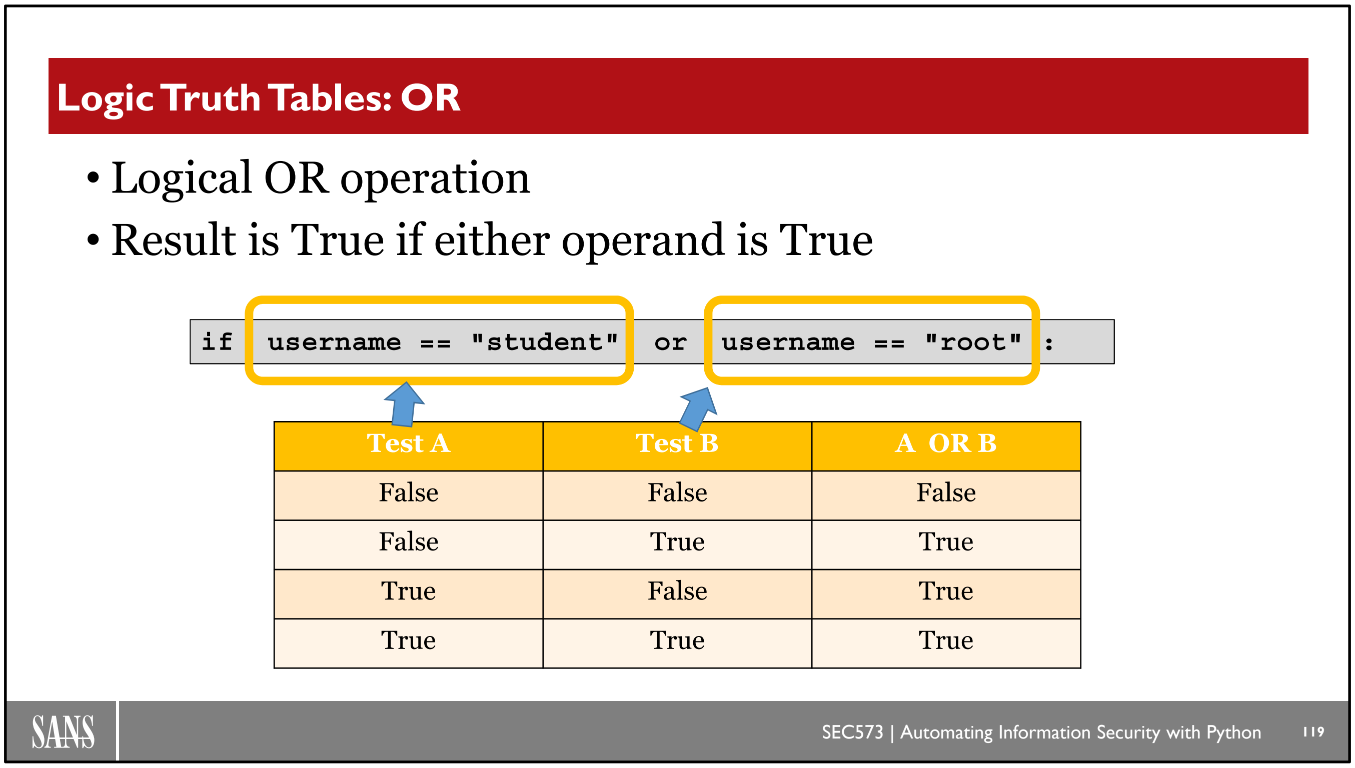
Logic Truth Tables: AND

- Logical AND operation
- Result is True only if BOTH tests are True

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

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.



Logic Truth Tables: OR

- Logical OR operation
- Result is True if either operand is True

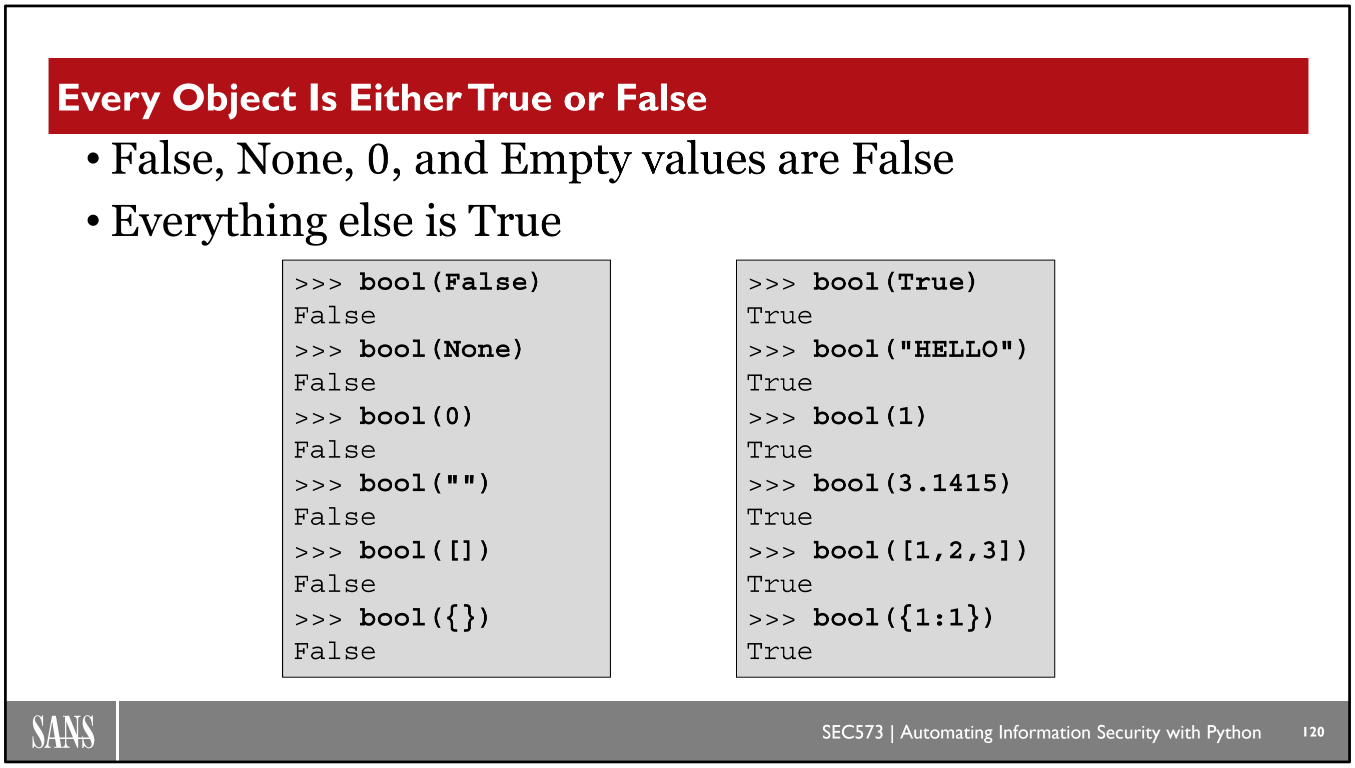
```
if  username == "student"  or  username == "root" :
```

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

SANS

SEC573 | Automating Information Security with Python 119

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.



Every Object Is Either True or False

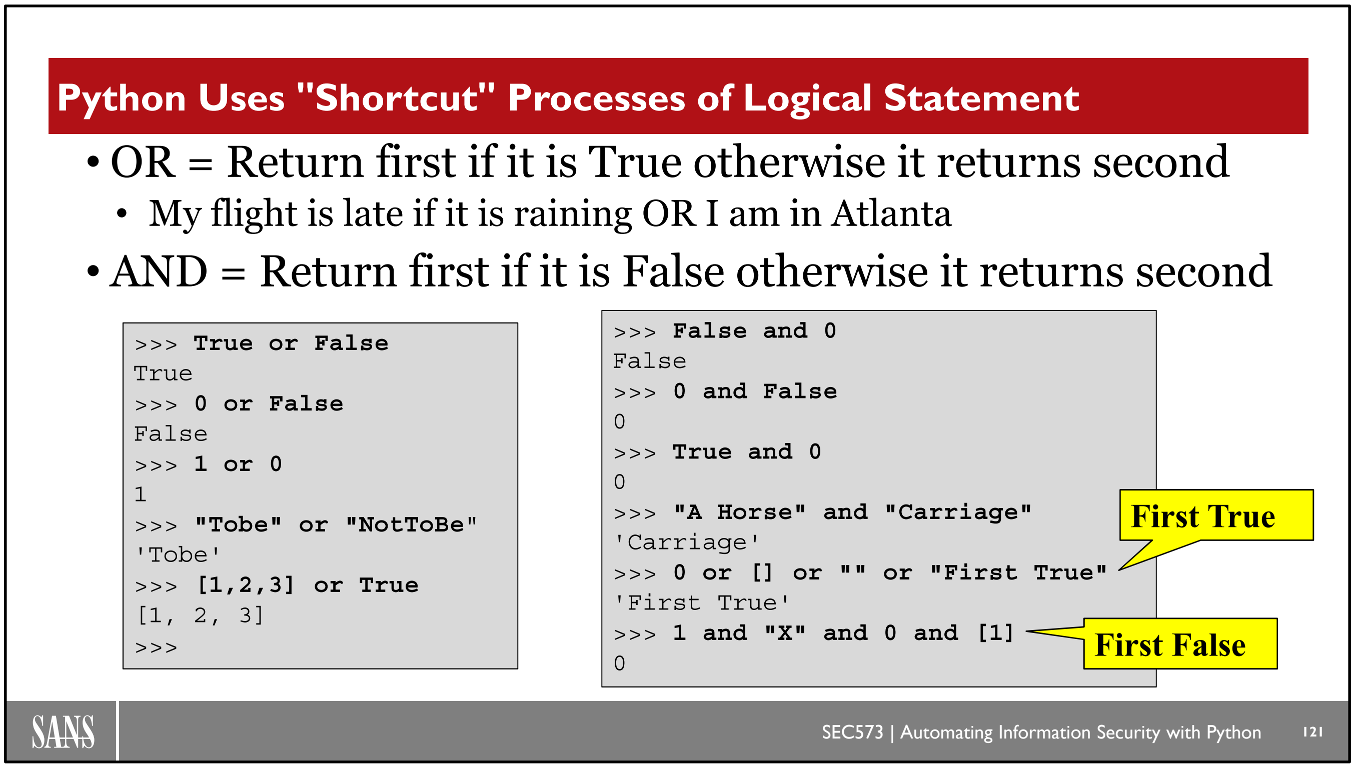
- False, None, 0, and Empty values are False
- Everything else is True

```
>>> 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
```

SANS | SEC573 | Automating Information Security with Python 120

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.



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
```

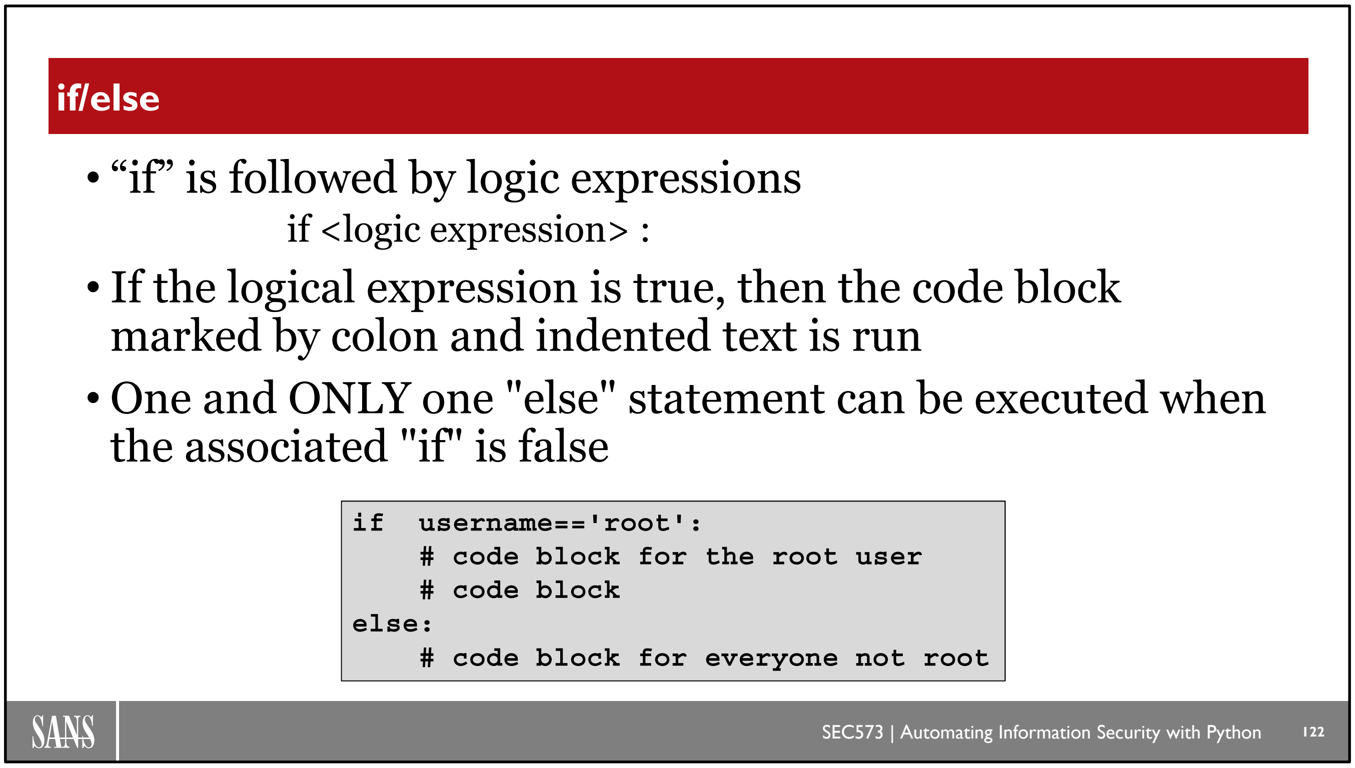
SANS SEC573 | Automating Information Security with Python 121

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 on 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.



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

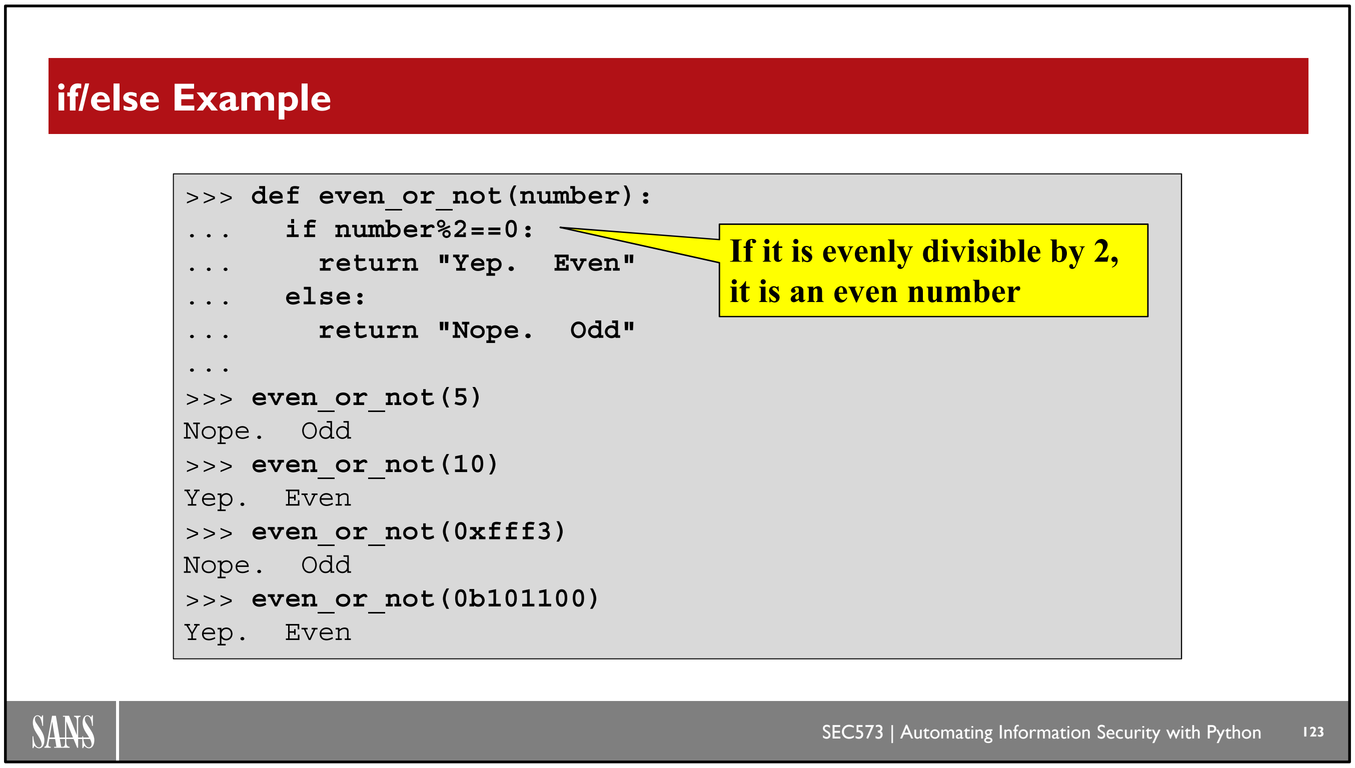
```
if  username=='root':
    # code block for the root user
    # code block
else:
    # code block for everyone not root
```

SANS

SEC573 | Automating Information Security with Python 122

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
```



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

SANS SEC573 | Automating Information Security with Python 123

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.

if/elif/else

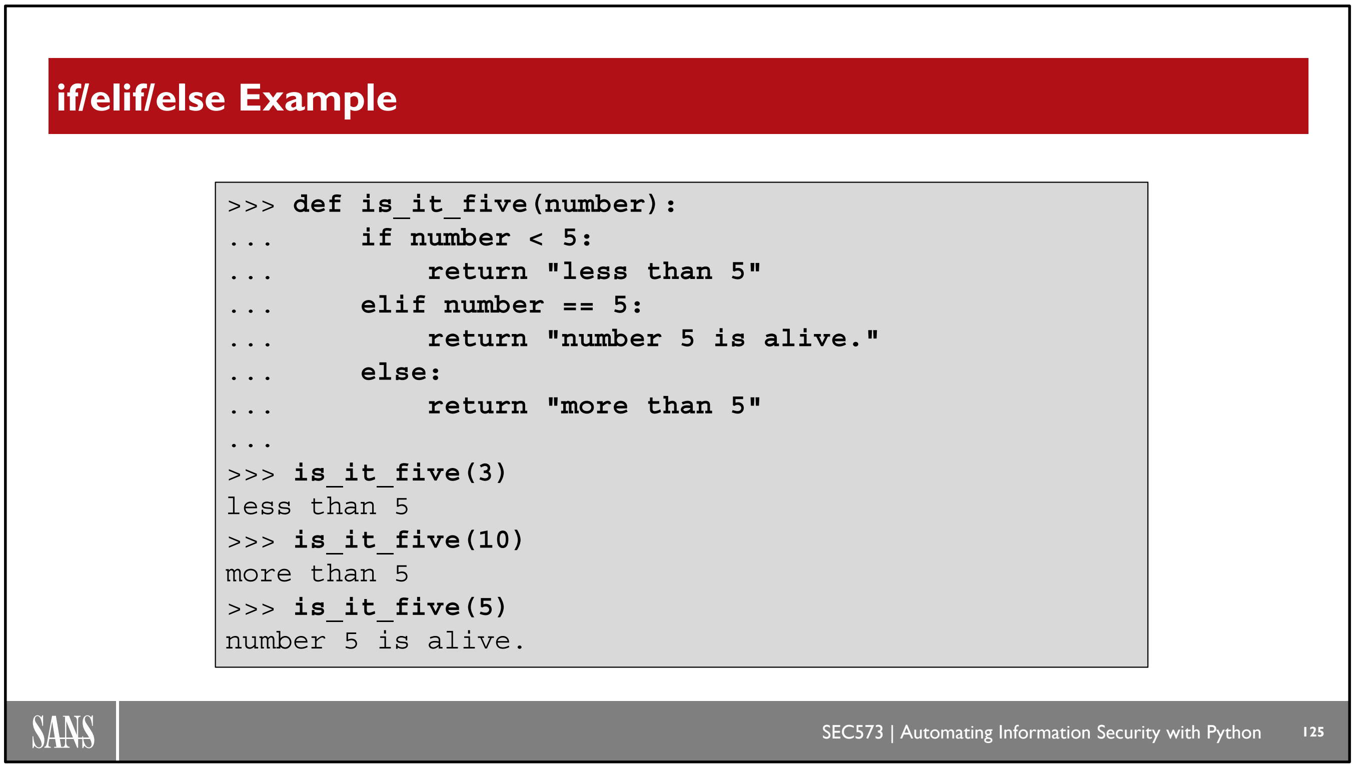
- "elif", which is short for "else if", can be added after the first "if"
- elif must also be followed by logic expressions

  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
```

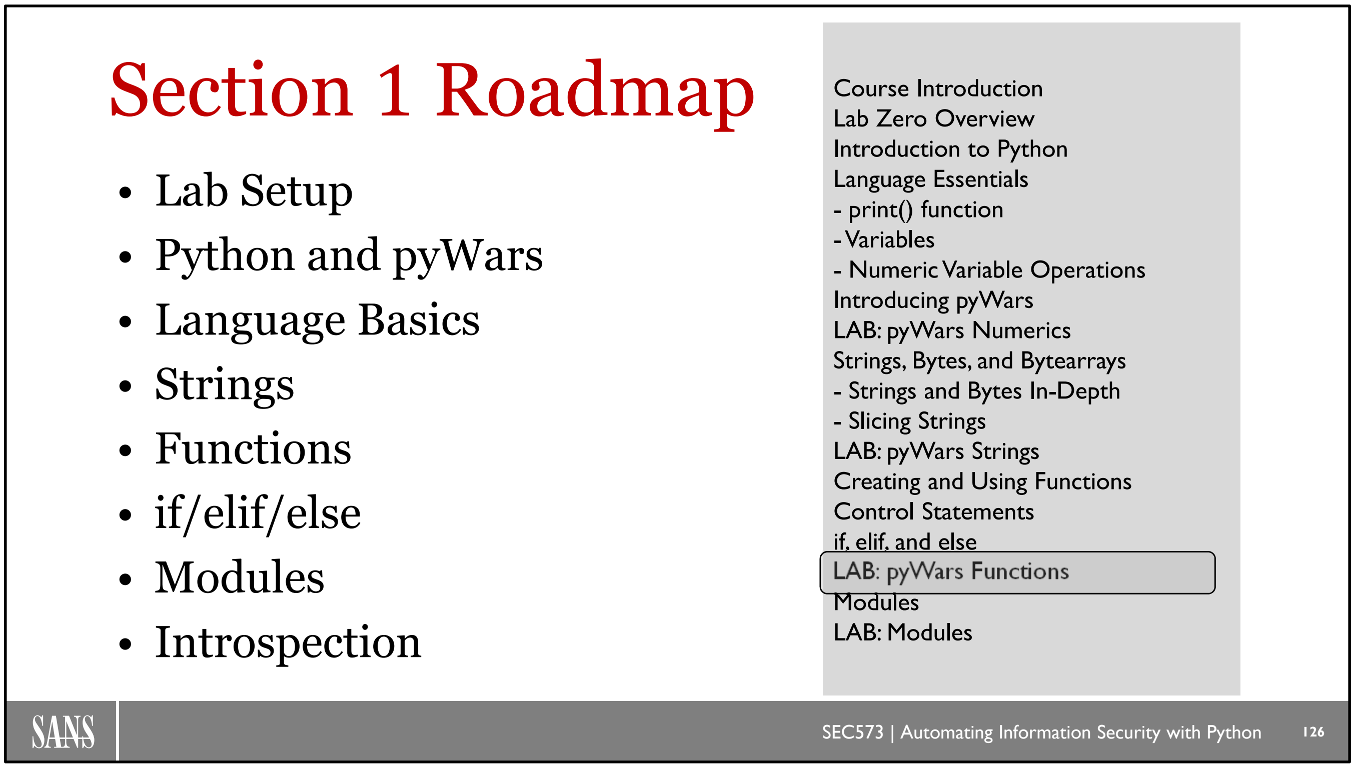


if/elif/else Example

```
>>> 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.
```

SANS | SEC573 | Automating Information Security with Python 125

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.

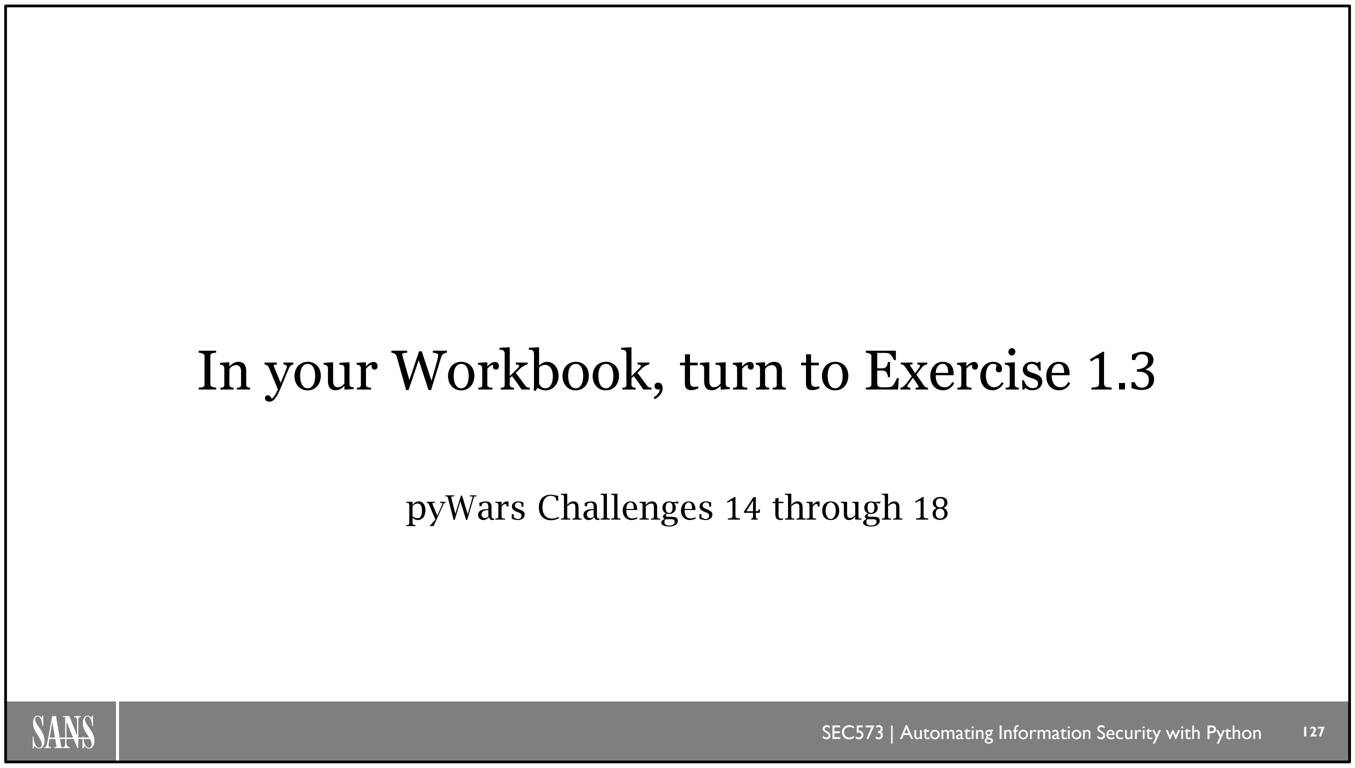


Section 1 Roadmap

- Lab Setup
- Python and pyWars
- Language Basics
- Strings
- Functions
- if/elif/else
- Modules
- Introspection

SANS

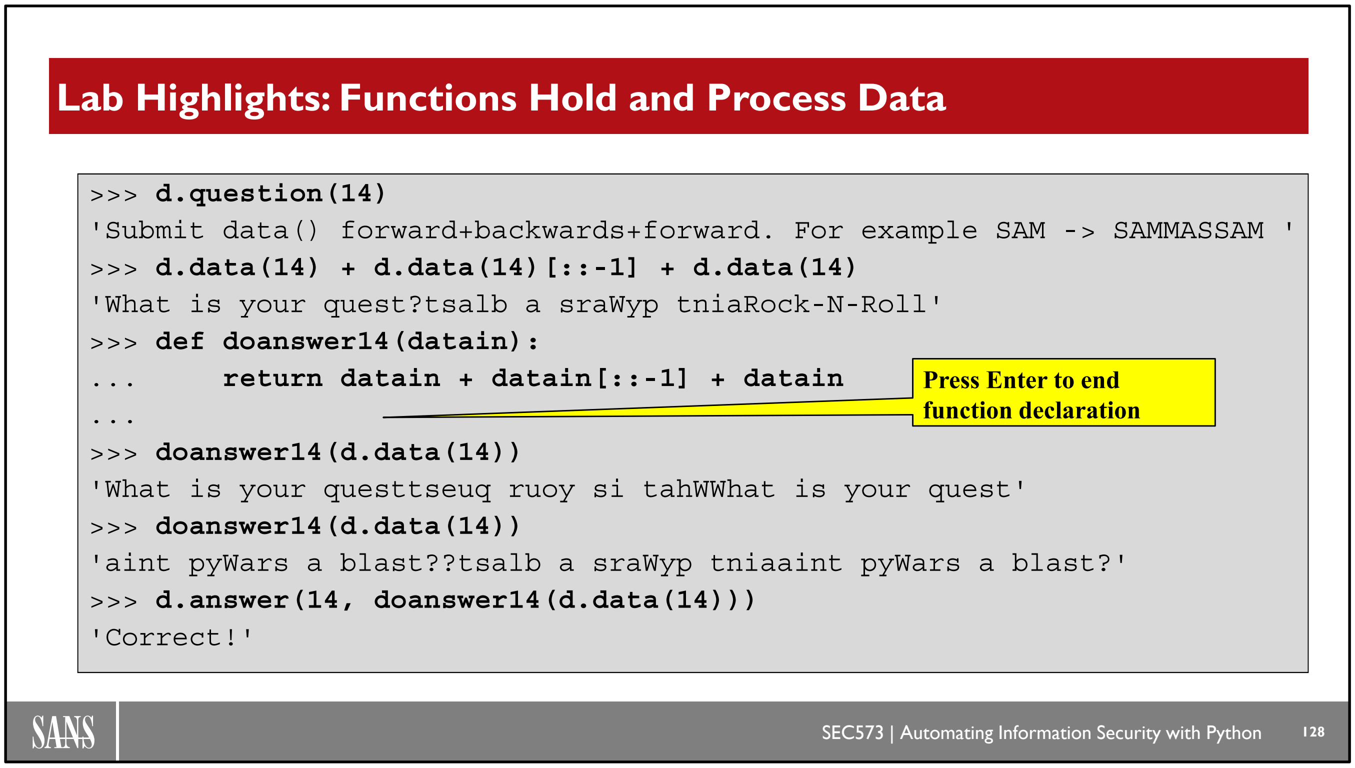
This is a Roadmap slide.



In your Workbook, turn to Exercise 1.3

pyWars Challenges 14 through 18

Please complete the exercise in your Workbook.



Lab Highlights: Functions Hold and Process Data

```
>>> d.question(14)
'Submit data() forward+backwards+forward. For example SAM -> SAMMASSAM '
>>> d.data(14) + d.data(14)[::-1] + d.data(14)
'What is your quest?tsalb a sraWyp tniaRock-N-Roll'
>>> def doanswer14(datain):
...     return datain + datain[::-1] + datain
...
>>> doanswer14(d.data(14))
'What is your questtseuq ruoy si tahWWhat is your quest'
>>> doanswer14(d.data(14))
'aint pyWars a blast??tsalb a sraWyp tniaaint pyWars a blast?'
>>> d.answer(14, doanswer14(d.data(14)))
'Correct!'
```

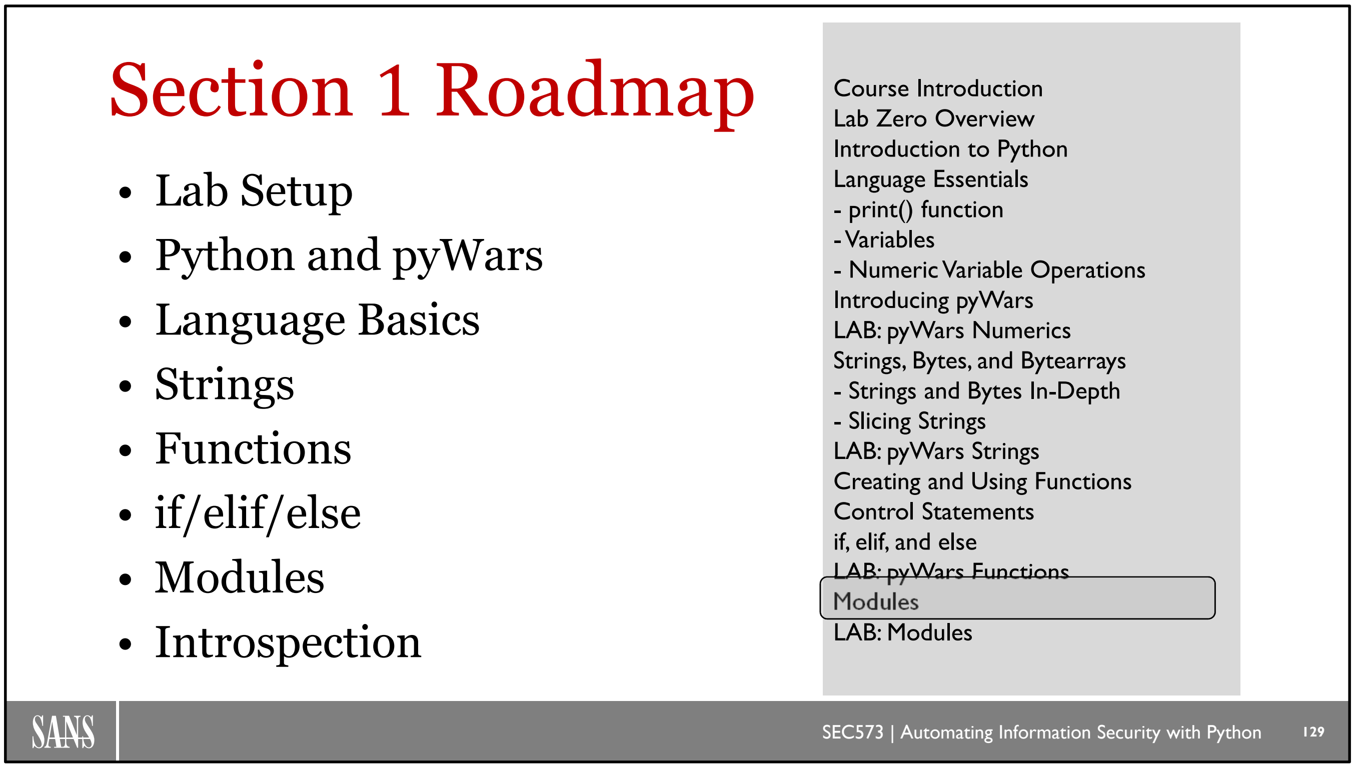
Press Enter to end function declaration

SANS SEC573 | Automating Information Security with Python 128

Question 14 illustrates how functions can solve one type of common programming challenge for us. Here you have to grab a piece of data once and use it three times. The string returned by d.data(14) changes every time you request it. If you try to do something like this, it does not work properly.

```
>>> d.data(14) + d.data(14)[::-1] + d.data(14)
```

This solution returns three different strings. When you use a function, it holds the contents of a single copy of the data in the function's argument. When the function is called, the variable 'datain' is assigned the value retrieved by 'd.data(14)'. Then the variable is used three times to produce the return string.



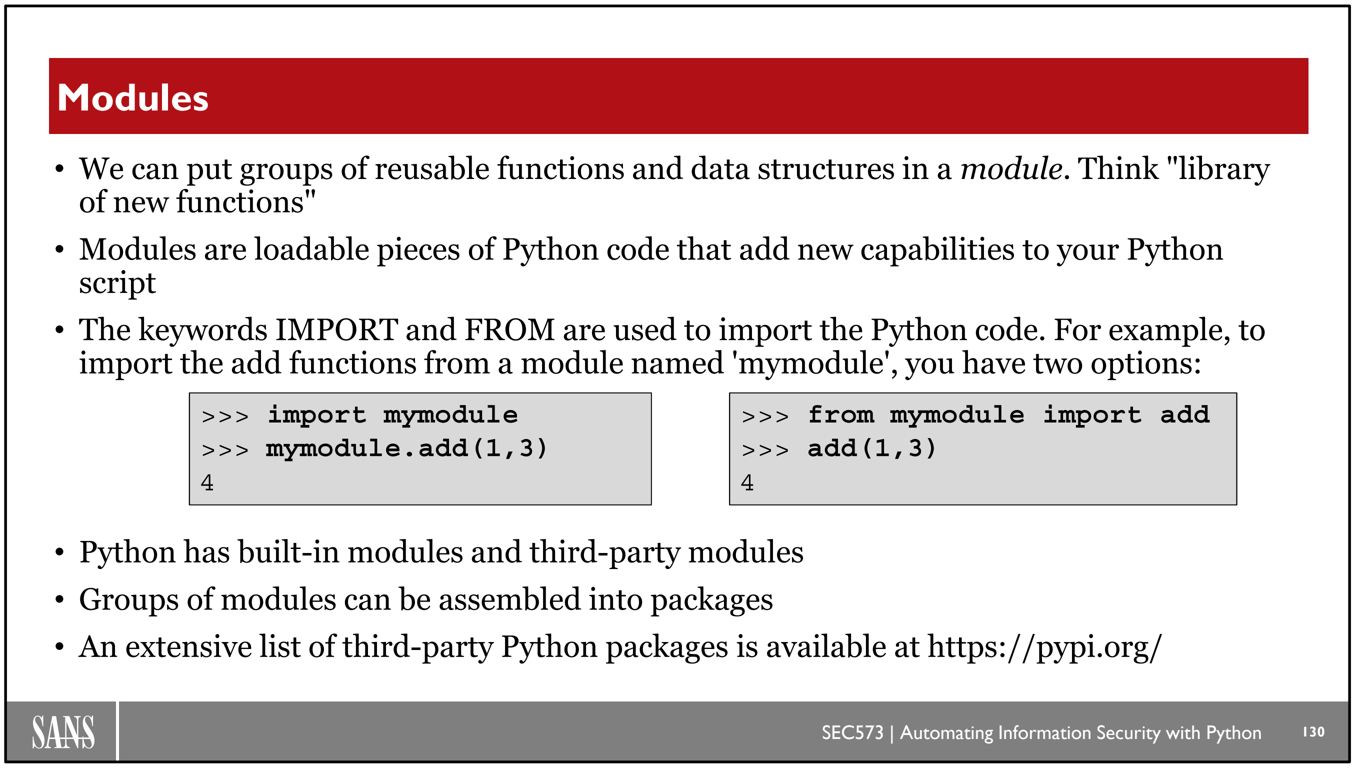
Section 1 Roadmap

- Lab Setup
- Python and pyWars
- Language Basics
- Strings
- Functions
- if/elif/else
- Modules
- Introspection

SANS

SEC573 | Automating Information Security with Python 129

This is a Roadmap slide.



Modules

- We can put groups of reusable functions and data structures in a *module*. Think "library of new functions"
- Modules are loadable pieces of Python code that add new capabilities to your Python script
- 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:

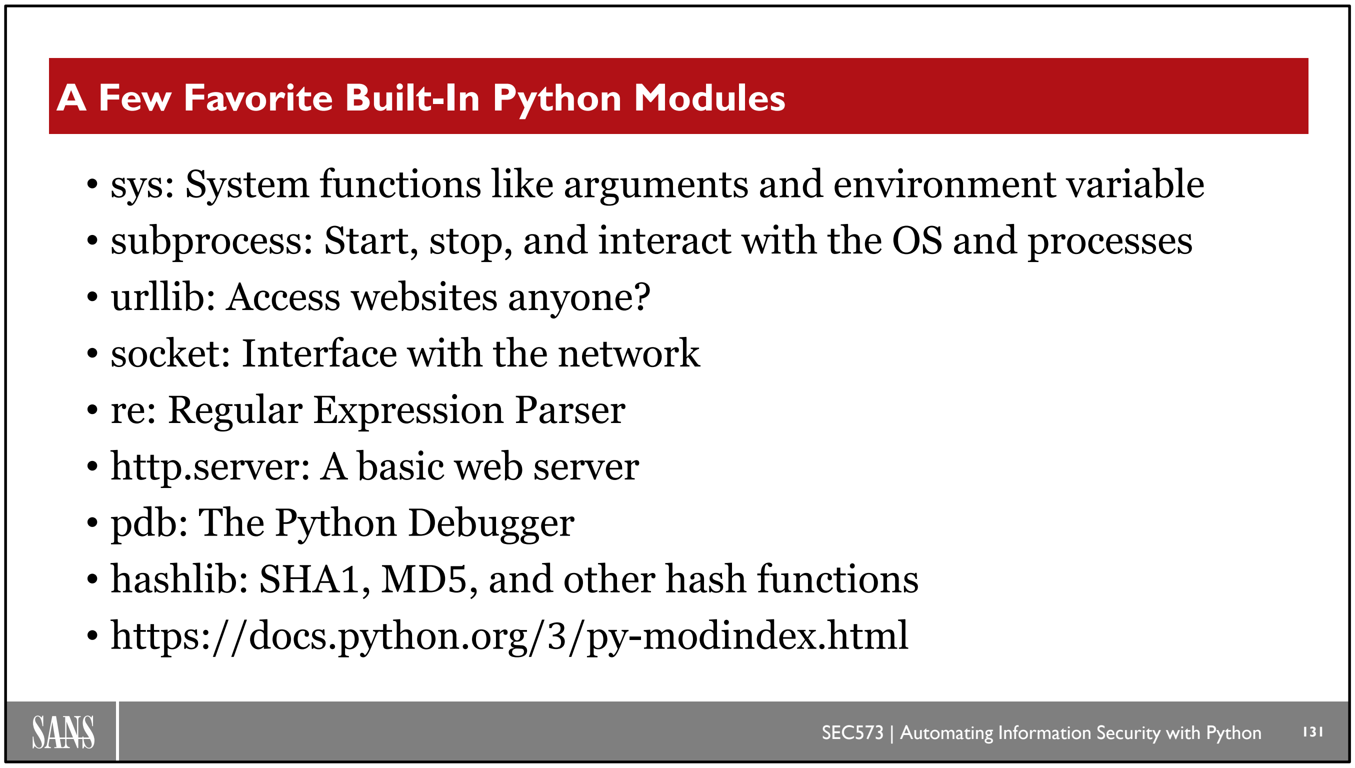
```
>>> import mymodule
>>> mymodule.add(1,3)
4
```

```
>>> 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.



A Few Favorite Built-In Python Modules

- sys: System functions like arguments and environment variable
- subprocess: Start, stop, and interact with the OS and processes
- urllib: Access websites anyone?
- 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

SANS SEC573 | Automating Information Security with Python 131

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 "urllib" module enables you to speak HTTP and HTTPS to interact with websites on the internet.
- 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.

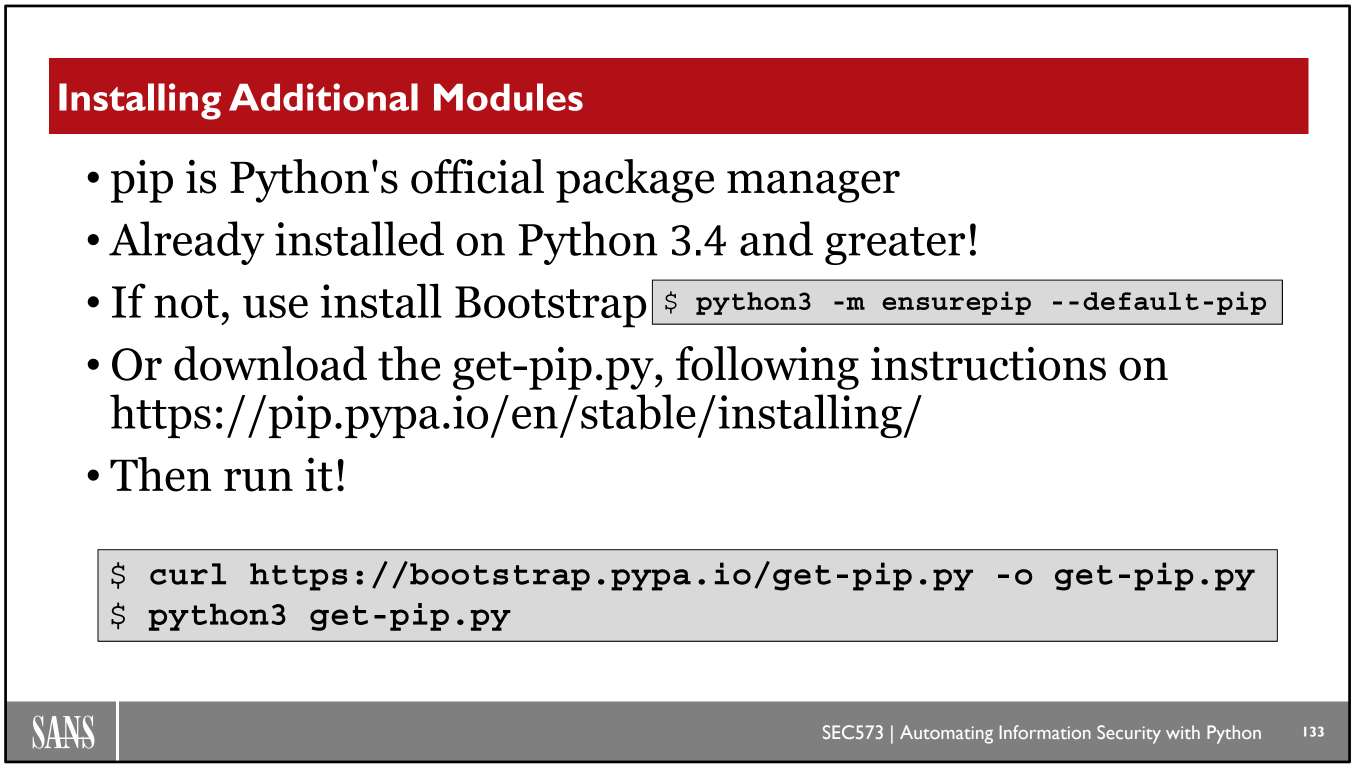
Just a Few Third-Party Modules

- 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
- We will use the following two modules in our recon tool:
  - Scapy: Parses network packet and sniffing (http://www.secdev.org/projects/scapy/)
  - PIL: Python Image Library used to parse images
- Scripts that use these modules will run only on systems with the libraries installed

Here are just a few popular modules that are available for download and installation to extend the functionality in Python:

- 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.



Installing Additional Modules

- pip is Python's official package manager
- Already installed on Python 3.4 and greater!
- If not, use install Bootstrap `$ python3 -m ensurepip --default-pip`
- Or 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
```

SANS SEC573 | Automating Information Security with Python 133

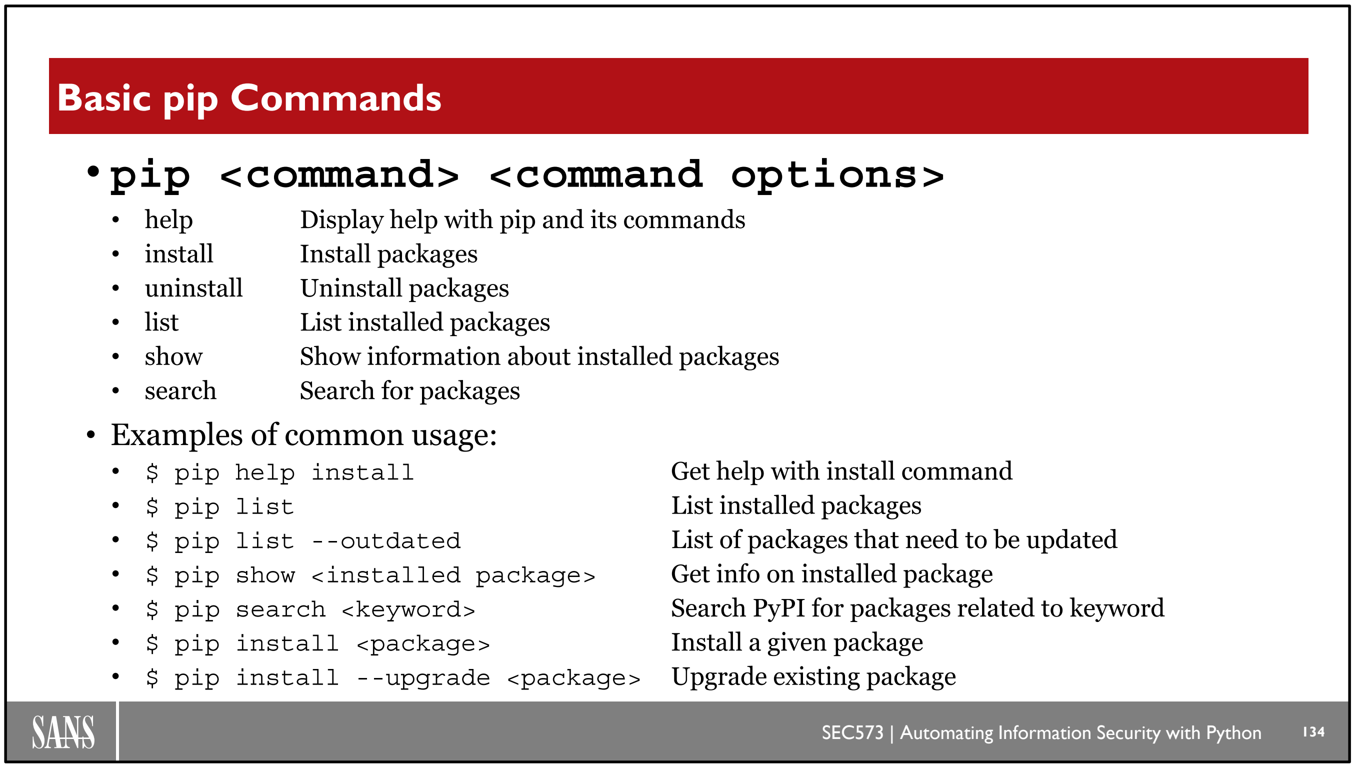
Your best option for managing your Python packages is pip. In addition to installing packages, pip enables you to uninstall a package, and it recovers gracefully if a package fails during the installation process. You can also search through the packages available in the online package repository with pip.

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
```

If that doesn't work, then install pip with the get-pip.py installation script. You download the Python script called 'get-pip.py', following the instructions on https://pip.pypa.io/en/stable/installing/. After downloading the script, you launch it with Python.

```
$ python3 get-pip.py
```



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
search	Search for 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 search <keyword>`	Search PyPI for packages related to keyword
`$ 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. pip supports the install, uninstall, list, show, and search commands. Each command has its own unique options. The 'help' command can be used to get help on any specific command. For example:

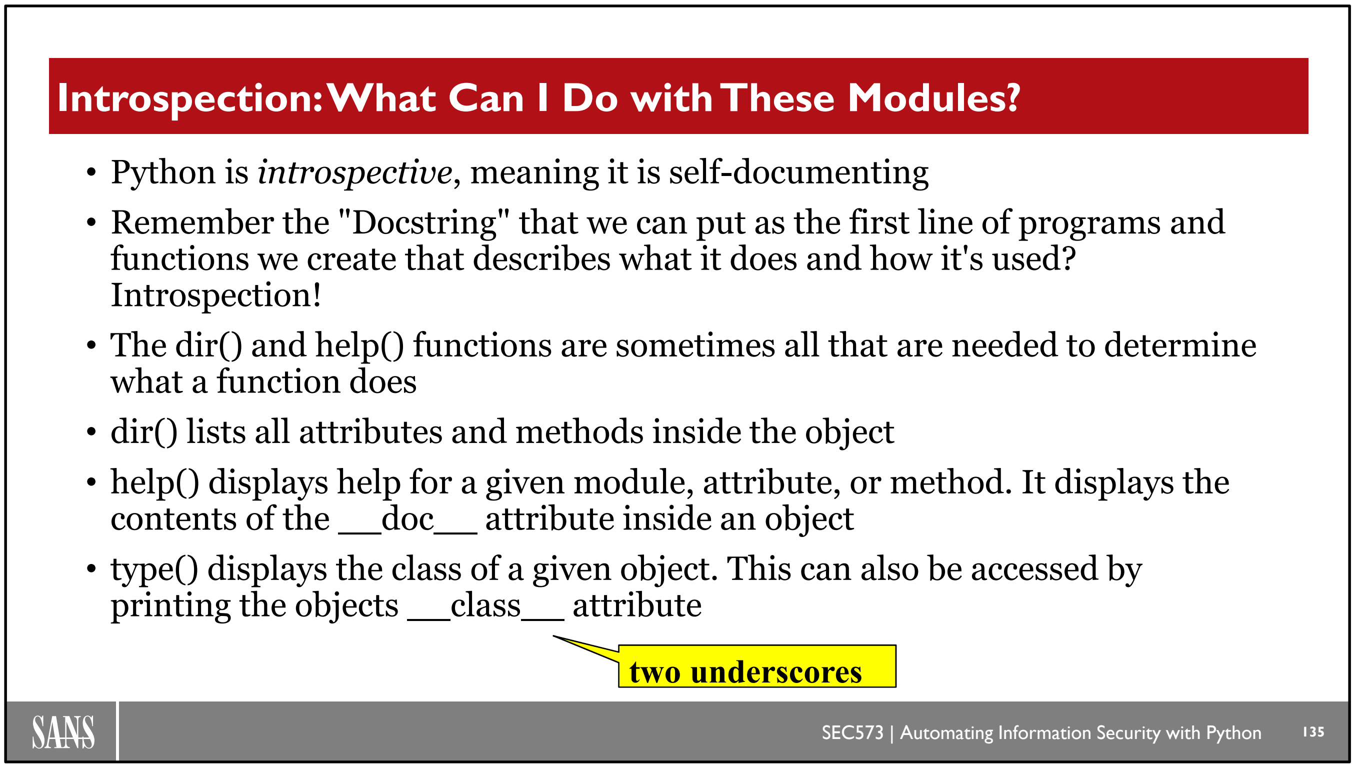
$ **pip help search**

This command will provide help on pip’s search command. The search command takes a keyword and will look at all of the packages available for download at PyPI for a match. PyPI is the official Python package repository at https://pypi.org/. For example, if you want to find a package related to Metasploit, you would type the following:

```
$ pip search metasploit
pymsfrpc (1.3)         - A Python client for the metasploit rpc
pymetasploit (1.1)     - A full-fledged msfrpc library for Metasploit framework
pymetasploit3 (1.0.1)  - A full-fledged msfrpc library for Metasploit framework
```

Then you could install the first package in the list by typing **pip install pymetasploit3**. You can also upgrade an existing installed package by adding the **--upgrade** option to the install command. For example, typing **pip install --upgrade requests** will upgrade your installed requests modules to the latest version. Conversely, you can use the uninstall command to remove already-installed packages that you no longer need. You can use the 'pip list' command to get a list of all the installed packages:

```
$ pip list --format=columns
Package                      Version
---------------------------- -------------------
asn1crypto                   0.24.0
astroid                      2.1.0
attrs                        19.1.0......
```



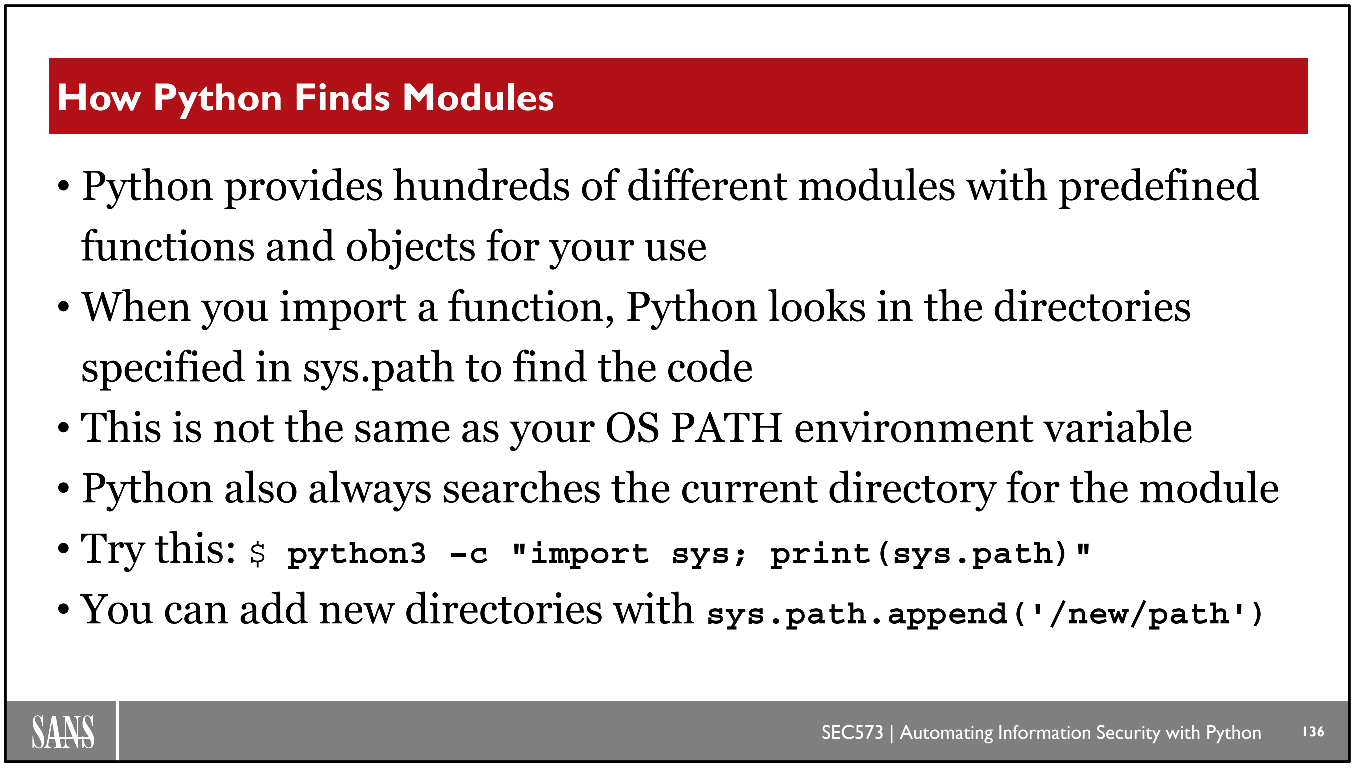
Introspection: What Can I Do with These Modules?

- Python is *introspective*, meaning it is self-documenting
- Remember the "Docstring" that we can put as the first line of programs and functions we create that describes what it does and how it's used? Introspection!
- The dir() and help() functions are sometimes all that are needed to determine what a function does
- dir() lists all attributes and methods inside the object
- help() displays help for a given module, attribute, or method. It displays the contents of the __doc__ attribute inside an object
- type() displays the class of a given object. This can also be accessed by printing the objects __class__ attribute

two underscores

SEC573 | Automating Information Security with Python 135

Python introspection can be useful when trying to determine how to use different Python modules. Remember the "Docstring" that we could provide as the first string in our program or function? Those strings along with other functions are examples of Python code being *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. The type function can be used to determine if something is an attribute or a method. This is often but not always the same as the information stored in the __class__ attribute.



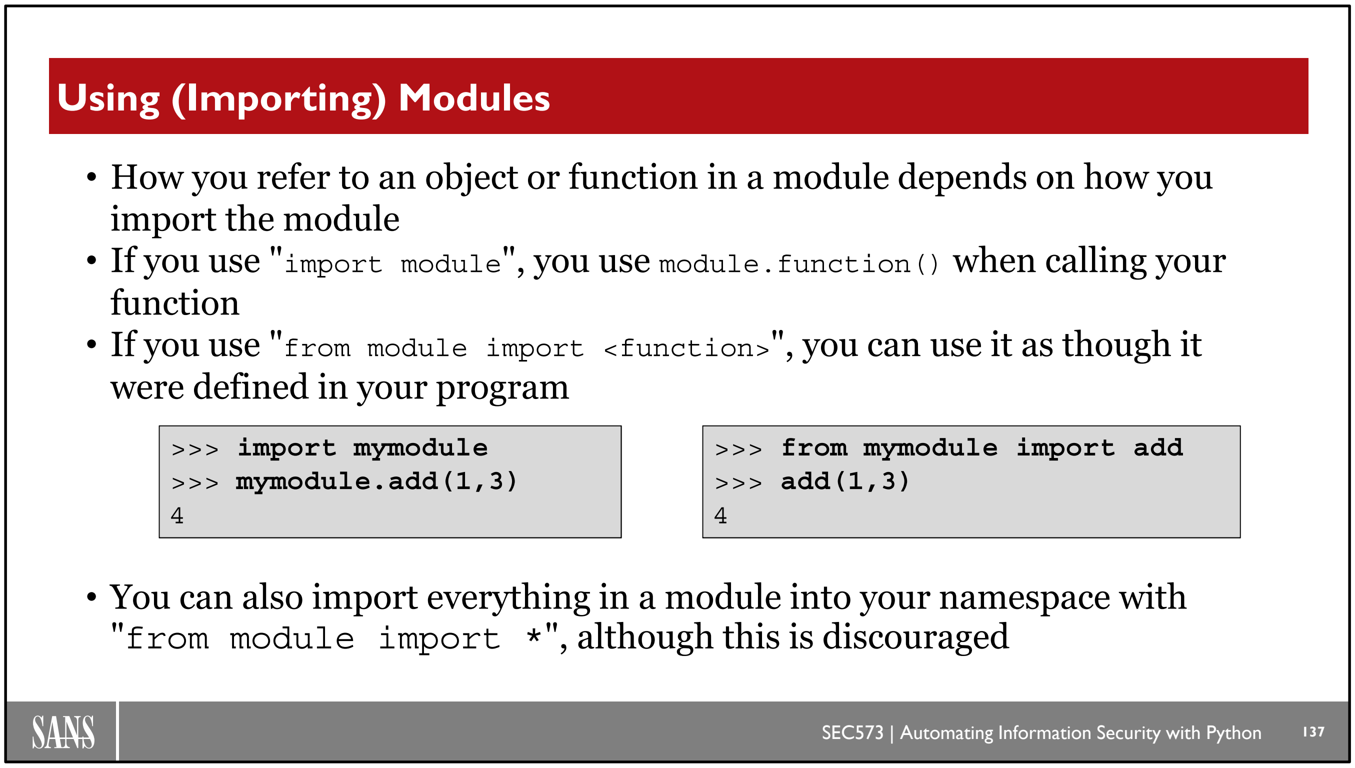
How Python Finds Modules

- 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')`

SANS SEC573 | Automating Information Security with Python 136

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.



Using (Importing) Modules

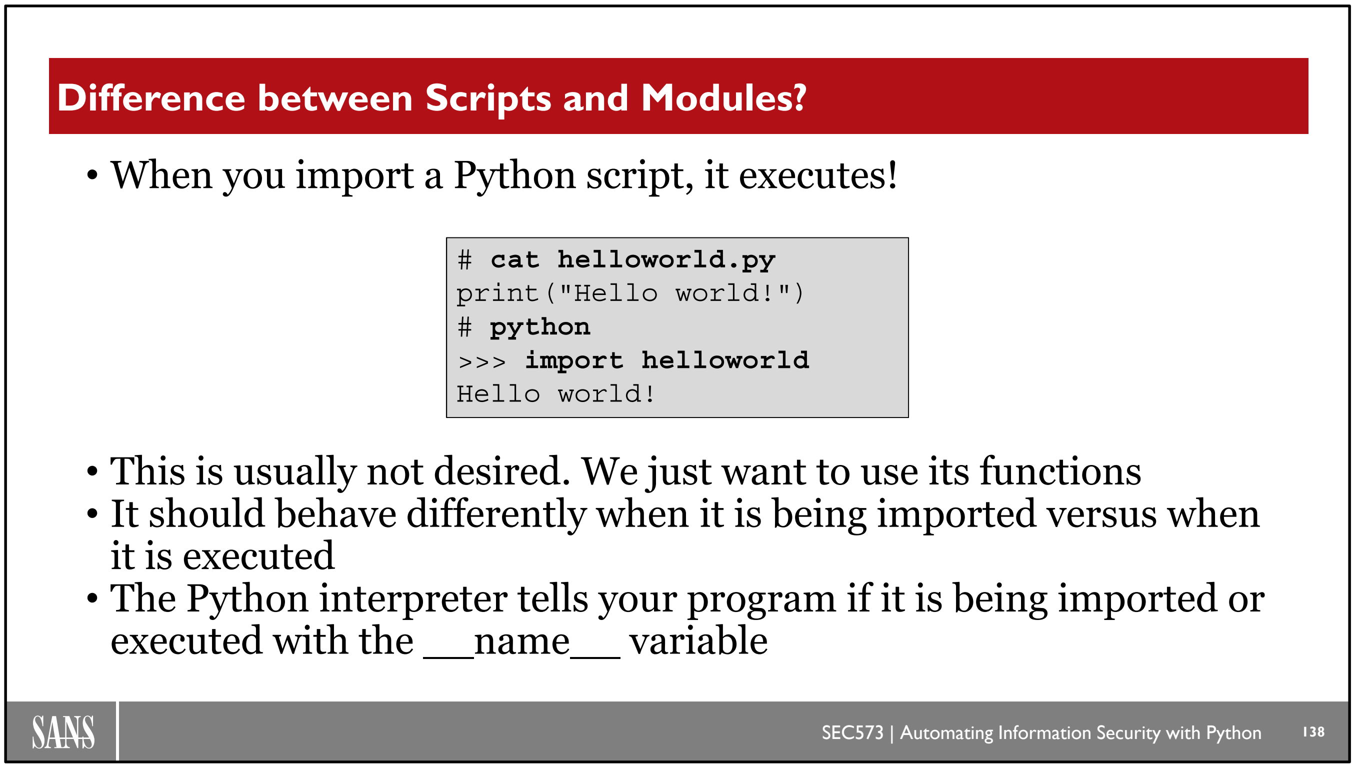
- How you refer to an object or function in a module depends on how you import the module
- If you use "`import module`", you use `module.function()` when calling your function
- If you use "`from module import <function>`", you can use it as though it were defined in your program

```
>>> 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.



Difference between Scripts and Modules?

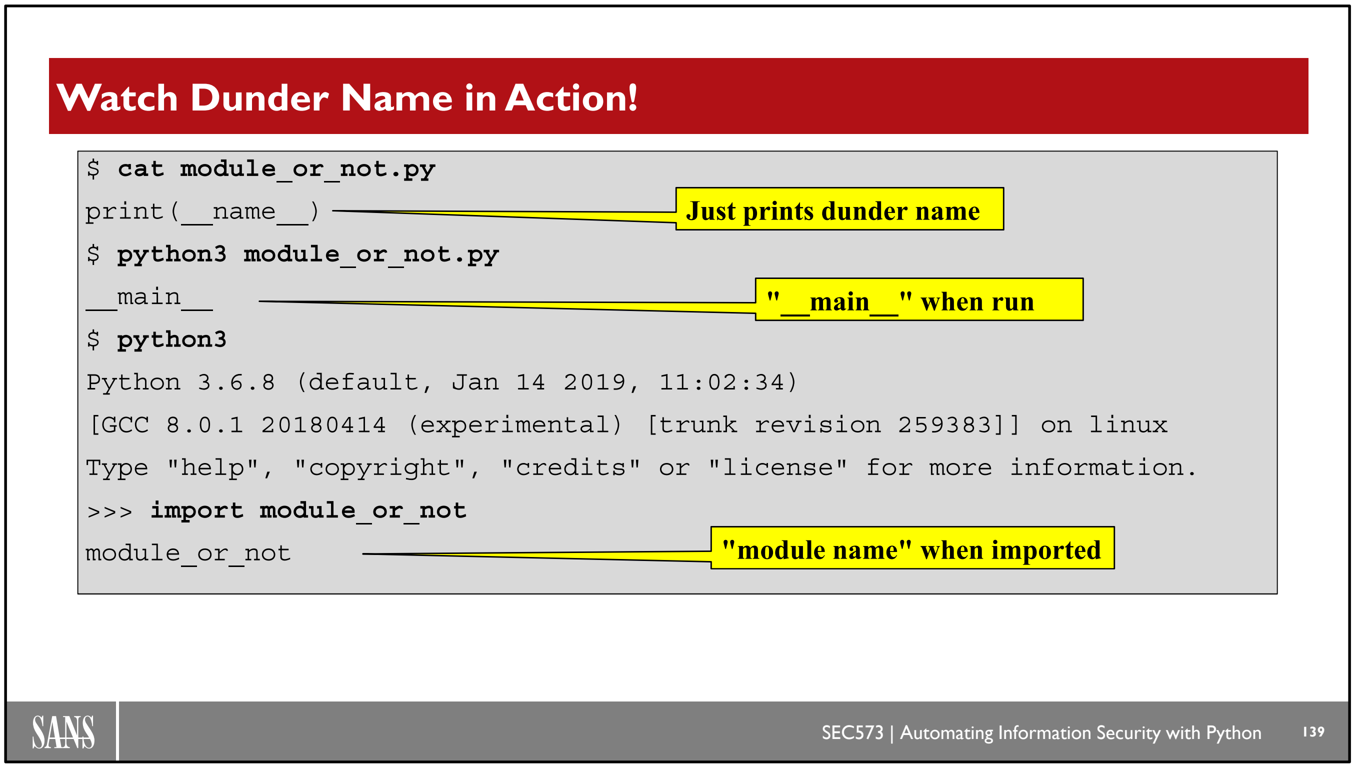
- When you import a Python script, it executes!

```
# 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

SANS SEC573 | Automating Information Security with Python 138

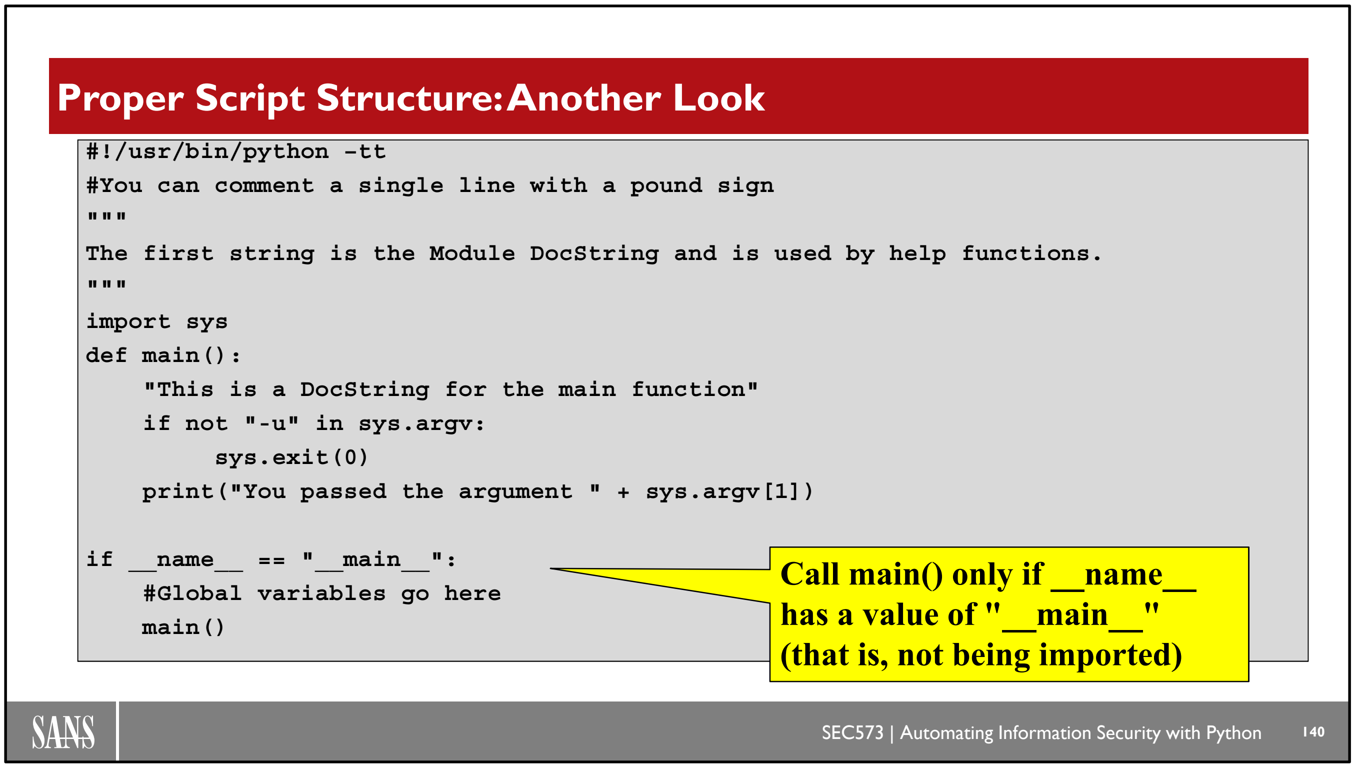
Is there any real difference between a Python module and a normal program? Well, you can import any program into another using the import statement. And when it is imported into your program, Python executes the script. This will create any functions that exist in the script and execute any code that is there also. 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.



Watch Dunder Name in Action!

```
$ cat module_or_not.py
print(__name__)
$ python3 module_or_not.py
__main__
$ python3
Python 3.6.8 (default, Jan 14 2019, 11:02:34)
[GCC 8.0.1 20180414 (experimental) [trunk revision 259383]] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import module_or_not
module_or_not
```

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 have it behave differently in each circumstance. If the script is being executed, we can have 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.



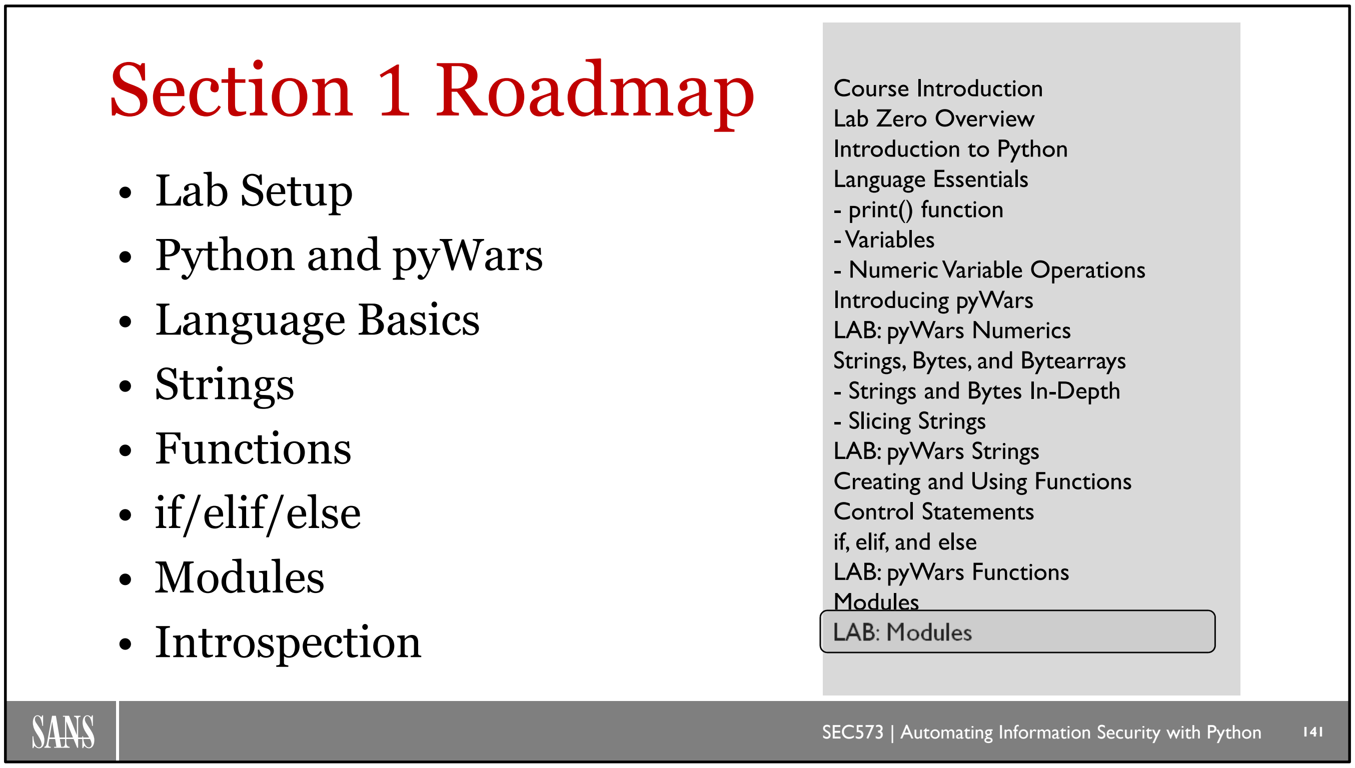
Proper Script Structure: Another Look

```
#!/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)

SANS | SEC573 | Automating Information Security with Python 140

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.

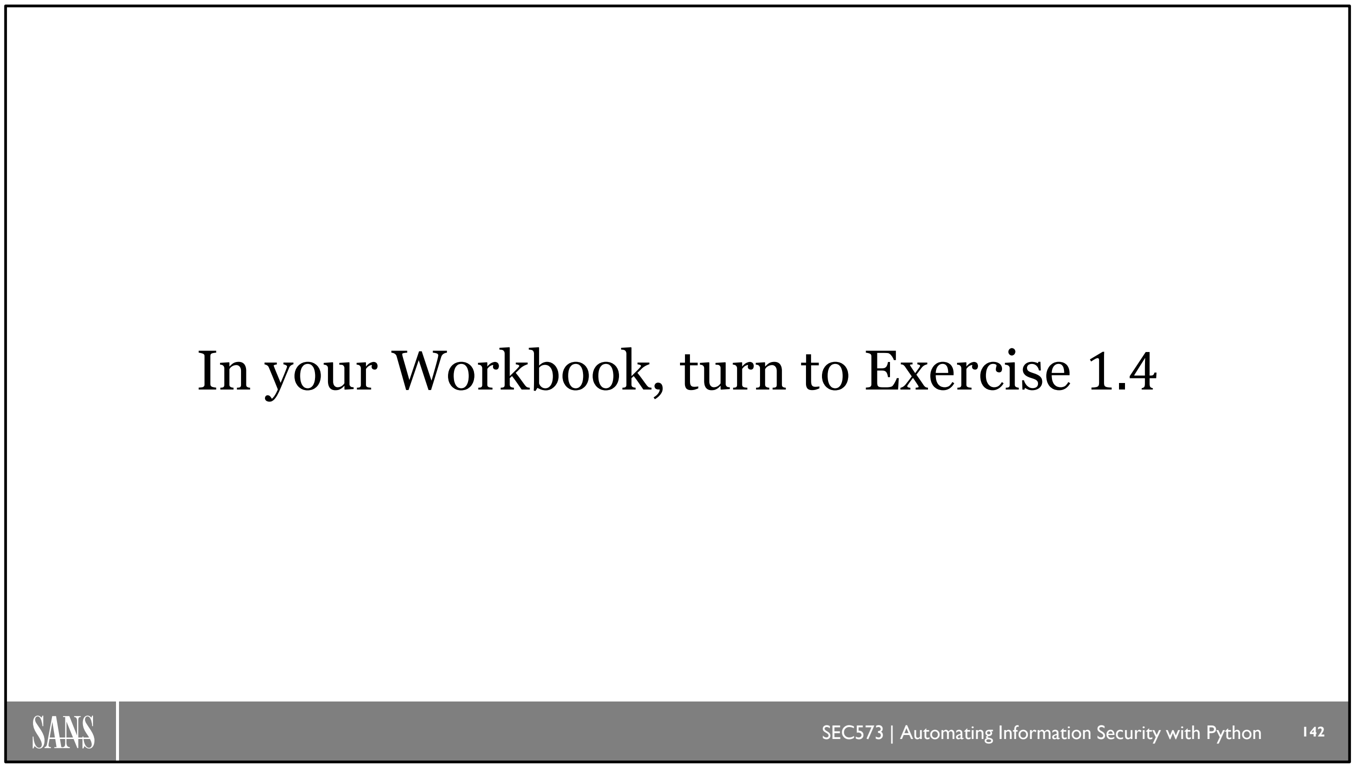


Section 1 Roadmap

- Lab Setup
- Python and pyWars
- Language Basics
- Strings
- Functions
- if/elif/else
- Modules
- Introspection

SANS

This is a Roadmap slide.



In your Workbook, turn to Exercise 1.4

SANS | SEC573 | Automating Information Security with Python 142

Please complete the exercise in your Workbook.

Lab Highlights: Written Properly, Modules Can Be Run or Imported

Your Python program behaves like a program

```
$ cd ~/Documents/pythonclass/essentials-workshop/
$ python3 pywars_answers.py
  #1 Correct!
```

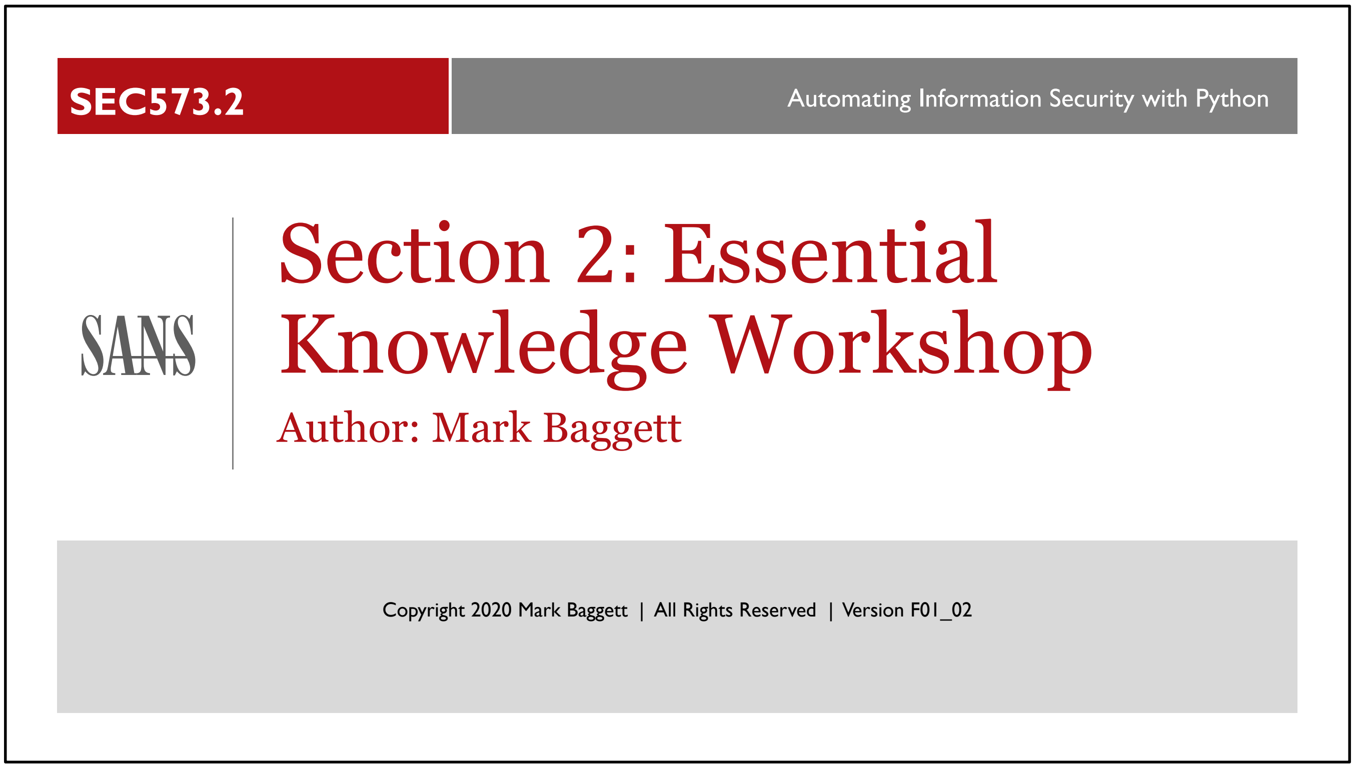
It can also be used as a module with the "import" syntax

```
$ python3
>>> import pywars_answers
>>> pywars_answers.answer1(20)
25
```

It can also be used as a module with the "from import" syntax

```
$ python3
>>> from pywars_answers import *
>>> answer1(30)
35
```

Written properly, your Python program can be used as a program or a module. This will give you the greatest level of flexibility and let you reuse your code inside of other programs.



SEC573.2

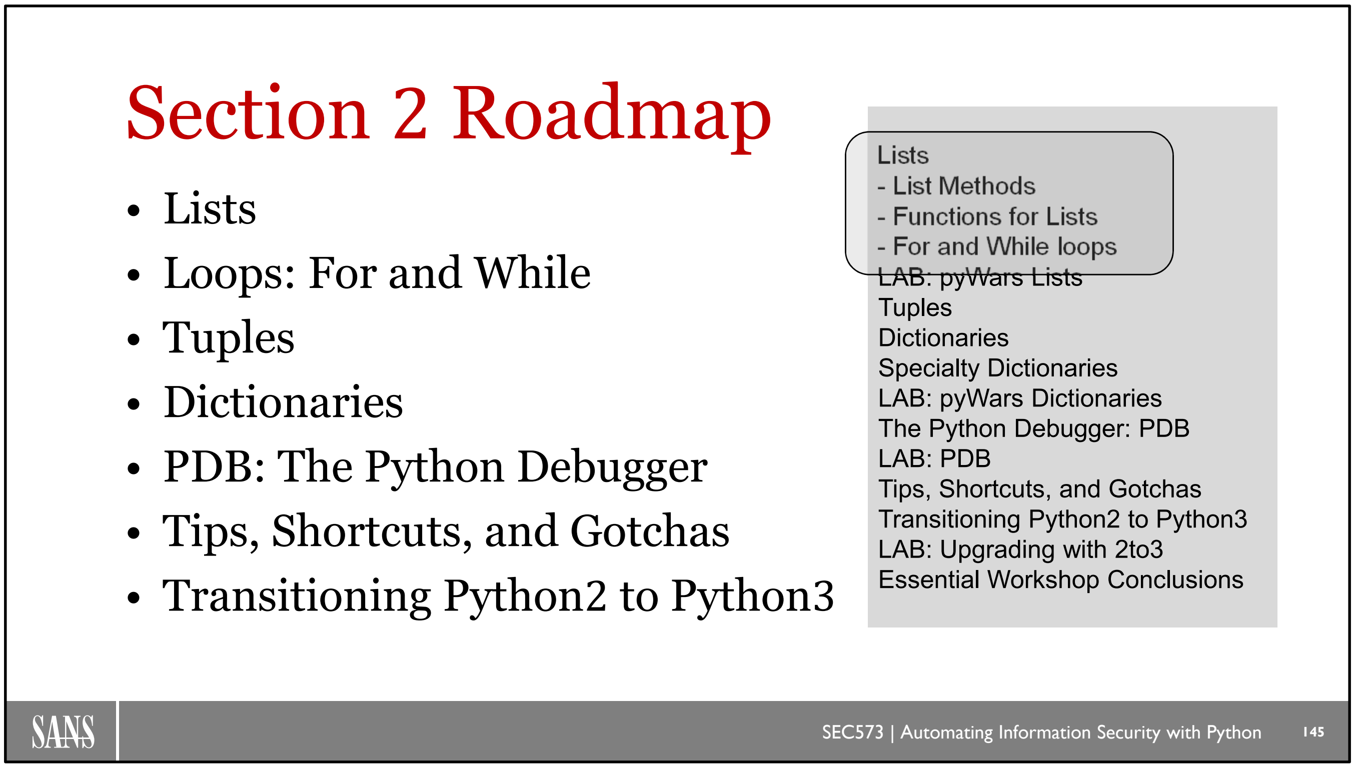
Automating Information Security with Python

SANS

Section 2: Essential Knowledge Workshop

Author: Mark Baggett

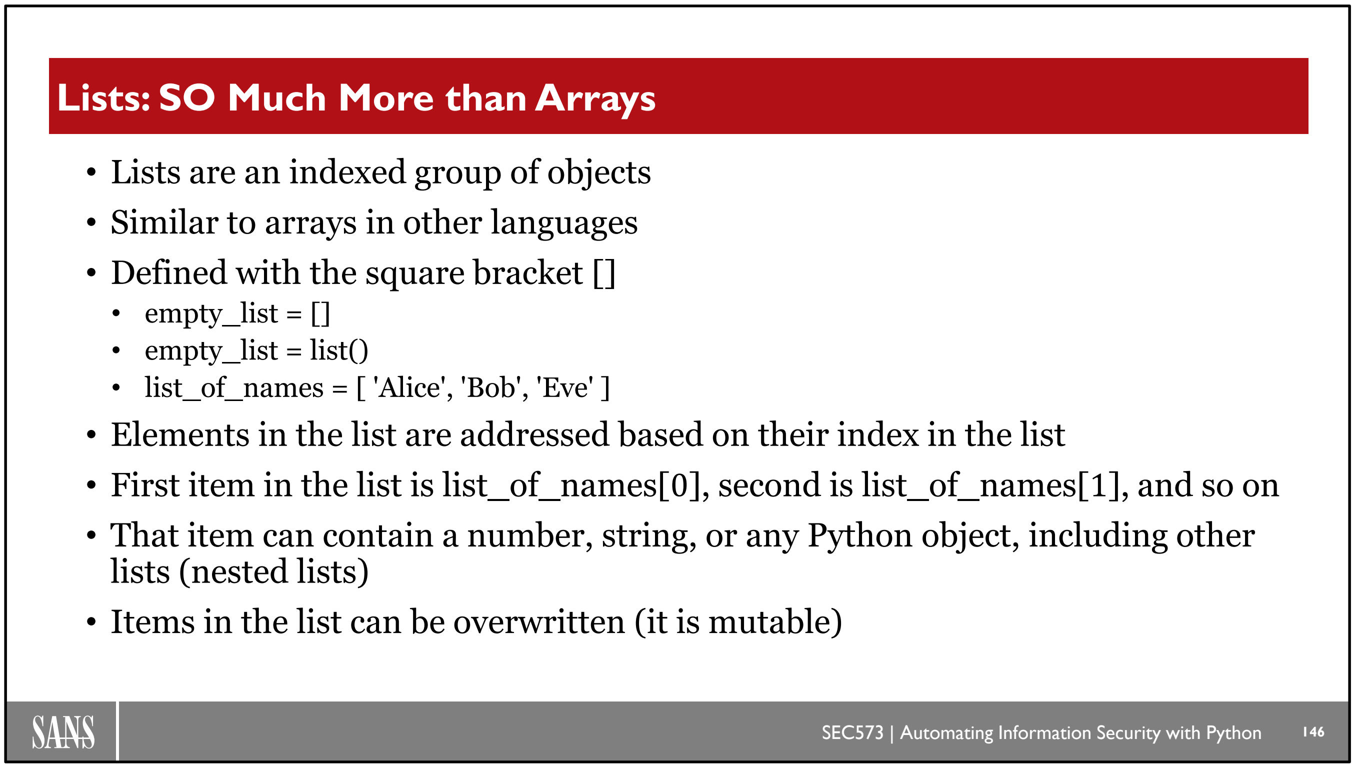
Welcome to Section 2. Today we will finish our overview of the language essentials as we discuss a few more important data and control structures of the Python language.



Section 2 Roadmap

- Lists
- Loops: For and While
- Tuples
- Dictionaries
- PDB: The Python Debugger
- Tips, Shortcuts, and Gotchas
- Transitioning Python2 to Python3

This is a Roadmap slide.



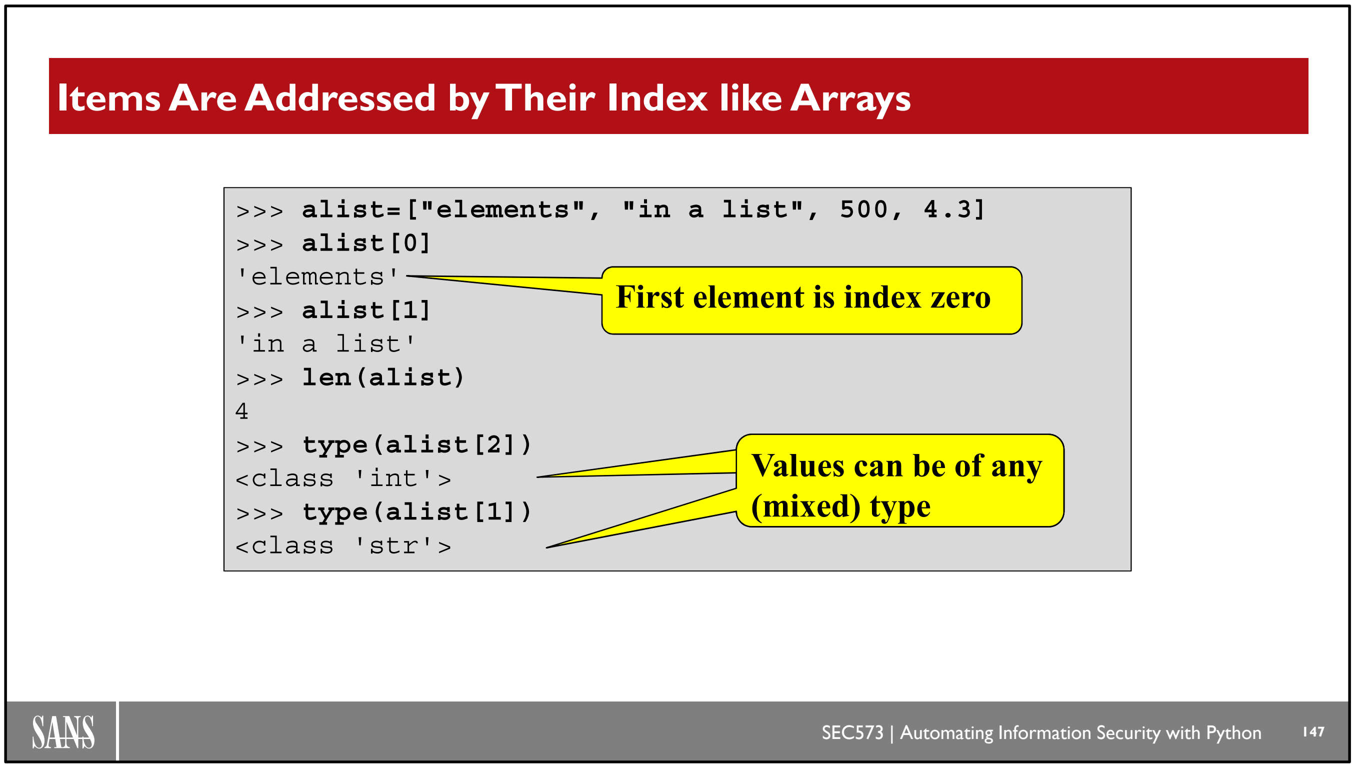
Lists: SO Much More than Arrays

- 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*.



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'>
```

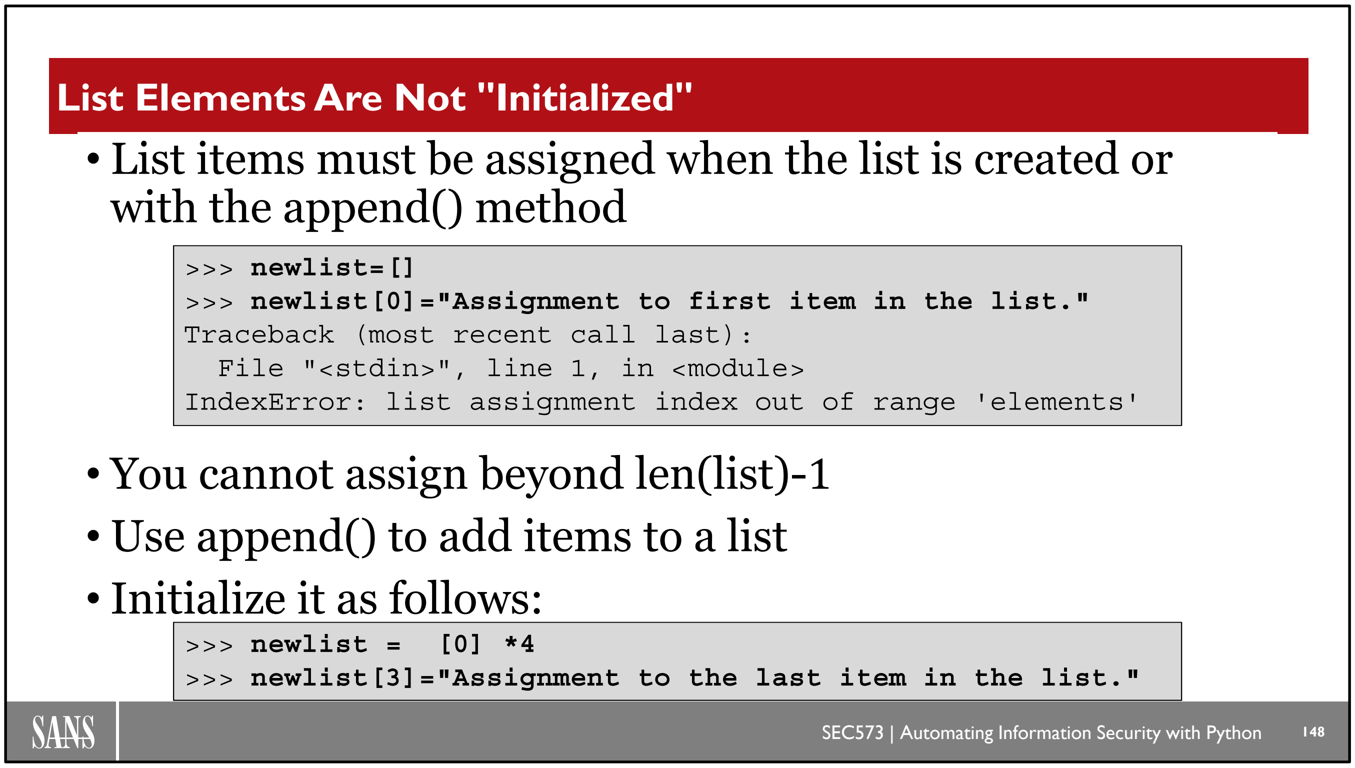
First element is index zero

Values can be of any (mixed) type

SANS | SEC573 | Automating Information Security with Python 147

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.



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 'elements'
```

- 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."
```

SANS SEC573 | Automating Information Security with Python 148

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:

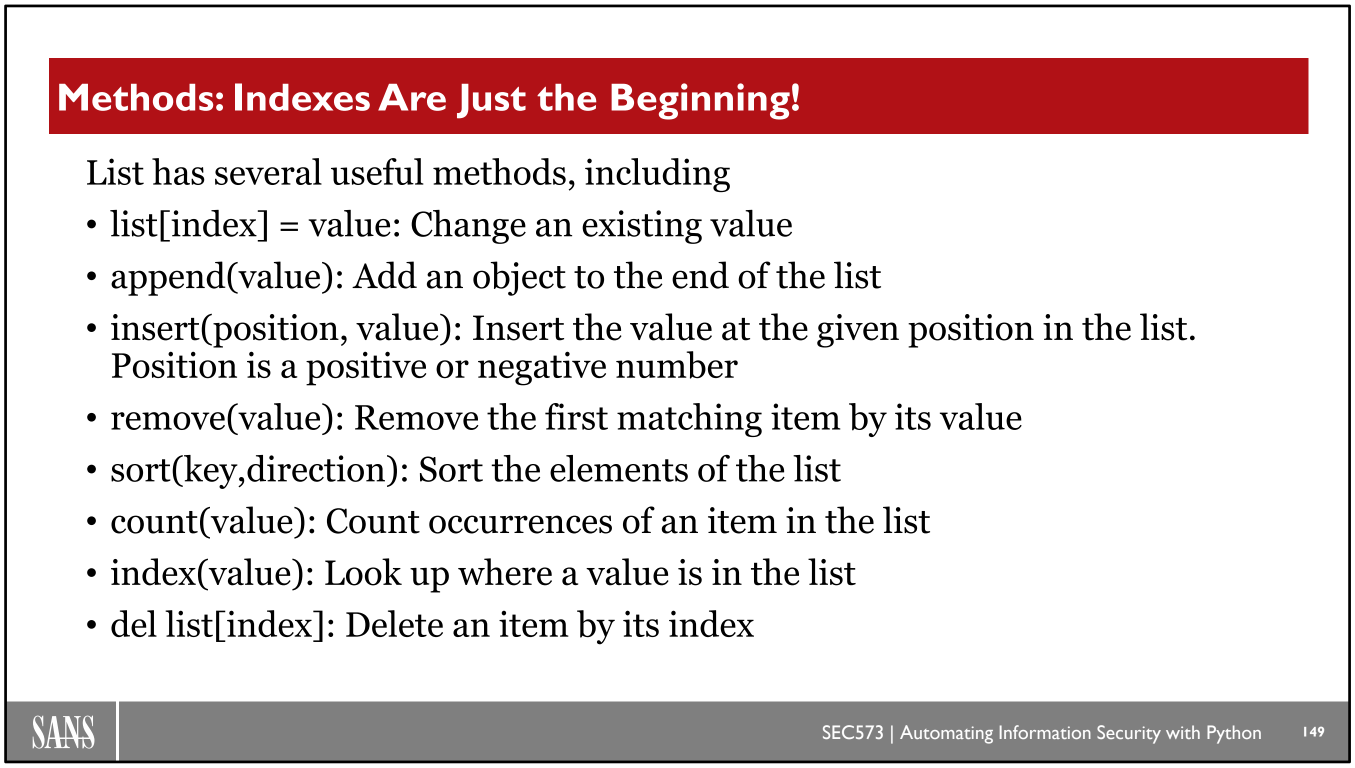
```
>>> newlist =   [default value] * <size of the "array">
```

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
```



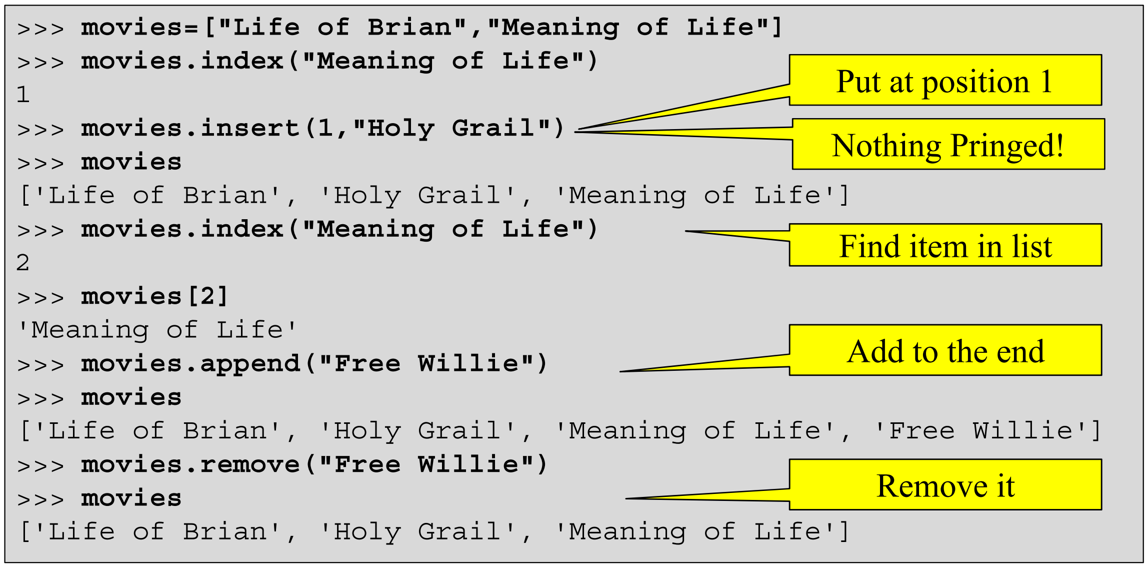
Methods: Indexes Are Just the Beginning!

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

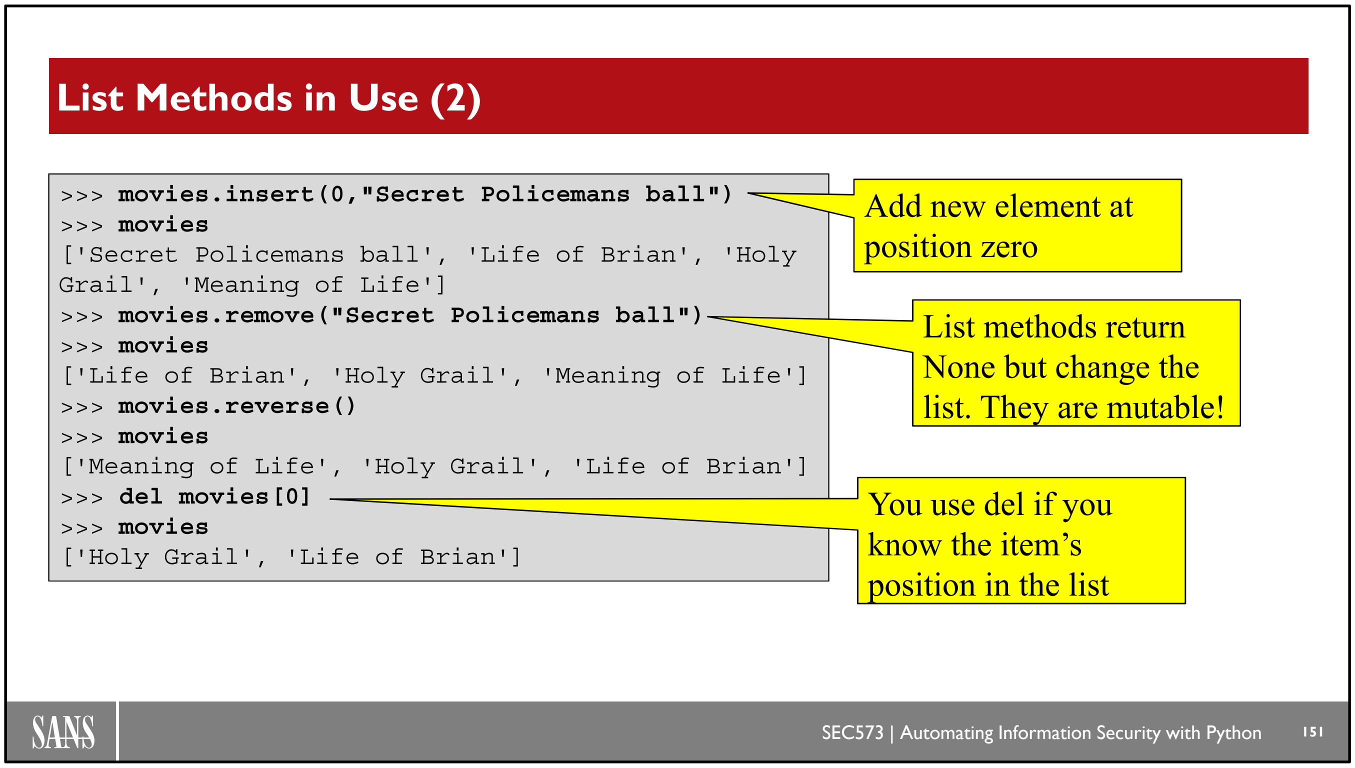
Lists are much more than arrays. Lists are objects with several methods available to manage the data contained in the list. Of course, we have already seen that we can assign individual elements in a list using their index. You can use the list's append() method to add things to the end of the list. 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.

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']
```

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.



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']
```

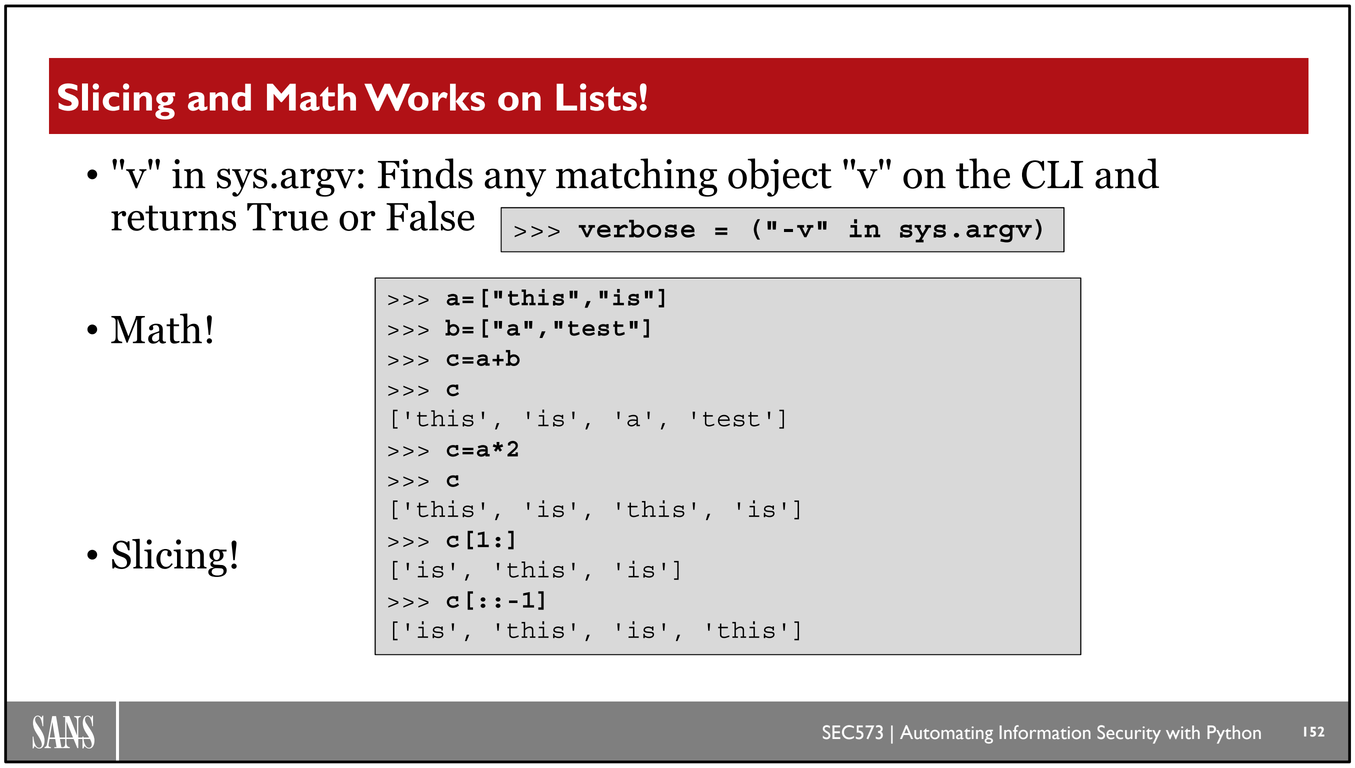
Add new element at position zero

List methods return None but change the list. They are mutable!

You use del if you know the item's position in the list

SANS SEC573 | Automating Information Security with Python 151

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.



Slicing and Math Works on Lists!

- "v" in sys.argv: Finds any matching object "v" on the CLI and returns True or False

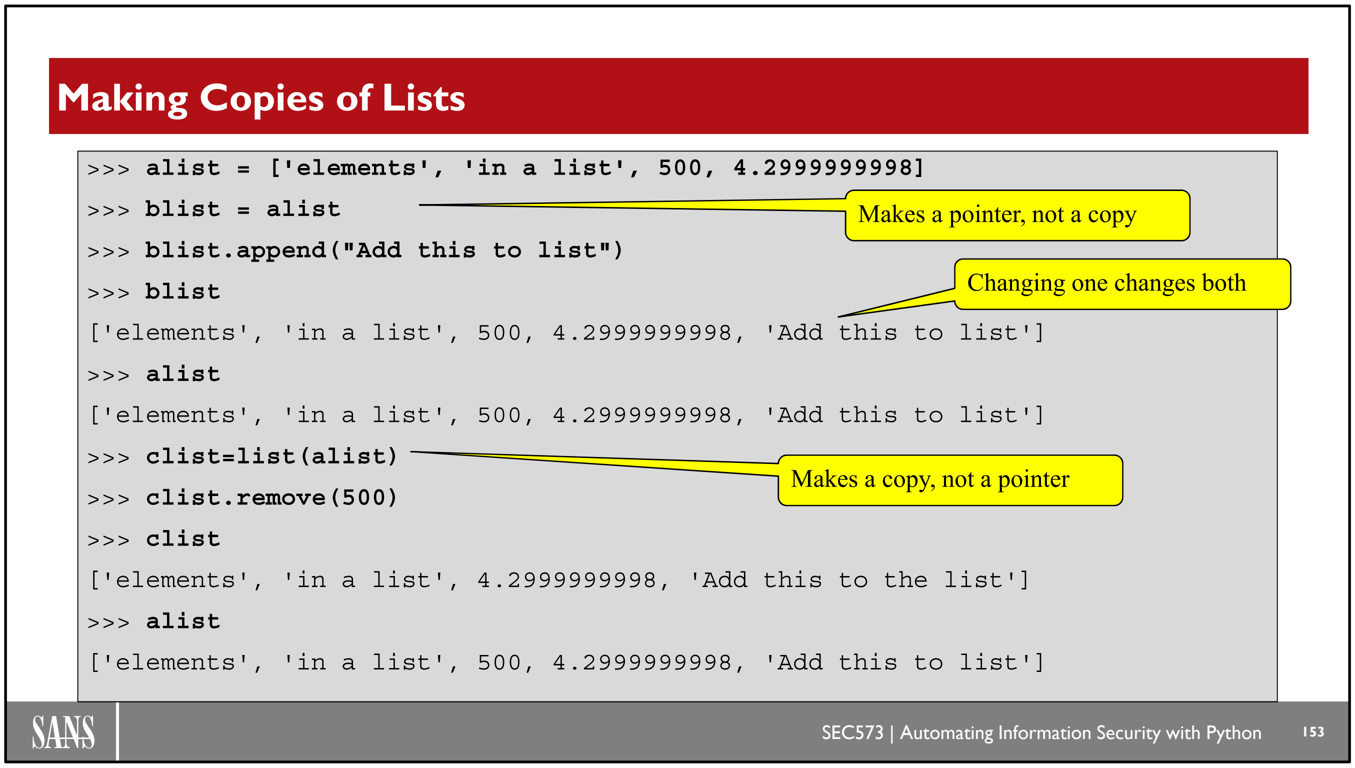
```
>>> verbose = ("-v" in sys.argv)
```

- 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']
```

SANS SEC573 | Automating Information Security with Python 152

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 zero and go until the end.” All of the same rules apply with lists that apply to strings, including negative numbers and stepping.



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.

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 character or characters

```
>>> "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 and tabs. 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 or a substring. The argument used to split the string will not appear in the resulting list.

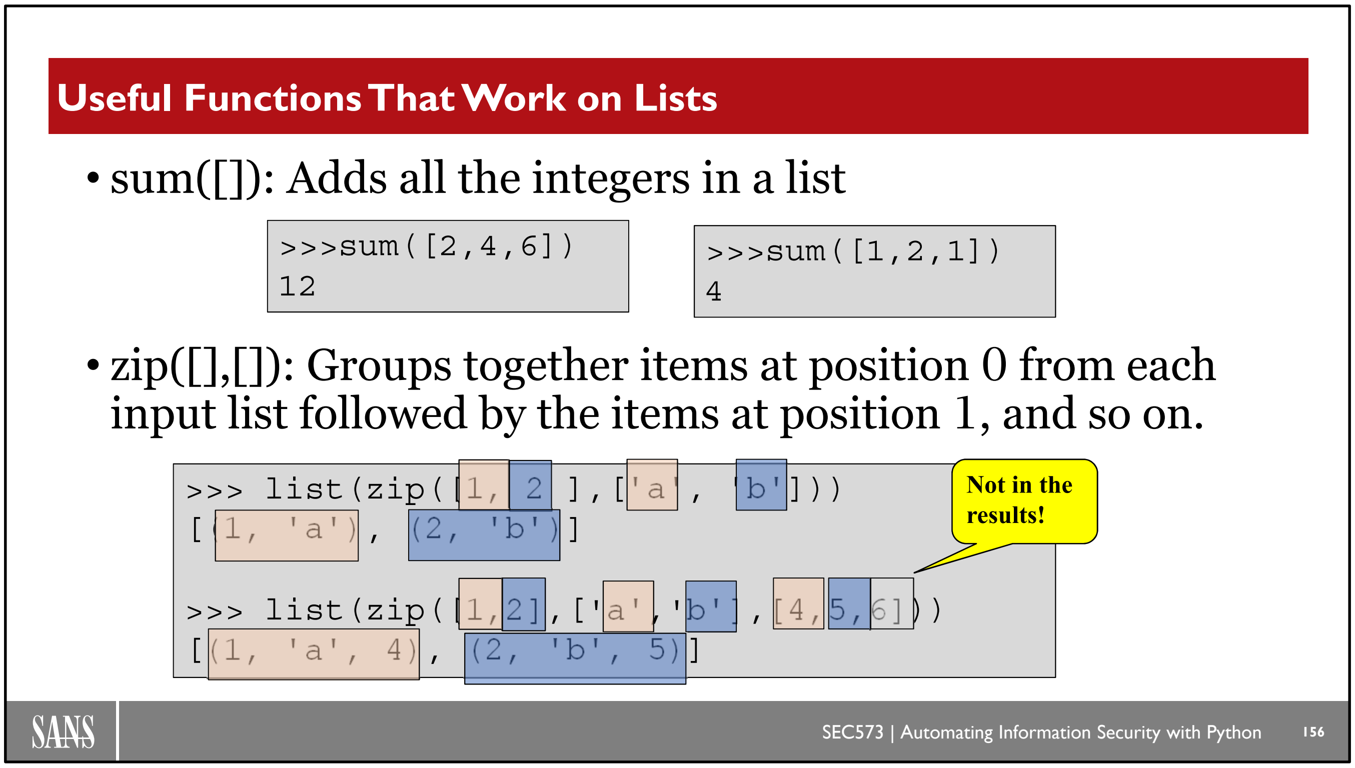
Convert Lists to Strings with "".join()

- 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.



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.

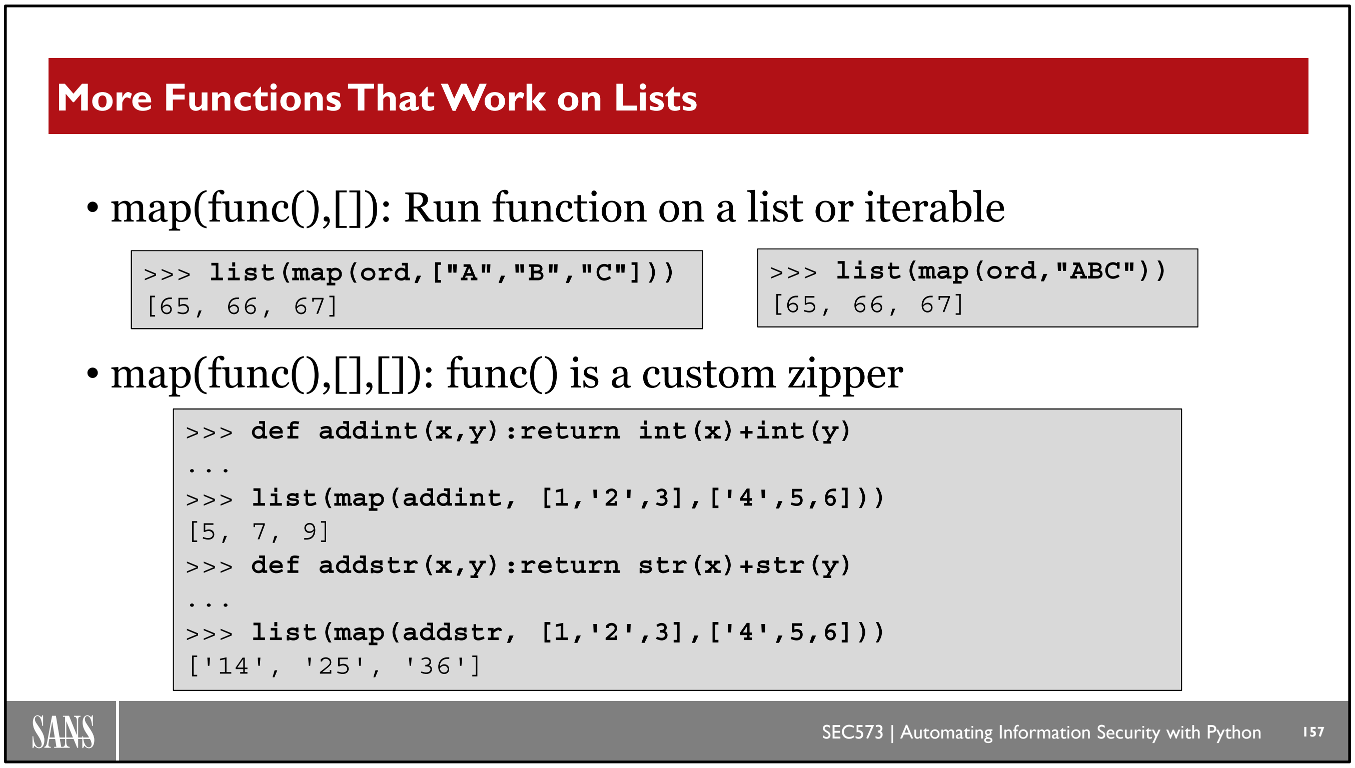
Consider these two examples. First, we zip two lists together. Each list has two items in it.

```
>>> 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.



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

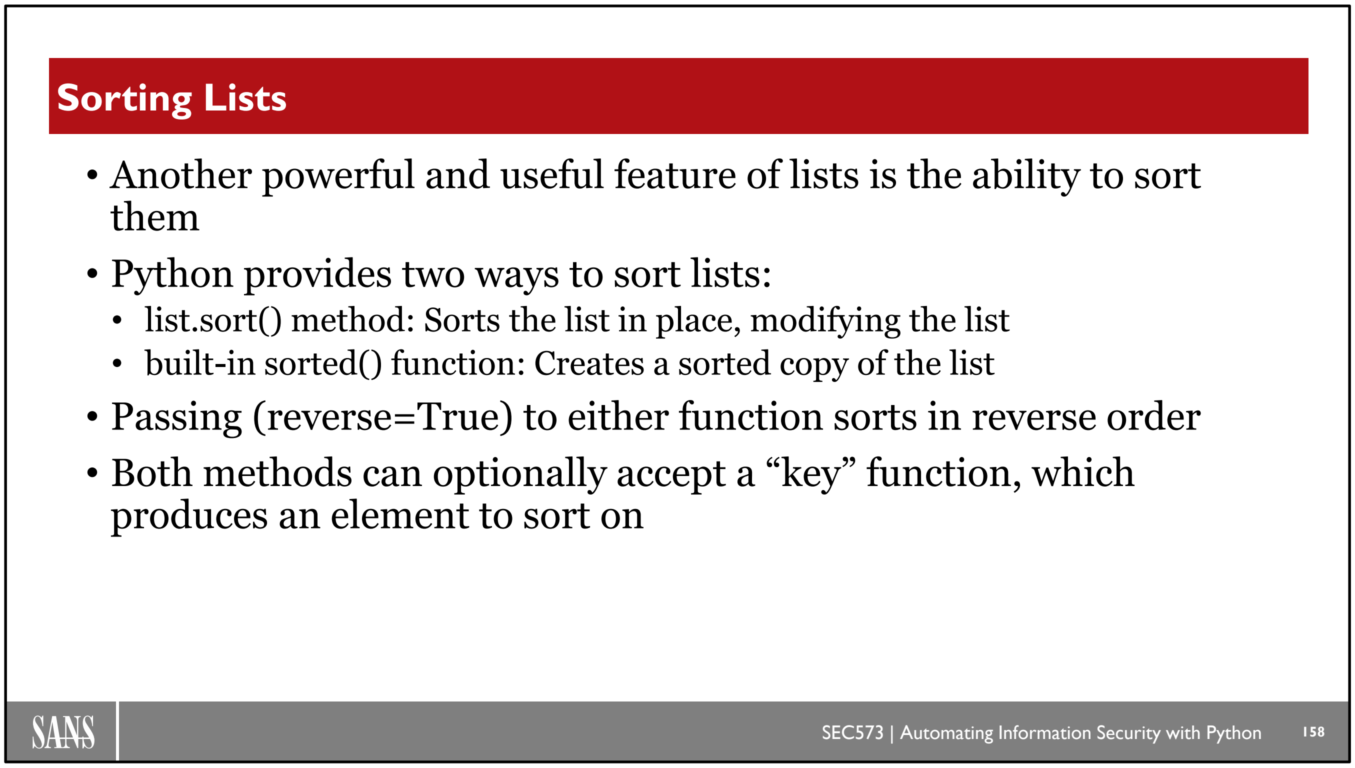
```
>>> def addint(x,y):return int(x)+int(y)
...
>>> list(map(addint, [1,'2',3],['4',5,6]))
[5, 7, 9]
>>> def addstr(x,y):return str(x)+str(y)
...
>>> list(map(addstr, [1,'2',3],['4',5,6]))
['14', '25', '36']
```

SANS | SEC573 | Automating Information Security with Python 157

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.



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

SANS SEC573 | Automating Information Security with Python 158

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]
```

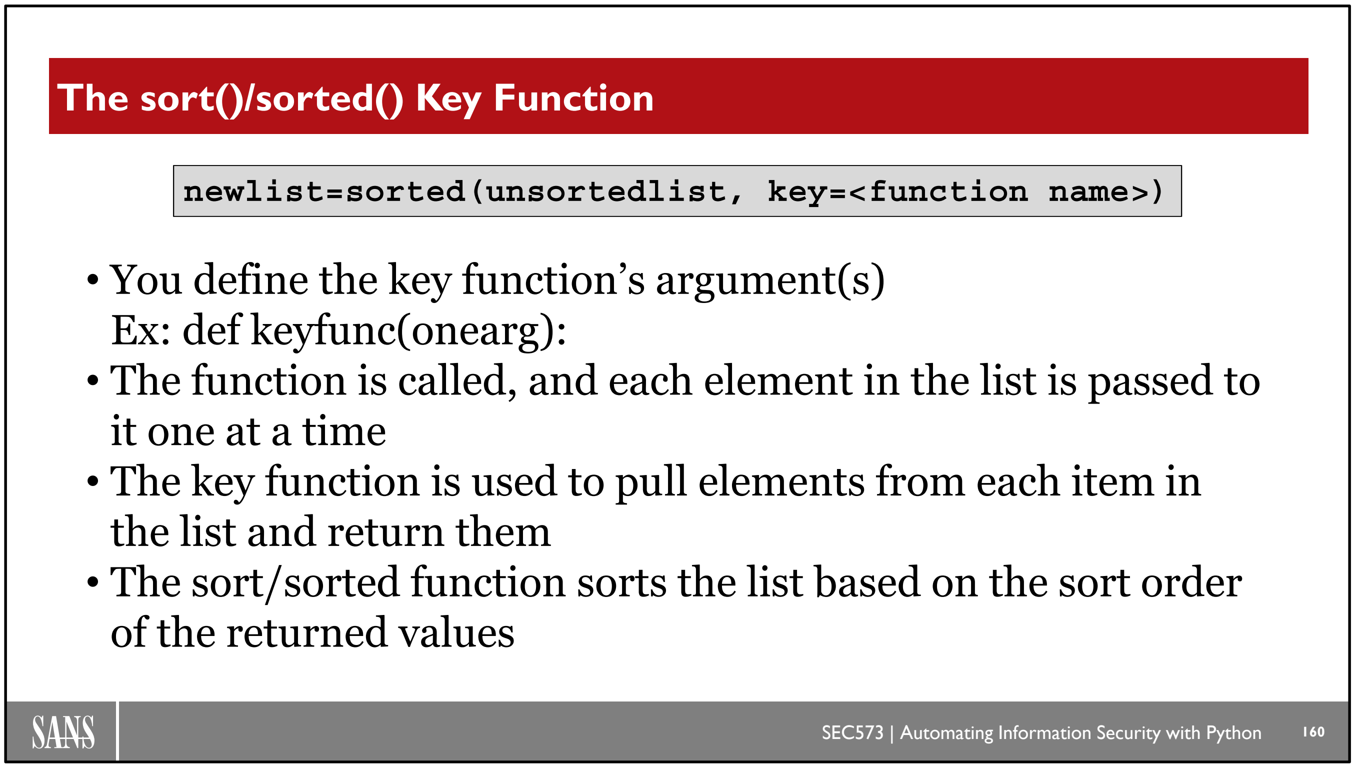
Often, you need to sort on something other than just the first letter of the data element. Python enables you to specify a "key" function to be used by sort() and sorted() to determine the order of elements.

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 man 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.



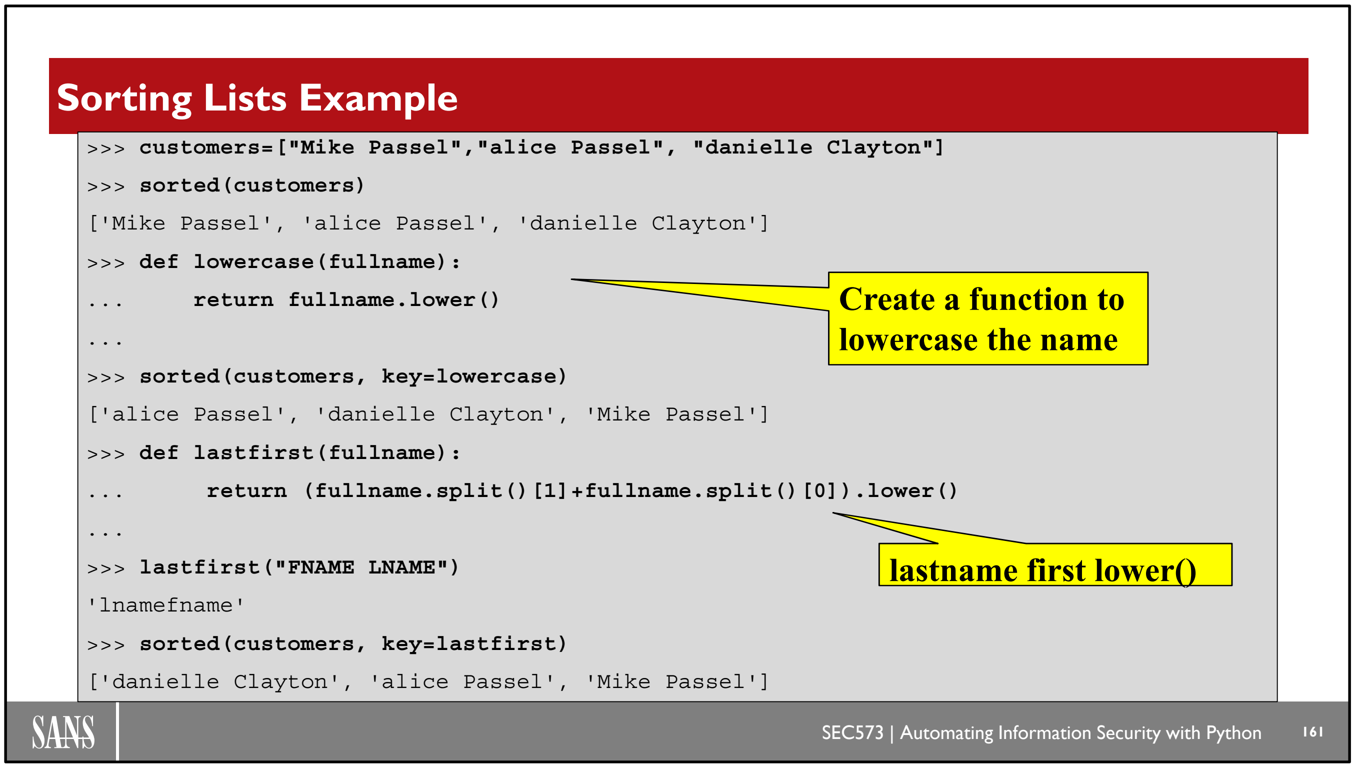
The sort()/sorted() Key Function

```
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

SANS SEC573 | Automating Information Security with Python 160

The key function is passed 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.



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']
```

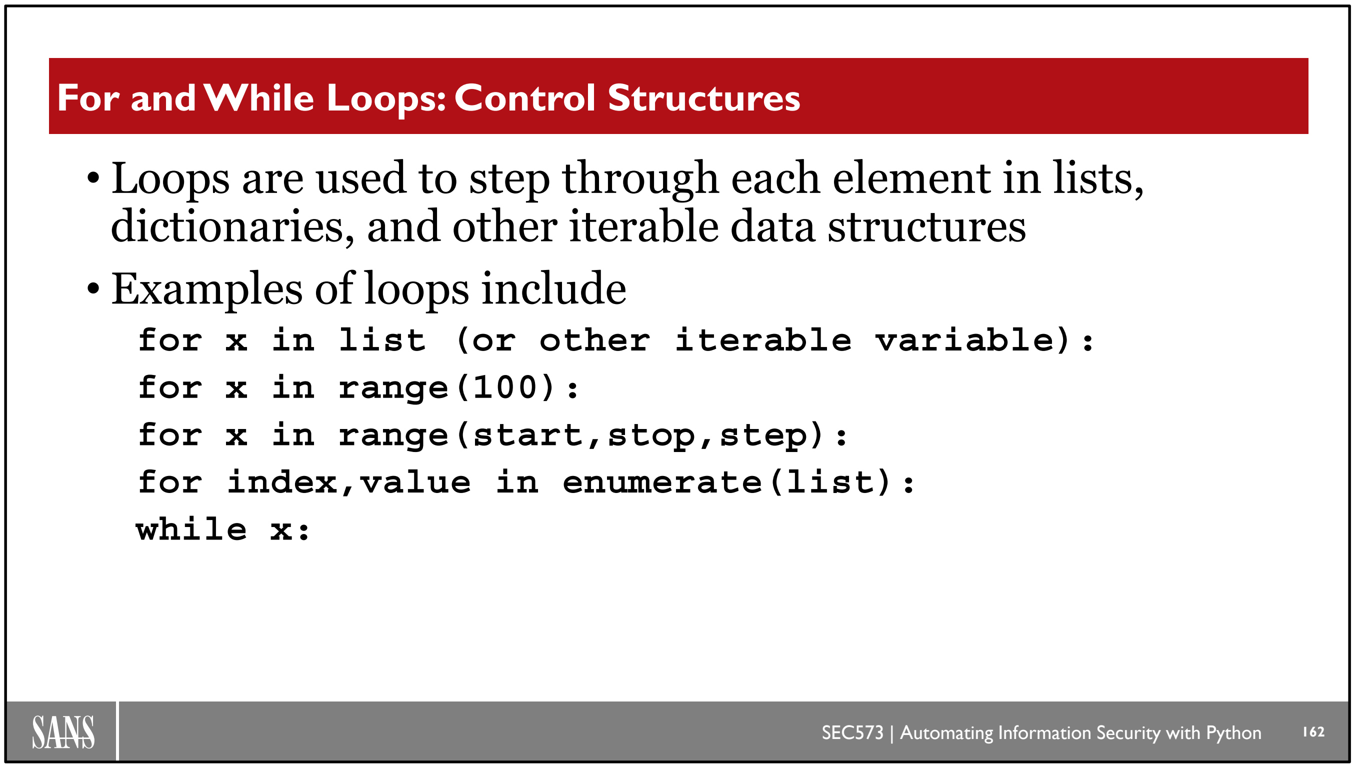
Here is a simple example of how we could use our key function to sort on something other than just an alphabetical listing of the data elements. Suppose we wanted to sort based on the last name of a list of customers. By default, sort will sort from left to right. Usually, when we sort text, we want 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 lastname(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 lowercased 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.



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:
```

SANS | SEC573 | Automating Information Security with Python 162

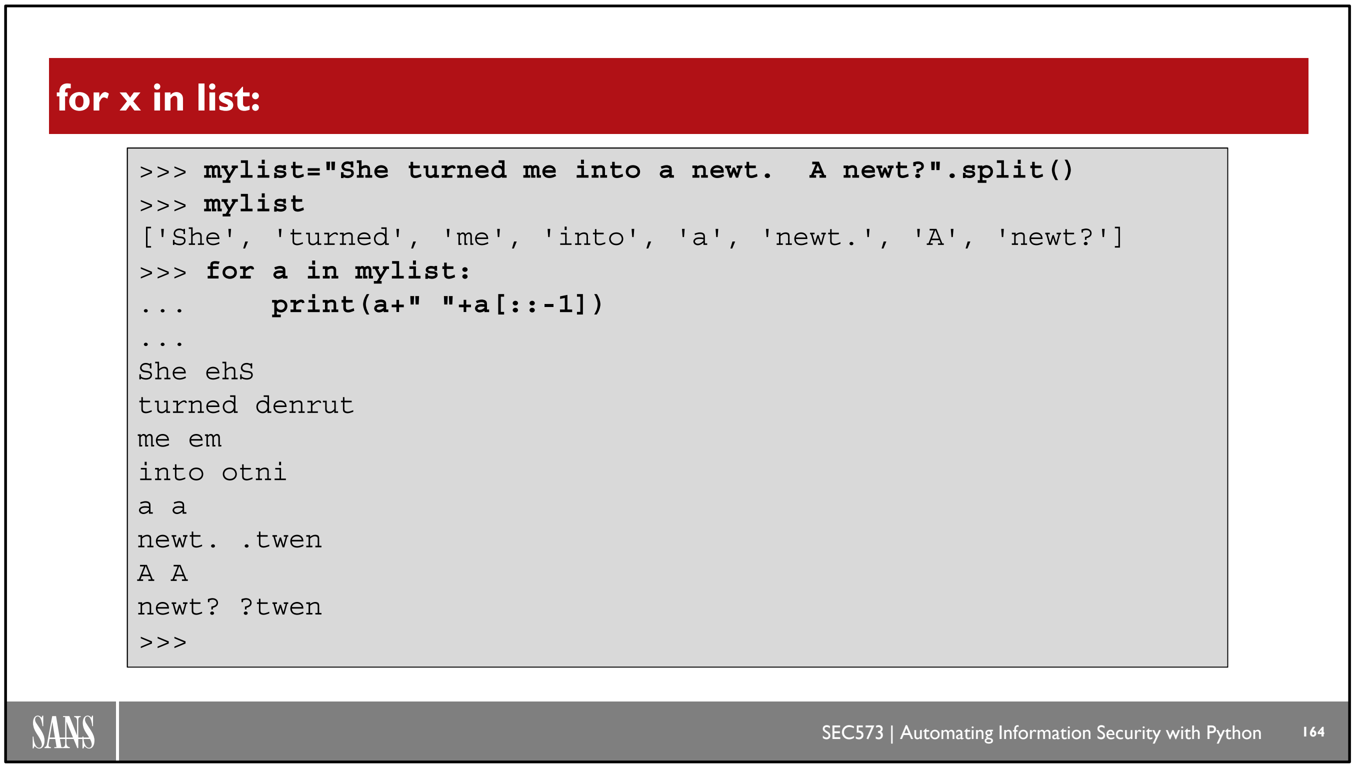
FOR loops and WHILE loops are used to step through elements of a list, dictionary, or other iterable data structure. Using a for or a while loop, you define a block of code that you want to execute for each element in a data structure. We will look at a couple of different ways to use loops over the next few pages.

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)]

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.



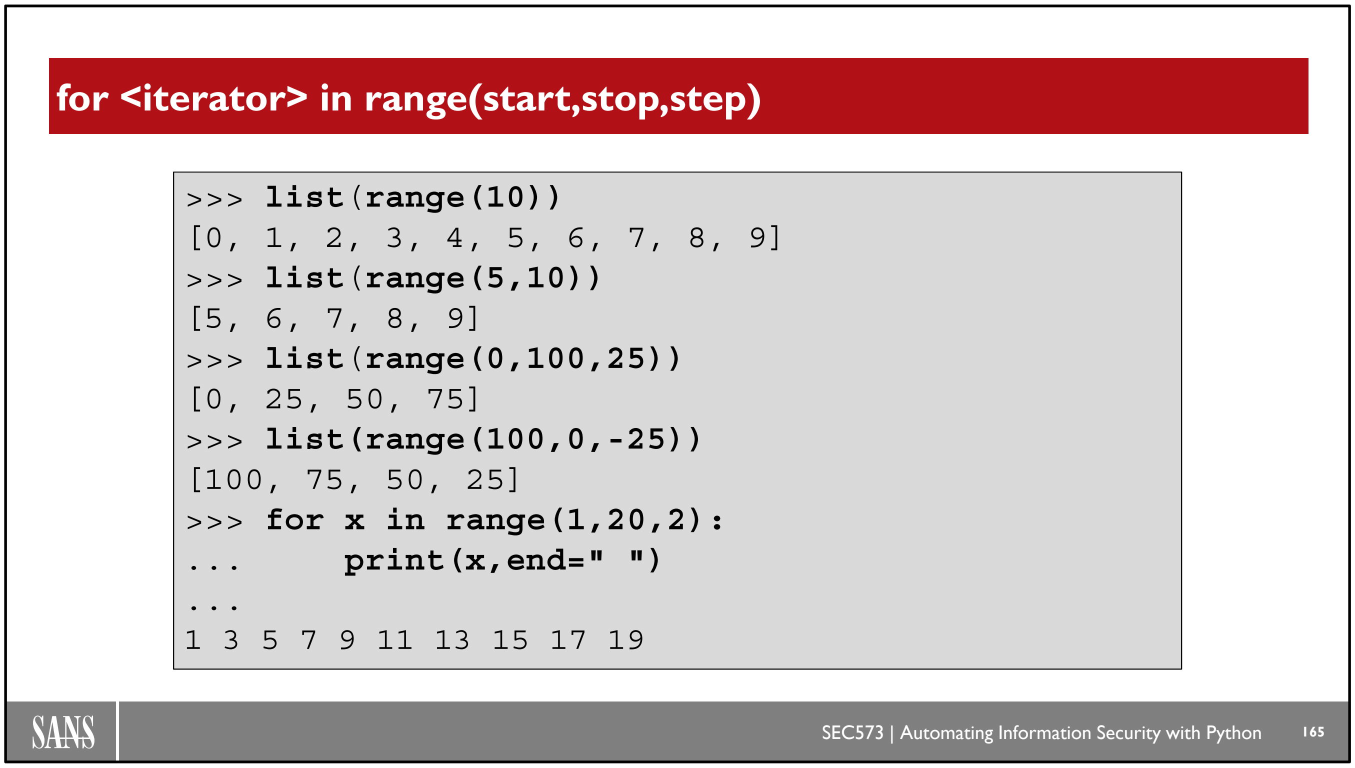
for x in list:

```
>>> 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
>>>
```

SANS

SEC573 | Automating Information Security with Python 164

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.



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
```

SANS

SEC573 | Automating Information Security with Python 165

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):
```

for index,value in enumerate(alist):

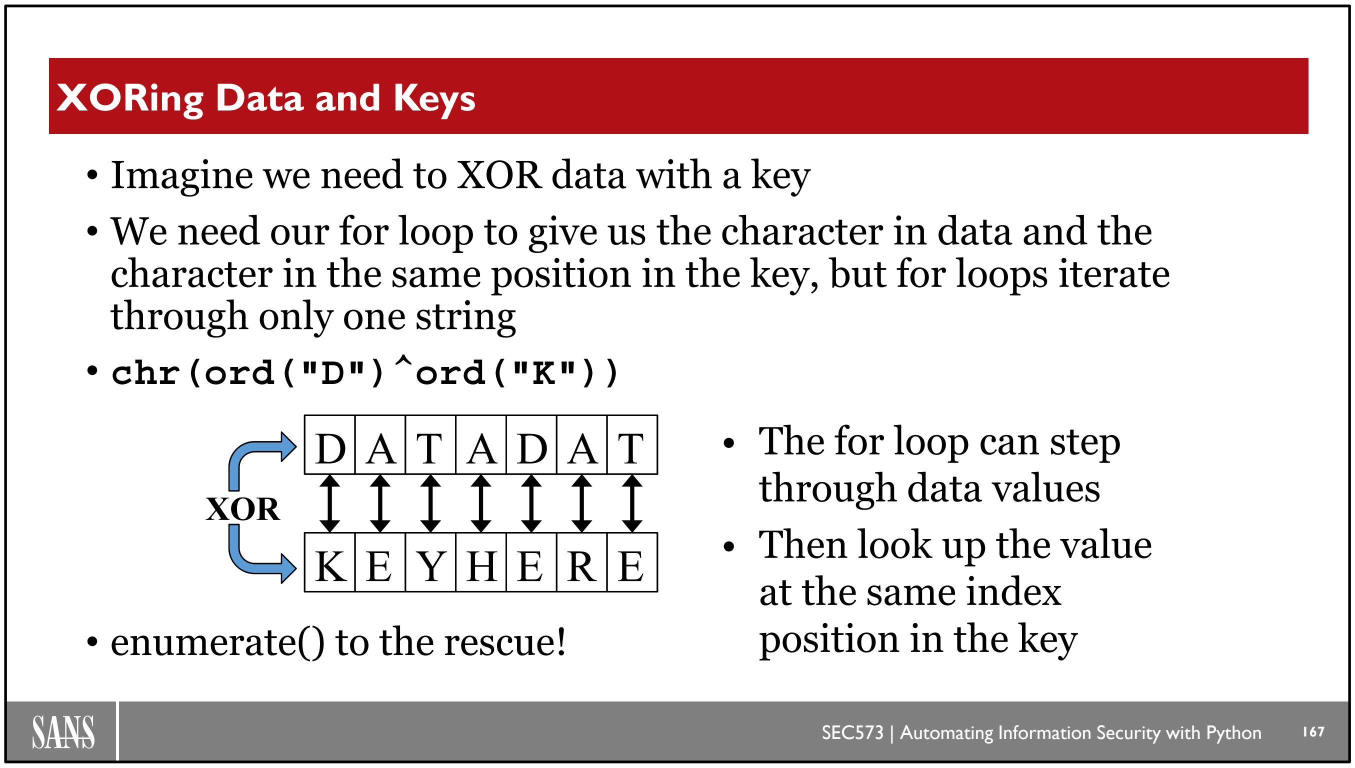
- "index" will be positioned in the list that you are currently processing
- "index" will start at zero and increase by one until it reaches len(alist)-1
- "value" will be the current value in the list, for example, alist[index]

```
>>> list(enumerate(movies))
[(0, 'Life of Brian'), (1, 'Holy Grail'), (2, 'Meaning of Life')]
>>> for index, value in enumerate(movies):
...     print("{} is in position {}".format(value, 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.

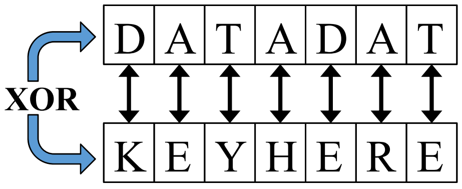
```
>>> list(enumerate(movies))
[(0, 'Life of Brian'), (1, 'Holy Grail'), (2, 'Meaning of Life')]
```

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.



XORing Data and Keys

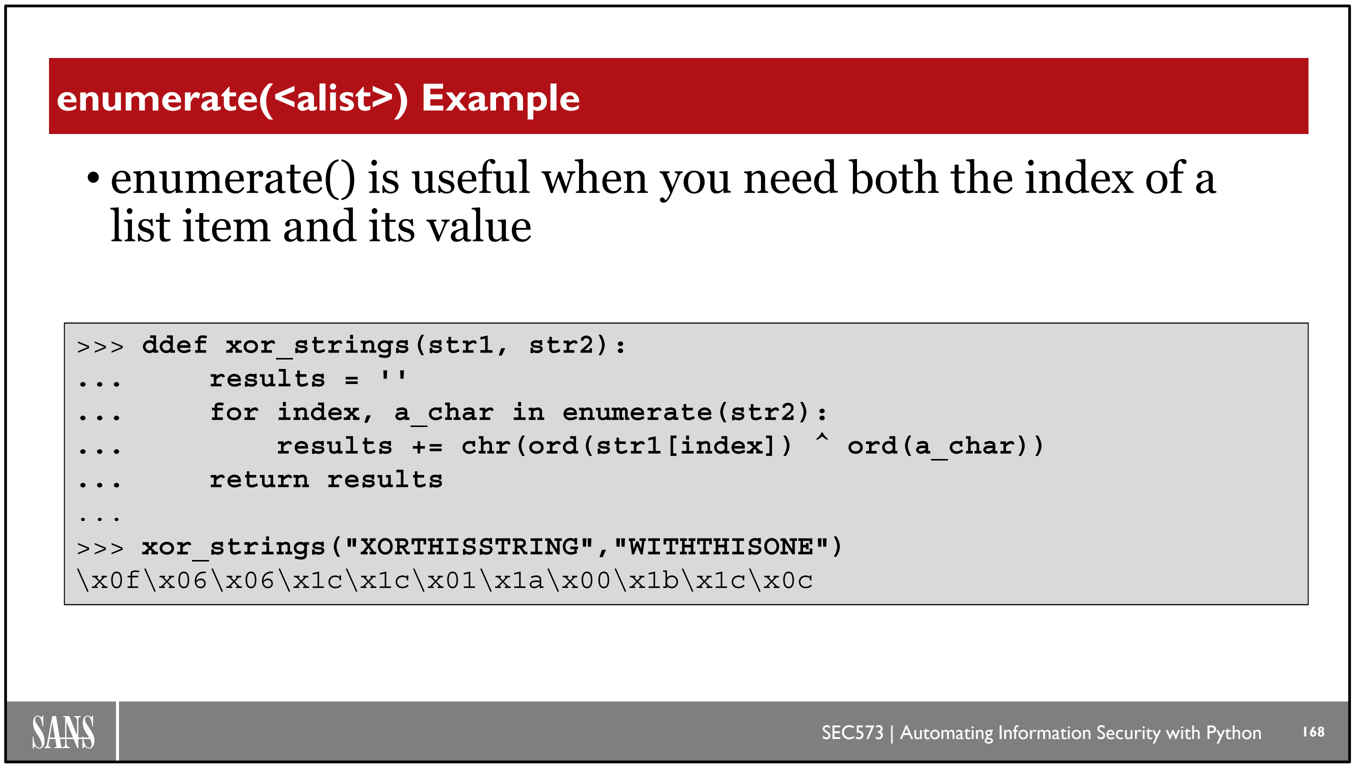
- Imagine we need to XOR data with a key
- We need our for loop to give us the character in data and the character in the same position in the key, but for loops iterate through only one string
- `chr(ord("D")^ord("K"))`



- The for loop can step through data values
- Then look up the value at the same index position in the key

- enumerate() to the rescue!

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.



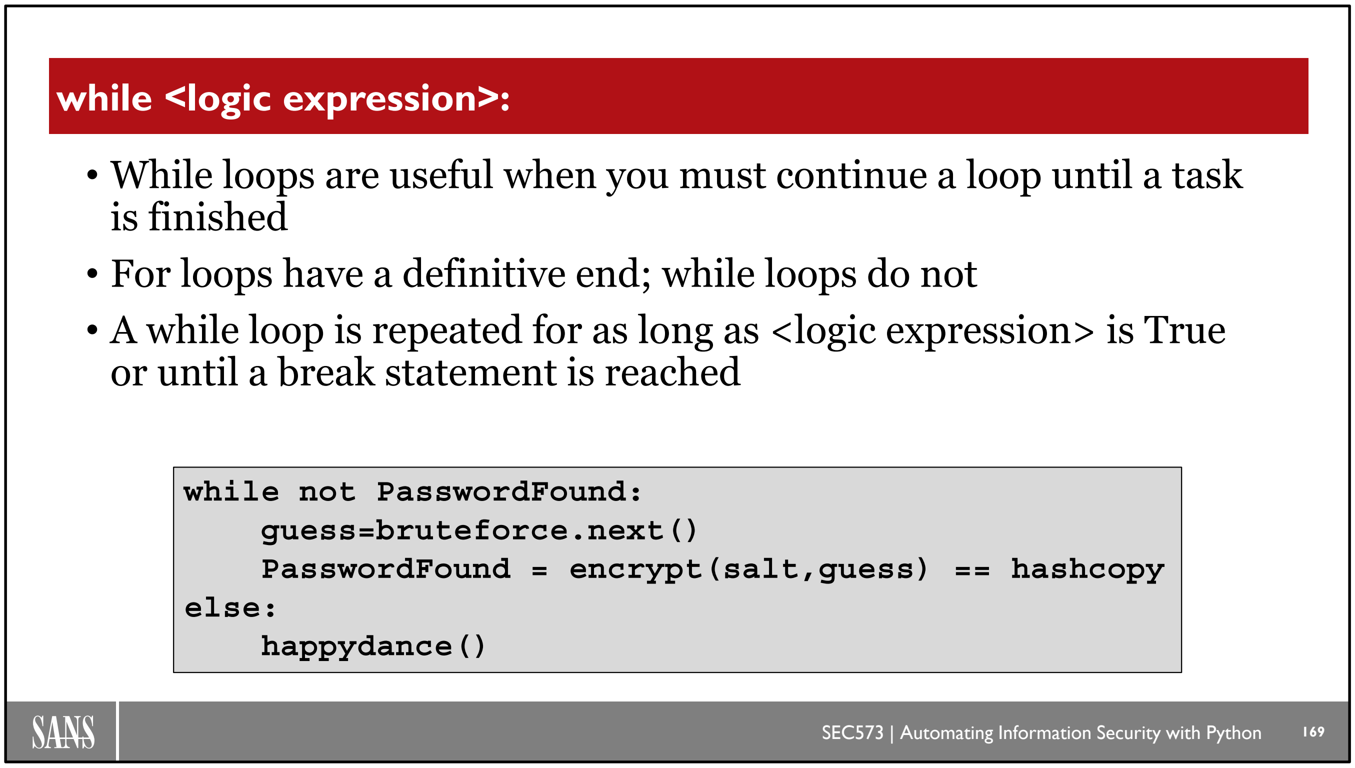
enumerate(<alist>) Example

- enumerate() is useful when you need both the index of a list item and its value

```
>>> ddef 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
```

SANS | SEC573 | Automating Information Security with Python 168

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.



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()
```

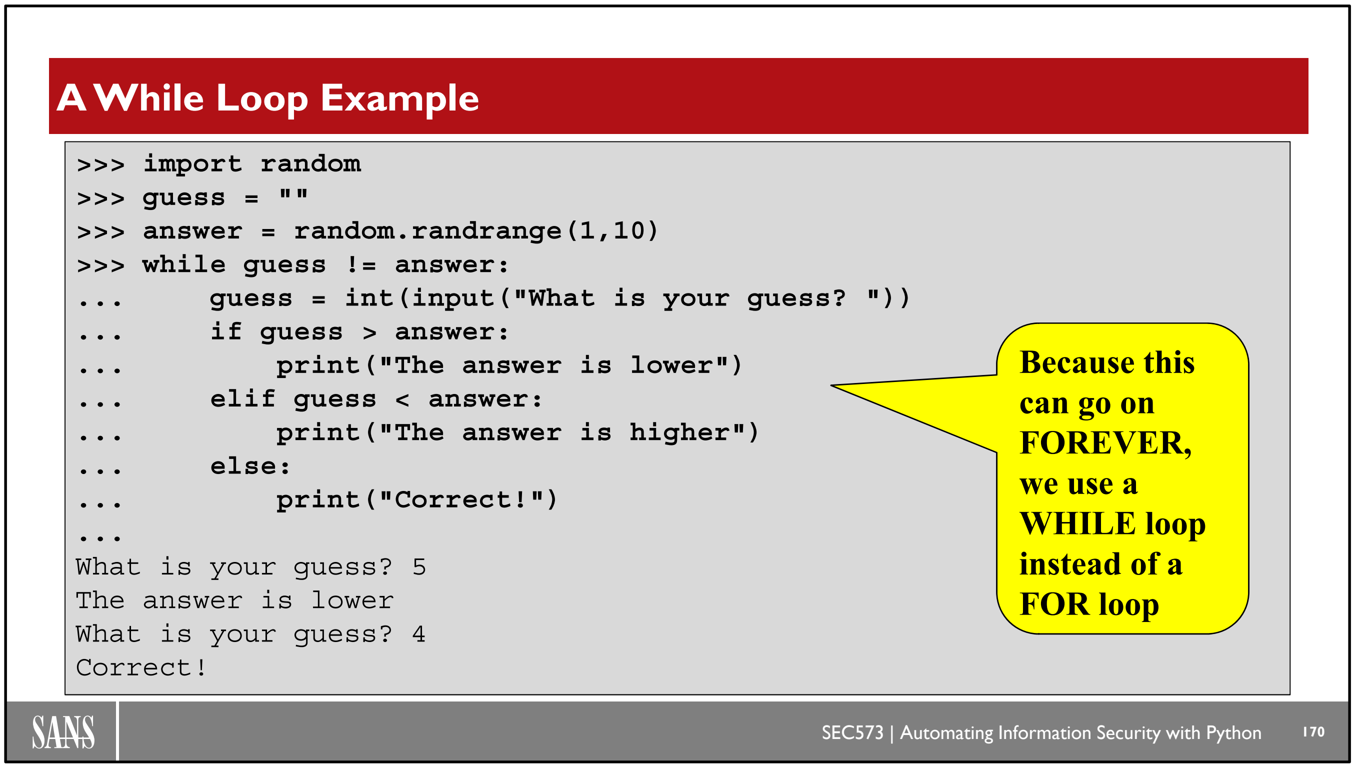
SANS SEC573 | Automating Information Security with Python 169

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.

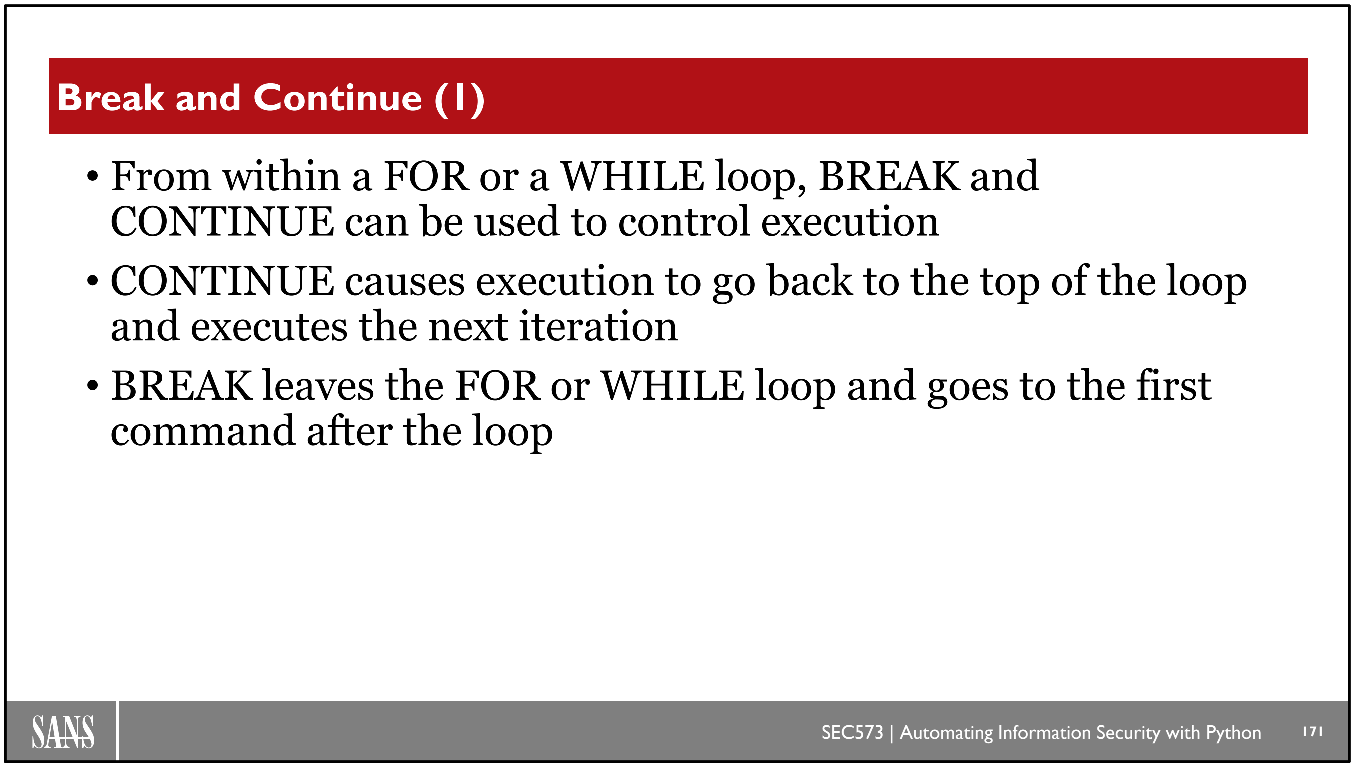


A While Loop Example

```
>>> 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 WHERE loop is the best option is a guessing game in which the user has an unlimited number of guesses to get the correct answer.



Break and Continue (1)

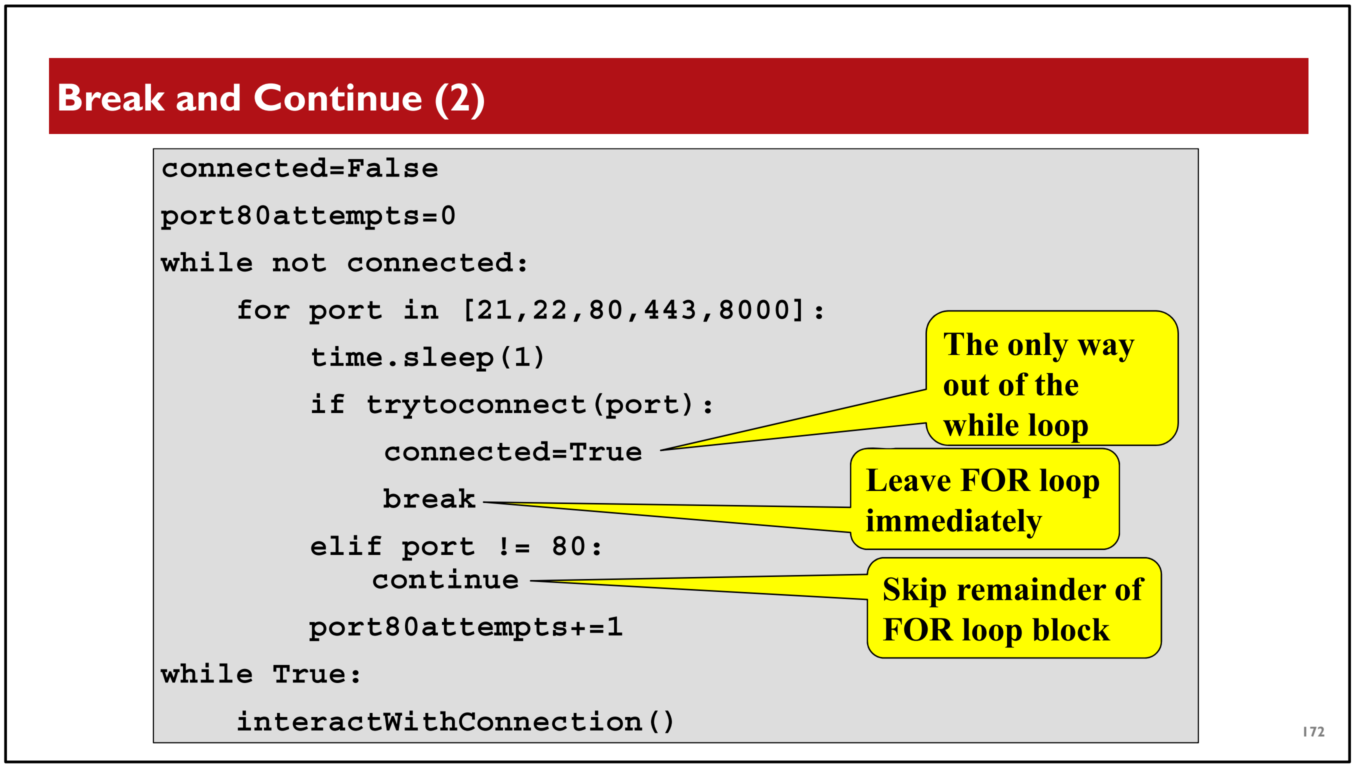
- From within a FOR or a WHILE loop, BREAK and CONTINUE can be used to control execution
- CONTINUE causes execution to go back to the top of the loop and executes the next iteration
- BREAK leaves the FOR or WHILE loop and goes to the first command after the loop

SANS SEC573 | Automating Information Security with Python 171

BREAK and CONTINUE are used within a for or a while loop to control the execution of code.

If a CONTINUE statement is encountered, no more code in the code block is executed for the current iteration. Instead, it goes back up to the top of the loop. With a for loop, it will start with the next item in the list. With a while loop, it will test to see whether the test case is still true and continue executing the loop.

When a BREAK is encountered, the for or while loop is immediately exited. The next command after the loop is executed. No further iterations of a for loop are executed. If a while loop has an else clause, it is not executed.



Break and Continue (2)

```
connected=False
port80attempts=0
while not connected:
    for port in [21,22,80,443,8000]:
        time.sleep(1)
        if trytoconnect(port):
            connected=True
            break
        elif port != 80:
            continue
        port80attempts+=1
while True:
    interactWithConnection()
```

The only way out of the while loop
Leave FOR loop immediately
Skip remainder of FOR loop block

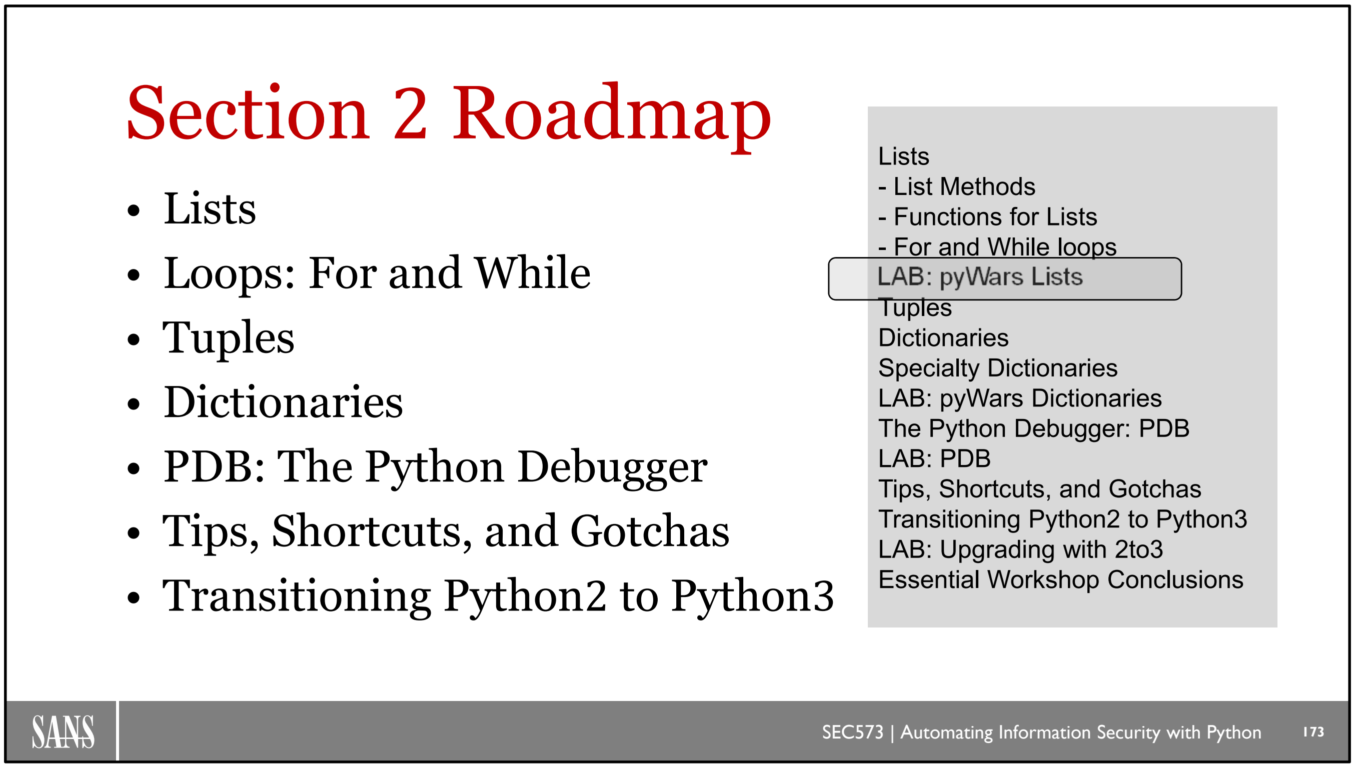
172

To illustrate how break and continue can be used, consider this pseudocode. *Pseudocode* isn't really a functioning program; certain elements of the code aren't syntactically correct, but the program illustrates the logical flow of what you are trying to accomplish.

In this program, we want to repeatedly go through a list of five different TCP ports forever until we get a connection. If we get no response, we should immediately try the next port on our list in the for loop. To try the next port, we use the "continue" statement. Here, when the continue is encountered, it will immediately go to the next iteration of the for or while loop. In this case, the continue is part of the for loop. If we are trying port 80, then we want to increment a counter. To exit both, we set the "connected" variable used by the while loop to TRUE and we call break. Break will immediately leave the for loop. When it does, the condition on the while loop is tested again and the while loop is exited.

What would happen if we didn't set connected=True and just called break? If we just called break, the program would exit the for loop, but the while loop would just cause it to start the for loop again. This result is undesirable because the program will never stop scanning, and it will never reach the interactWithConnection() function.

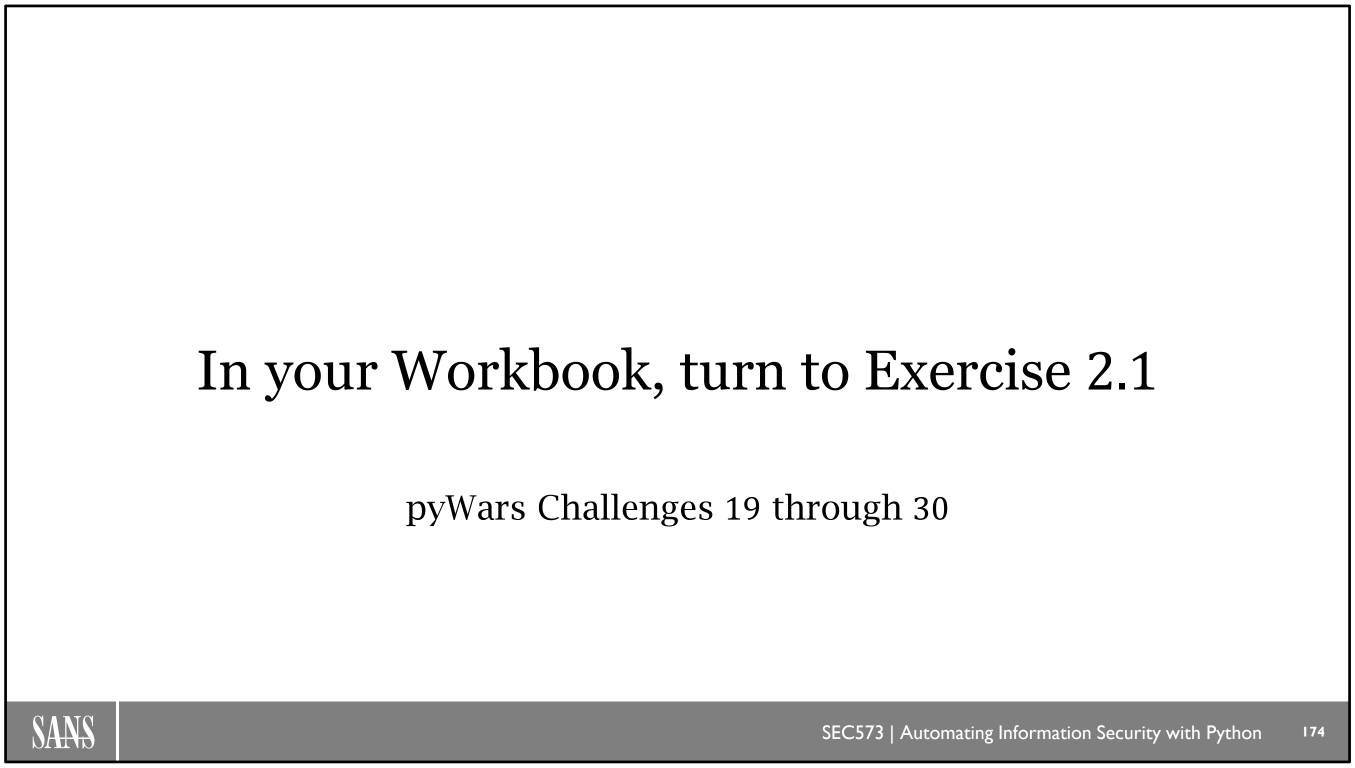
What would happen if we just set connected=True and didn't call break? If we just set connected to True, then the for loop would continue trying the rest of the ports in its list. When it reaches the end of the list, the while logical expression is tested. Because we set connected=True, the while loop will exit. This outcome is undesirable because the program still tries to connect to port after it has already established a connection.



Section 2 Roadmap

- Lists
- Loops: For and While
- Tuples
- Dictionaries
- PDB: The Python Debugger
- Tips, Shortcuts, and Gotchas
- Transitioning Python2 to Python3

This is a Roadmap slide.



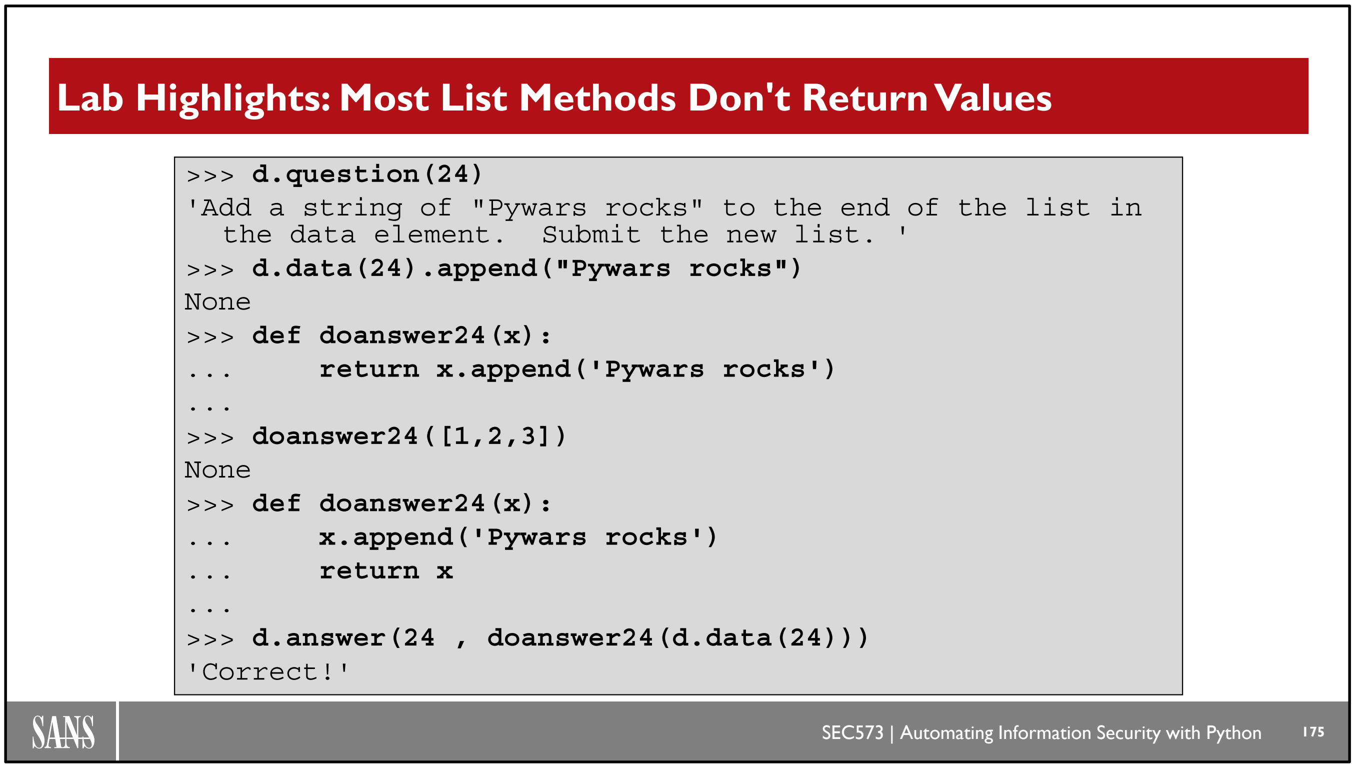
In your Workbook, turn to Exercise 2.1

pyWars Challenges 19 through 30

SANS

SEC573 | Automating Information Security with Python 174

Please complete the exercise in your Workbook.

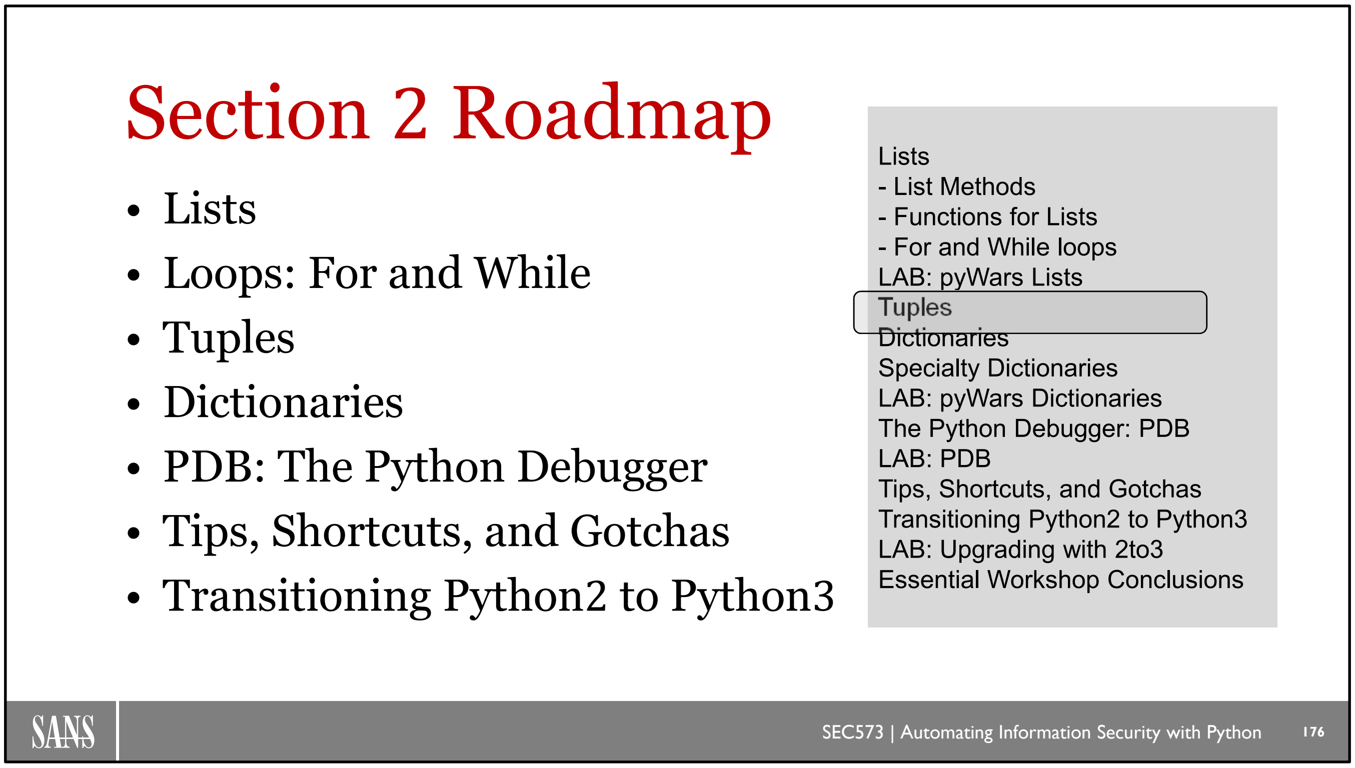


Lab Highlights: Most List Methods Don't Return Values

```
>>> d.question(24)
'Add a string of "Pywars rocks" to the end of the list in
  the data element.  Submit the new list. '
>>> d.data(24).append("Pywars rocks")
None
>>> def doanswer24(x):
...     return x.append('Pywars rocks')
...
>>> doanswer24([1,2,3])
None
>>> def doanswer24(x):
...     x.append('Pywars rocks')
...     return x
...
>>> d.answer(24 , doanswer24(d.data(24)))
'Correct!'
```

SEC573 | Automating Information Security with Python 175

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('Python rocks')." The problem is that the .append() method doesn't return a new copy of the list. It returns NOTHING. The result is your function will return nothing. Instead, you should just return x after calling the append function on a different line.

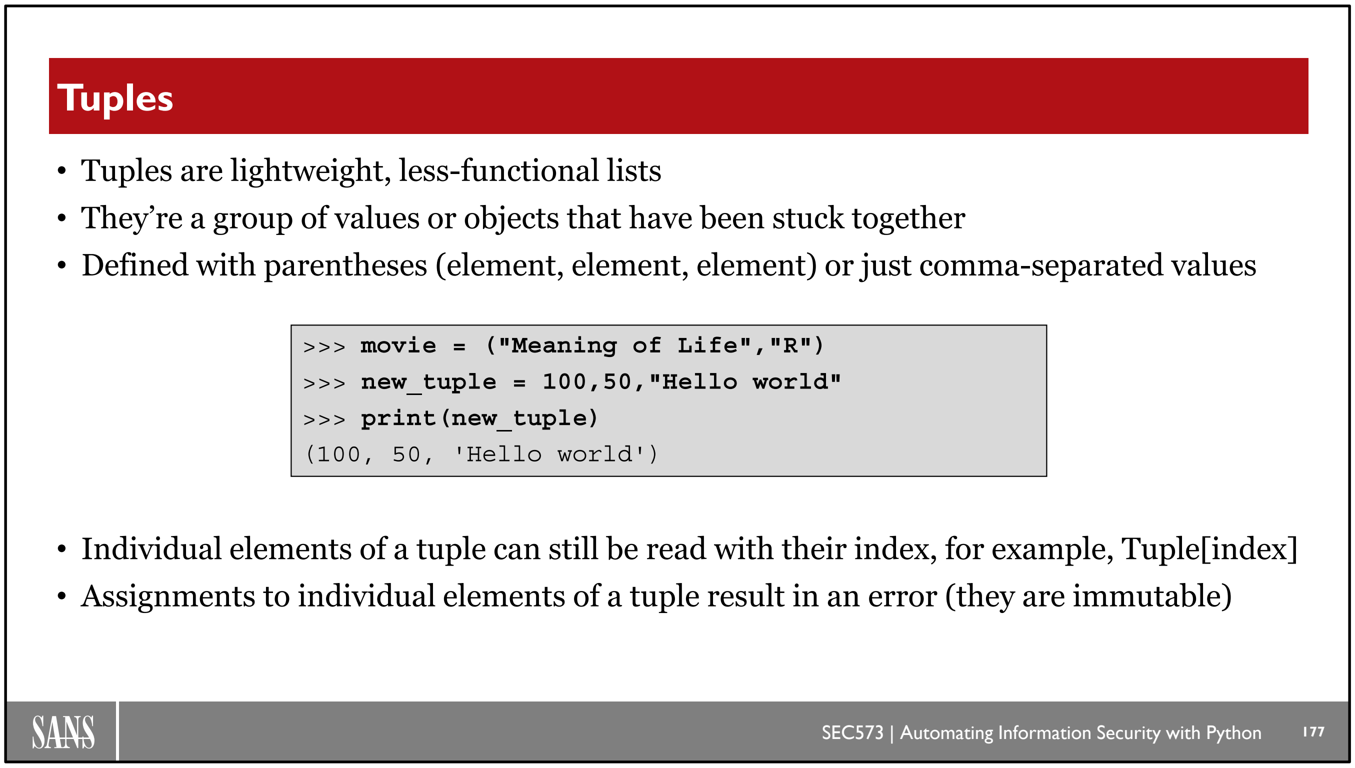


Section 2 Roadmap

- Lists
- Loops: For and While
- Tuples
- Dictionaries
- PDB: The Python Debugger
- Tips, Shortcuts, and Gotchas
- Transitioning Python2 to Python3

SANS SEC573 | Automating Information Security with Python 176

This is a Roadmap slide.



Tuples

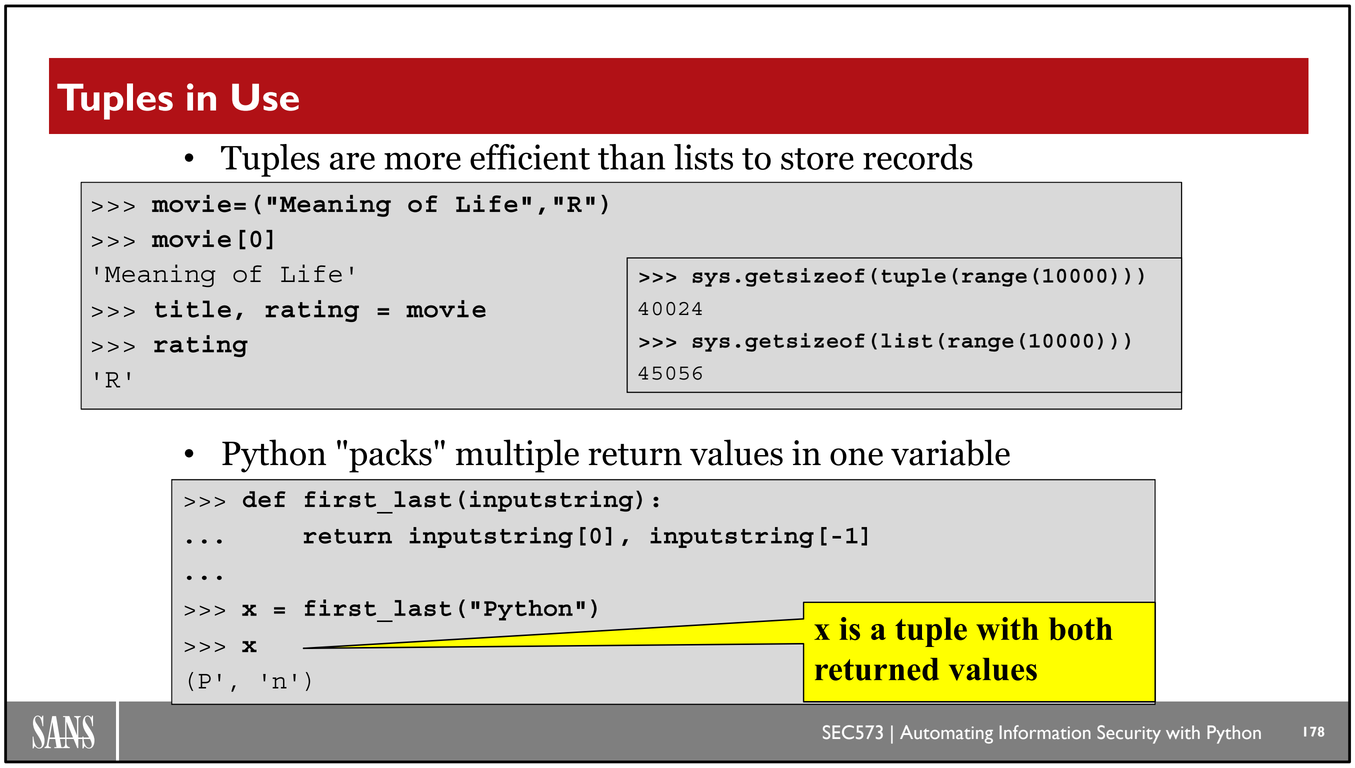
- Tuples are lightweight, less-functional lists
- They're a group of values or objects that have been stuck together
- Defined with parentheses (element, element, element) or just comma-separated values

```
>>> 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)

SANS SEC573 | Automating Information Security with Python 177

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.



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

SEC573 | Automating Information Security with Python 178

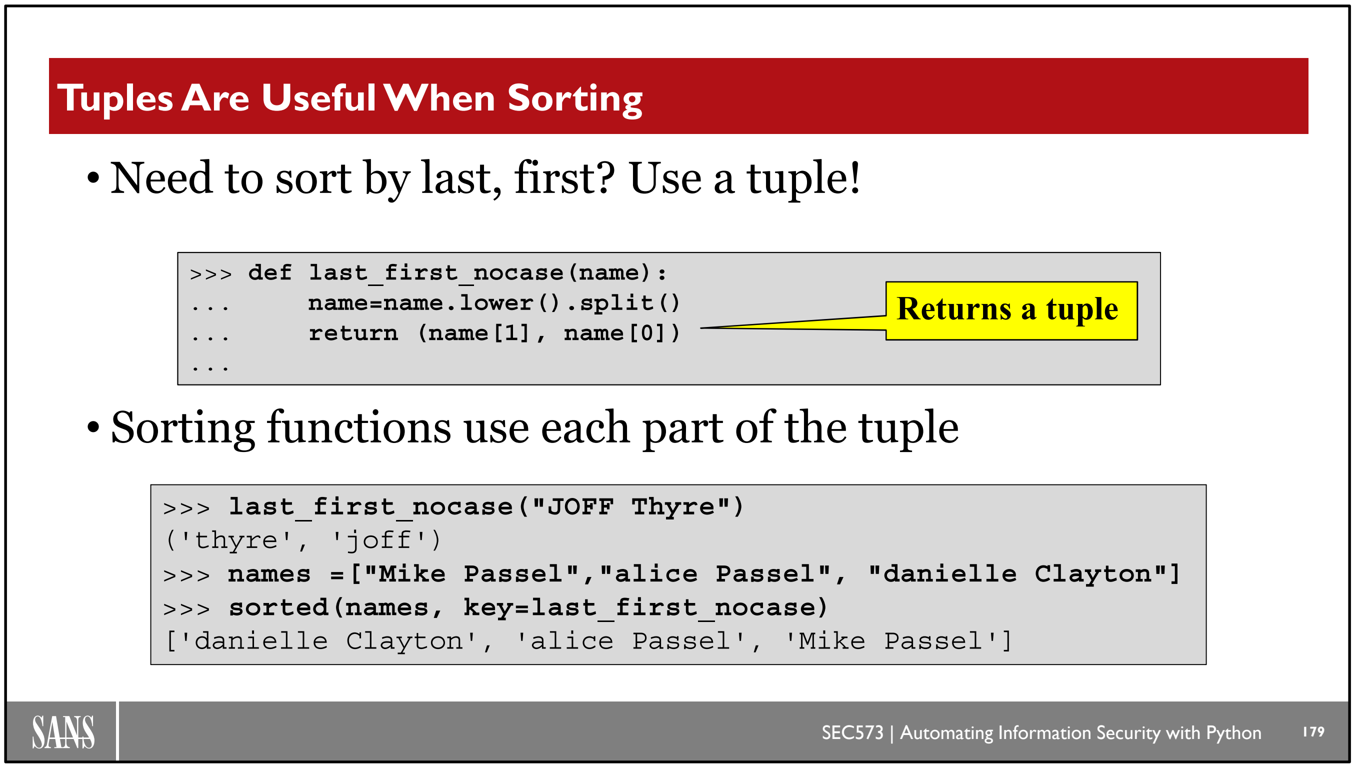
Tuples are lightweight because they do not have as many methods as lists. They have some of the basic functionality of lists. For example, you can create a variable named "movie" that contains a title and a rating and store them in a single variable. You can also pull out the individual items with their 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.



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

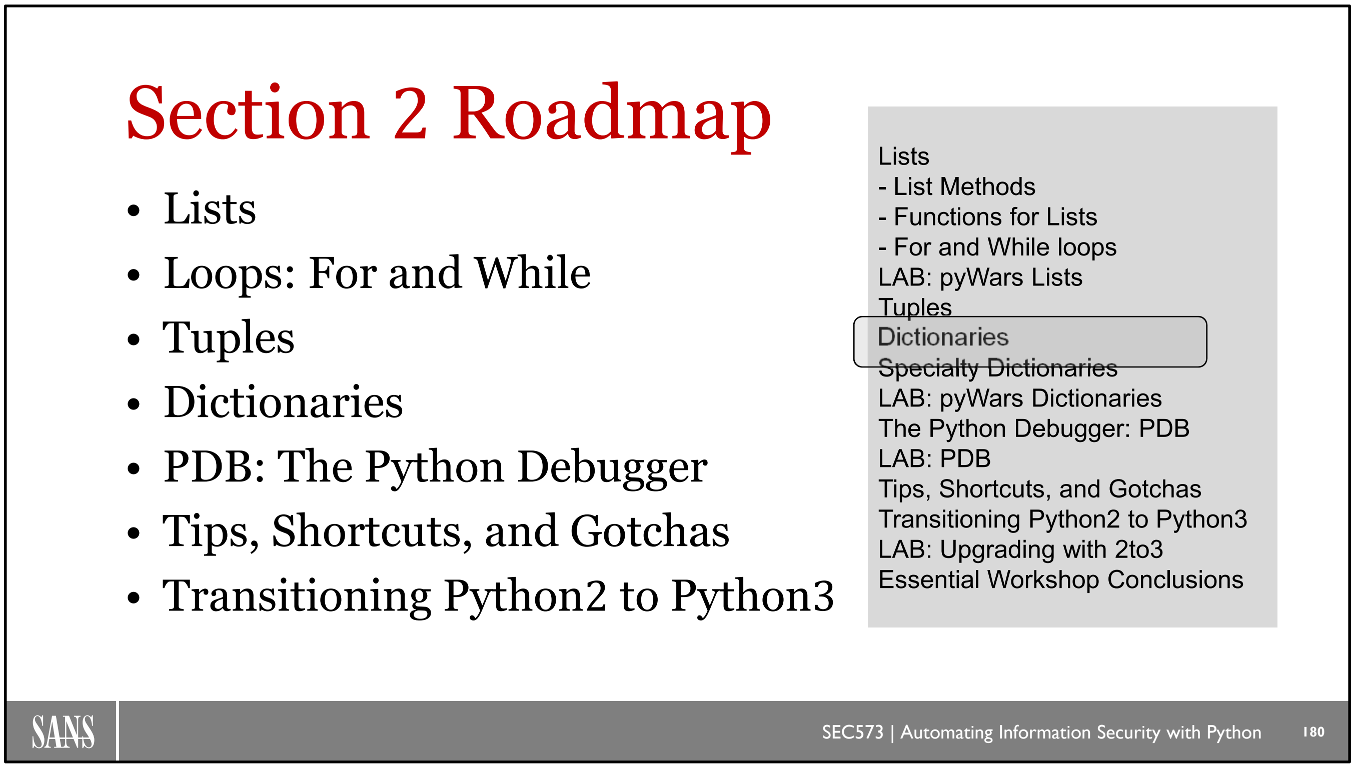
- Sorting functions use each part of the tuple

```
>>> last_first_nocase("JOFF Thyre")
('thyre', 'joff')
>>> 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.

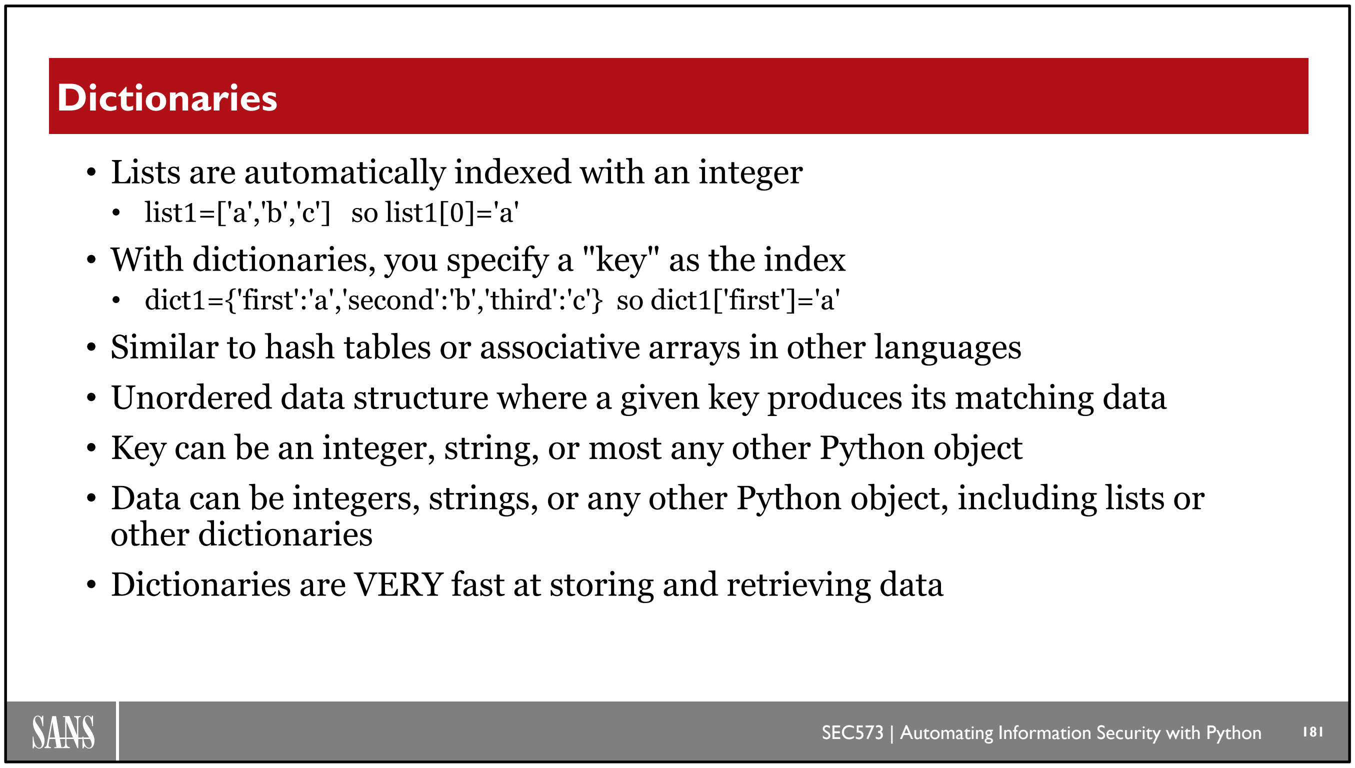
First, you create a function that takes in a string. It converts the string to lowercase and then turns it into a list by splitting on any white space. Then it returns a tuple with the last name in position 0 and the first name is position 1. When this is given to sorted or sort as a key function, the 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.



Section 2 Roadmap

- Lists
- Loops: For and While
- Tuples
- Dictionaries
- PDB: The Python Debugger
- Tips, Shortcuts, and Gotchas
- Transitioning Python2 to Python3

This is a Roadmap slide.



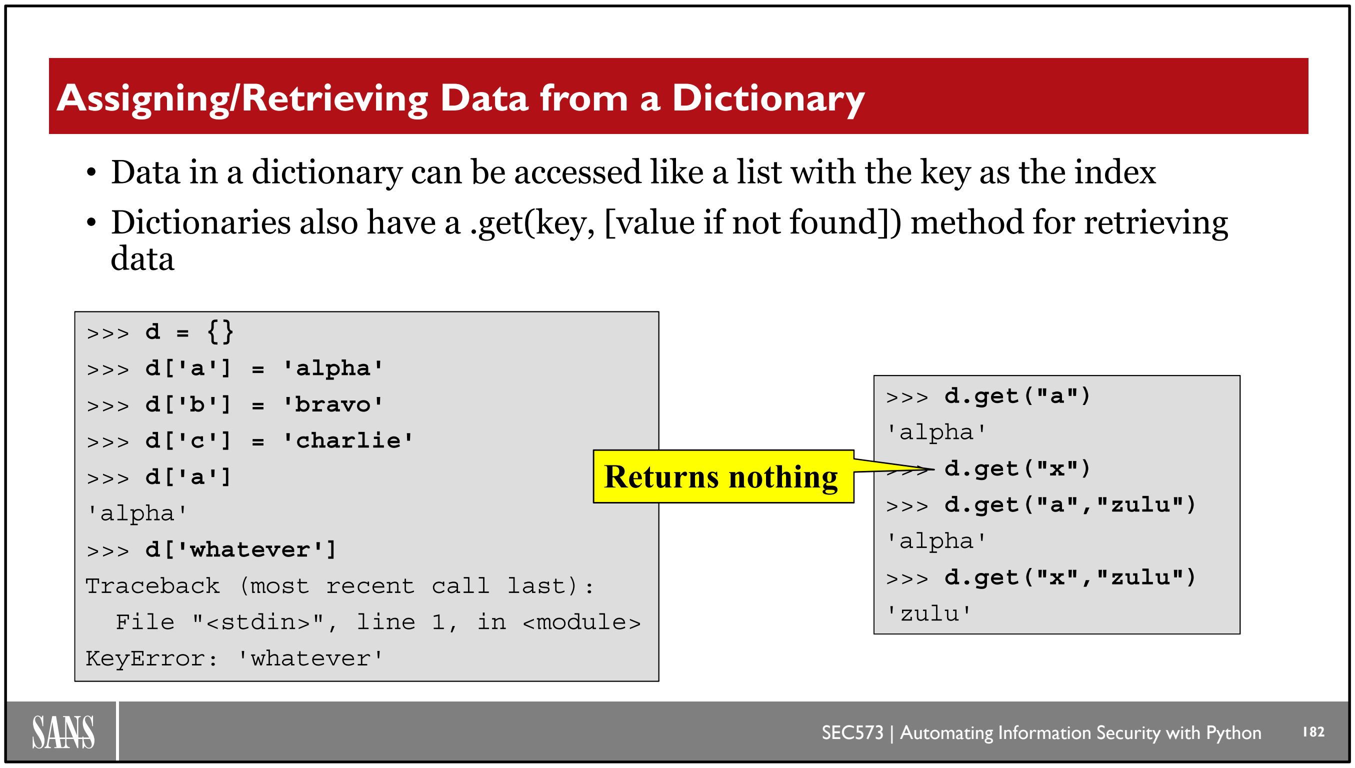
Dictionaries

- 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

SANS

SEC573 | Automating Information Security with Python 181

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.



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

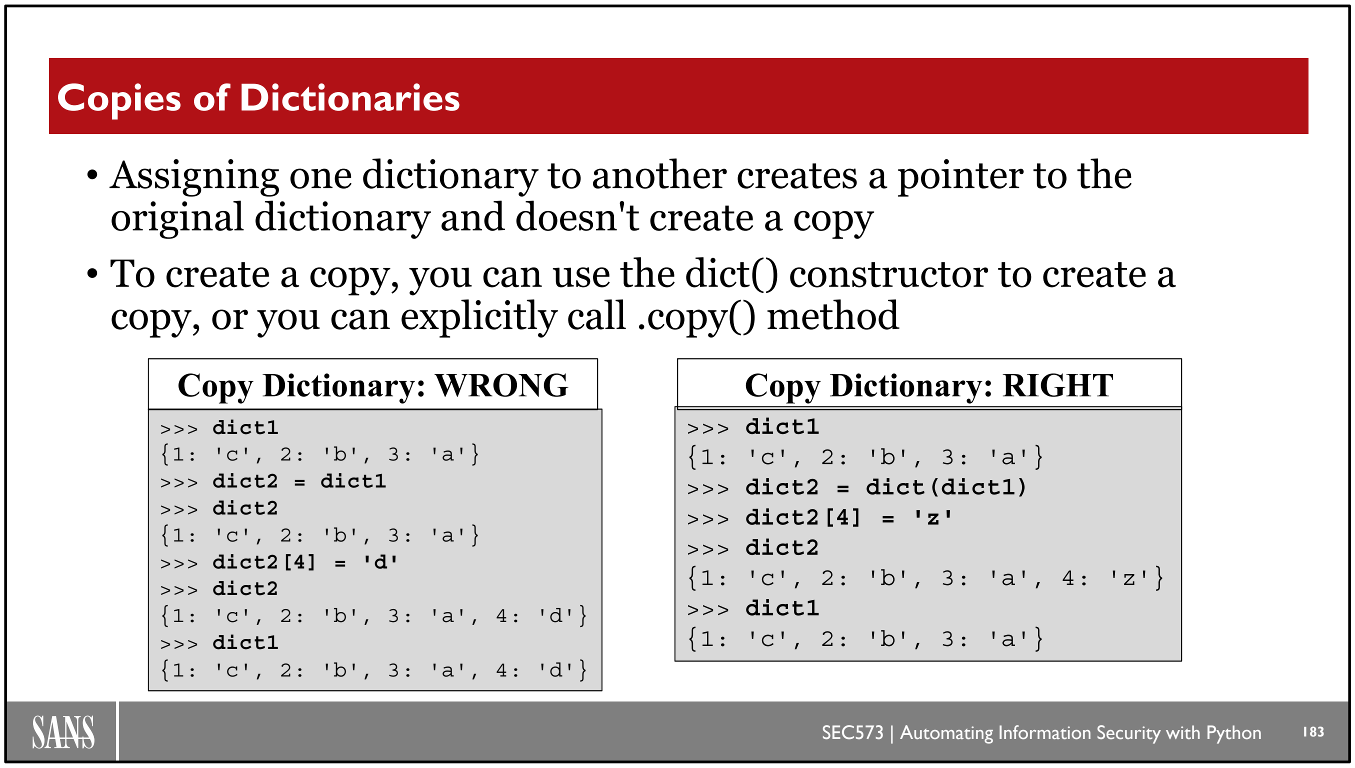
```
>>> d = {}
>>> d['a'] = 'alpha'
>>> d['b'] = 'bravo'
>>> d['c'] = 'charlie'
>>> d['a']
'alpha'
>>> d['whatever']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'whatever'
```

Returns nothing

```
>>> d.get("a")
'alpha'
>>> d.get("x")
>>> d.get("a","zulu")
'alpha'
>>> d.get("x","zulu")
'zulu'
```

SANS SEC573 | Automating Information Security with Python 182

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.



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

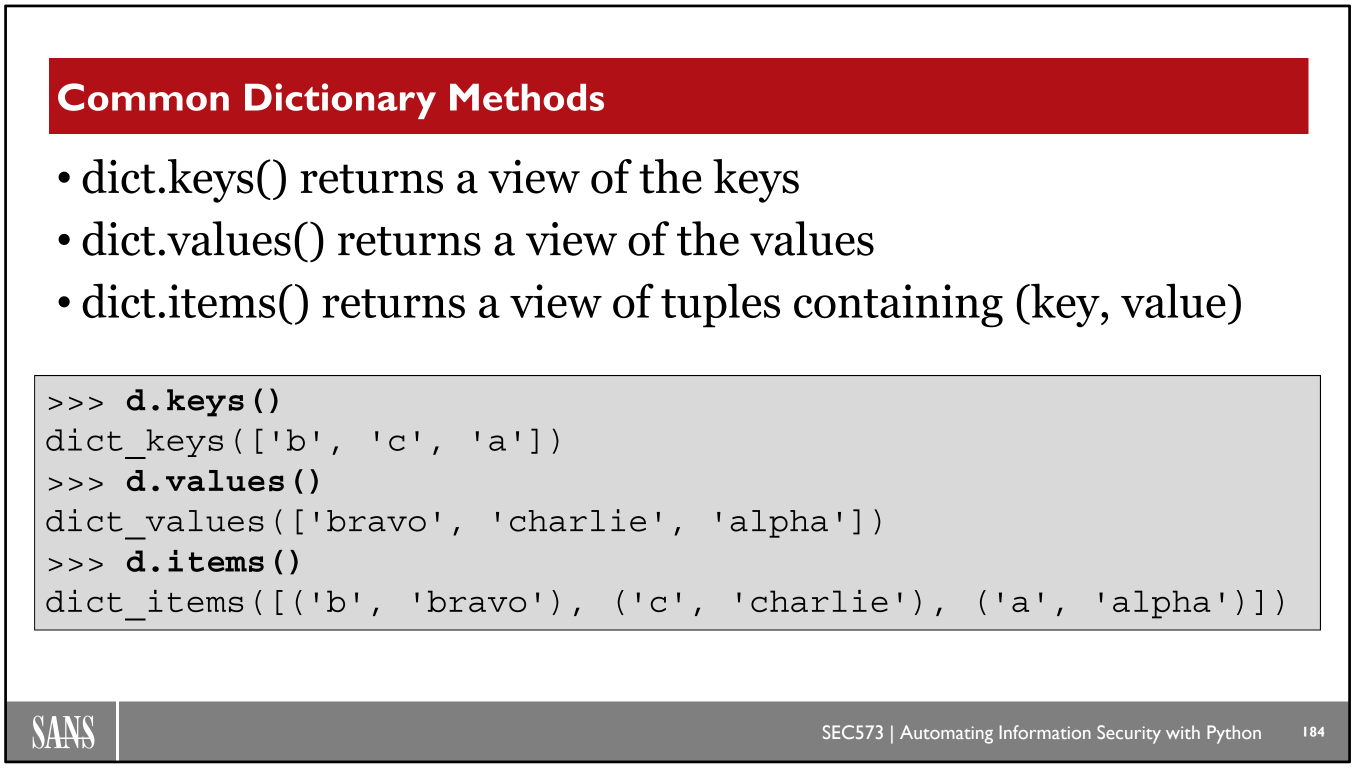
```
>>> dict1
{1: 'c', 2: 'b', 3: 'a'}
>>> dict2 = dict(dict1)
>>> dict2[4] = 'z'
>>> dict2
{1: 'c', 2: 'b', 3: 'a', 4: 'z'}
>>> dict1
{1: 'c', 2: 'b', 3: 'a'}
```

SANS SEC573 | Automating Information Security with Python 183

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. 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
```



Common Dictionary Methods

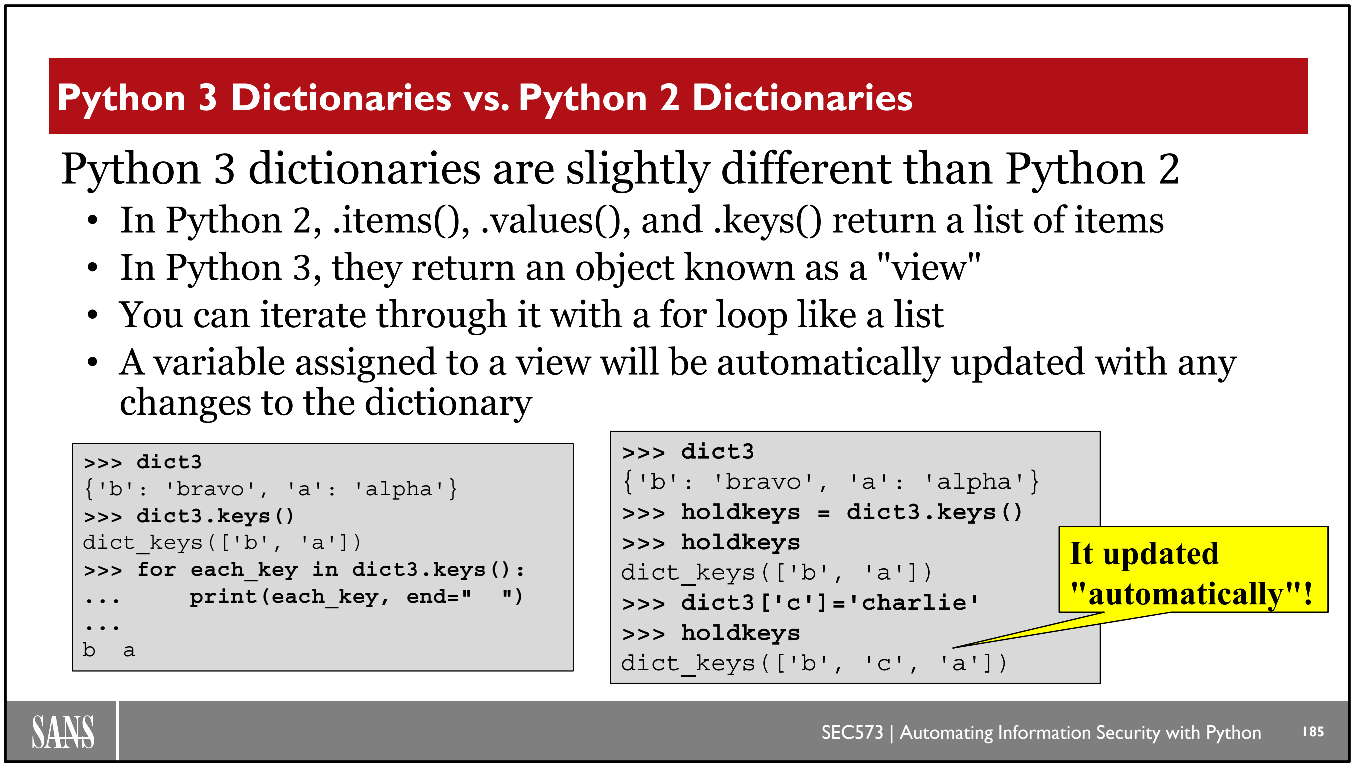
- dict.keys() returns a view of the keys
- dict.values() returns a view of the values
- dict.items() returns a view of tuples containing (key, value)

```
>>> d.keys()
dict_keys(['b', 'c', 'a'])
>>> d.values()
dict_values(['bravo', 'charlie', 'alpha'])
>>> d.items()
dict_items([('b', 'bravo'), ('c', 'charlie'), ('a', 'alpha')])
```

SANS SEC573 | Automating Information Security with Python 184

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, you can use an OrderedDict from the collections module.



Python 3 Dictionaries vs. Python 2 Dictionaries

Python 3 dictionaries are slightly different than Python 2

- In Python 2, .items(), .values(), and .keys() return a list of items
- In Python 3, they return an object known as a "view"
- 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

```
>>> dict3
{'b': 'bravo', 'a': 'alpha'}
>>> dict3.keys()
dict_keys(['b', 'a'])
>>> for each_key in dict3.keys():
...     print(each_key, end="  ")
...
b  a
```

```
>>> dict3
{'b': 'bravo', 'a': 'alpha'}
>>> holdkeys = dict3.keys()
>>> holdkeys
dict_keys(['b', 'a'])
>>> dict3['c']='charlie'
>>> holdkeys
dict_keys(['b', 'c', 'a'])
```

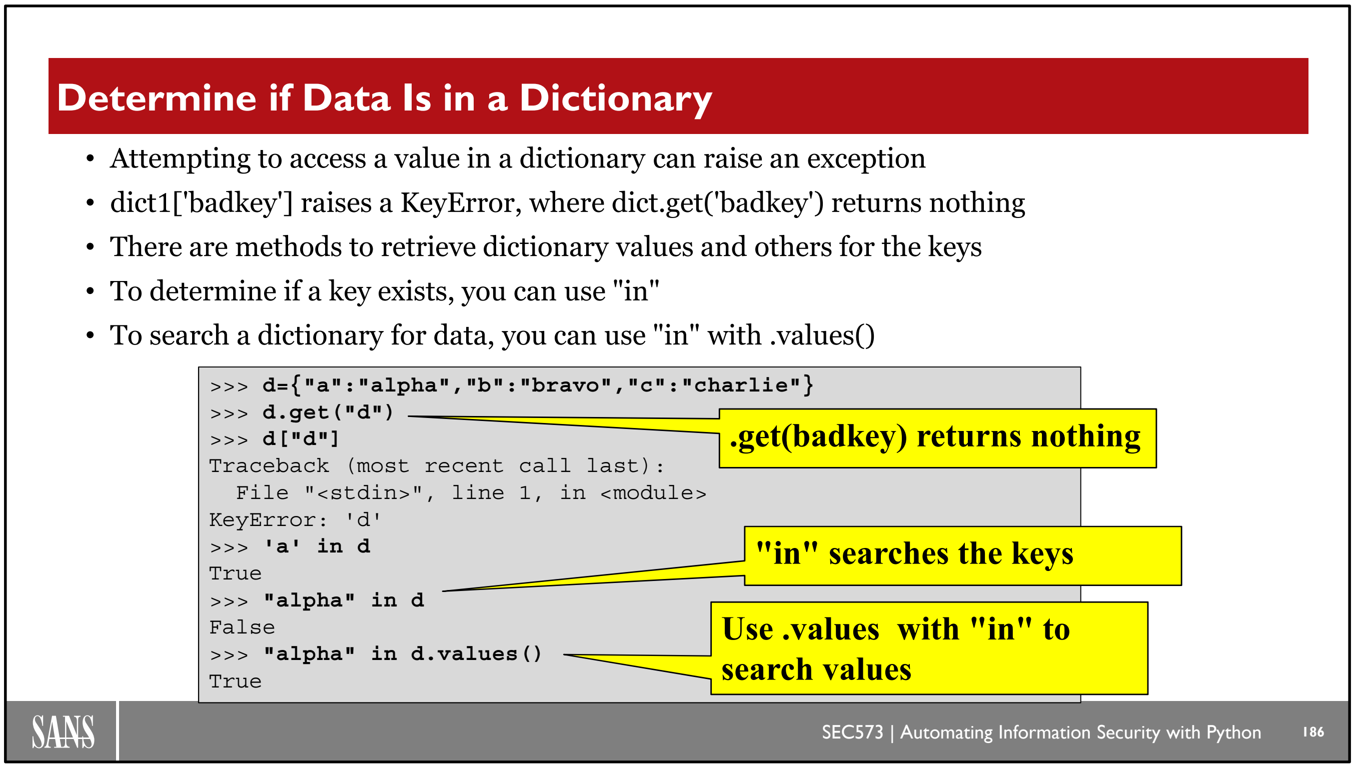
It updated "automatically"!

SANS

SEC573 | Automating Information Security with Python 185

Dictionaries in Python 3 are only slightly different than in Python 2. In Python 2, .items(), .values(), and .keys() returned a list of tuples, values, and keys, respectively. In Python 3, they return a "view" into the database. These view objects can be iterated through with a for loop just like in Python 2. But you will not be able to use the slice operation or other list methods on the values returned. These views' objects are identical to the objects returned by .viewkeys(), .viewitems(), and .viewvalues() in Python 2. 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.

Python 2 dictionaries have a .has_key() method that can be used to check to see if a key exists, but it should not be used, as it is not forward compatible with Python 3. Let's talk about how to check for a key in a way that is compatible with both Python 2 and Python 3.



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
SANS
SEC573 | Automating Information Security with Python
186

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.

Looping through Dictionary Keys

- A for loop can step through a dictionary as follows:

  `for <some var> in adict:`

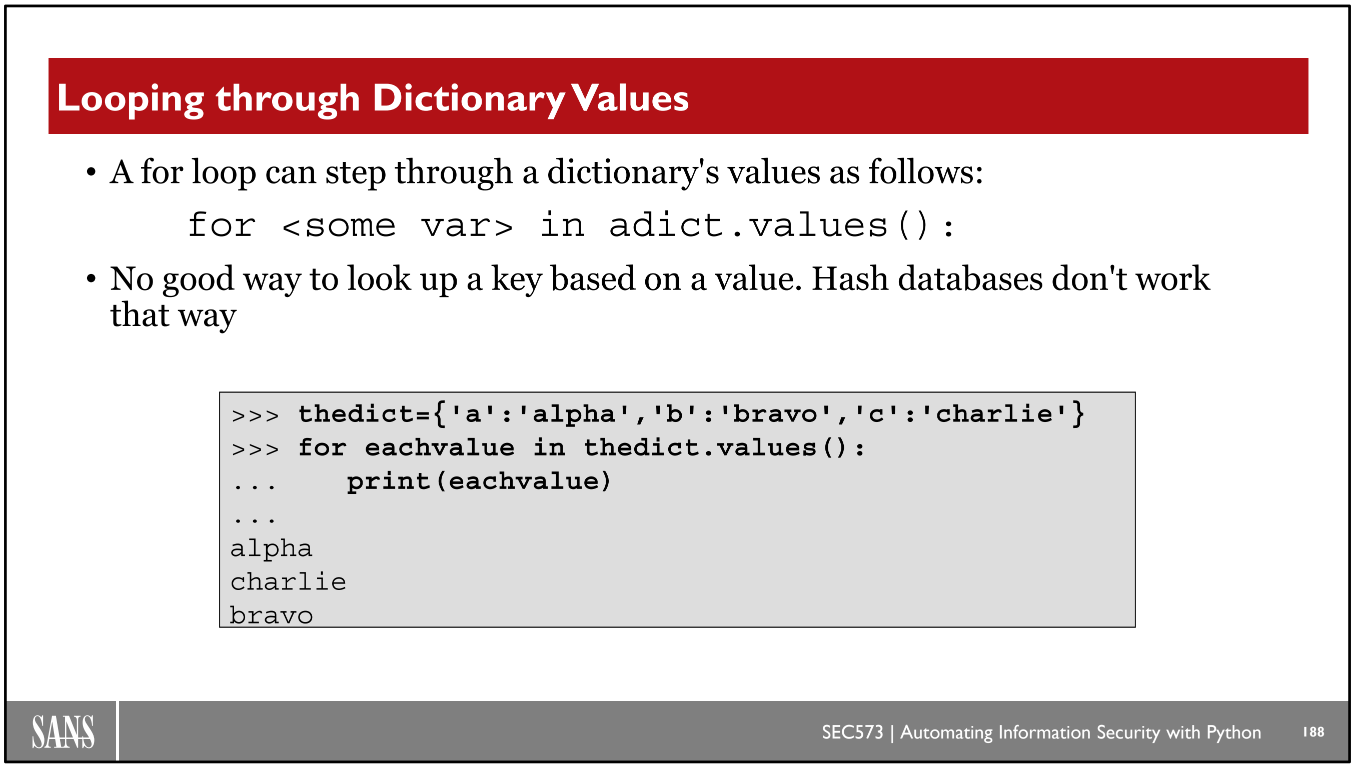
- This is the same as doing this:

  `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
c charlie
b bravo
```

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.



Looping through Dictionary Values

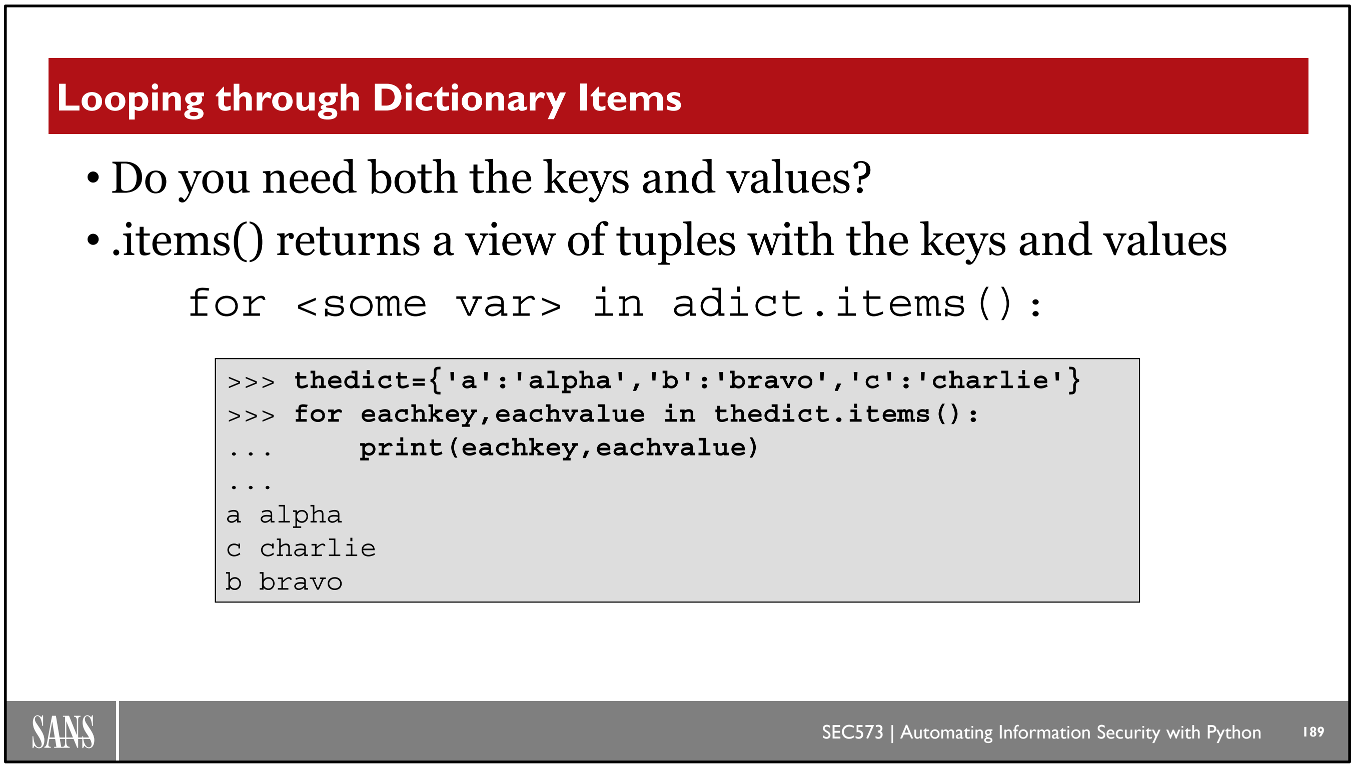
- A for loop can step through a dictionary's values as follows:

  `for <some var> in adict.values():`

- No 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
charlie
bravo
```

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.



Looping through Dictionary Items

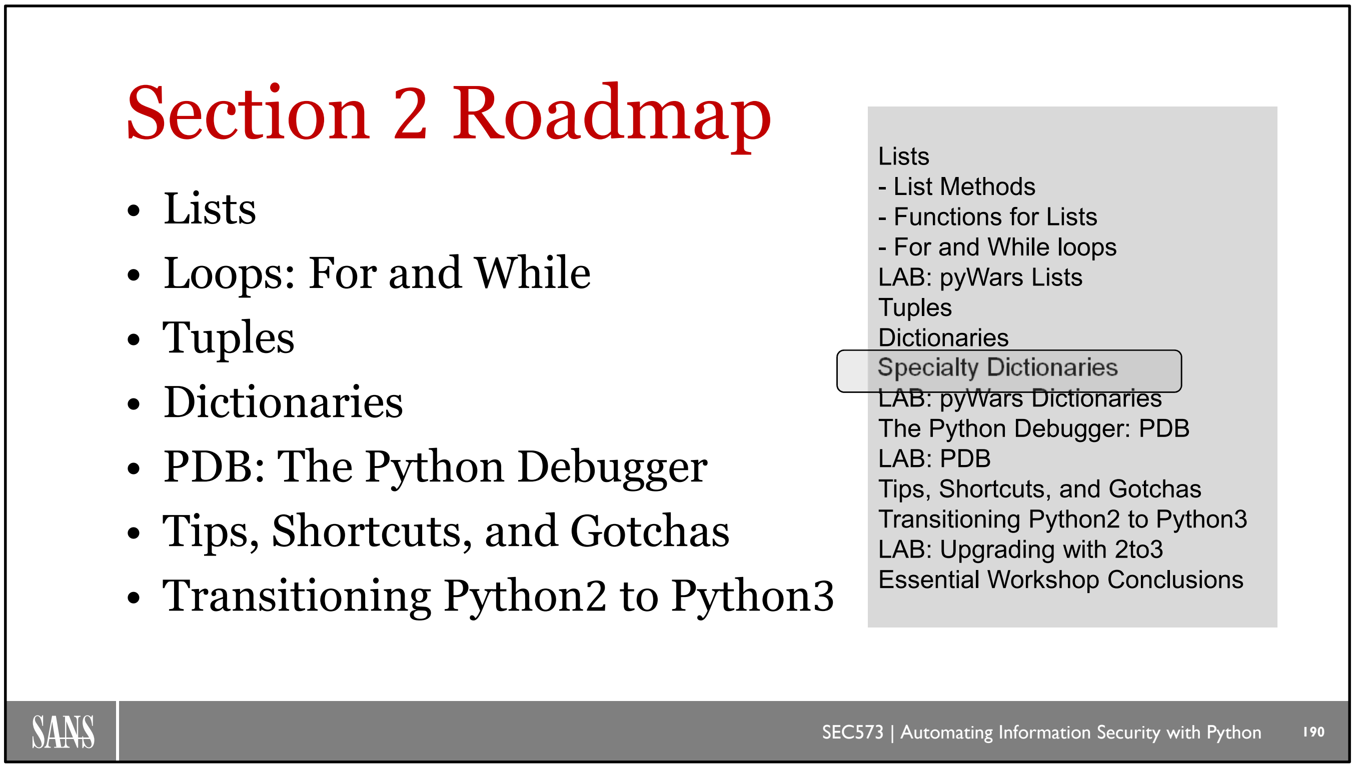
- Do you need both the keys and values?
- .items() returns a view of tuples with the keys and values

```
for <some var> in adict.items():
```

```
>>> thedict={'a':'alpha','b':'bravo','c':'charlie'}
>>> for eachkey,eachvalue in thedict.items():
...     print(eachkey,eachvalue)
...
a alpha
c charlie
b bravo
```

SANS | SEC573 | Automating Information Security with Python 189

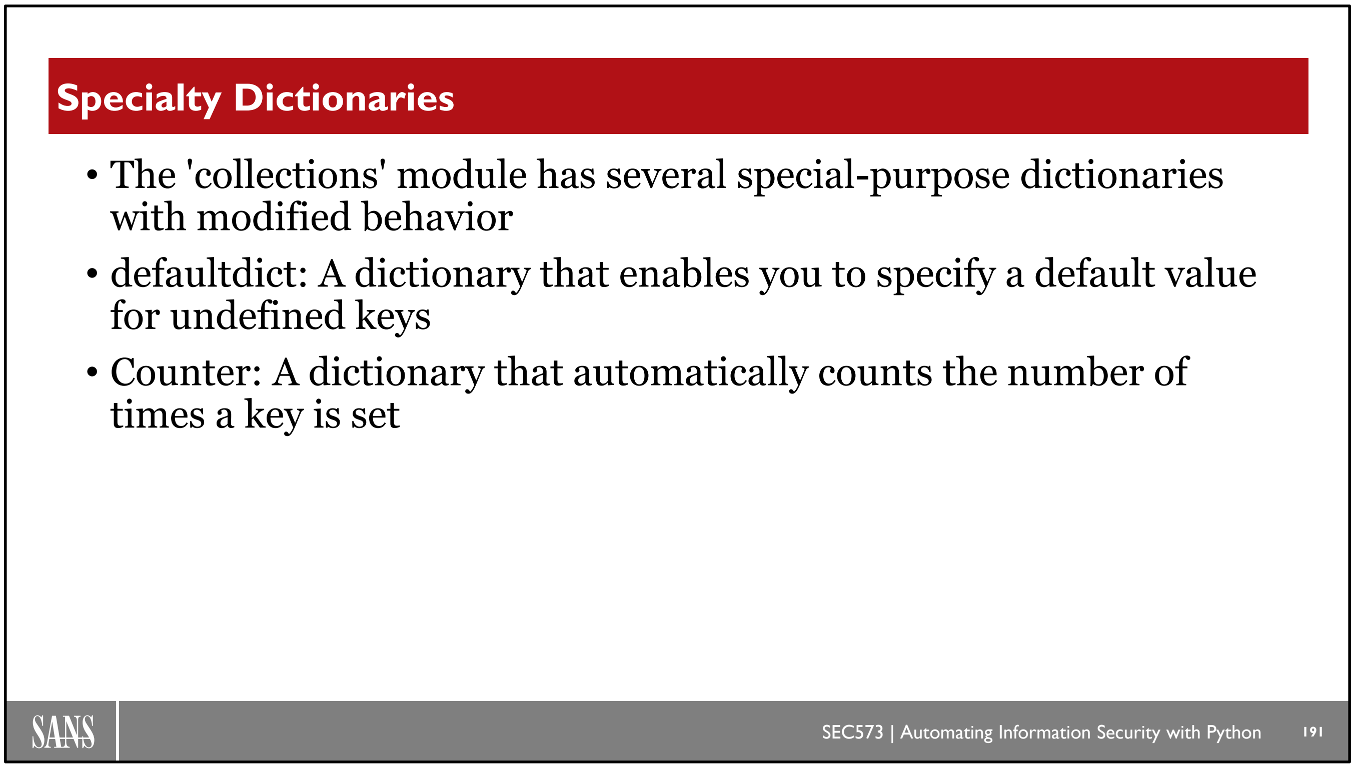
The dictionary's .items() method returns a view of tuples. Each tuple contains a key and the associated value. The items() method can be used to step through each of these tuples without creating a list in memory first. If you know you need access to every key and item in the dictionary, then step through the dictionary with a for loop and the dictionary's .items() method.



Section 2 Roadmap

- Lists
- Loops: For and While
- Tuples
- Dictionaries
- PDB: The Python Debugger
- Tips, Shortcuts, and Gotchas
- Transitioning Python2 to Python3

This is a Roadmap slide.



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

SANS SEC573 | Automating Information Security with Python 191

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.

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 newval at 0x024A1C30>, {'src#1': ['dst'], 'src#2': []})
```

SANS SEC573 | Automating Information Security with Python 192

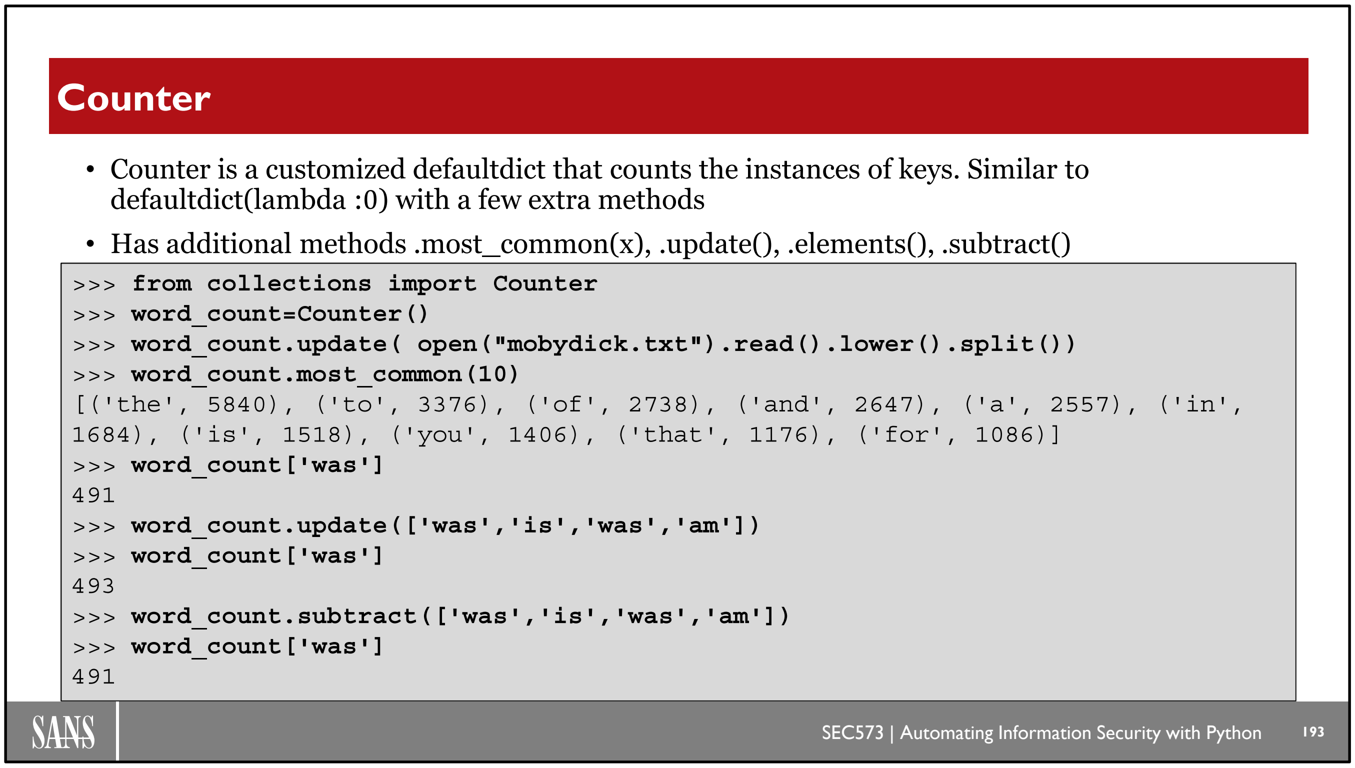
Remember that if you attempt to get data from a dictionary using a key that doesn't exist, you will get an error:

```
>>> 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 when it is called. Whatever it returns is the value of "empty" dictionary items. In this example, our dictionary calls the function newval anytime we try to retrieve a key that doesn't exist. The newval function returns an empty list. Now we can append items to nonexistent lists in the dictionary.



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
```

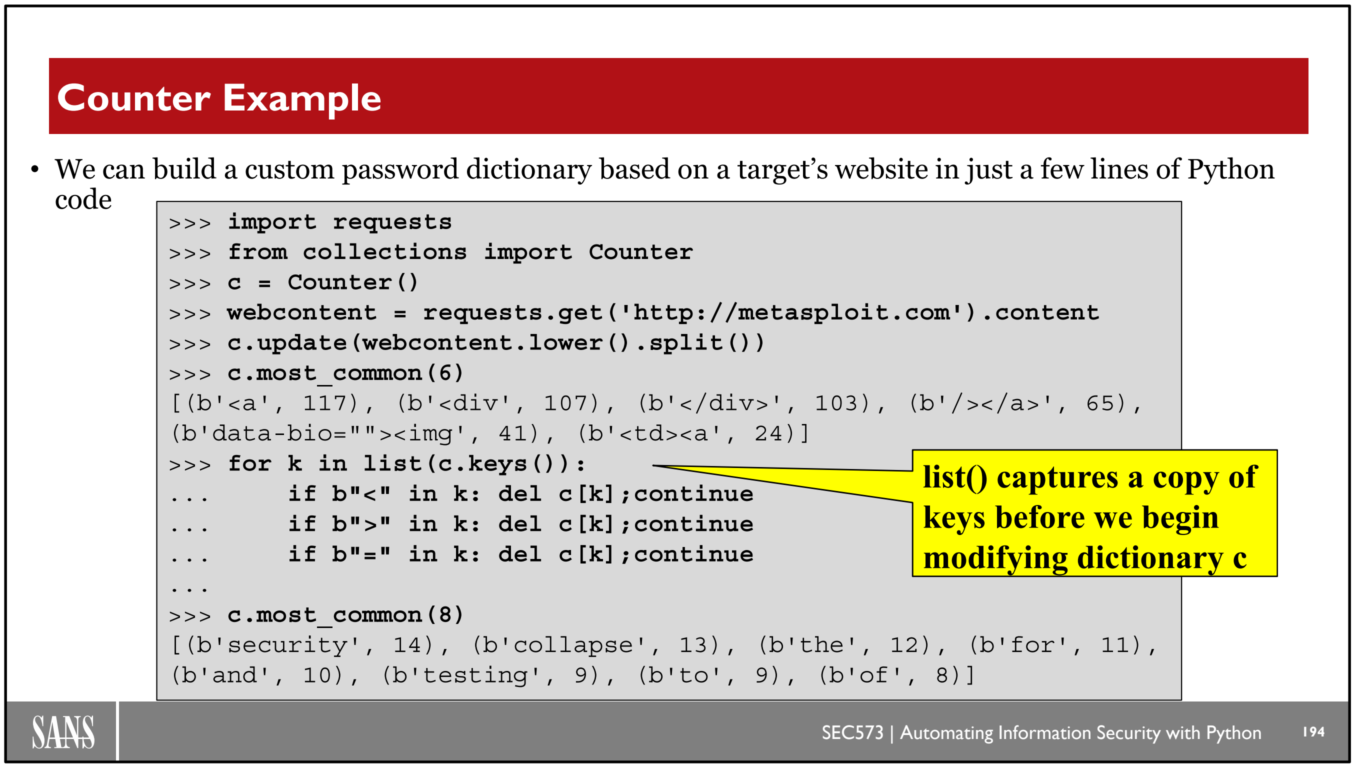
SANS | SEC573 | Automating Information Security with Python 193

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.



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').content
>>> c.update(webcontent.lower().split())
>>> c.most_common(6)
[(b'<a', 117), (b'<div', 107), (b'</div>', 103), (b'/></a>', 65),
(b'data-bio=""><img', 41), (b'<td><a', 24)]
>>> for k in list(c.keys()):
...     if b"<" in k: del c[k];continue
...     if b">" in k: del c[k];continue
...     if b"=" in k: del c[k];continue
...
>>> c.most_common(8)
[(b'security', 14), (b'collapse', 13), (b'the', 12), (b'for', 11),
(b'and', 10), (b'testing', 9), (b'to', 9), (b'of', 8)]
```

list() captures a copy of keys before we begin modifying dictionary c

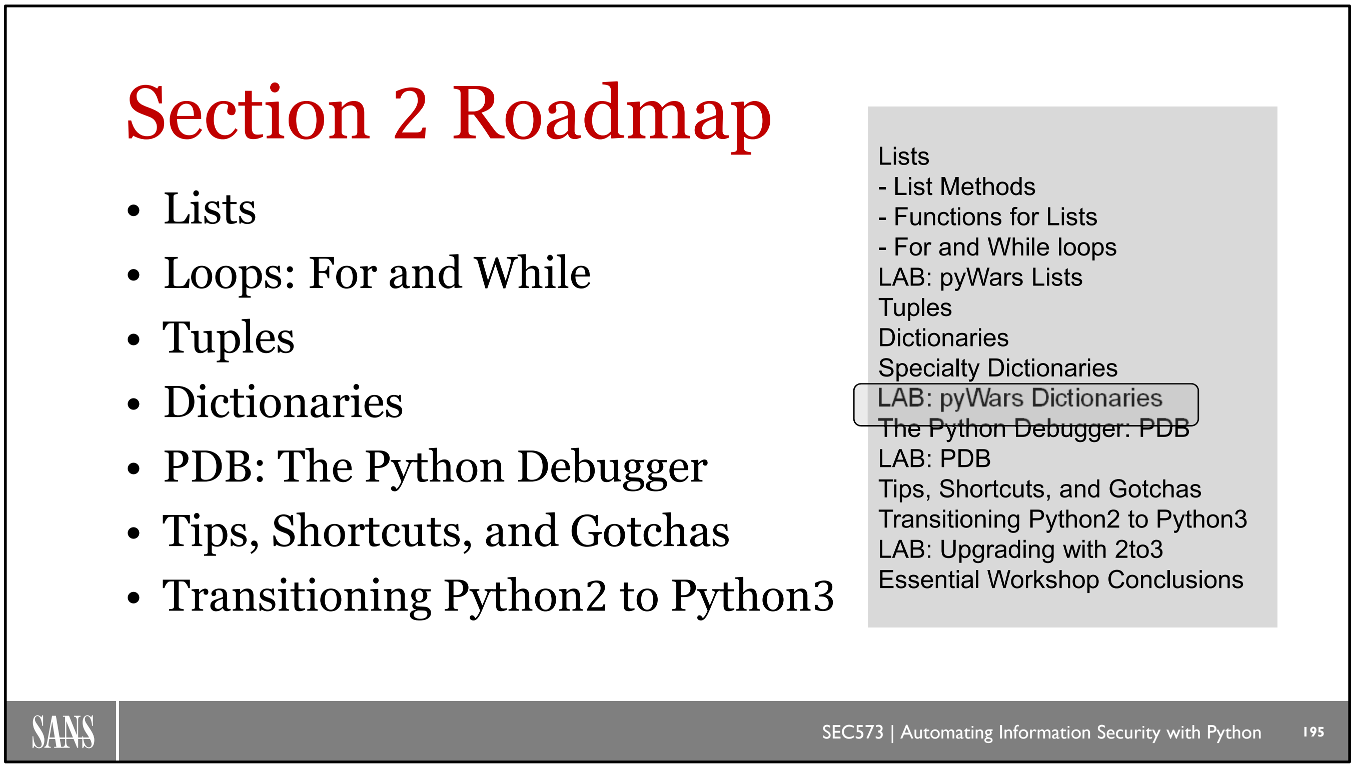
SANS   SEC573 | Automating Information Security with Python   194

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 b'<' in k: del c[k]; continue
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
RuntimeError: dictionary changed size during iteration
```

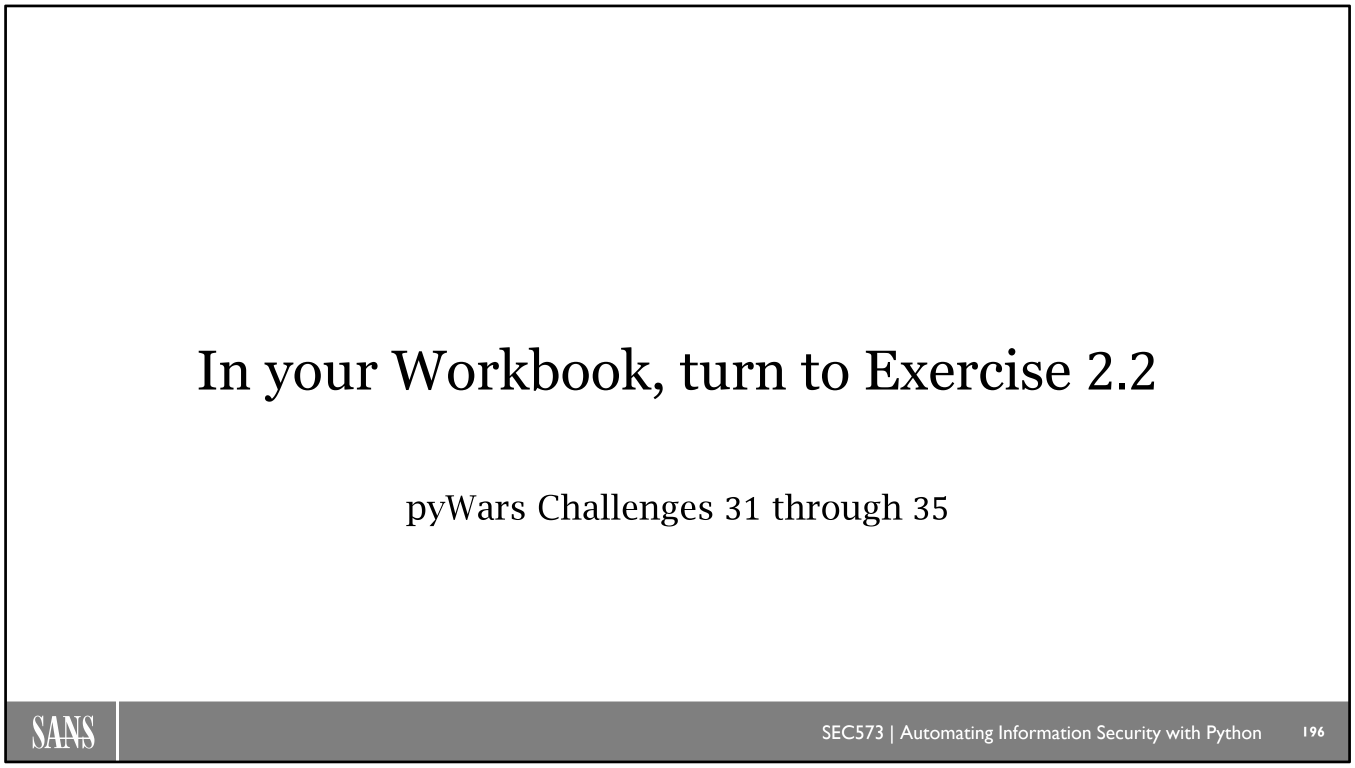
Then, after you've deleted the HTML elements, you can retrieve the most frequently occurring words using the most_common() method.



Section 2 Roadmap

- Lists
- Loops: For and While
- Tuples
- Dictionaries
- PDB: The Python Debugger
- Tips, Shortcuts, and Gotchas
- Transitioning Python2 to Python3

This is a Roadmap slide.



In your Workbook, turn to Exercise 2.2

pyWars Challenges 31 through 35

Please complete the exercise in your Workbook.

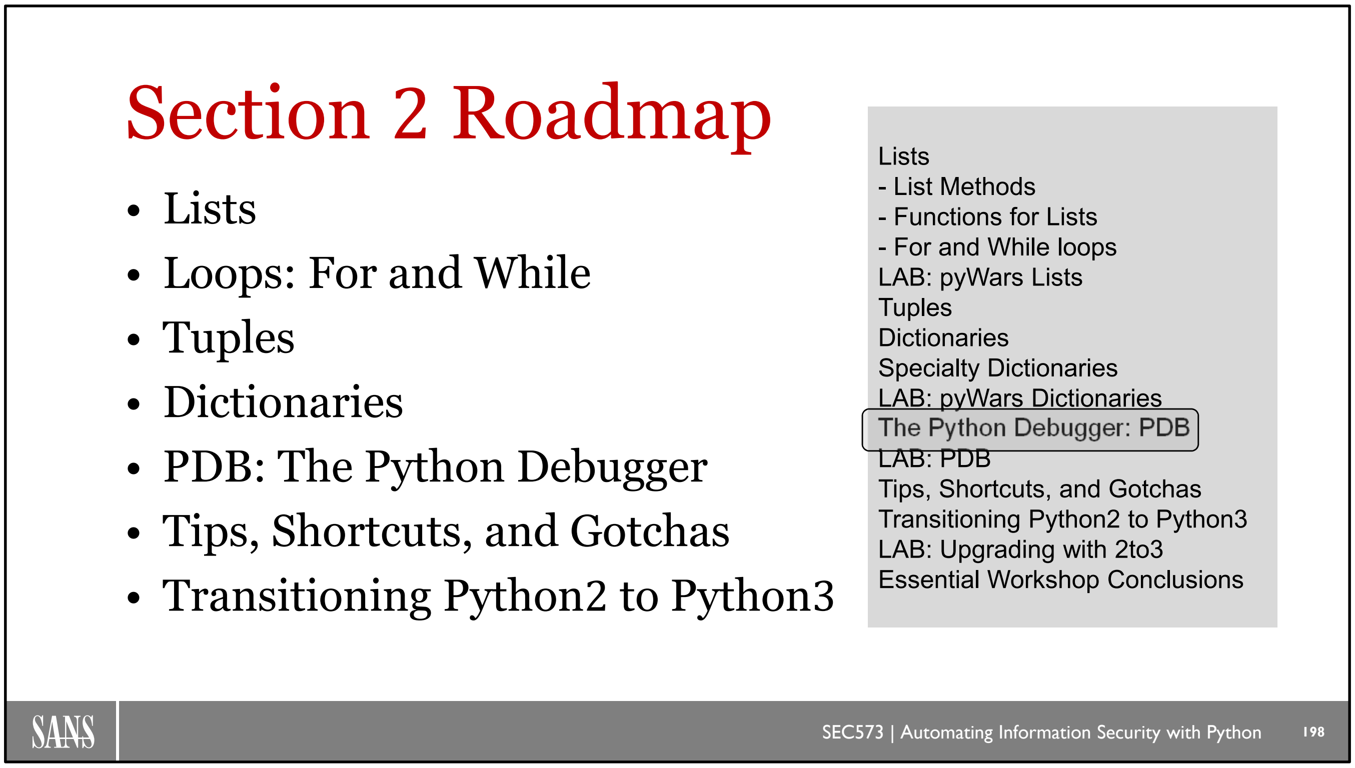
Lab Highlights: Getting Data In and Out of Dictionaries Is Easy/Fast

- .get(), .keys(), .values(), and .items() give you easy access to the data in the dictionary

```
>>> d.question(34)
'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(34)
>>> 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.

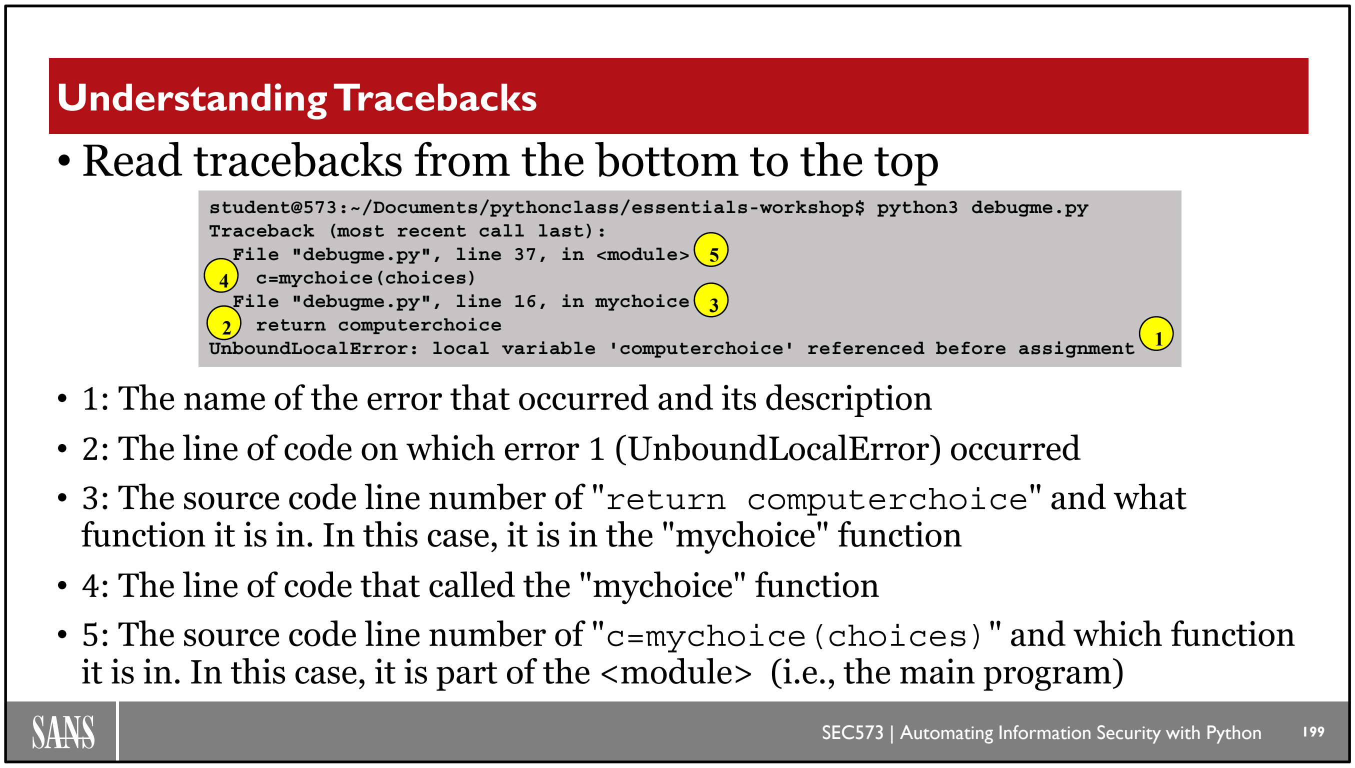
```
def answer34(thedata):
    return thedata.get('python')+thedata.get('rocks')
```



Section 2 Roadmap

- Lists
- Loops: For and While
- Tuples
- Dictionaries
- PDB: The Python Debugger
- Tips, Shortcuts, and Gotchas
- Transitioning Python2 to Python3

This is a Roadmap slide.



Understanding Tracebacks

- Read tracebacks from the bottom to the top

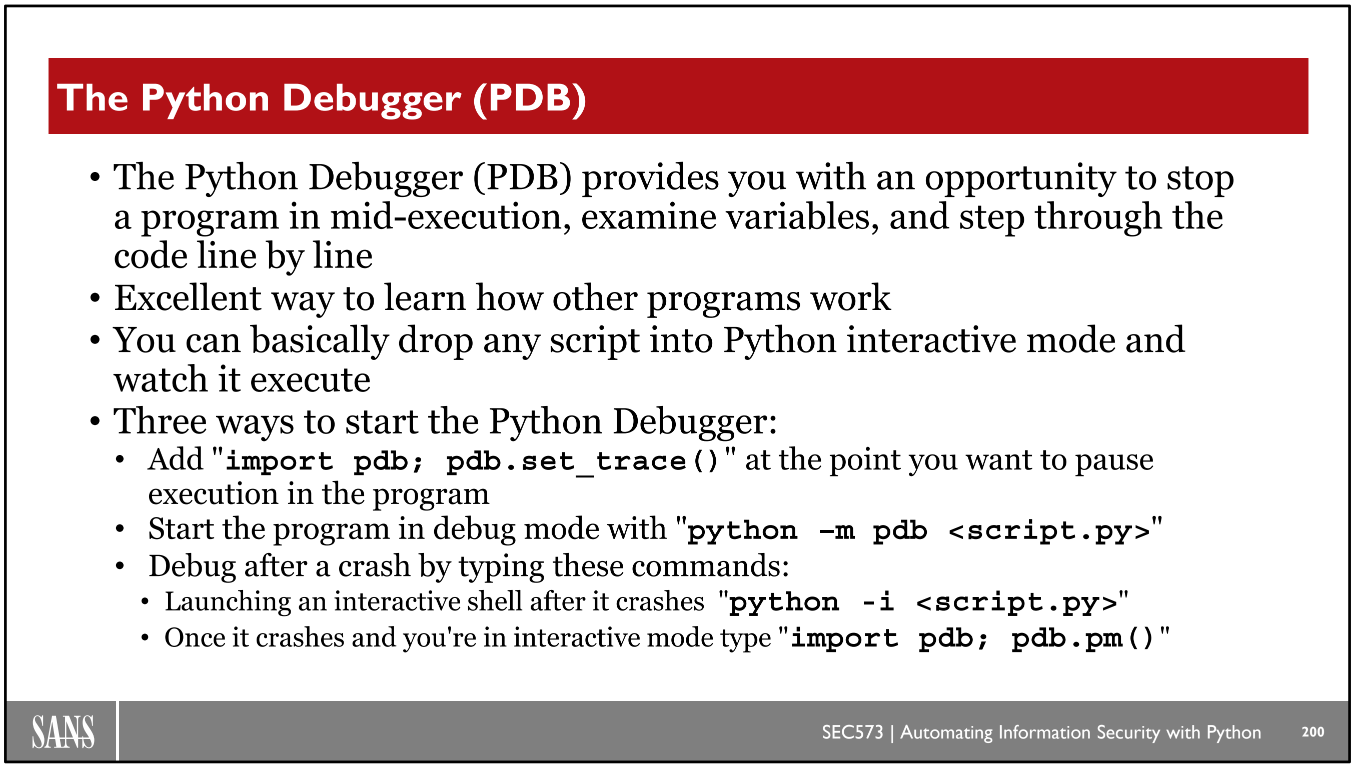
```
student@573:~/Documents/pythonclass/essentials-workshop$ python3 debugme.py
Traceback (most recent call last):
  File "debugme.py", line 37, in <module> 5
4   c=mychoice(choices)
  File "debugme.py", line 16, in mychoice 3
2   return computerchoice
UnboundLocalError: local variable 'computerchoice' referenced before assignment 1
```

- 1: The name of the error that occurred and its description
- 2: The line of code on which error 1 (UnboundLocalError) occurred
- 3: The source code line number of "`return computerchoice`" and what function it is in. In this case, it is in the "mychoice" function
- 4: The line of code that called the "mychoice" function
- 5: The source code line number of "`c=mychoice(choices)`" and which function it is in. In this case, it is part of the <module> (i.e., the main program)

SANS SEC573 | Automating Information Security with Python 199

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, an "UnboundLocalError" occurred because the "local variable 'computerchoice'" was "referenced before assignment". 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 "return computerchoice" executed the variable, "computerchoice" is being referenced but it has not been assigned yet. Moving up another line (line 3) tells us that the code "return computerchoice" is on line 16 of the script and that it is part of the function "mychoice" in the script. Up one more line we can see the line of code that called the mychoice() function. The line of code in our script "c = mychoice(choices)" is what called the mychoice function. Moving up to line 5, we are told that the line "c = mychoice(choices)" is on line 37 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 "computerchoice" inside of the mychoice function 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 16 and see what is going wrong. Python PDB gives us the ability to create breakpoints and stop the program’s execution when it reaches these critical lines in the code.



The Python Debugger (PDB)

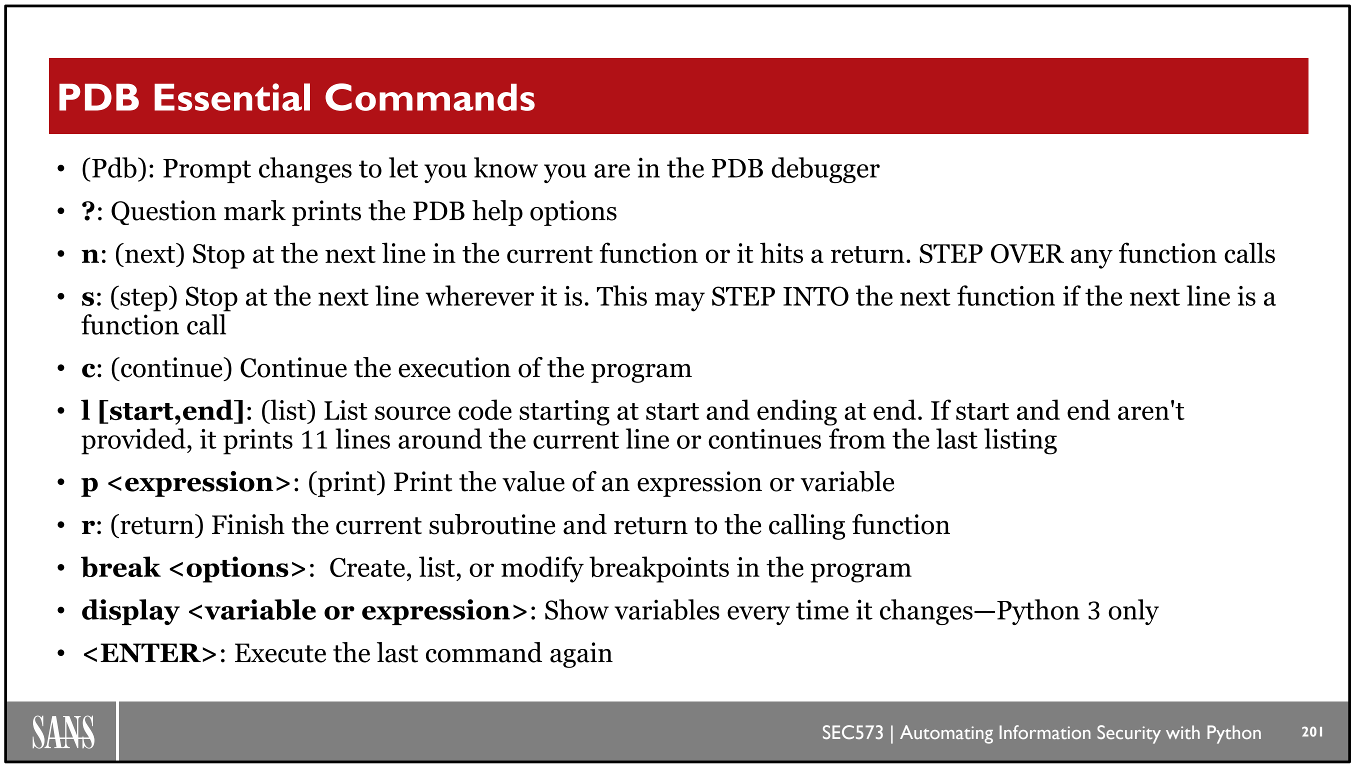
- The Python Debugger (PDB) provides you with an opportunity to stop a program in mid-execution, examine variables, and step through the code line by line
- Excellent way to learn how other programs work
- You can basically drop any script into Python interactive mode and watch it execute
- Three ways to start the Python Debugger:
  - Add "`import pdb; pdb.set_trace()`" at the point you want to pause execution in the program
  - 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()`"

The Python Debugger is a debugging module that is distributed with all Python installations. It is very easy to use when you know a few simple commands, and it is a great way to learn how other Python programs work. By running any Python program through the PDB debugger, you can step through the code line by line and observe the variables changing to see what is happening under the hood.

There are three ways to typically start debugging a program—first is to import the PDB debugger module and then to 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.

A second 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 third 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. The "where" command shows you which function you are currently in inside the program's function stack. The "args" command will show you the arguments that were passed to the current function you are currently inspecting.



PDB Essential Commands

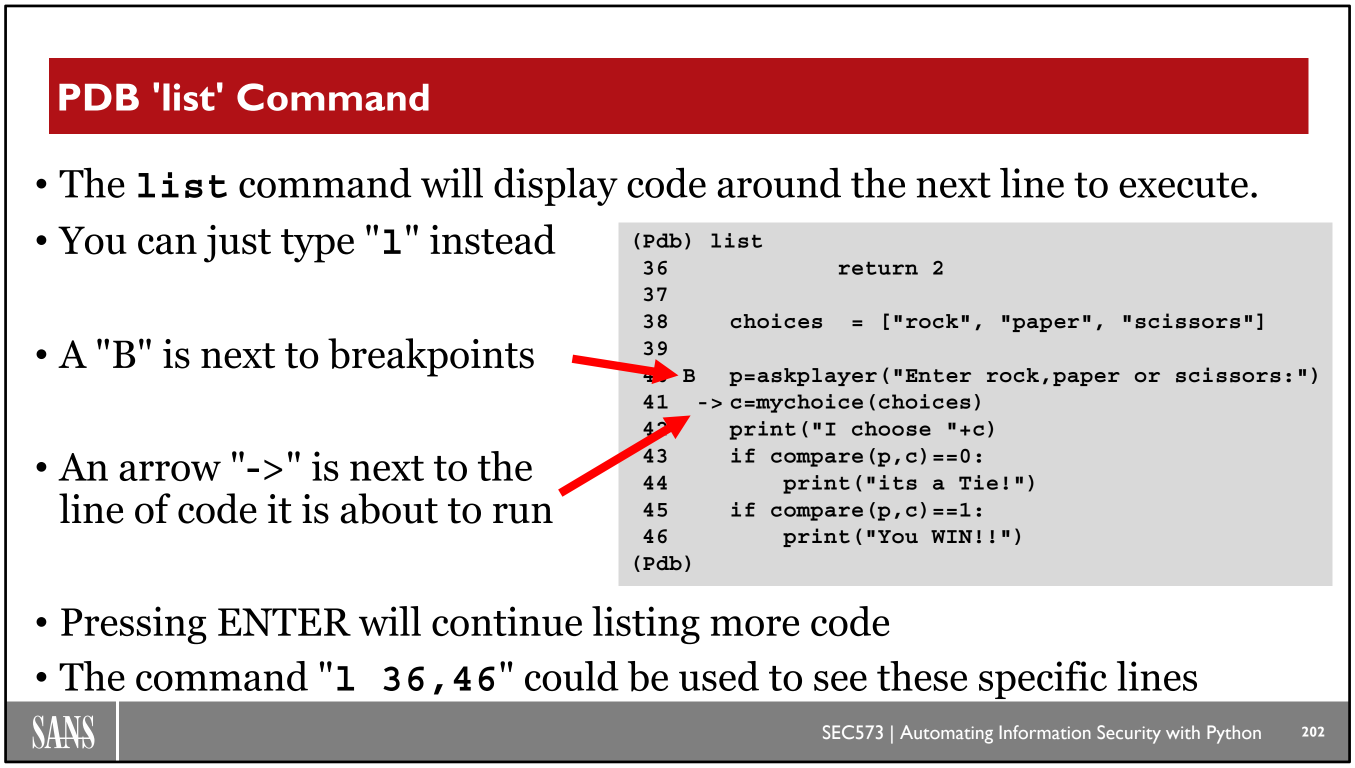
- (Pdb): Prompt changes to let you know you are in the PDB debugger
- **?**: Question mark prints the PDB help options
- **n**: (next) Stop at the next line in the current function or it hits a return. STEP OVER any function calls
- **s**: (step) Stop at the next line wherever it is. This may STEP INTO the next function if the next line is a function call
- **c**: (continue) Continue the execution of the program
- **l [start,end]**: (list) List source code starting at start and ending at end. If start and end aren't provided, it prints 11 lines around the current line or continues from the last listing
- **p <expression>**: (print) Print the value of an expression or variable
- **r**: (return) Finish the current subroutine and return to the calling function
- **break <options>**: Create, list, or modify breakpoints in the program
- **display <variable or expression>**: Show variables every time it changes—Python 3 only
- **<ENTER>**: Execute the last command again

SANS SEC573 | Automating Information Security with Python 201

After you have entered the Python Debugger, your prompt will change to (PDB). Now, you can issue various PDB commands to step through, examine, and alter your program's execution.

- Typing ? (question mark) will print the PDB help.
- The n command executes until it reaches the next line of code.
- The s command executes a single step in your program. We will discuss these two in a little more detail in a second.
- The c command is short for CONTINUE (that is, RUN). It will execute the program until it terminates or until it reaches a line that has been defined as a BREAK using the BREAK command.
- The l command is short for LIST, and it can be used to display lines of source code. The LIST command will also place an ARROW on the listing of the code to show which line of code the debugger is paused on.
- The p command is short for print. It can be used to print the content of variables or other expressions that you want to test.
- The r command is short for return. This will execute the program until the current function (def:) reaches a return and is ready to go back to the calling program.
- The break command is used to create, list, or modify breakpoints in your program.
- The display command only works in Python 3, and it can be used to display the contents of variables every time the contents of a variable changes. This is basically a shortcut to defining break commands that work in both Python 2 and Python 3.

After any of these commands are entered, you can REPEAT the last command entered by simply pressing the **ENTER** key.



PDB 'list' Command

- The `list` command will display code around the next line to execute.
- You can just type "`l`" instead
- A "B" is next to breakpoints
- An arrow "->" is next to the line of code it is about to run

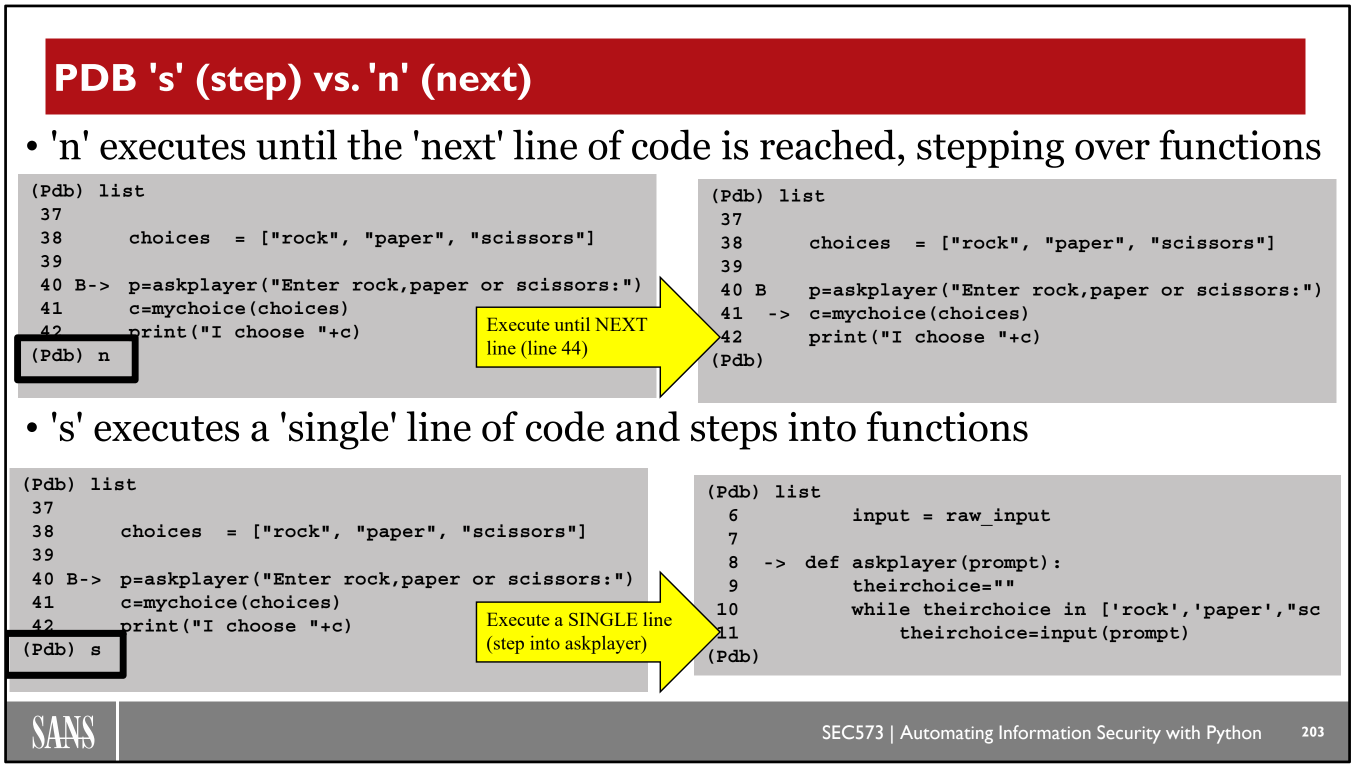
```
(Pdb) list
 36             return 2
 37
 38     choices  = ["rock", "paper", "scissors"]
 39
 40 B   p=askplayer("Enter rock,paper or scissors:")
 41  -> c=mychoice(choices)
 42     print("I choose "+c)
 43     if compare(p,c)==0:
 44         print("its a Tie!")
 45     if compare(p,c)==1:
 46         print("You WIN!!")
(Pdb)
```

- Pressing ENTER will continue listing more code
- The command "`l 36,46`" could be used to see these specific lines

The PDB 'list' command will display lines of source code. It can be abbreviated by just typing its first letter, 'l'. When you list your code, you will see line numbers and source code. You will also see a capital "B" next to any lines where the debugger has a "Breakpoint" defined. An arrow "->" will be printed next to the next line of code that the debugger is going to execute.

The list command will show the lines of code around the line that is about to execute the first time it is run. Each subsequent run will display more lines of code, continuing to list lines from where the previous list command finished. When another line of code is executed, then the list command will again center itself around that line of code. This makes it very easy to execute a command and then list the contents to watch the arrow move through the lines of code and see what is happening in your program.

The list command can also be followed by a starting and ending line. If those are provided, the list command will display those specific lines.



PDB 's' (step) vs. 'n' (next)

- 'n' executes until the 'next' line of code is reached, stepping over functions

```
(Pdb) list
 37
 38      choices  = ["rock", "paper", "scissors"]
 39
 40 B->  p=askplayer("Enter rock,paper or scissors:")
 41      c=mychoice(choices)
 42      print("I choose "+c)
(Pdb) n
```

Execute until NEXT line (line 44)

```
(Pdb) list
 37
 38      choices  = ["rock", "paper", "scissors"]
 39
 40 B    p=askplayer("Enter rock,paper or scissors:")
 41   -> c=mychoice(choices)
 42      print("I choose "+c)
(Pdb)
```

- 's' executes a 'single' line of code and steps into functions

```
(Pdb) list
 37
 38      choices  = ["rock", "paper", "scissors"]
 39
 40 B->  p=askplayer("Enter rock,paper or scissors:")
 41      c=mychoice(choices)
 42      print("I choose "+c)
(Pdb) s
```

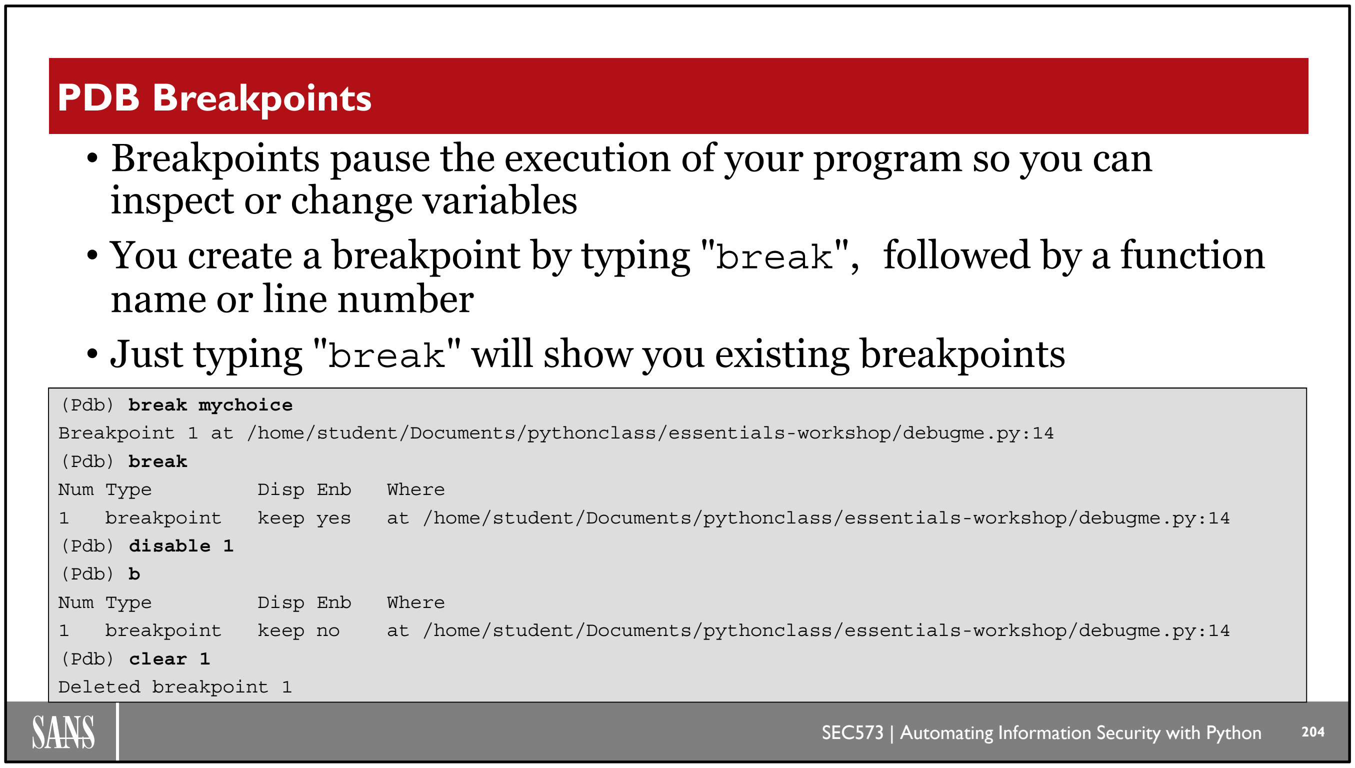
Execute a SINGLE line (step into askplayer)

```
(Pdb) list
  6          input = raw_input
  7
  8   -> def askplayer(prompt):
  9          theirchoice=""
 10          while theirchoice in ['rock','paper',"sc
 11              theirchoice=input(prompt)
(Pdb)
```

SANS SEC573 | Automating Information Security with Python 203

The 'n' command will execute the program, pausing at the NEXT line in the source code. The next line in the source code is not always the next line in the program's execution. For example, consider the lines of code shown in the upper-left corner of the slide above. Notice the arrow "->" is on line 40. If I enter the command 'n', it will stop on line 41. This is not a single line of code. There may be thousands of lines of code that execute between line 40 and line 41. Line 40 contains a call to the function askplayer. The function askplayer may in turn call several other functions. Those functions may call other functions and so on. The 'n' command will execute all of those lines until it reaches the next line. This, in effect, will STEP OVER any function calls.

The s command will step into the execution of a program and execute the next line in the program's logical execution. The s command executes one line of code and pauses again, showing you the next command to be executed. The bottom left corner of the slide above shows the arrow on line 40. When we issue the 's' command, we STEP INTO the askplayer function stopping on line 8 and watch that function execute.



PDB Breakpoints

- Breakpoints pause the execution of your program so you can inspect or change variables
- You create a breakpoint by typing "`break`", followed by a function name or line number
- Just typing "`break`" will show you existing breakpoints

```
(Pdb) break mychoice
Breakpoint 1 at /home/student/Documents/pythonclass/essentials-workshop/debugme.py:14
(Pdb) break
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at /home/student/Documents/pythonclass/essentials-workshop/debugme.py:14
(Pdb) disable 1
(Pdb) b
Num Type         Disp Enb   Where
1   breakpoint   keep no    at /home/student/Documents/pythonclass/essentials-workshop/debugme.py:14
(Pdb) clear 1
Deleted breakpoint 1
```

SANS | SEC573 | Automating Information Security with Python 204

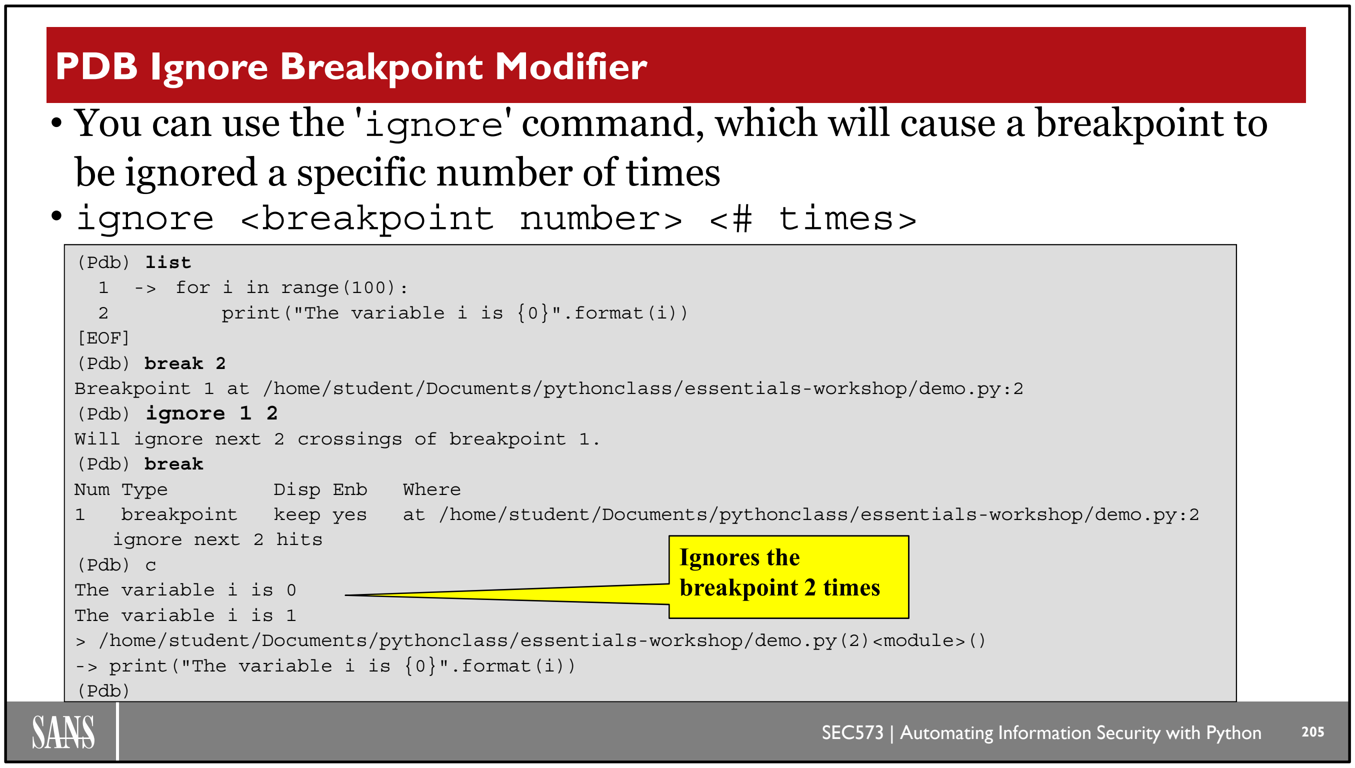
You create breakpoints in the program telling PDB that you want to do something when that line of code is reached. To create a breakpoint, you issue the command 'break', followed by either the name of a function or a line of code on which you want to create the breakpoint. PDB will create the breakpoint and print a status line that tells you the breakpoint number. In the slide above, after we type 'break mychoice', we get a message indicating that "Breakpoint 1" was created in the debugme.py program on line 14.

If you have a breakpoint with no modifiers on it, then every time that line of code is reached, PDB will pause the execution of the program and return to the PDB prompt, where you can use PDB commands to inspect or change variables and otherwise change the flow of your program’s execution.

Just typing 'break' will list the current breakpoints that are defined in PDB.

You can 'enable' or 'disable' breakpoints.

The 'clear' command will erase a breakpoint.



PDB Ignore Breakpoint Modifier

- You can use the '`ignore`' command, which will cause a breakpoint to be ignored a specific number of times
- `ignore <breakpoint number> <# times>`

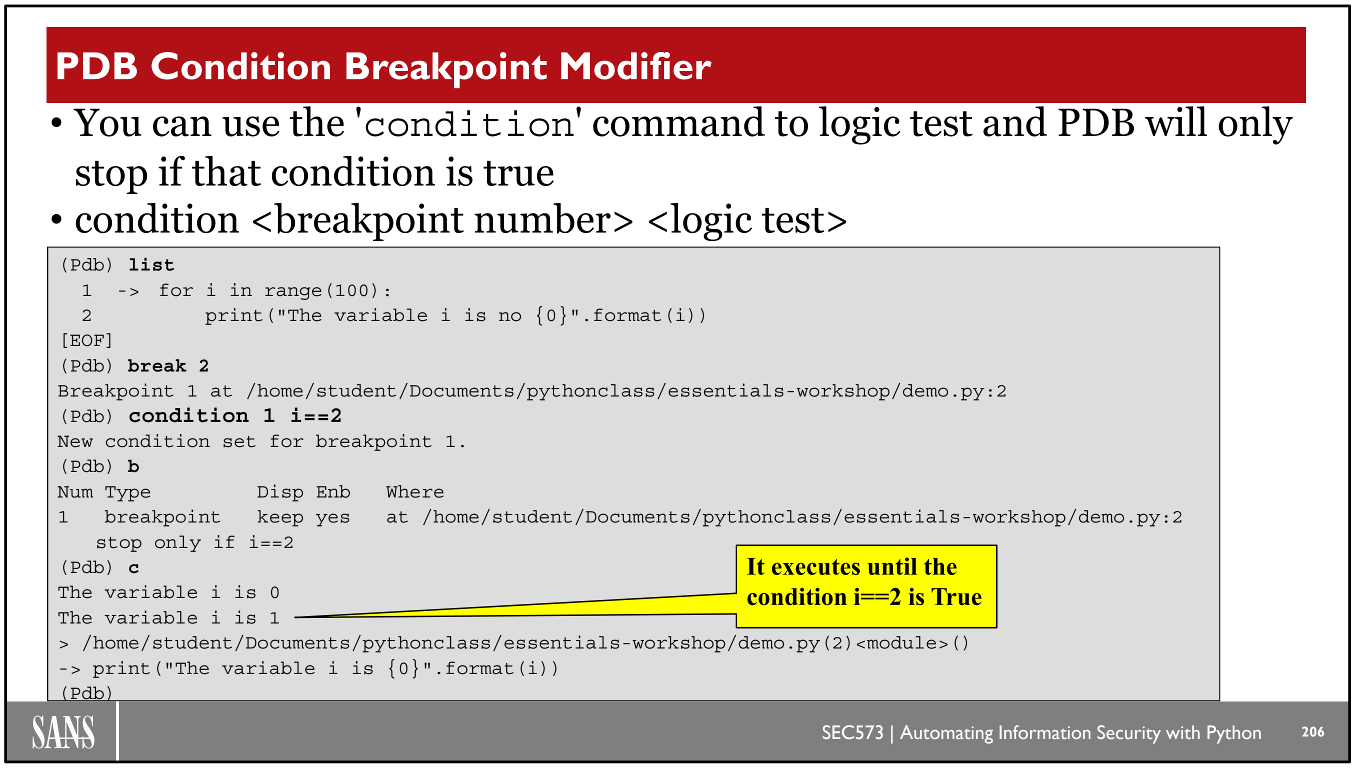
```
(Pdb) list
  1  -> for i in range(100):
  2         print("The variable i is {0}".format(i))
[EOF]
(Pdb) break 2
Breakpoint 1 at /home/student/Documents/pythonclass/essentials-workshop/demo.py:2
(Pdb) ignore 1 2
Will ignore next 2 crossings of breakpoint 1.
(Pdb) break
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at /home/student/Documents/pythonclass/essentials-workshop/demo.py:2
    ignore next 2 hits
(Pdb) c
The variable i is 0
The variable i is 1
> /home/student/Documents/pythonclass/essentials-workshop/demo.py(2)<module>()
-> print("The variable i is {0}".format(i))
(Pdb)
```

Ignores the breakpoint 2 times

SANS | SEC573 | Automating Information Security with Python 205

You can also use the ignore command to tell PDB to ignore the next X times that you reach a specific breakpoint. For example, if you have a breakpoint in a for loop but only need to pause execution on the 100th time you are in the loop, you could type **ignore <breakpoint number> 99**. The next 99 times the breakpoint is reached will not pause the execution of the script.

In this example, we have a for loop that executes 10 times. We want to watch what happens on line 2 the third time through the loop. First, we create a breakpoint on line 2 by typing 'break 2'. It comes back and tells us that "Breakpoint 1" was created. Then we tell PDB to ignore breakpoint 1 2 times by typing 'ignore 1 2'. Now when the code is executed by typing 'c' for continue, it ignores the breakpoint the first two times through the loop.



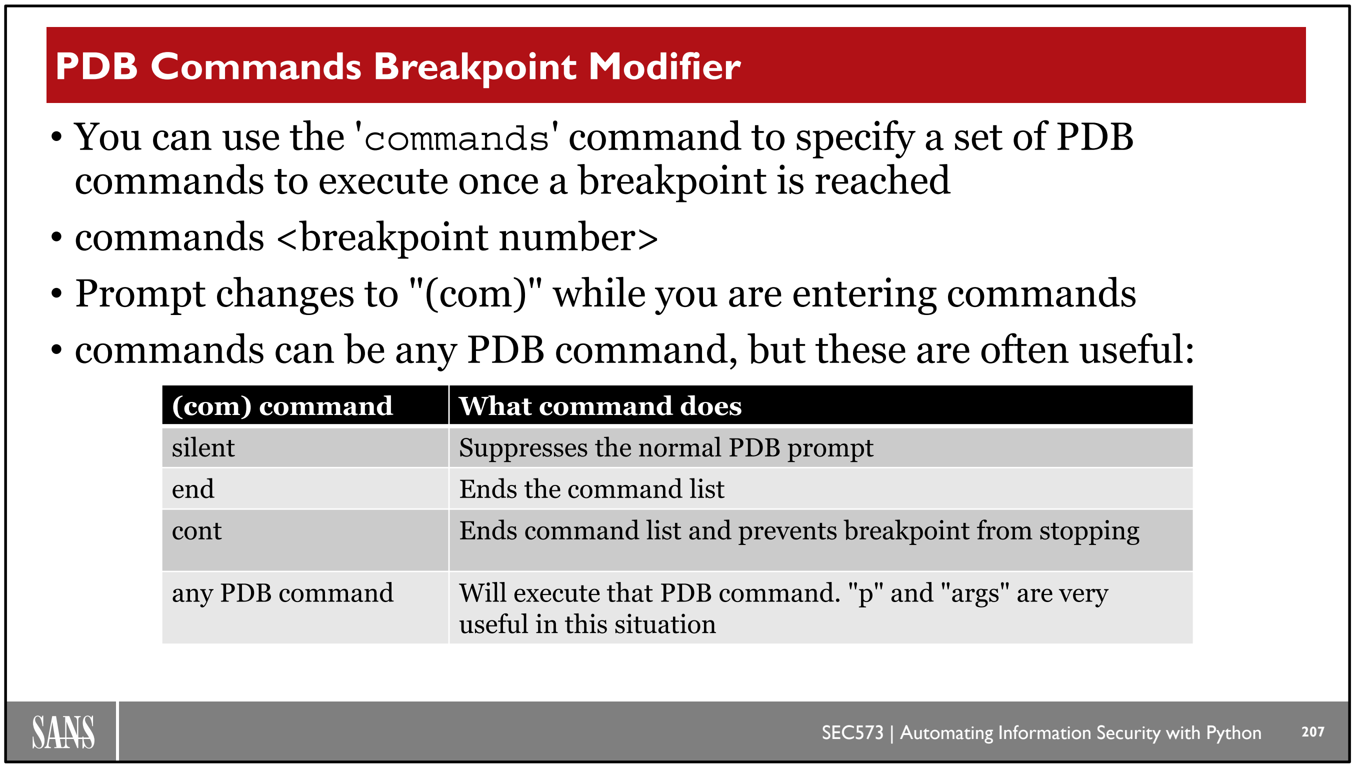
PDB Condition Breakpoint Modifier

- You can use the '`condition`' command to logic test and PDB will only stop if that condition is true
- condition <breakpoint number> <logic test>

```
(Pdb) list
  1  -> for i in range(100):
  2         print("The variable i is no {0}".format(i))
[EOF]
(Pdb) break 2
Breakpoint 1 at /home/student/Documents/pythonclass/essentials-workshop/demo.py:2
(Pdb) condition 1 i==2
New condition set for breakpoint 1.
(Pdb) b
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at /home/student/Documents/pythonclass/essentials-workshop/demo.py:2
    stop only if i==2
(Pdb) c
The variable i is 0
The variable i is 1
> /home/student/Documents/pythonclass/essentials-workshop/demo.py(2)<module>()
-> print("The variable i is {0}".format(i))
(Pdb)
```

SEC573 | Automating Information Security with Python 206

You can make breakpoints 'conditional' so that they will pause only when certain conditions are true. For example, if your program crashes only when a certain variable has the value of '2' in it, you can create a conditional breakpoint so that you do not have to step through the code for all of the conditions that are not causing the crash. To create a conditional breakpoint, you first create a normal breakpoint. Python will tell you what that breakpoint's number is. In this example, we create a breakpoint on line 2 by typing '**break 2**', and Python tells us that "breakpoint 1" was created. Now you use the "condition" command to specify the condition for breakpoint 1. In this case, the syntax is **`condition 1 i == 2`**. Now when the code is run by typing 'c' (for continue), the breakpoint is ignored until the variable i has a value of 2.



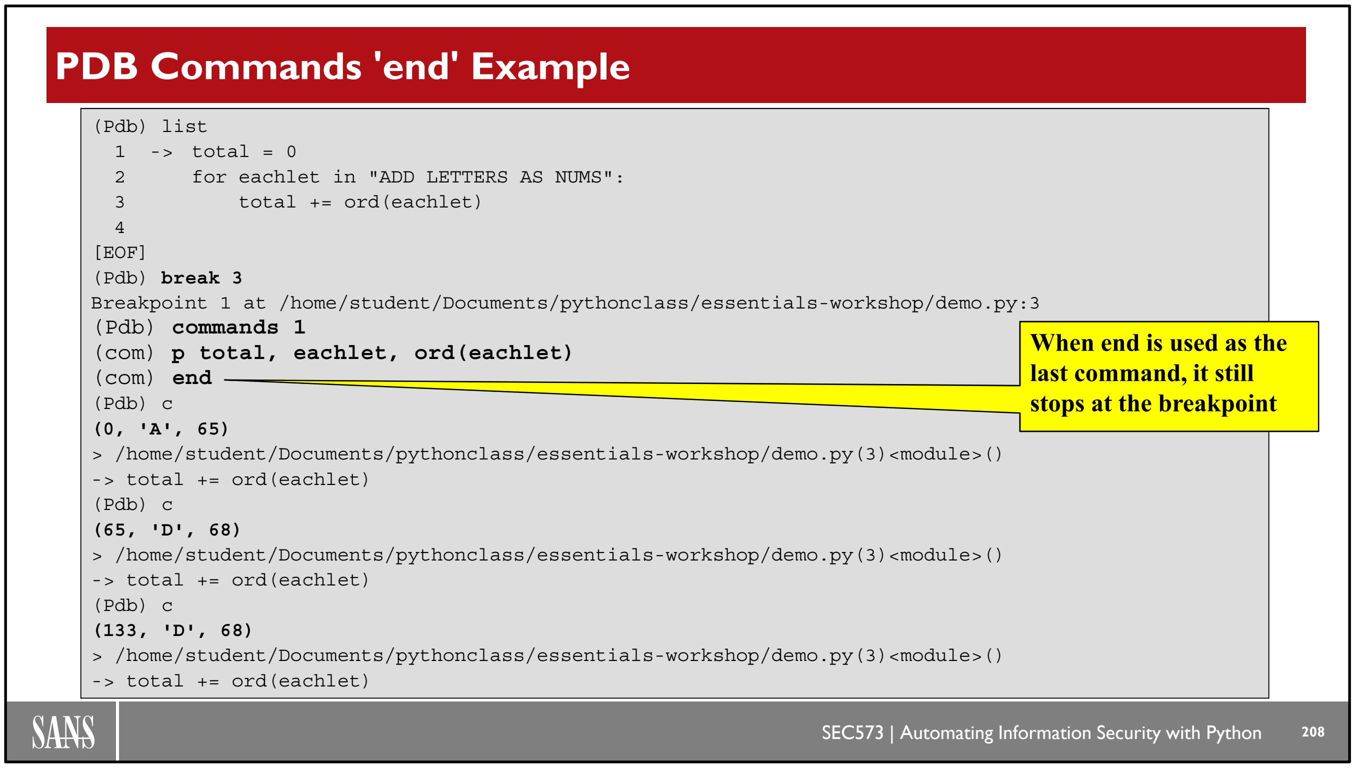
PDB Commands Breakpoint Modifier

- You can use the '`commands`' command to specify a set of PDB commands to execute once a breakpoint is reached
- commands <breakpoint number>
- Prompt changes to "(com)" while you are entering commands
- commands can be any PDB command, but these are often useful:

(com) command	What command does
silent	Suppresses the normal PDB prompt
end	Ends the command list
cont	Ends command list and prevents breakpoint from stopping
any PDB command	Will execute that PDB command. "p" and "args" are very useful in this situation

You can also attach a series of PDB commands to a breakpoint. These commands will automatically execute as soon as the breakpoint is reached. To associate PDB commands with a breakpoint, you type 'commands' followed by a breakpoint number. When you do, the prompt will change to '(com)'. Then you can type multiple PDB commands and end your list of commands with either 'end' or 'cont'.

For example, you can automatically run commands that display the contents of variables every time a breakpoint is reached. This is very useful for watching changes in variables as they occur. In Python 3, the display command will automatically perform these actions for you, but commands are much more powerful than the display command. In addition to all of the PDB commands, you can also issue the commands 'silent' and 'end'. 'silent' will prevent PDB from printing the normal PDB prompt. 'end' is how you tell PDB that you are done listing commands to associate with a breakpoint. You will typically type this as the last command in a list of PDB commands to run. Alternatively, you may end a list of commands with the command 'cont'. The 'cont' command is the normal PDB command that says to run until the next breakpoint. If that is the last line in your commands list, then the breakpoint doesn't stop execution after running your PDB commands. This gives you a lot of flexibility and allows you to trace program changes through large loops.



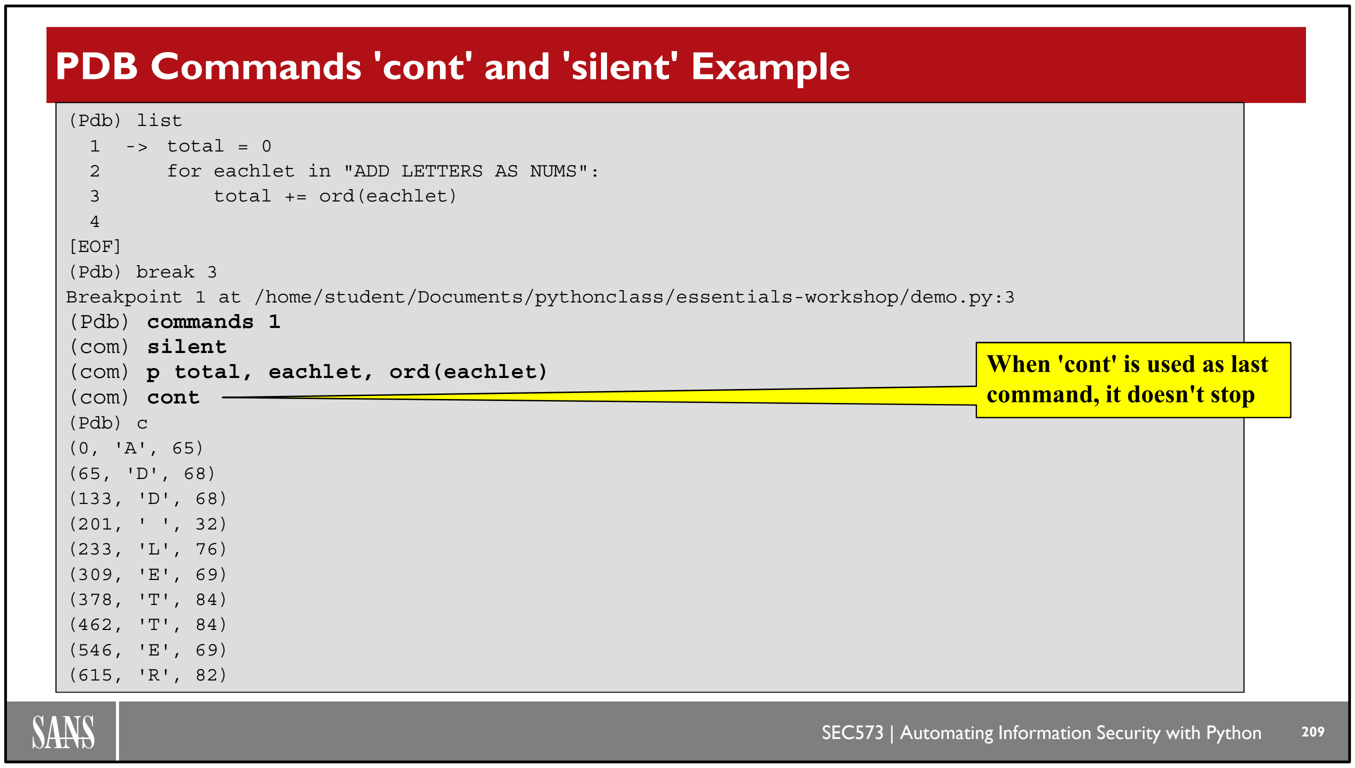
PDB Commands 'end' Example

```
(Pdb) list
  1  -> total = 0
  2     for eachlet in "ADD LETTERS AS NUMS":
  3         total += ord(eachlet)
  4
[EOF]
(Pdb) break 3
Breakpoint 1 at /home/student/Documents/pythonclass/essentials-workshop/demo.py:3
(Pdb) commands 1
(com) p total, eachlet, ord(eachlet)
(com) end
(Pdb) c
(0, 'A', 65)
> /home/student/Documents/pythonclass/essentials-workshop/demo.py(3)<module>()
-> total += ord(eachlet)
(Pdb) c
(65, 'D', 68)
> /home/student/Documents/pythonclass/essentials-workshop/demo.py(3)<module>()
-> total += ord(eachlet)
(Pdb) c
(133, 'D', 68)
> /home/student/Documents/pythonclass/essentials-workshop/demo.py(3)<module>()
-> total += ord(eachlet)
```

When end is used as the last command, it still stops at the breakpoint

SANS SEC573 | Automating Information Security with Python 208

We can use commands to display variables every time a breakpoint is reached. In this example, you see a for loop that will accumulate the ASCII values in the a string. First, we need a breakpoint to add commands to, so we issue the commands 'break 3'. PDB now tells you that "Breakpoint 1" was created. We start our list of commands by typing 'commands 1'. The 1 indicates we want to associate these with breakpoint 1. The prompt changes to (com). Then we type the normal PDB command 'p total, eachlet, ord(eachlet)‘, indicating that we want to print the contents or results of those variables and expressions every time the breakpoint is reached. Since that is the only command we want to run, we end our list of commands by typing 'end' and our prompt changes back to (Pdb). Now every time the breakpoint is reached, a tuple containing our three variables is automatically printed for us.



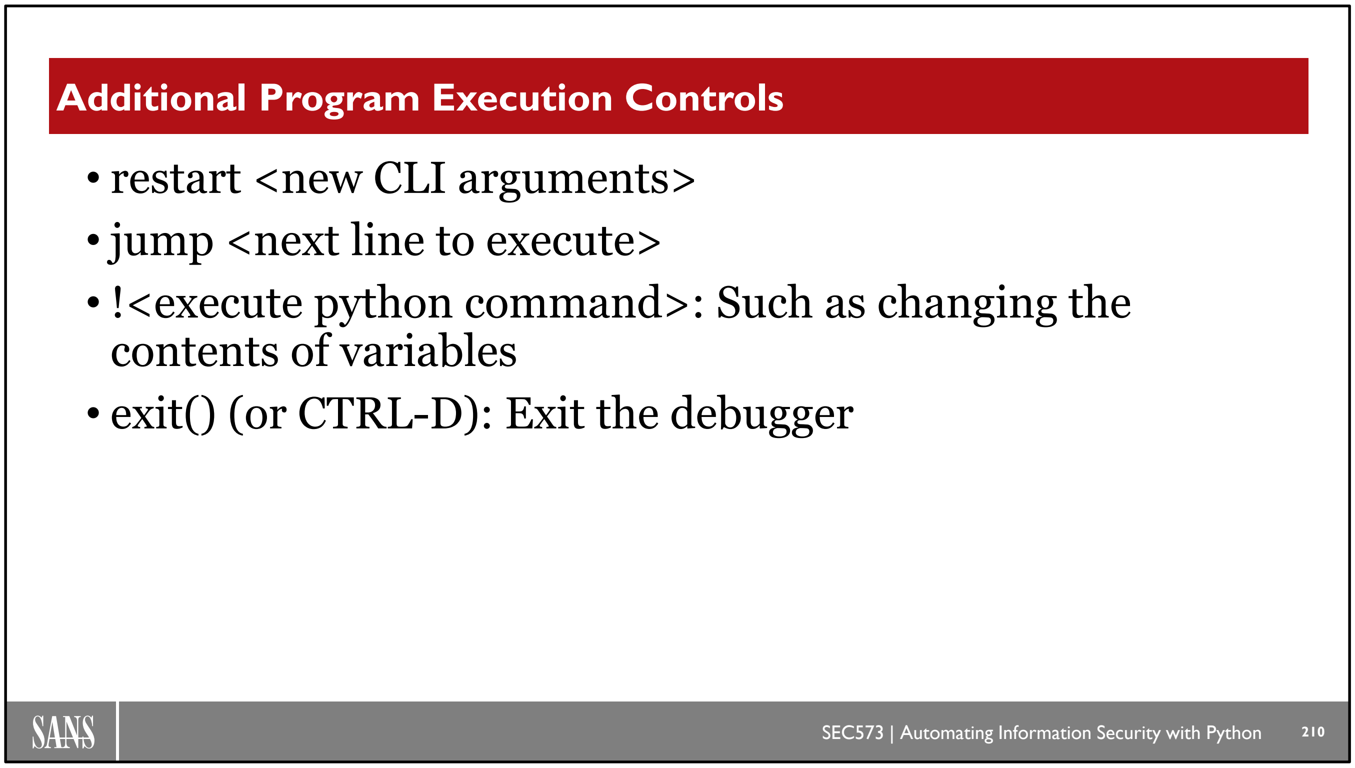
PDB Commands 'cont' and 'silent' Example

```
(Pdb) list
  1  -> total = 0
  2     for eachlet in "ADD LETTERS AS NUMS":
  3         total += ord(eachlet)
  4
[EOF]
(Pdb) break 3
Breakpoint 1 at /home/student/Documents/pythonclass/essentials-workshop/demo.py:3
(Pdb) commands 1
(com) silent
(com) p total, eachlet, ord(eachlet)
(com) cont
(Pdb) c
(0, 'A', 65)
(65, 'D', 68)
(133, 'D', 68)
(201, ' ', 32)
(233, 'L', 76)
(309, 'E', 69)
(378, 'T', 84)
(462, 'T', 84)
(546, 'E', 69)
(615, 'R', 82)
```

When 'cont' is used as last command, it doesn't stop

SANS SEC573 | Automating Information Security with Python 209

Here is another example that shows how we can use commands to watch changes in variables. We create our breakpoint on line 3 just as before. Then we type 'commands 1' to associate our commands with breakpoint 1. This time, we first issue the 'silent' command so that PDB will not print the normal PDB status messages. Then we issue the same print command as before to each of our variables. Last, instead of ending our command list with 'end', we end the command list with 'cont', which can be abbreviated as just the letter 'c'. This tells PDB to restart the program and stop again at the next breakpoint. Because of this command, the breakpoint appears not to stop the program execution. Instead, only the commands associated with the breakpoint are run.



Additional Program Execution Controls

- restart <new CLI arguments>
- jump <next line to execute>
- !<execute python command>: Such as changing the contents of variables
- exit() (or CTRL-D): Exit the debugger

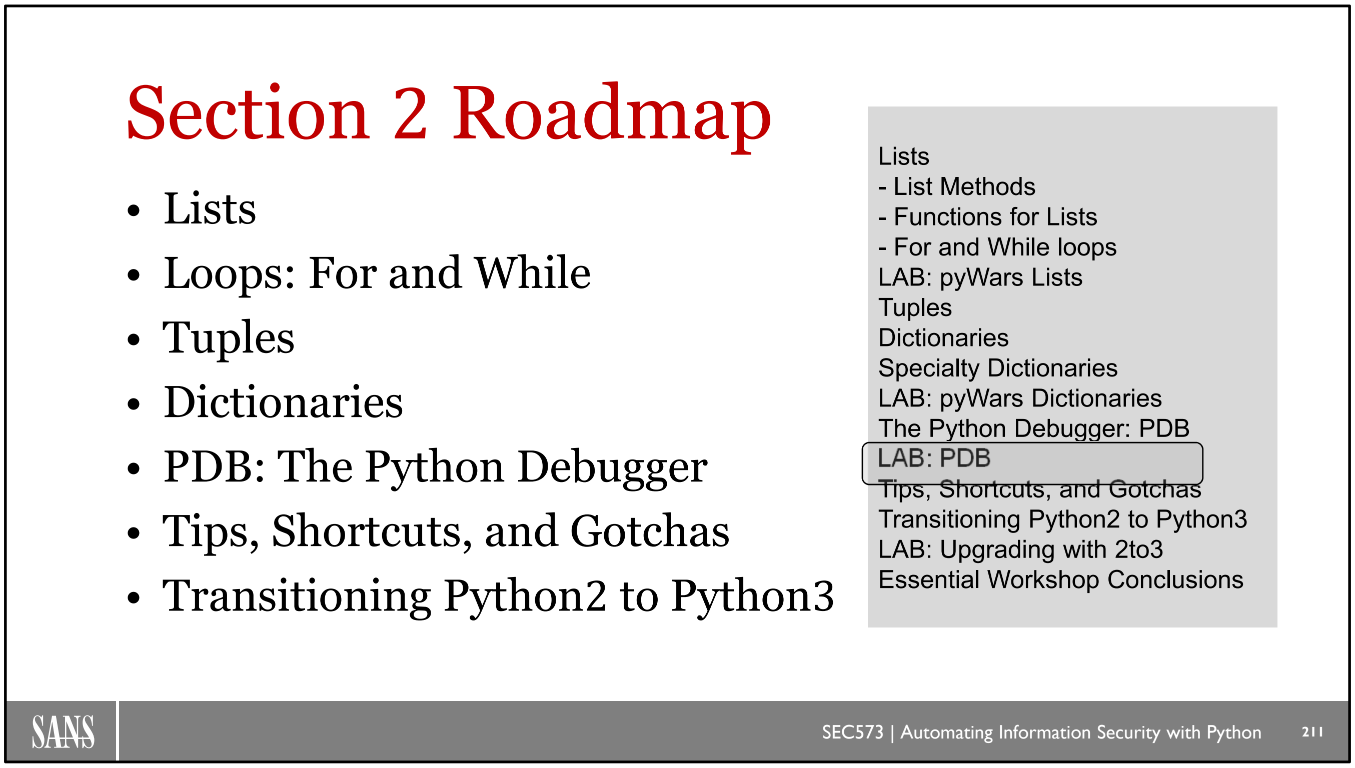
SANS SEC573 | Automating Information Security with Python 210

The restart command can be used to start over the execution of the program from the beginning of the script. All of your breakpoints are preserved when you restart. When you use the restart command, you can optionally provide a new setup of command line arguments that will be passed to the script as it is restarted.

While you are executing a script, you can change the next line of the script with the "jump <next line to execute>" command. In doing so, you can change the execution of your script from a top-down sequential execution to something else to see how it affects the program.

Any non-PDB command that you type at the PDB prompt will be passed on to the Python interpreter and executed. You can use this to change the contents of variables or change the environment in which the script is executing. However, relying on PDB to recognize what is and isn't a PDB command isn't the best option. Instead, you should precede any non-PDB Python commands with the exclamation point. For example, imagine that you're in the debugger and you want to change the contents of the variable "n" to be 100. So, at the PDB prompt, you would type **n = 100**. The letter N has meaning in PDB. It means execute the next command. PDB will assume this is a PDB command unless you precede it with the exclamation mark **"!n = 100"**.

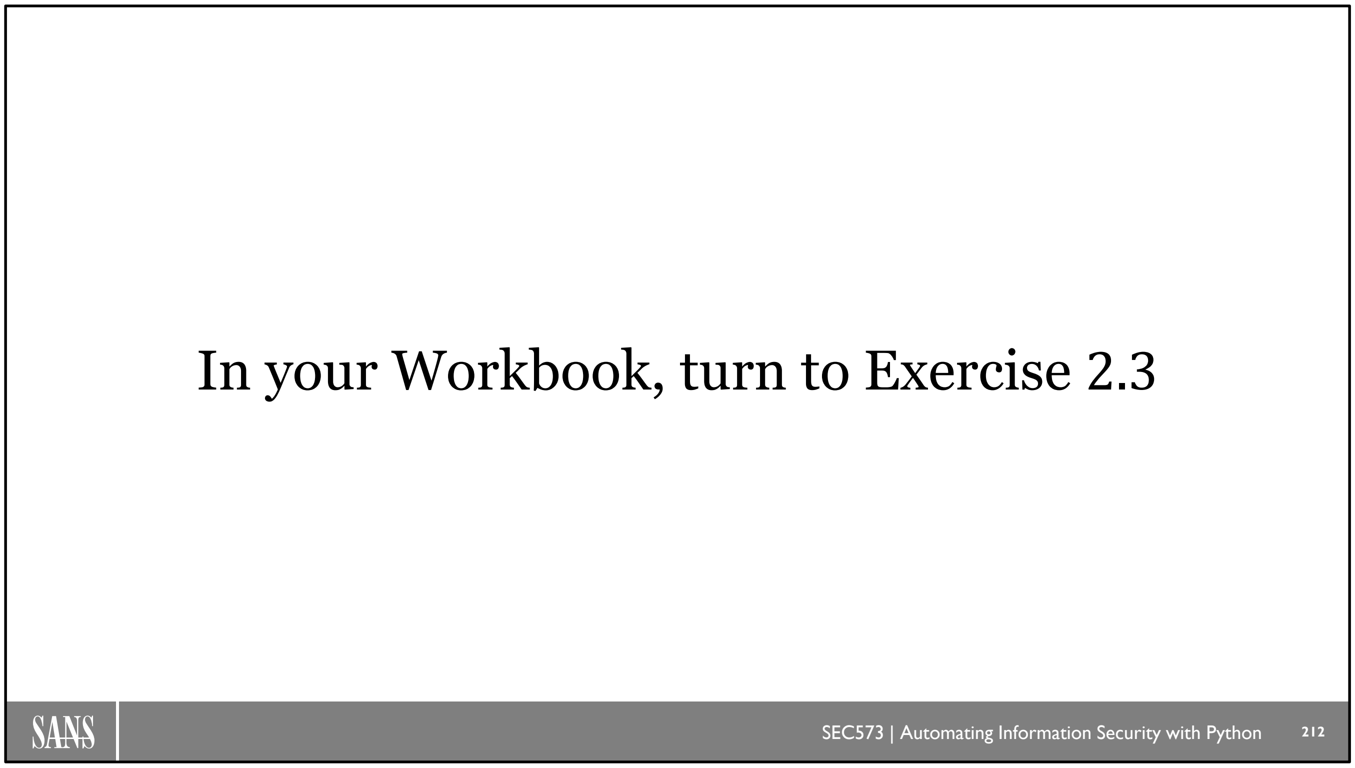
When you are done debugging your program, you can hold down the Control key and press D to quickly exit the debugger. If it doesn't return you to a bash prompt, repeat hitting CTRL-D until it does.



Section 2 Roadmap

- Lists
- Loops: For and While
- Tuples
- Dictionaries
- PDB: The Python Debugger
- Tips, Shortcuts, and Gotchas
- Transitioning Python2 to Python3

This is a Roadmap slide.

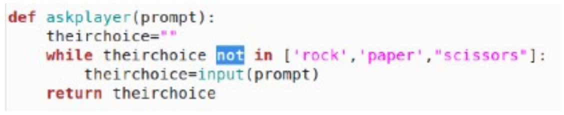


In your Workbook, turn to Exercise 2.3

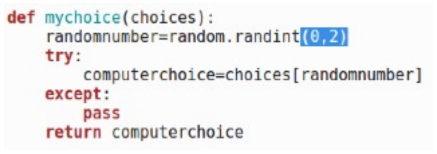
Please complete the exercise in your Workbook.

Lab Highlights: Two Programming Errors

- First, the programmer really wanted to prompt the user over and over until they enter either rock, paper, or scissors
- Add "not" to the code to fix it
- When choosing the random number, we incorrectly choose 1, 2, or 3. Lists start at offset zero! We should have chosen the numbers 0, 1, or 2.



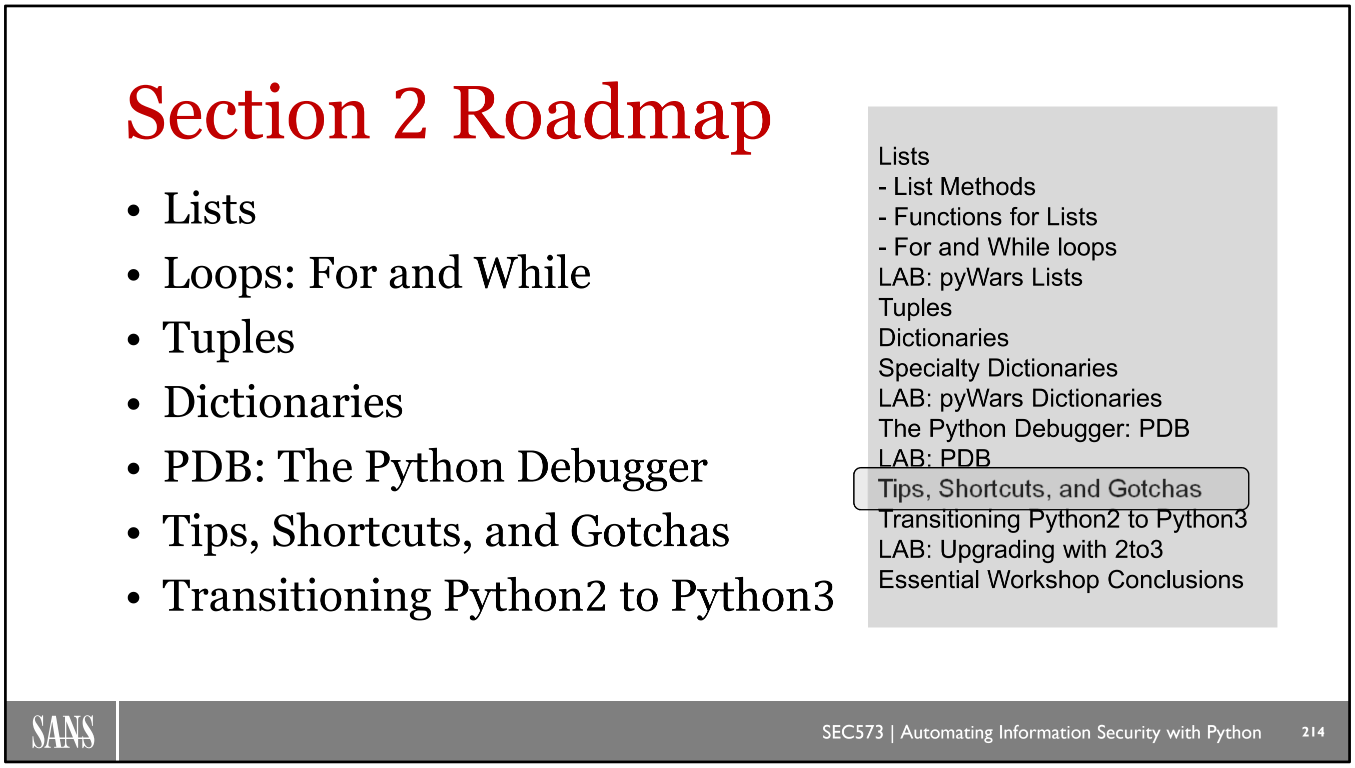
```
def askplayer(prompt):
    theirchoice=""
    while theirchoice not in ['rock','paper',"scissors"]:
        theirchoice=input(prompt)
    return theirchoice
```



```
def mychoice(choices):
    randomnumber=random.randint(0,2)
    try:
        computerchoice=choices[randomnumber]
    except:
        pass
    return computerchoice
```

There were two errors in our rock paper scissors application. First, we needed to put the keyword "not" in our while statement. This will cause Python to prompt the user over and over until they type either 'rock', 'paper', or 'scissors'.

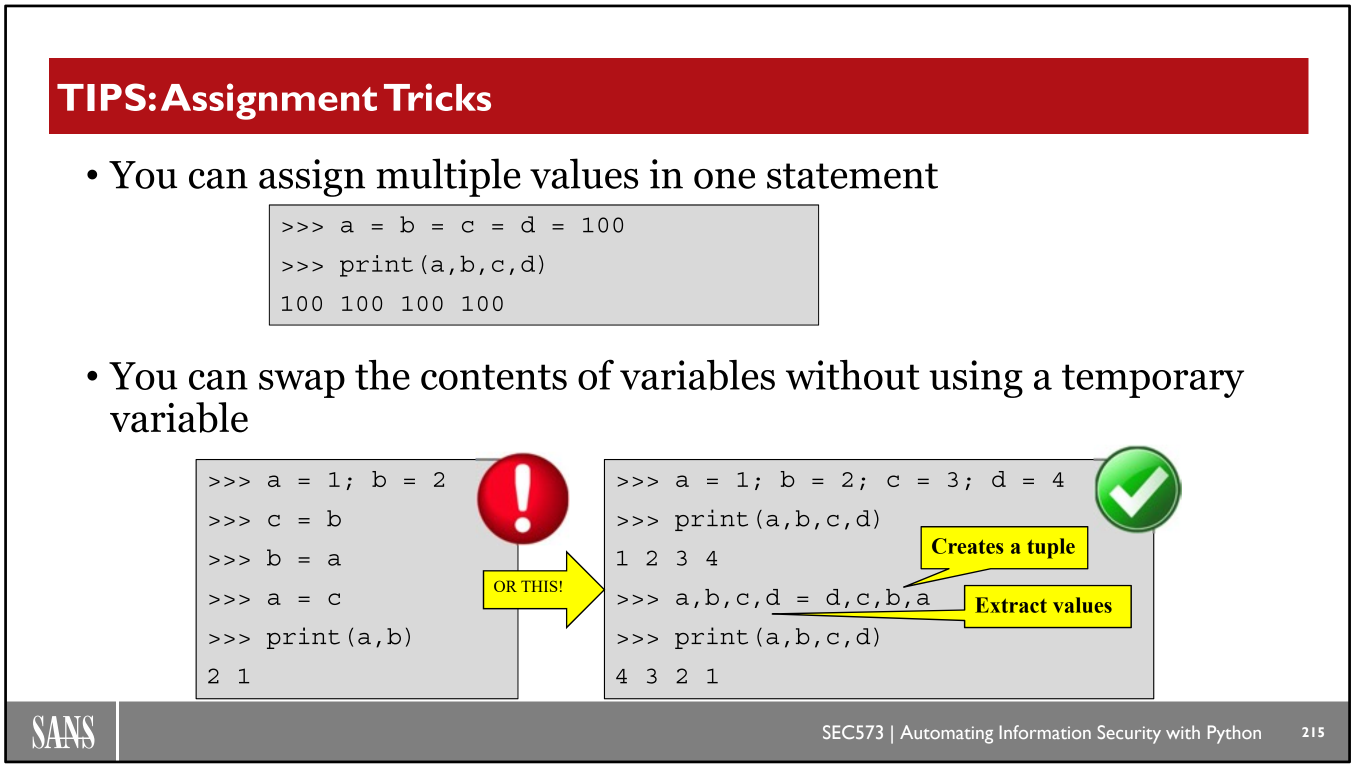
The second error was in our random number generation. Instead of choosing the numbers 0, 1, or 2, our random number is 1, 2, or 3. The list in the variable choices only has items in position 0, 1, and 2. When the number 3 is randomly chosen, it tries to retrieve the item in position 3 of the list choices, and an exception is raised. As a result, the variable computerchoice is not assigned. When the return statement tries to return computerchoice, an error occurs because the computerchoice variable was referenced before it was assigned.



Section 2 Roadmap

- Lists
- Loops: For and While
- Tuples
- Dictionaries
- PDB: The Python Debugger
- Tips, Shortcuts, and Gotchas
- Transitioning Python2 to Python3

This is a Roadmap slide.



TIPS: Assignment Tricks

- You can assign multiple values in one statement

```
>>> a = b = c = d = 100
>>> print(a,b,c,d)
100 100 100 100
```

- You can swap the contents of variables without using a temporary variable

```
>>> a = 1; b = 2
>>> c = b
>>> b = a
>>> a = c
>>> print(a,b)
2 1
```

```
>>> a = 1; b = 2; c = 3; d = 4
>>> print(a,b,c,d)
1 2 3 4
>>> a,b,c,d = d,c,b,a
>>> print(a,b,c,d)
4 3 2 1
```

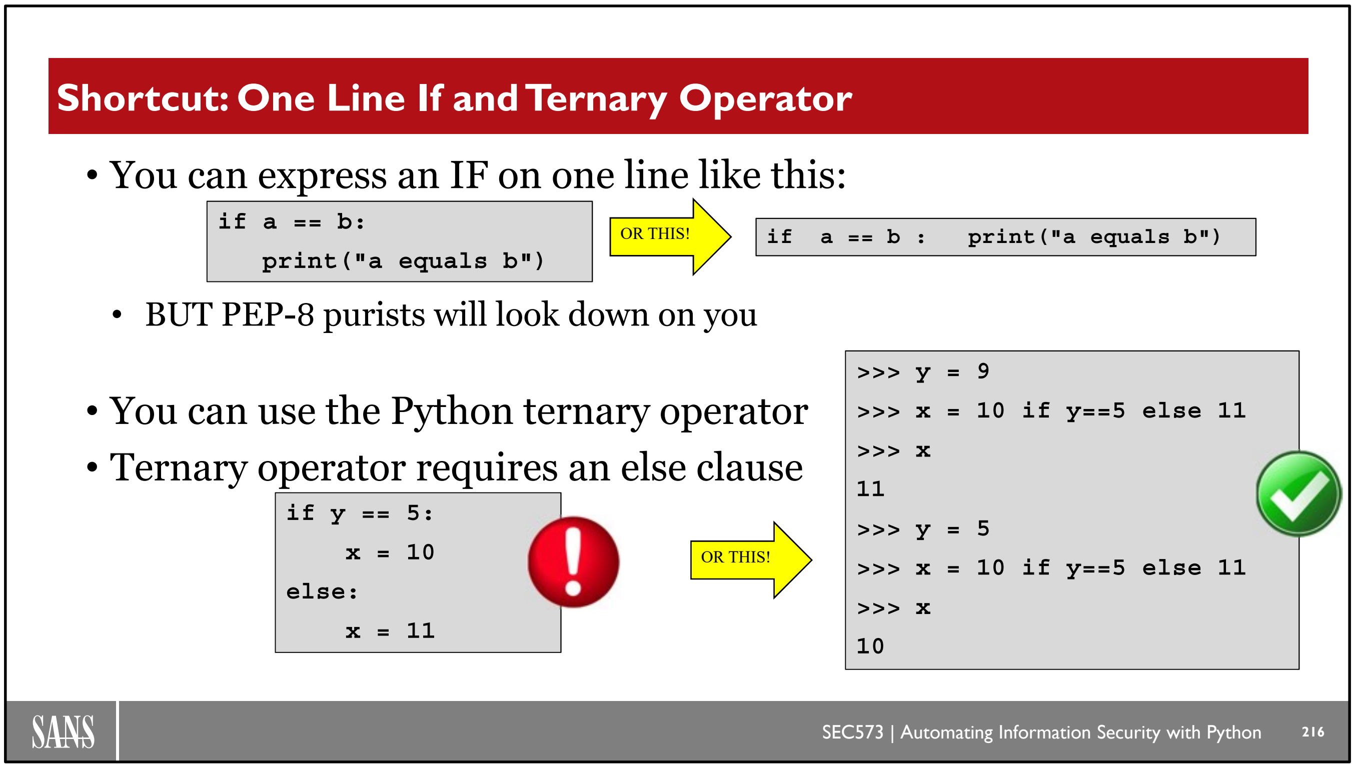
OR THIS!
Creates a tuple
Extract values

A nice shortcut in Python is assigning multiple variables at the same time. Imagine you had to assign multiple variables the value 100. Rather than four separate lines, you could do it in one line, as you see here. To understand why this works, you need to remember that Python processes the right side of the equal sign first. The first thing that happens is 100 is placed in memory. Then the label d is assigned to point to it. Next, the label c points to the contents of the variable d. Next, the label b is assigned to the contents of the variable c. Finally, a is assigned to the contents of variable b.

You can also swap the values stored in variables in one line without the use of temporary holding variables. Imagine you want to swap the contents of variables a and b:

```
>>> a = 1; b = 2
>>> a = b
>>> b = a
>>> print(a,b)
2 2
```

This approach doesn't work because we lose one of our variables with the first assignment. Instead, you have to use a temporary variable to hold one of the values. This becomes troublesome when swapping several variables. Python makes swapping them easy, enabling you to reassign them on a single line.



Shortcut: One Line If and Ternary Operator

- You can express an IF on one line like this:

```
if a == b:
   print("a equals b")
```

OR THIS!

```
if  a == b :   print("a equals b")
```

  - BUT PEP-8 purists will look down on you

- You can use the Python ternary operator
- Ternary operator requires an else clause

```
if y == 5:
    x = 10
else:
    x = 11
```

OR THIS!

```
>>> y = 9
>>> x = 10 if y==5 else 11
>>> x
11
>>> y = 5
>>> x = 10 if y==5 else 11
>>> x
10
```

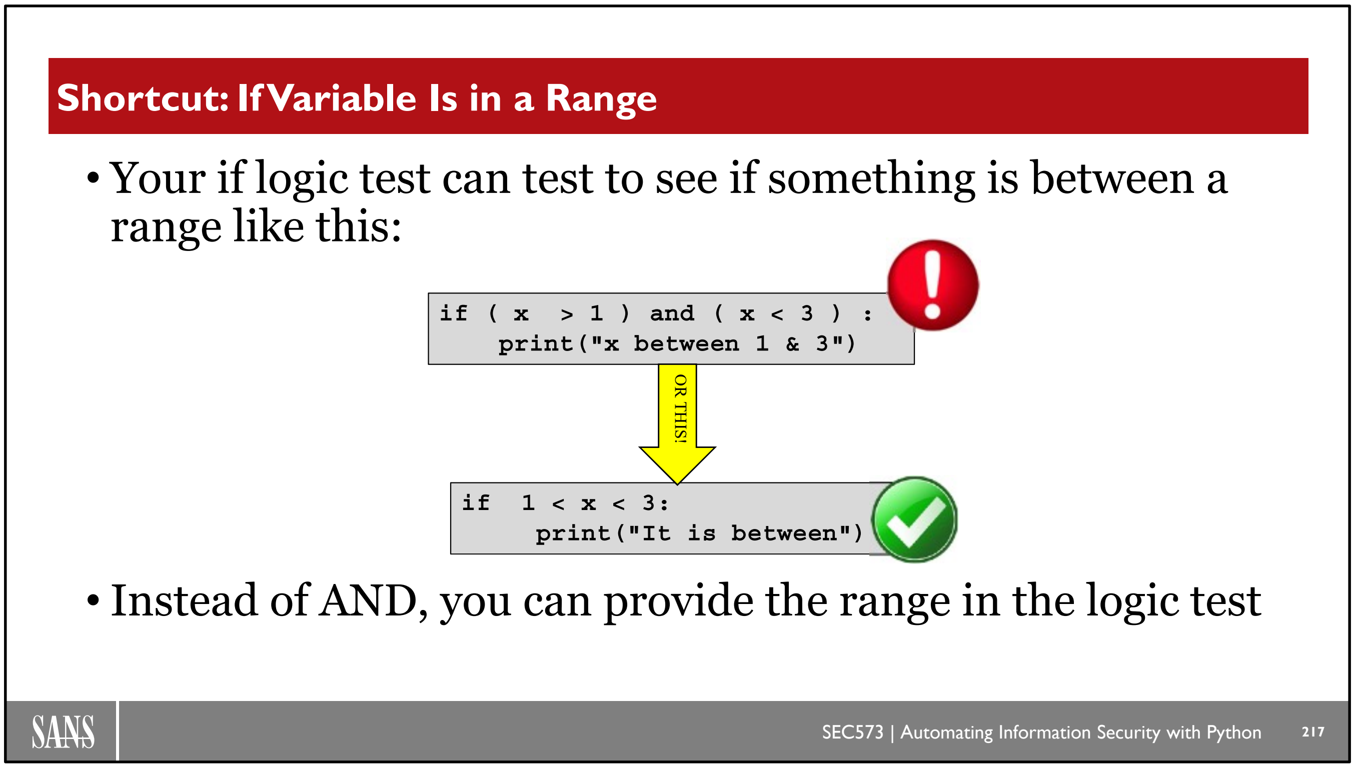
SANS SEC573 | Automating Information Security with Python 216

You can put your if statement on one line by putting the one-line code block after the colon. However, according to PEP-8, "Compound statements (multiple statements on the same line) are generally discouraged." Discouraged but not forbidden sounds like permission to me. That said, the Python purists in the crowd will give you a disapproving stare when they see your code.

If you are assigning a variable, then the Python ternary operator is a shortcut that will keep the purists happy. The line "x = 10 if y==5 else 11" will assign x to the value 10 if the logical test (in this case, y == 5) is true. If it isn't, it will assign x to be 11.

Reference

https://www.python.org/dev/peps/pep-0008/



Shortcut: If Variable Is in a Range

- Your if logic test can test to see if something is between a range like this:

```
if ( x  > 1 ) and ( x < 3 ) :
    print("x between 1 & 3")
```

```
if  1 < x < 3:
     print("It is between")
```

- Instead of AND, you can provide the range in the logic test

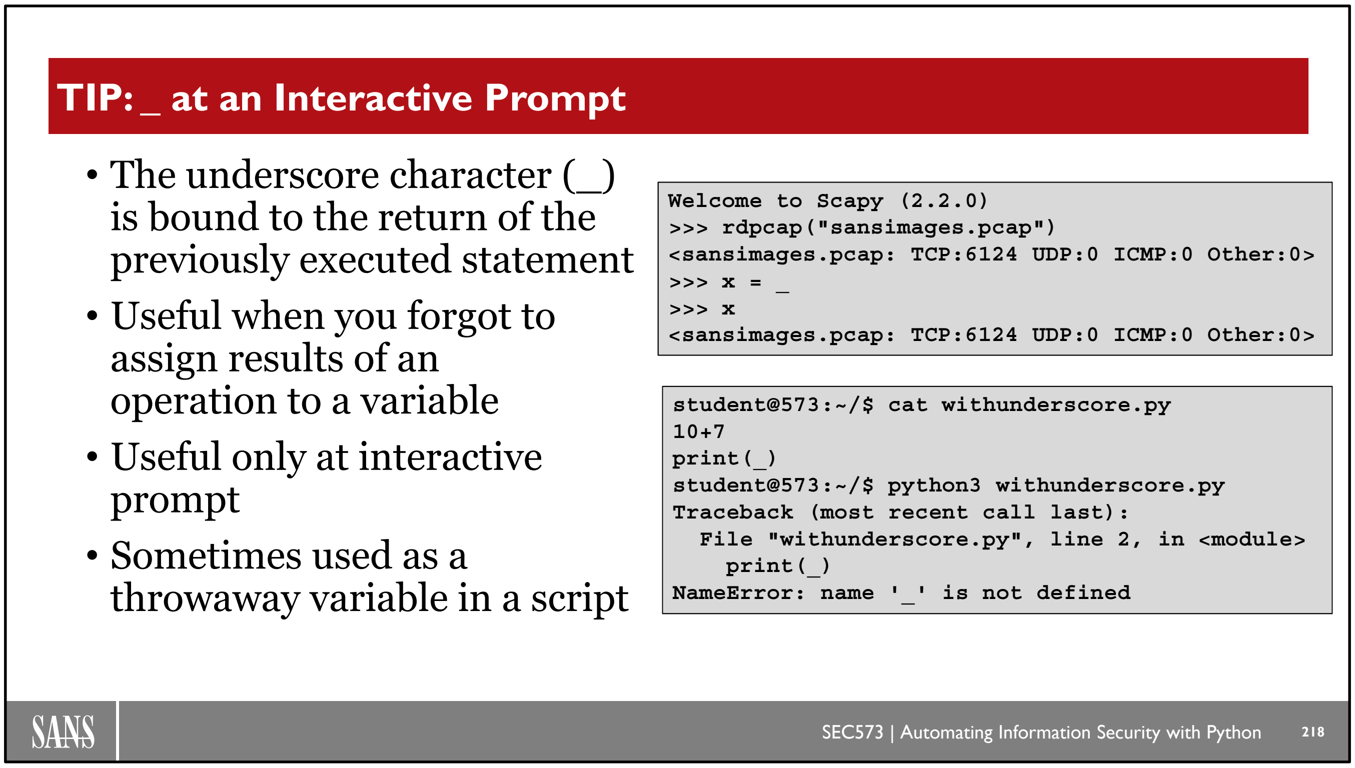
SANS SEC573 | Automating Information Security with Python 217

If you are checking to see whether the content of a variable is between two values, then you can do it with an AND and two tests like this:

```
>>> if ( x  > 1 ) and ( x < 3 ) : print("x between 1 & 3")
```

Or you can do it in one step like this:

```
>>> x = 2
>>> if 1 < x < 3: print("it is between")
...
it is between
>>> x = 5
>>> if 1 < x < 3: print("it is between")
...
>>> x = 0
>>> if 1 < x < 3: print("it is between")
...
>>>
```



TIP: _ at an Interactive Prompt

- The underscore character (_) is bound to the return of the previously executed statement
- Useful when you forgot to assign results of an operation to a variable
- Useful only at interactive prompt
- Sometimes used as a throwaway variable in a script

```
Welcome to Scapy (2.2.0)
>>> rdpcap("sansimages.pcap")
<sansimages.pcap: TCP:6124 UDP:0 ICMP:0 Other:0>
>>> x = _
>>> x
<sansimages.pcap: TCP:6124 UDP:0 ICMP:0 Other:0>
```

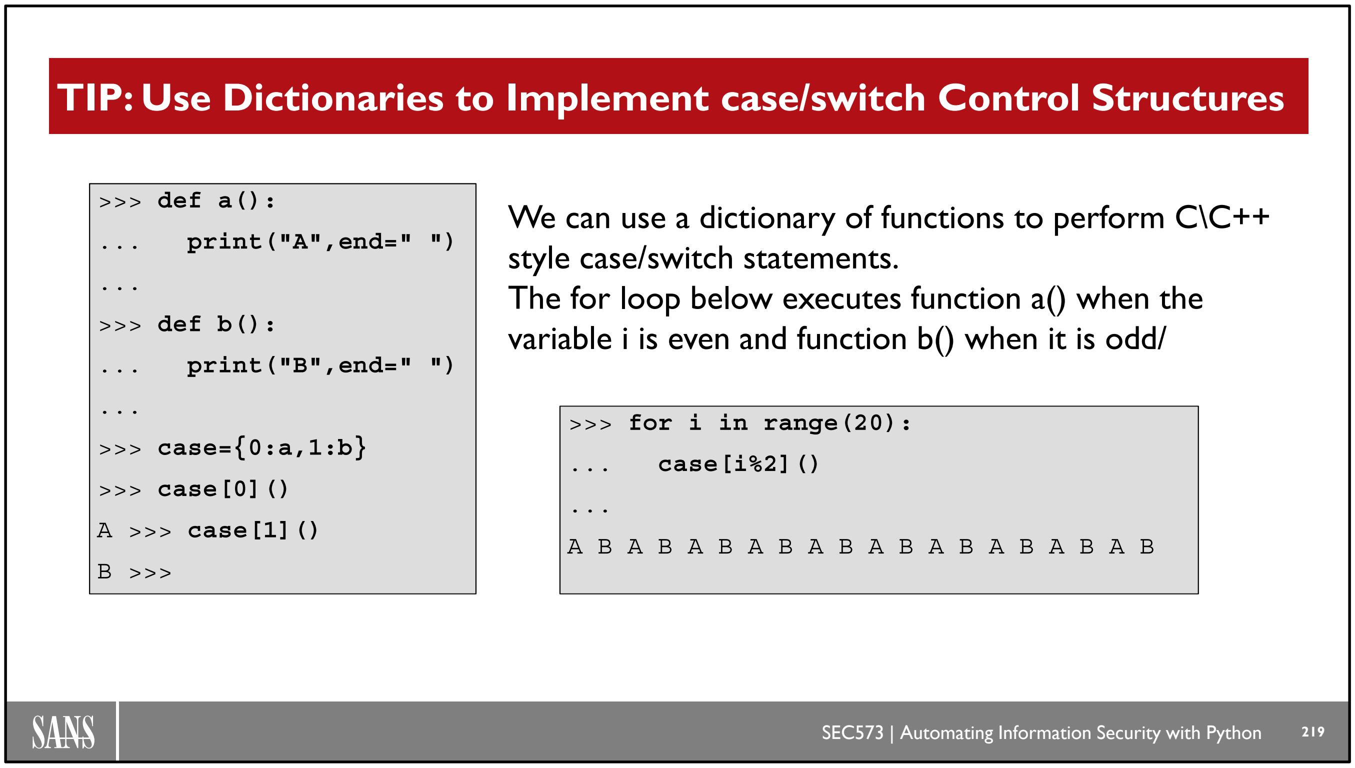
```
student@573:~/$ cat withunderscore.py
10+7
print(_)
student@573:~/$ python3 withunderscore.py
Traceback (most recent call last):
  File "withunderscore.py", line 2, in <module>
    print(_)
NameError: name '_' is not defined
```

SANS SEC573 | Automating Information Security with Python 218

While you are in an interactive Python shell, the _ character always holds the results of the last thing executed in the interactive shell. This character can be useful if you run some command but forget to capture the result into a variable. In this example, we used scapy's rdpcap to read the contents of a PCAP file into the system but forgot to assign it to a variable. Rather than reading that file again (a potentially slow operation), we can just assign the contents of the file to the variable x by running " x = _ ".

Note that when Python is executing a .py or .pyc script, the underscore is not specifically assigned values and is not as useful. You will sometimes see developers use the underscore as a throwaway variable in a script. For example, if you wanted to create a for loop that executed exactly 10 times but you didn't have a use for the iterator variable, you might use the code `"for _ in range(10):"` or if you were only interested in one part of a tuple that contained four parts, you might do something like this:

```
>>> _,_,keepthis,_ = ('ignored','ignored','kept','ignored')
>>> keepthis
'kept'
```



TIP: Use Dictionaries to Implement case/switch Control Structures

```
>>> def a():
...   print("A",end=" ")
...
>>> def b():
...   print("B",end=" ")
...
>>> case={0:a,1:b}
>>> case[0]()
A >>> case[1]()
B >>>
```

We can use a dictionary of functions to perform C\C++ style case/switch statements.
The for loop below executes function a() when the variable i is even and function b() when it is odd/

```
>>> for i in range(20):
...   case[i%2]()
...
A B A B A B A B A B A B A B A B A B A B
```

SANS SEC573 | Automating Information Security with Python 219

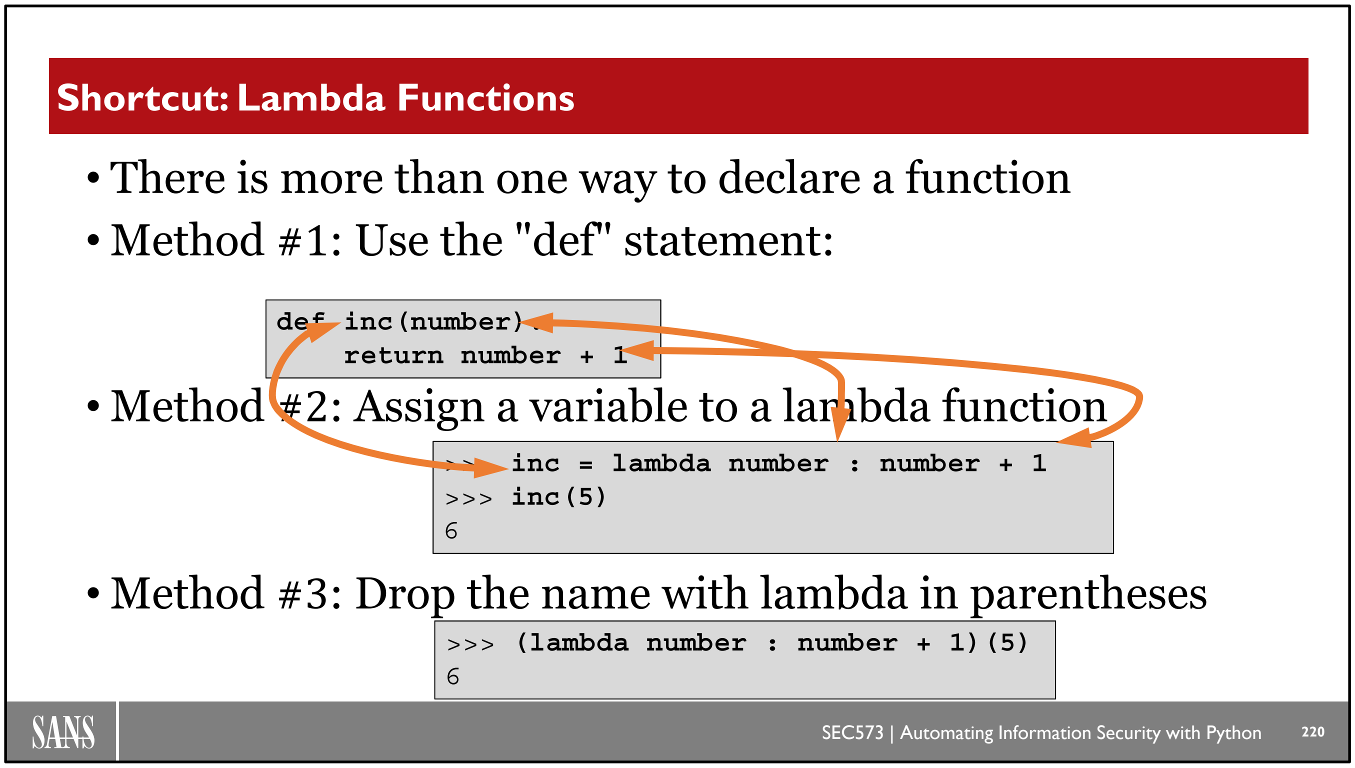
One nice use of dictionaries is to create a "case statement" in Python. Python doesn't have a native "case statement". A case statement is used in situations in which you have multiple code branches that are taken based on some key element. Consider the following if statement:

```
if a==1:
    callfunction1()
elif a==2:
    callfunction2()
elif a==3:
    callfunction3()
```

Other languages provide case statements that enable you to branch based on a given index value like this:

```
 switch ( a )
      {
         case 1:
            callfunction1();
         case 2:
            callfunction2();
}
```

In Python, you can use dictionaries to create a case statement. In the dictionary, set the index as the switched variable and the data element to the name of the function. Then you execute the function by calling Dictionary[switch case](), and the associated data element in the dictionary is executed.



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
```

SEC573 | Automating Information Security with Python 220

We have already discussed how to use the keyword def to define a function. There is another way to define functions. Python has what are 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:

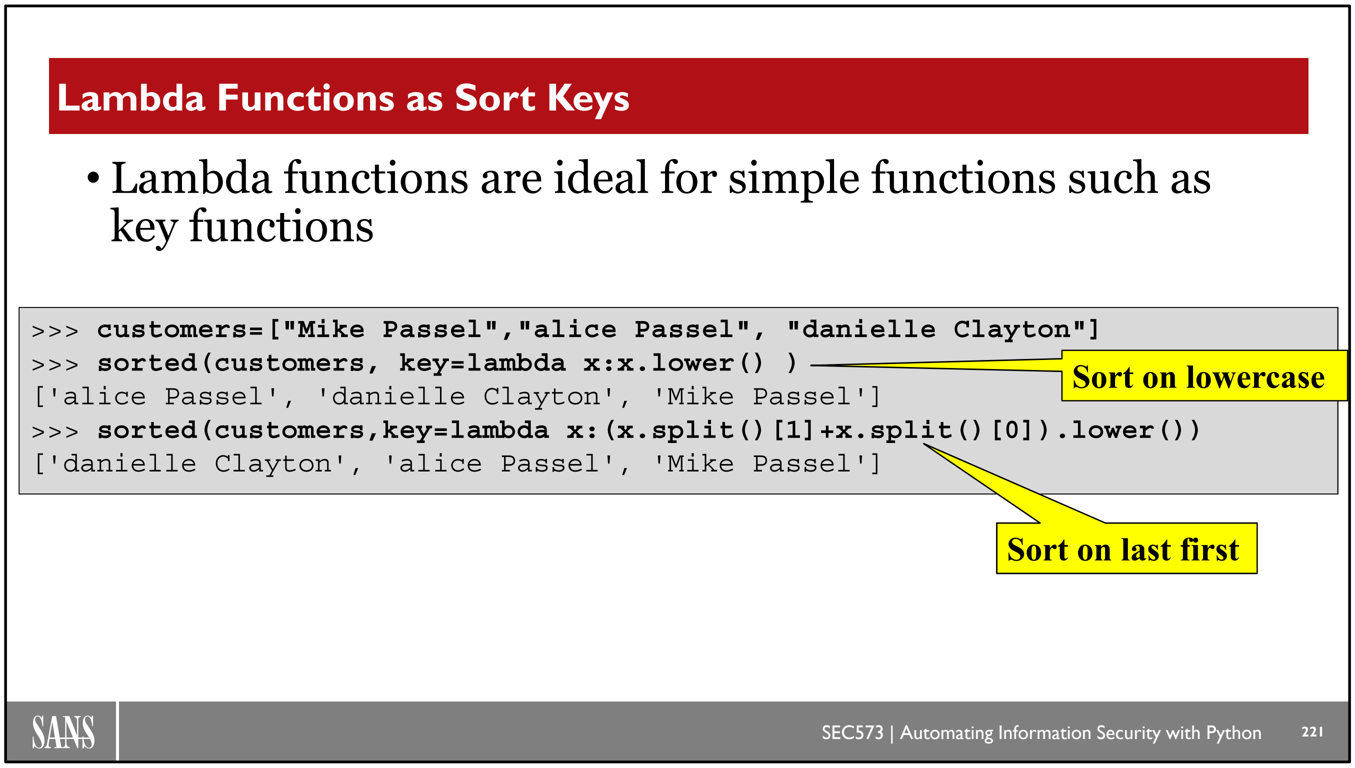
```
>>> def inc(number):
...     return number + 1
```

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
```



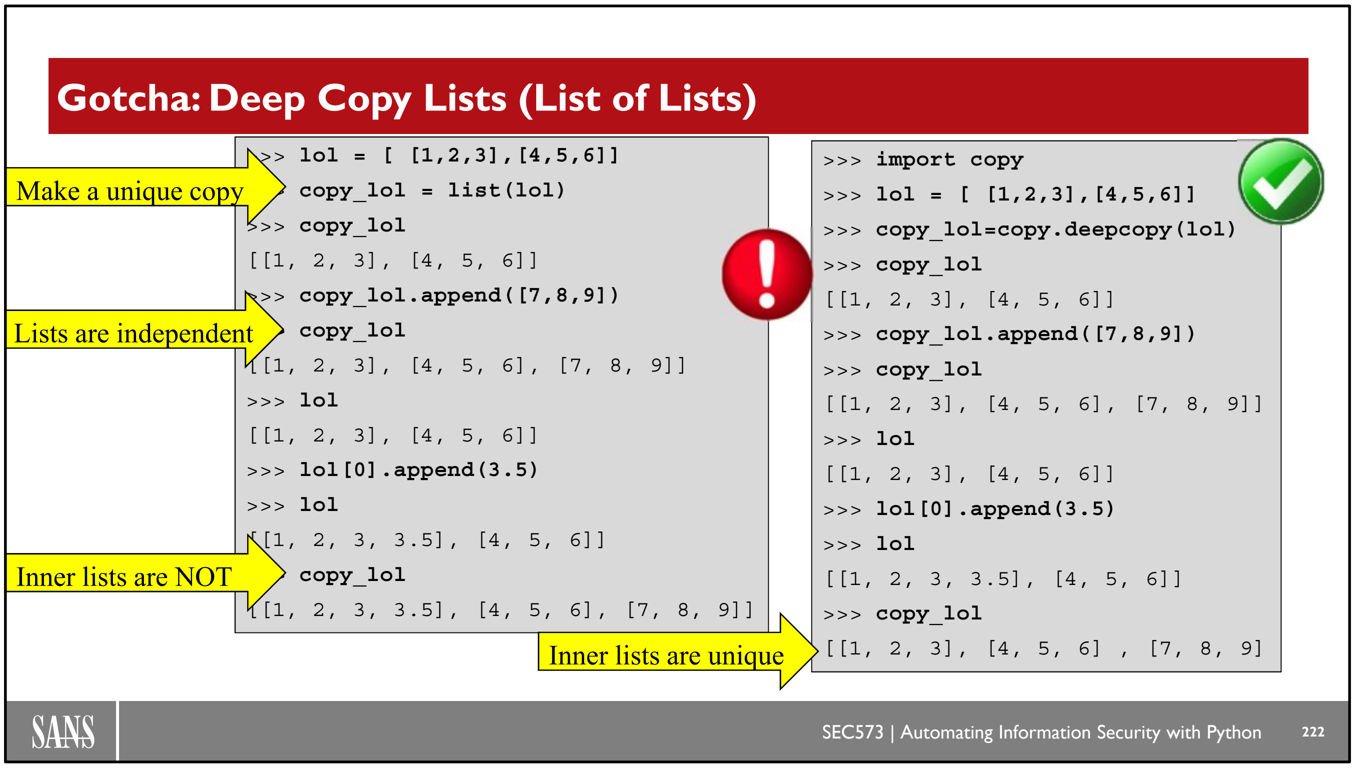
Lambda Functions as Sort Keys

- Lambda functions are ideal for simple functions such as key functions

```
>>> 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.



Gotcha: Deep Copy Lists (List of Lists)

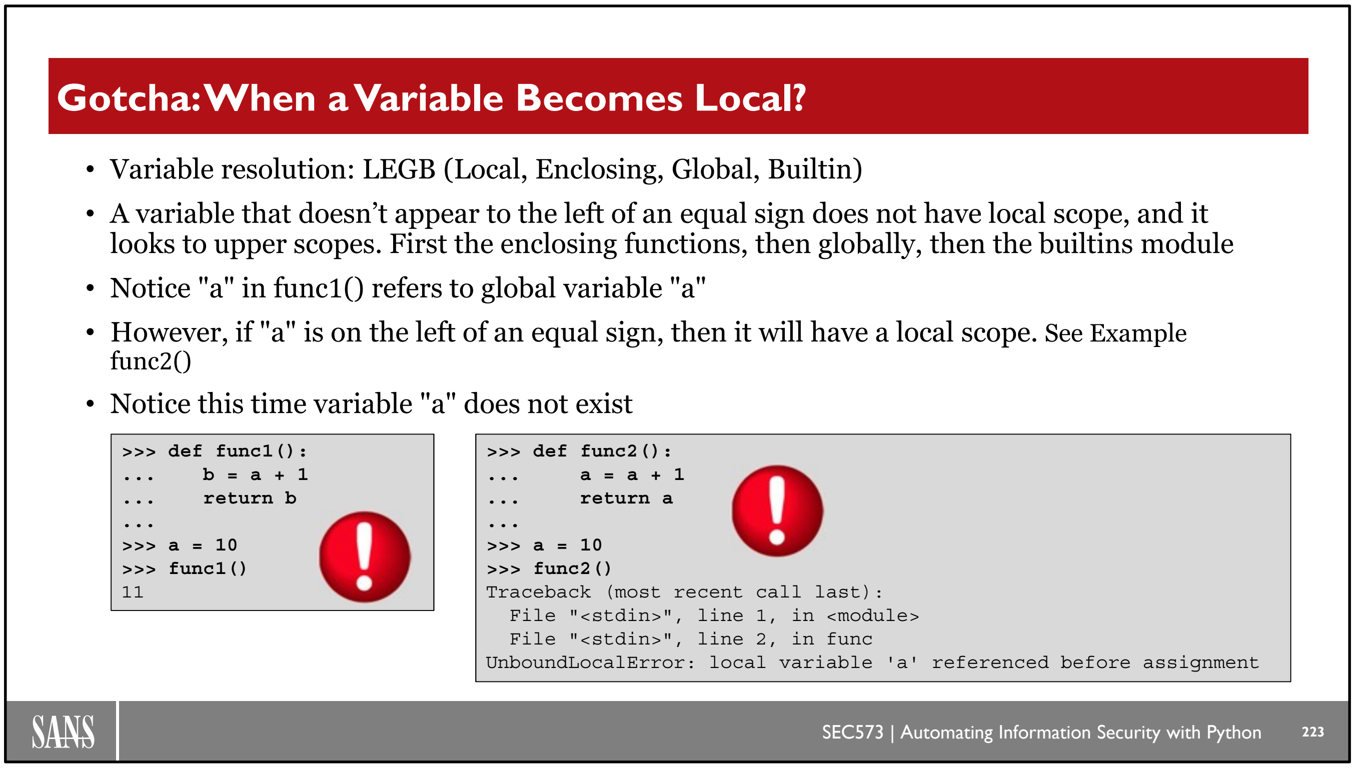
```
>>> lol = [ [1,2,3],[4,5,6]]
>>> copy_lol = list(lol)
>>> copy_lol
[[1, 2, 3], [4, 5, 6]]
>>> copy_lol.append([7,8,9])
>>> copy_lol
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> lol
[[1, 2, 3], [4, 5, 6]]
>>> lol[0].append(3.5)
>>> lol
[[1, 2, 3, 3.5], [4, 5, 6]]
>>> copy_lol
[[1, 2, 3, 3.5], [4, 5, 6], [7, 8, 9]]
```

```
>>> import copy
>>> lol = [ [1,2,3],[4,5,6]]
>>> copy_lol=copy.deepcopy(lol)
>>> copy_lol
[[1, 2, 3], [4, 5, 6]]
>>> copy_lol.append([7,8,9])
>>> copy_lol
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> lol
[[1, 2, 3], [4, 5, 6]]
>>> lol[0].append(3.5)
>>> lol
[[1, 2, 3, 3.5], [4, 5, 6]]
>>> copy_lol
[[1, 2, 3], [4, 5, 6] , [7, 8, 9]
```

SANS SEC573 | Automating Information Security with Python 222

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.



Gotcha: When a Variable Becomes Local?

- Variable resolution: LEGB (Local, Enclosing, Global, Builtin)
- A variable that doesn't appear to the left of an equal sign does not have local scope, and it looks to upper scopes. First the enclosing functions, then globally, then the builtins module
- Notice "a" in func1() refers to global variable "a"
- However, if "a" is on the left of an equal sign, then it will have a local scope. See Example func2()
- Notice this time variable "a" does not exist

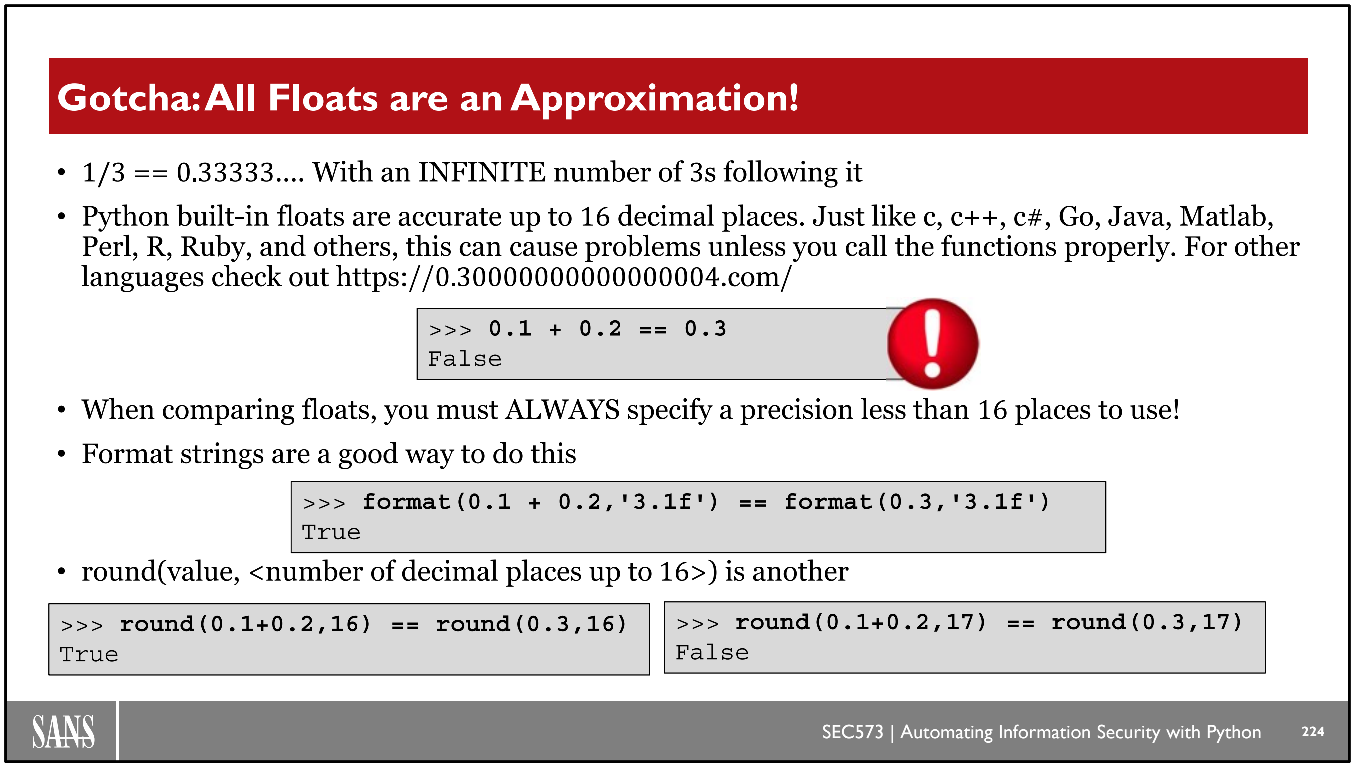
```
>>> def func1():
...     b = a + 1
...     return b
...
>>> a = 10
>>> func1()
11
```

```
>>> def func2():
...     a = a + 1
...     return a
...
>>> a = 10
>>> func2()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in func
UnboundLocalError: local variable 'a' referenced before assignment
```

SANS | SEC573 | Automating Information Security with Python 223

Remember, when Python is trying to resolve a variable name, it first looks inside the local scope. A variable is considered local if it is either declared as an input argument in the definition line or if it appears on the left-hand side of an equal sign in the function. If the variable doesn't exist in the local scope, Python then looks in any "enclosing" functions. An enclosed function is a function that is declared inside of another function. Then it looks in the global() namespace for any existing variables with that name. Finally, it looks inside the __builtin__ module to see if the variable is declared there.

In a function, a variable is considered to be local if it appears on the left side of an equal sign or as an input argument in the definition. In func1() above, variable b is assigned the contents of global variable a plus 1. This works perfectly fine. In func2(), because variable a is on the left side of an equal sign, it is considered local. So the assignment of a = a + 1 fails because local variable a was never assigned an initial value before it tries to execute "a + 1".



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
```

SANS | SEC573 | Automating Information Security with Python 224

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:

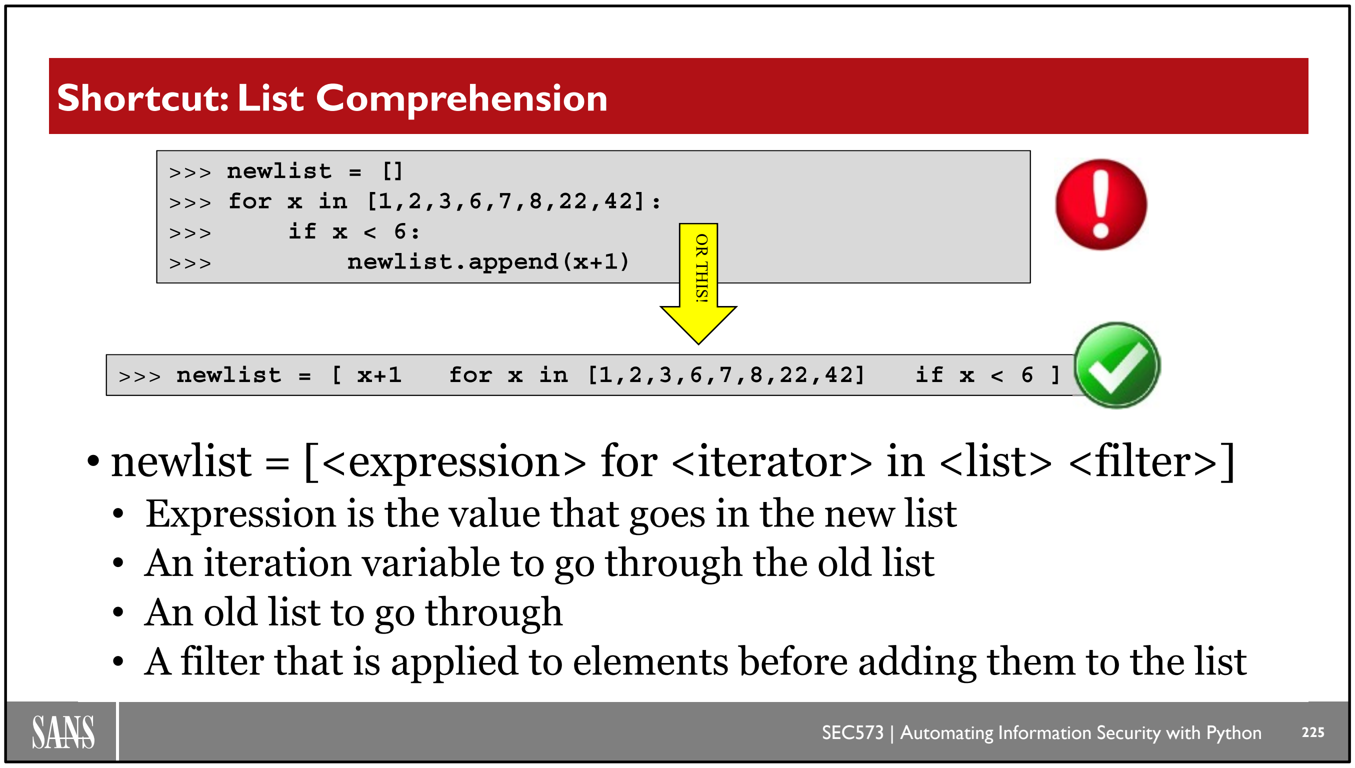
```
>>> 0.1 + 0.2 == 0.3000000000000000444089209850062616169452667236328125
True
```

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.



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

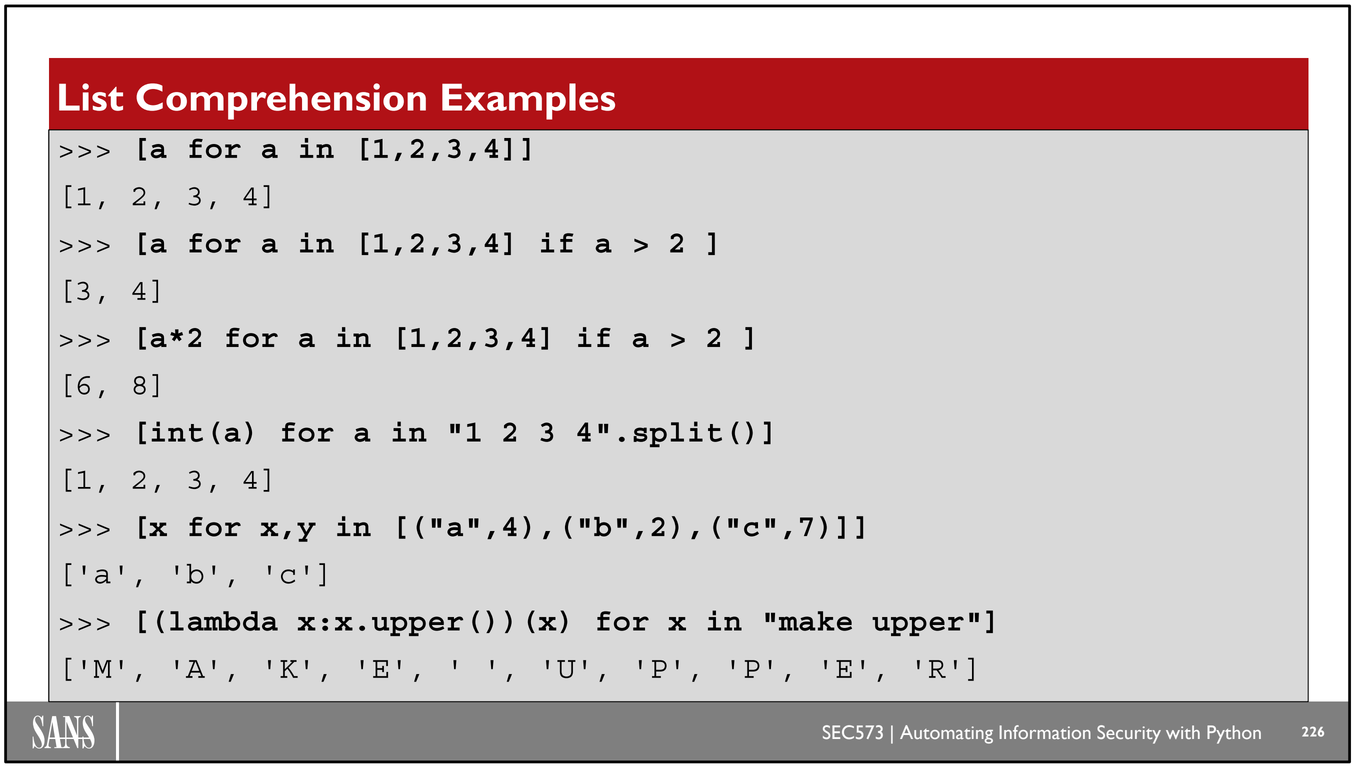
SANS SEC573 | Automating Information Security with Python 225

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.



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:

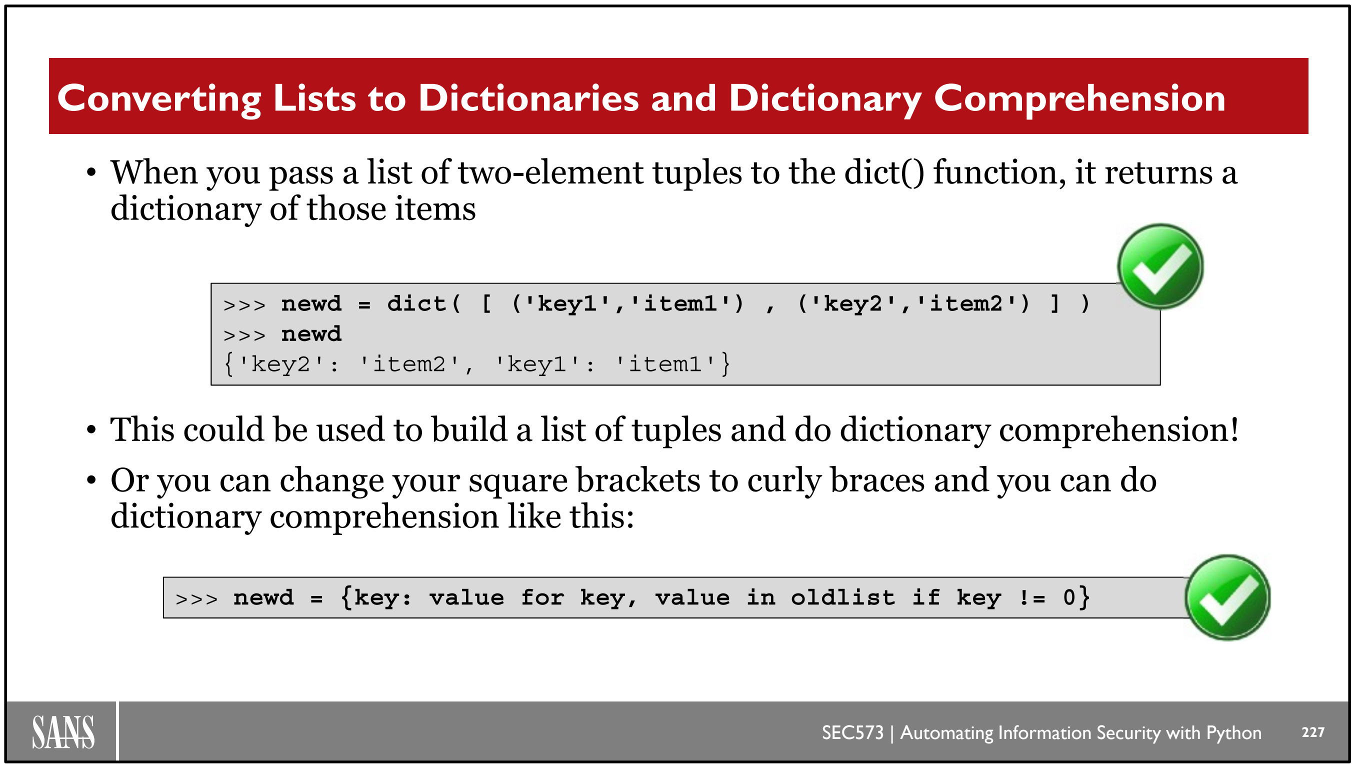
```
>>> [a for a in alist]
[1, 2, 3, 4, 5]
```

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.



Converting Lists to Dictionaries and Dictionary Comprehension

- When you pass a list of two-element tuples to the dict() function, it returns a dictionary of those items

```
>>> newd = dict( [ ('key1','item1') , ('key2','item2') ] )
>>> newd
{'key2': 'item2', 'key1': 'item1'}
```

- This could be used to build a list of tuples and do dictionary comprehension!
- Or you can change your square brackets to curly braces and you can do dictionary comprehension like this:

```
>>> newd = {key: value for key, value in oldlist if key != 0}
```

SANS SEC573 | Automating Information Security with Python 227

A nice feature of dictionaries is that you can initialize them with a list of two-item tuples. When you call the dict() function and pass it a list of tuples that contain a key and a value, it will return a dictionary make-up of those values. Consider the following:

```
>>> newdict = dict( [ ('thekey', 1 ) ] )
>>> newdict
{'thekey': 1}
```

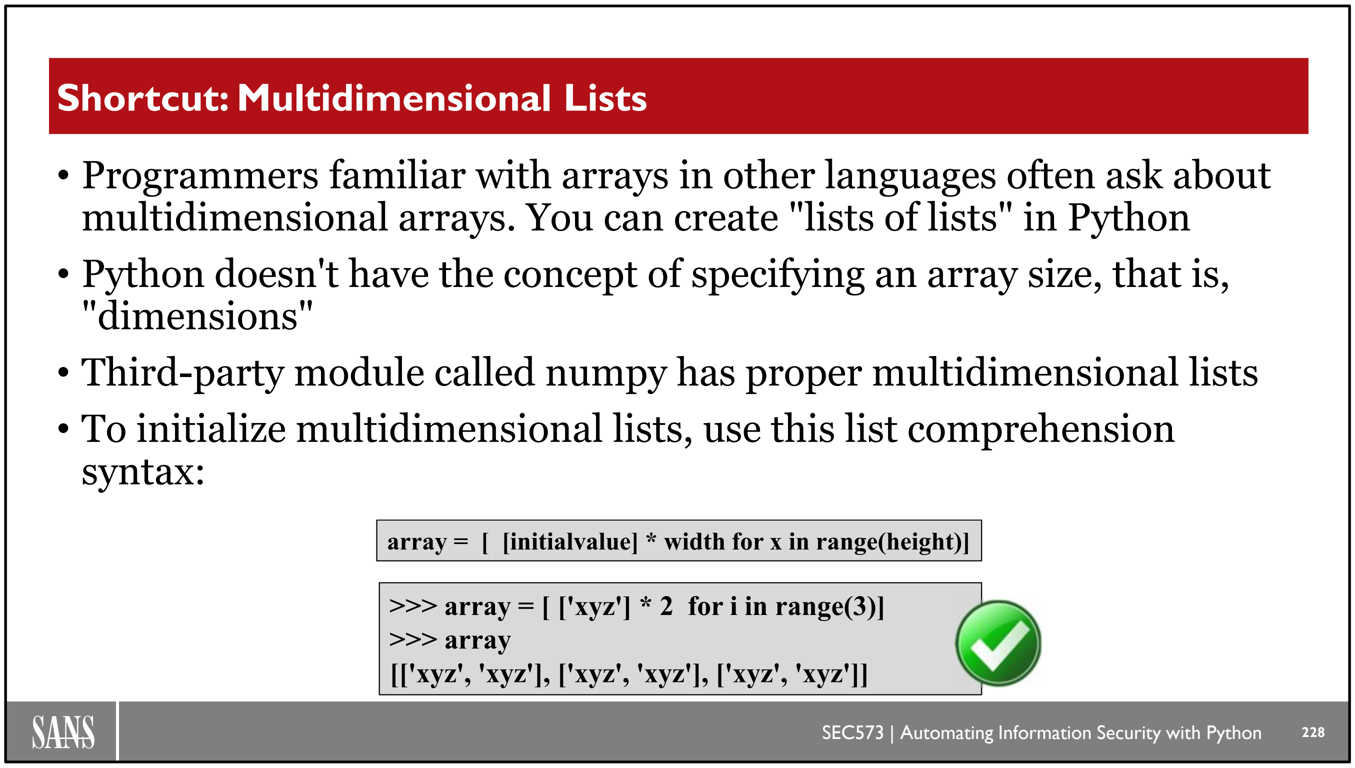
This example creates a dictionary with one key in it. The key is called 'thekey'. The dictionary item has a value of 1. Here, there is only one item (that is, one tuple) in the list, but you can have as many tuples in the list as you want. Each of the tuples will be converted into a record in the dictionary, with the first item in the tuple being the key and the second item in the tuple being the value.

You can also use list comprehension to create a list of tuples and pass that to the dict() function. Doing so effectively becomes dictionary comprehension:

```
>>> newd = dict([( key,value) for key,value in oldlist if key != 0 ])
```

Alternatively, in all modern versions of Python, you can also do dictionary comprehension by just changing the square brackets to curly braces.

```
>>> newd = {1: 'b', 2: 'c'}dict[(key,value) for key,value in oldlist if
key != 0 ])
```



Shortcut: Multidimensional Lists

- Programmers familiar with arrays in other languages often ask about multidimensional arrays. You can create "lists of lists" in Python
- Python doesn't have the concept of specifying an array size, that is, "dimensions"
- Third-party module called numpy has proper multidimensional lists
- To initialize multidimensional lists, use this list comprehension syntax:

```
array =  [  [initialvalue] * width for x in range(height)]
```

```
>>> array = [ ['xyz'] * 2  for i in range(3)]
>>> array
[['xyz', 'xyz'], ['xyz', 'xyz'], ['xyz', 'xyz']]
```

SANS

SEC573 | Automating Information Security with Python 228

A common question is, “How do I create a multidimensional list?” In Python, you would create a list of lists. In other languages, you might declare the height and width of your array with some type of declaration statement. Python has no such concept. There is a module called numpy that does provide great support for these types of data structures. To initialize a multidimensional array in Python, you would use the following syntax:

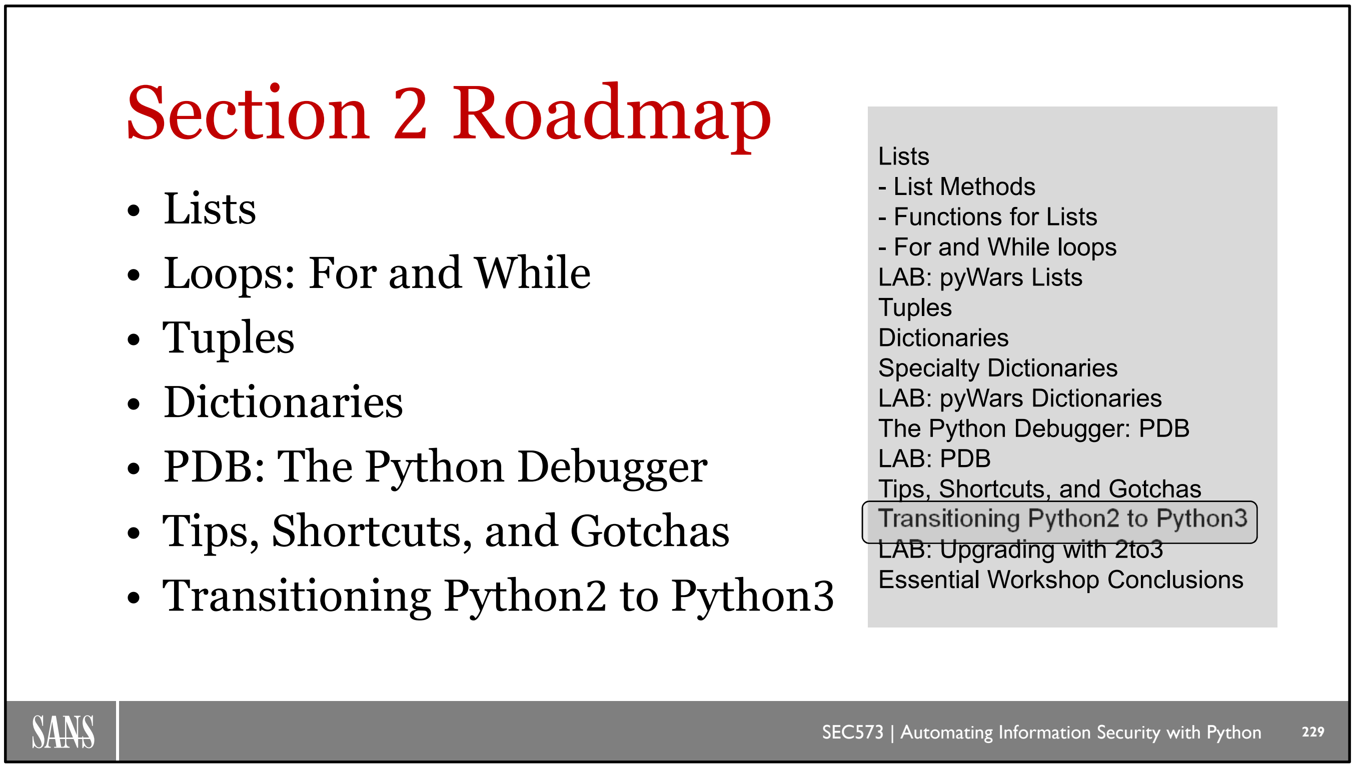
```
>>> array =  [  [initialvalue] * width for x in range(height)]
```

To create an array that is 4 × 5 with all elements set to an initial value of zero, you would use the following:

```
>>> array = [ [0]* 4 for i in range(5) ]
>>> array
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
```

Then you can treat it like a multidimensional array:

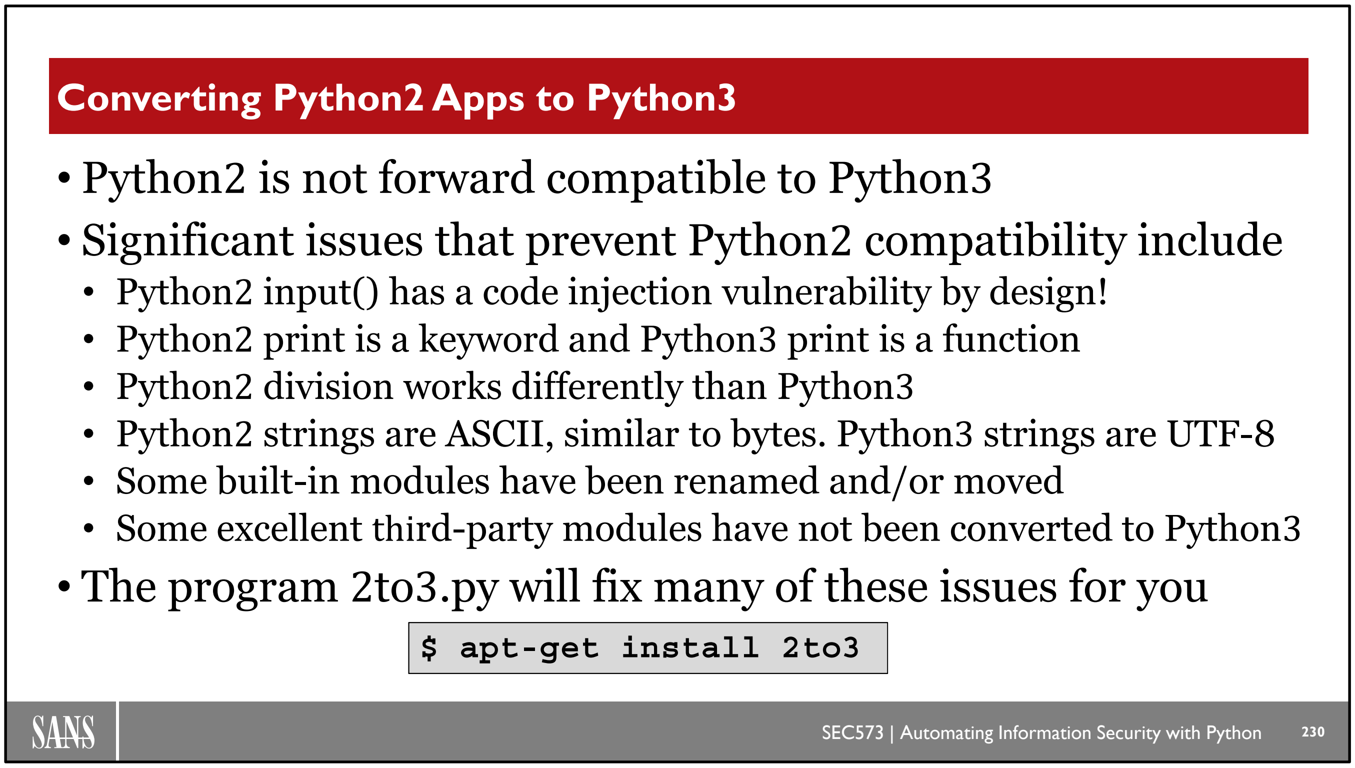
```
>>> array[4][3]=9
>>> array[1][1]=4
>>> array[0][3]=1
>>> array
[[0, 0, 0, 1], [0, 4, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 9]]
```



Section 2 Roadmap

- Lists
- Loops: For and While
- Tuples
- Dictionaries
- PDB: The Python Debugger
- Tips, Shortcuts, and Gotchas
- Transitioning Python2 to Python3

This is a Roadmap slide.



Converting Python2 Apps to Python3

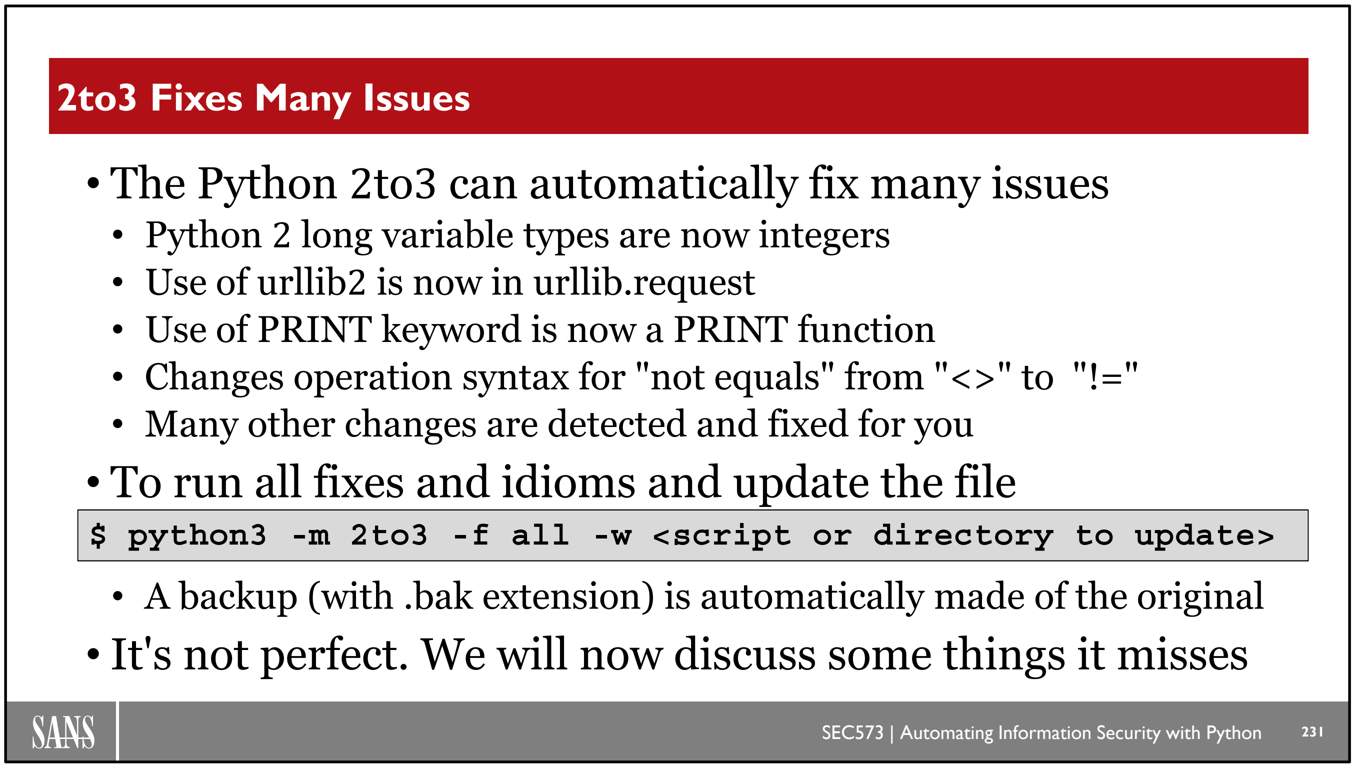
- Python2 is not forward compatible to Python3
- Significant issues that prevent Python2 compatibility include
  - Python2 input() has a code injection vulnerability by design!
  - Python2 print is a keyword and Python3 print is a function
  - Python2 division works differently than Python3
  - Python2 strings are ASCII, similar to bytes. Python3 strings are UTF-8
  - Some built-in modules have been renamed and/or moved
  - Some excellent third-party modules have not been converted to Python3
- The program 2to3.py will fix many of these issues for you

```
$ apt-get install 2to3
```

SANS SEC573 | Automating Information Security with Python 230

Python2 code is not strictly forward compatible with Python3. However, it is possible to write Python2 code that will run perfectly in a Python3 interpreter. For the last five years, this curse has been teaching people to write code that works in both Python2 and Python3. But it requires discipline on the part of the developer and most developers don't make this effort. Given a Python2 program, it is very likely that it will require some work to make it compatible with Python3.

There are many issues that prevent Python2 programs from working properly in Python3. Some are very easy to spot, such as modules being renamed and the print command becoming a function. Others, such as the use of bytes and strings, are not as easy to spot. To assist in the upgrade process, the program 2to3 attempts to automate rewriting pieces of your code for you. It is a Python module, so it can be installed with PIP or with package managers such as apt shown in the slide above.



2to3 Fixes Many Issues

- The Python 2to3 can automatically fix many issues
  - Python 2 long variable types are now integers
  - Use of urllib2 is now in urllib.request
  - Use of PRINT keyword is now a PRINT function
  - Changes operation syntax for "not equals" from "<>" to "!="
  - Many other changes are detected and fixed for you
- To run all fixes and idioms and update the file

```
$ python3 -m 2to3 -f all -w <script or directory to update>
```

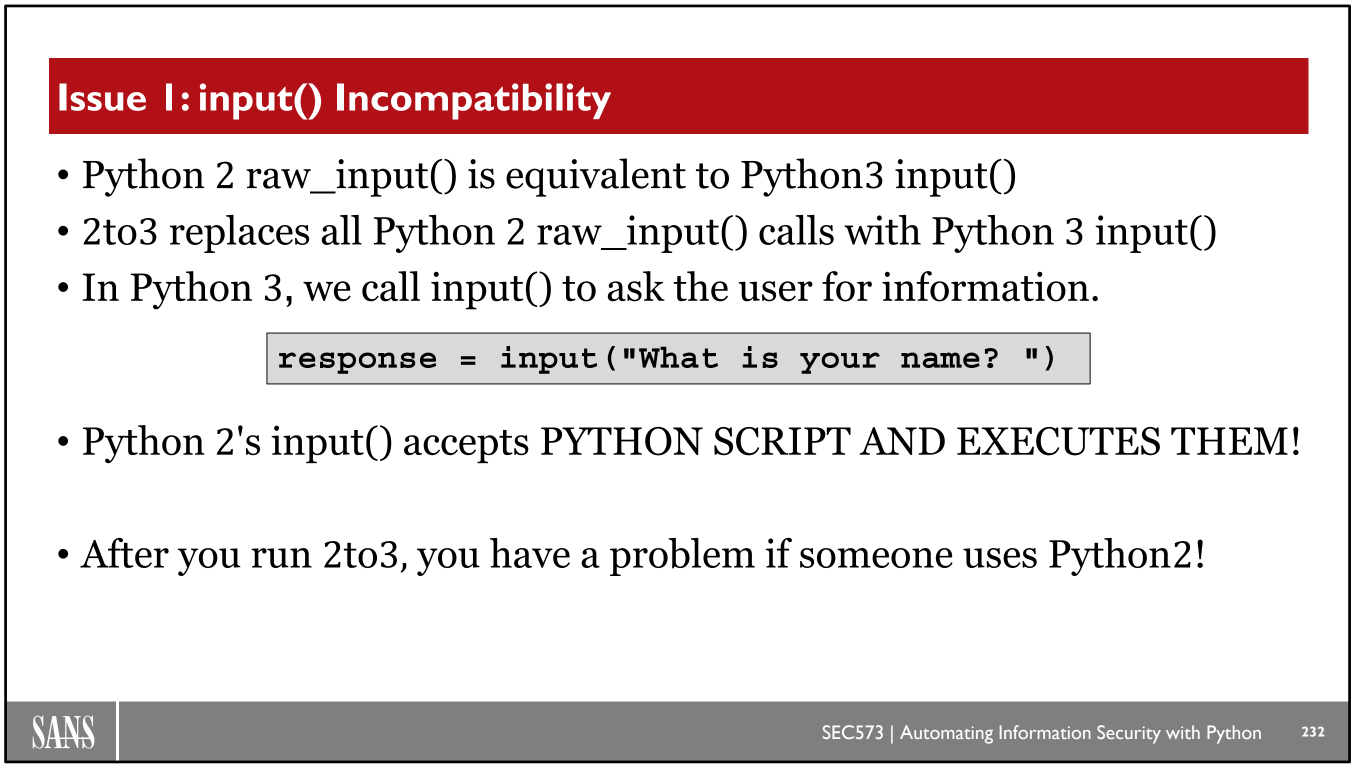
  - A backup (with .bak extension) is automatically made of the original
- It's not perfect. We will now discuss some things it misses

SANS | SEC573 | Automating Information Security with Python 231

2to3 has a long list of fixes that it will automatically apply to your code. 2to3 will change all reverence to the "long" variable type to integers. It changes references to urllib2 to urllib.request because the modules were renamed. It changes all use of the print keyword to calls to the print function. It changes references to <>, which was commonly used in Python 2 for "not equals" comparisons, to the preferred Python 3 syntax of "!=". These and many other changes are automatically made for you to your code.

To use 2to3, you run the module as shown above. The "-f all" option tells it to apply all of the fixes it has in its library. Without this option, it will only fix a limited subset of issues. The "-w" option tells it to update the original source code, rewriting the lines that had errors in it. Without the "-w" option, it will only display the problems that it would fix. It is safe to apply your changes, as 2to3 will automatically make a backup of the original program before it makes the changes. The file will have the same name as the original program, with a .bak extension added to the end of it. The last argument is either a single script or a directory containing multiple scripts that you want upgraded.

2to3 was good, but not perfect. There are several issues it didn't fix, and if you are not careful, a security issue will be created by the update. Let's look at a few of them now.



Issue 1: input() Incompatibility

- Python 2 raw_input() is equivalent to Python3 input()
- 2to3 replaces all Python 2 raw_input() calls with Python 3 input()
- In Python 3, we call input() to ask the user for information.

```
response = input("What is your name? ")
```

- Python 2's input() accepts PYTHON SCRIPT AND EXECUTES THEM!
- After you run 2to3, you have a problem if someone uses Python2!

SANS SEC573 | Automating Information Security with Python 232

One significant issue is the difference between input and raw_input. If you want to ask the person using your script a question and collect an answer from them, you call the Python3 function named input(). Python 2 also has an input() command that is used for collecting **and running** python commands. This is horribly dangerous and Python has removed this functionality from version 3. Python 2 also had a function named raw_input that does exactly what Python 3's input() function does today.

To safely collect input in Python 2, you used to call raw_input(). To safely collect input in Python 3, you call input(). Both of these functions return a string. If you want to collect something other than a string, then you can use another function such as int() or bytes() to convert the string into the variable type that you want.

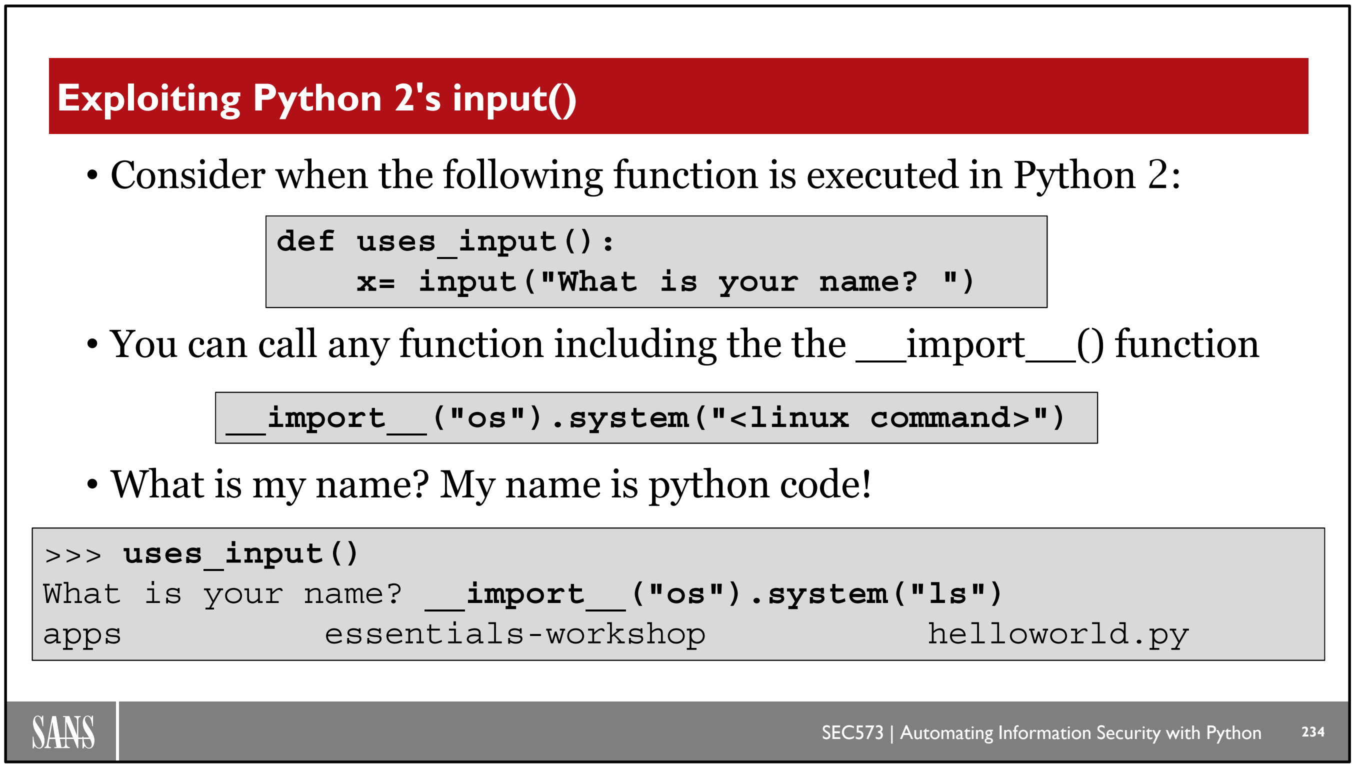
Fixing input() after Using 2to3

- 2to3 will change all the raw_input() calls to input()
- Your Python3 program works PERFECTLY!
- When someone runs the program with Python 2, they are going to be vulnerable to code injection attacks
- Add these lines to the top of the program to protect the program if it's run in Python 2

```
import sys

if sys.version_info.major == 2:
    input = raw_input
```

In Python 3, if you use the input() function, it will return a string instead of executing a script. In other words, in Python 3, input() does what raw_input() does in Python 2. So 2to3 will change all of the calls to raw_input() into calls to input(). As a result, if we run the program with Python 2, our code is exploitable.

To solve this problem, we can add a small block of code to our programs that will reassign the input() function to call raw_input(). Then we can safely make calls to input(), and our programs will work in both Python 2 and Python 3. However, since there is no raw_input() function in Python 3, assigning input() to call raw_input() will fail. So we have to check the sys.version_info.major variable to make sure we are in Python2 before doing the reassignment.



Exploiting Python 2's input()

- Consider when the following function is executed in Python 2:

```
def uses_input():
    x= input("What is your name? ")
```

- You can call any function including the the __import__() function

```
__import__("os").system("<linux command>")
```

- What is my name? My name is python code!

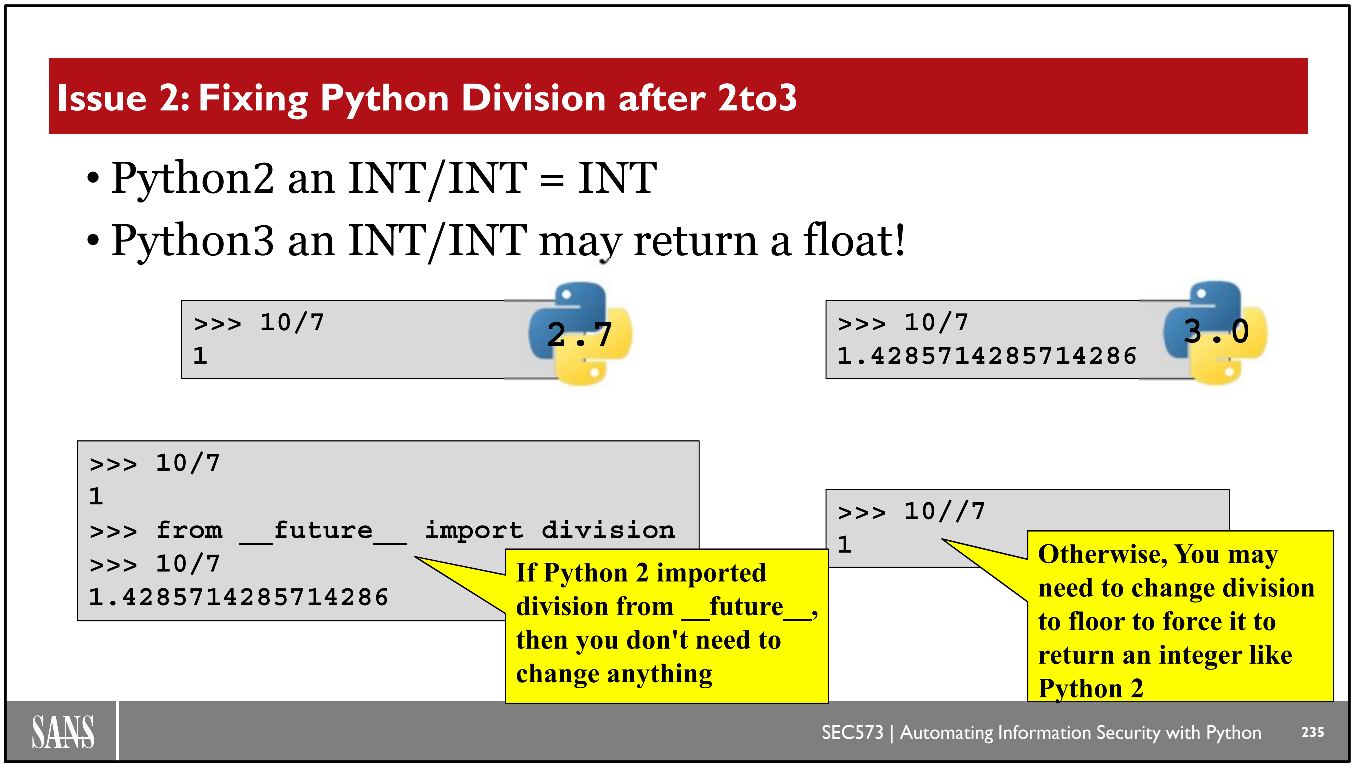
```
>>> uses_input()
What is your name? __import__("os").system("ls")
apps          essentials-workshop          helloworld.py
```

SEC573 | Automating Information Security with Python 234

After you have run 2to3, if someone runs your code through a Python2 interpreter then the code will be vulnerable to code injection attacks. Because Python 2 input processes the input, users can put in Python expressions as input, and they will be evaluated before they are assigned to the variable. In many situations, this isn't a horrible problem. However, if your Python script runs with elevated privilege, then it is a privilege escalation attack. If your Python program is accessible through a web server or over the network, then it is a remote code execute attack. It is best to eliminate the vulnerability because it is very trivial to exploit. Here is an example of exploiting the vulnerability. The following lines are run in Python 2.

```
>>> import os
>>> a=str(input("What is your name? "))
What is your name? os.system("id")
uid=501(userx) gid=20(staff),groups=20(staff),98(_lpadmin)
```

Ouch! It is executing a command of the user's choosing! You can provide any command you want to inside of the call to system(), and it will run the command for you!



Issue 2: Fixing Python Division after 2to3

- Python2 an INT/INT = INT
- Python3 an INT/INT may return a float!

```
>>> 10/7
1
```

```
>>> 10/7
1.4285714285714286
```

```
>>> 10/7
1
>>> from __future__ import division
>>> 10/7
1.4285714285714286
```

If Python 2 imported division from __future__, then you don't need to change anything

```
>>> 10//7
1
```

Otherwise, You may need to change division to floor to force it to return an integer like Python 2

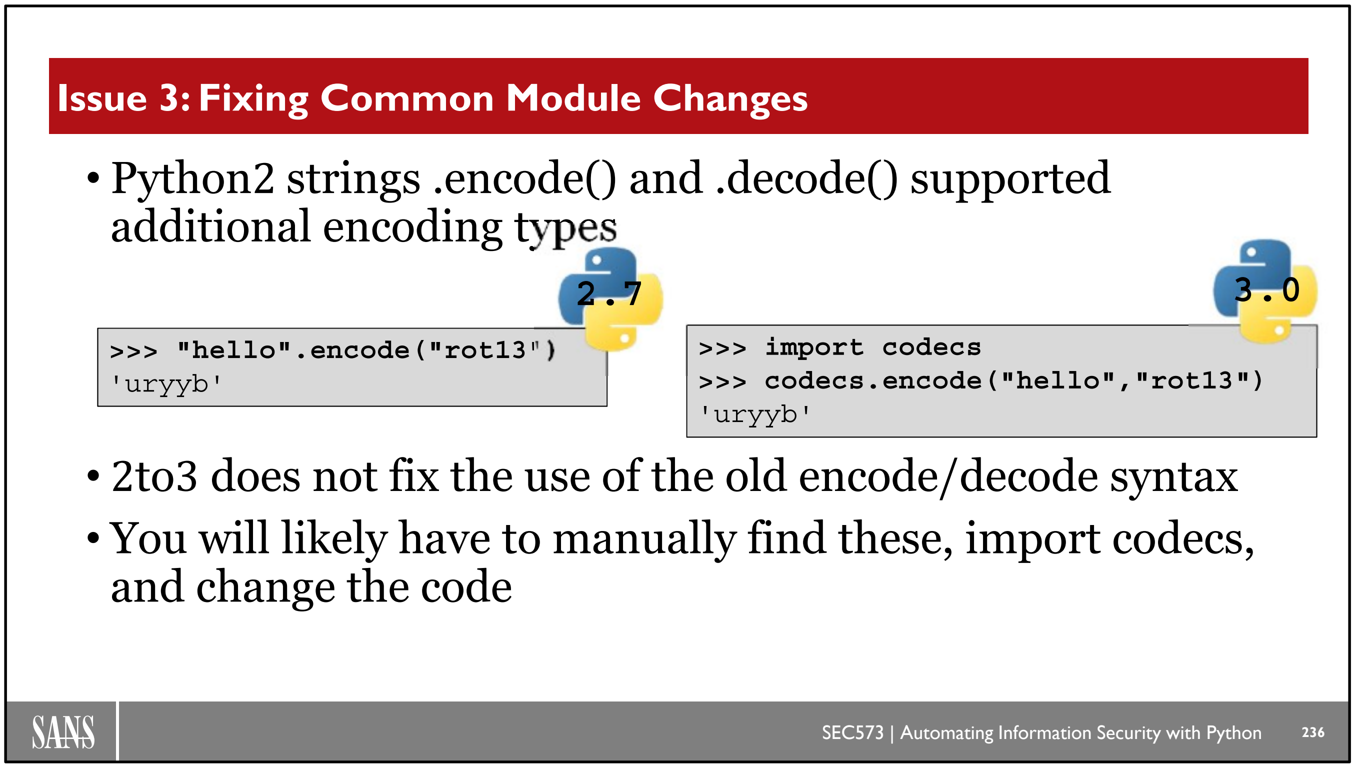
SANS SEC573 | Automating Information Security with Python 235

2to3 also misses some nuances associated with how division changed between Python 2 and Python 3. It is possible that the developers wrote excellent Python2 code that anticipated these errors. If the Python 2 program had the line "from __future__ import division" at the top of it, then division acted the same in Python 2 as it does in Python 3. If you see that line at the top of the Python 2 code, then there is likely no fix required to upgrade the code.

In the absence of that line. Python 2 and Python 3 behave very differently when dividing integers. In Python 2, an integer divided by an integer would always return an integer. For example, 10/7 would return 1 and not 1.42857 like Python3 does. In Python 2, you had to turn either the numerator or the denominator into a float if you wanted a float as a result. 2to3 will not catch errors where the developer was expecting an integer result and is now getting a float. For example, consider this Python 2 program that finds the first half of a word.

```
>>> word = "find first half"
>>> word[:len(word)/2]
'find fi'
```

In Python 2, this would usually return an integer, but in Python 3 the middle of the string might be character 8.5, which would cause an error. You can fix this by using the floor operator "//" for your division. It always returns an integer. It is the Python 3 equivalent of "/" in Python 2 when you use two integers.



Issue 3: Fixing Common Module Changes

- Python2 strings .encode() and .decode() supported additional encoding types

2.7

```
>>> "hello".encode("rot13")
'uryyb'
```

3.0

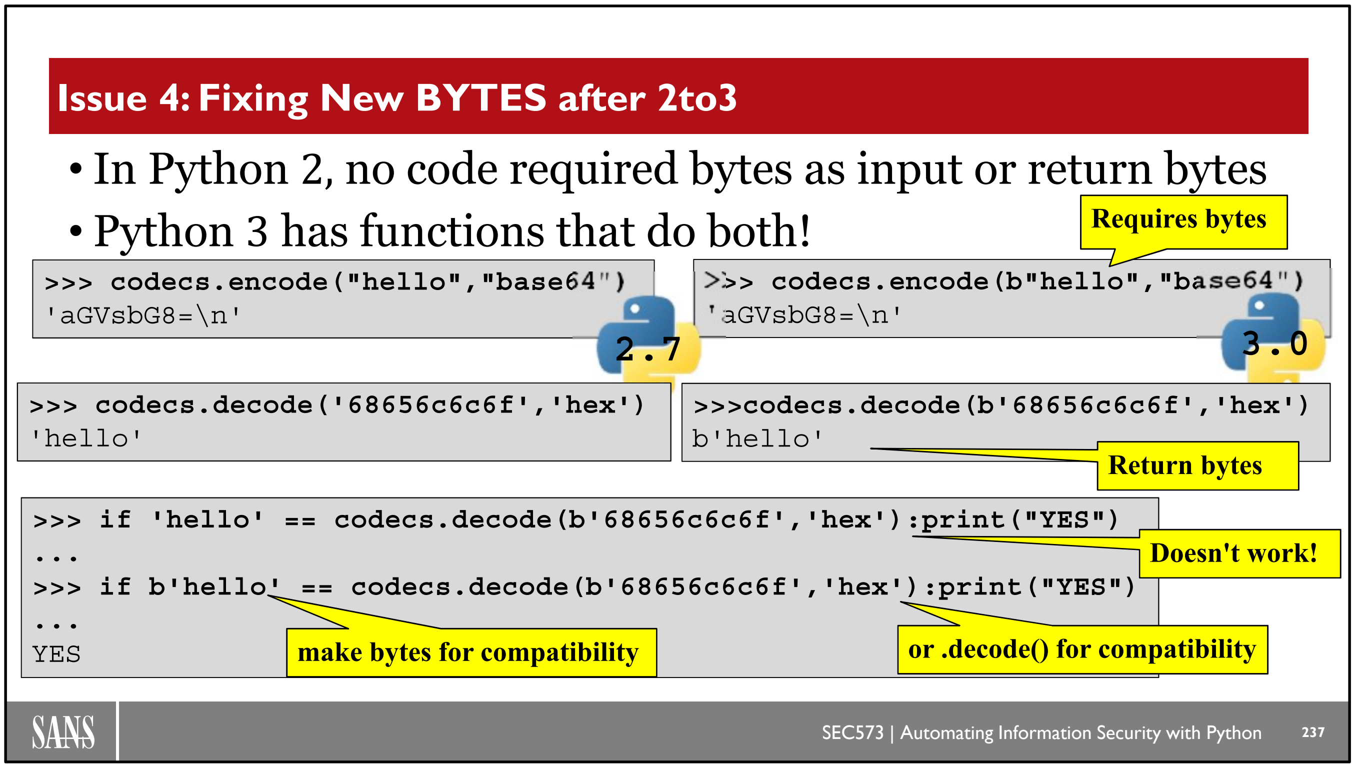
```
>>> import codecs
>>> codecs.encode("hello","rot13")
'uryyb'
```

- 2to3 does not fix the use of the old encode/decode syntax
- You will likely have to manually find these, import codecs, and change the code

SANS SEC573 | Automating Information Security with Python 236

2to3 does not fix the use of text-encoding schemes in your program. In Python2, it was common practice to put an .encode() or .decode() at the end of a string and use that to apply different encoding standards such as "zip", "rot13", "base64", "hex", and others. In Python3, these encoding standards have been moved into the codecs module.

You will still find .encode() at the end of strings to handle conversions into bytes, but it can no longer be used for special text encodings such as those mentioned earlier. If those encoders or decoders are used, you will have to manually import codecs and change the code to use that module, as shown above.



Issue 4: Fixing New BYTES after 2to3

- In Python 2, no code required bytes as input or return bytes
- Python 3 has functions that do both!

```
>>> codecs.encode("hello","base64")
'aGVsbG8=\n'
```

```
>>> codecs.encode(b"hello","base64")
'aGVsbG8=\n'
```

```
>>> codecs.decode('68656c6c6f','hex')
'hello'
```

```
>>>codecs.decode(b'68656c6c6f','hex')
b'hello'
```

```
>>> if 'hello' == codecs.decode(b'68656c6c6f','hex'):print("YES")
...
>>> if b'hello' == codecs.decode(b'68656c6c6f','hex'):print("YES")
...
YES
```

SANS SEC573 | Automating Information Security with Python 237

In Python3, you now have functions that expect bytes as input and return bytes. These same functions took in strings and returned strings in Python2. These can be very subtle tricky bugs that are not easily found unless you know to look for them. This problem often shows up in if statements that, as a result of conversion, are now comparing bytes to strings that will never match. For example, consider this if statement that looks for the string 'hello' in the response that comes back from decode. In Python2, this worked fine because codecs.decode returned a string. In Python3, the bytes returned by codecs.decode will never match the string 'hello'. It will only match the bytes b'hello'.

```
>>> if 'hello' == codecs.decode(b'68656c6c6f','hex'):print("YES")
...
```

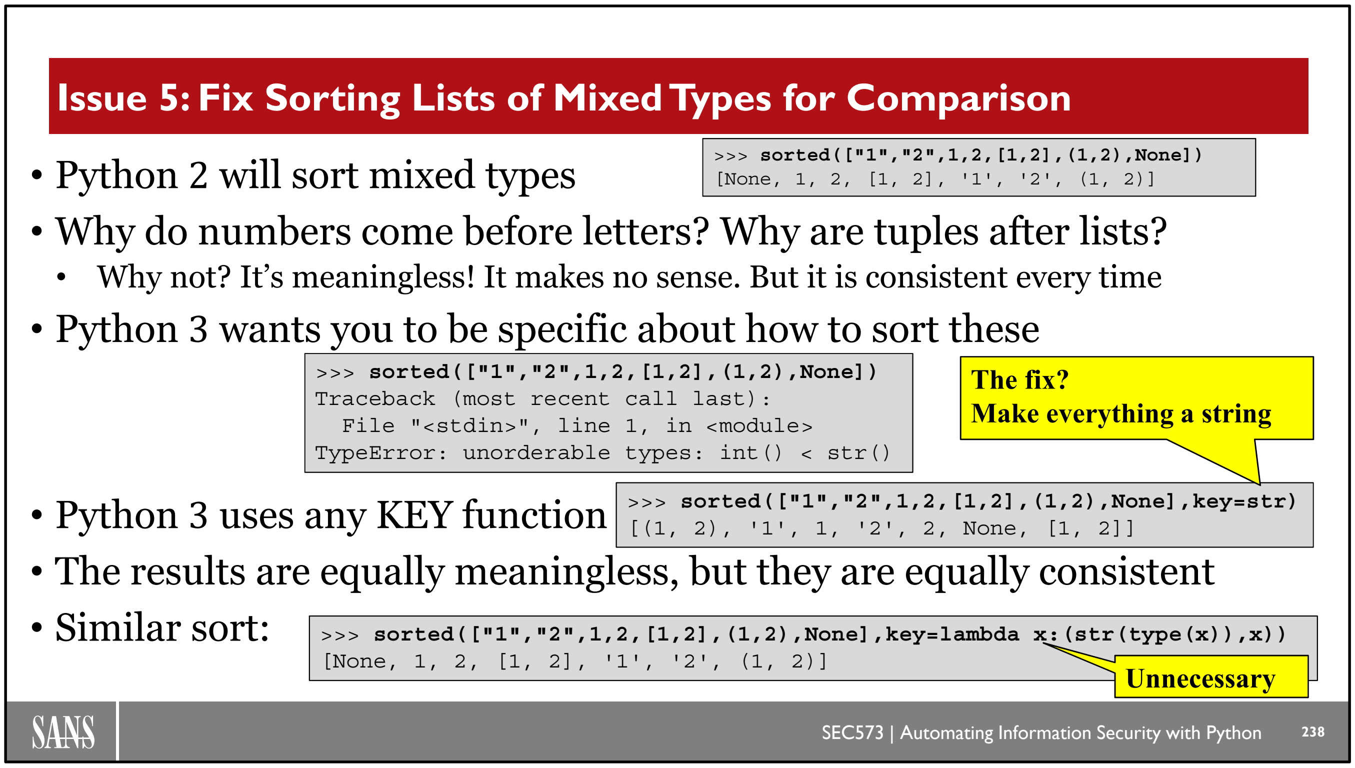
This didn't print 'YES' because the string 'hello' isn't in the bytes that were returned. To fix this, we can look for the bytes b'hello' in what is returned.

```
>>> if b'hello' == codecs.decode(b'68656c6c6f','hex'):print("YES")
...
YES
```

Alternatively, we can .decode() the results of codecs.decode and turn it into a string.

```
>>> if 'hello' == codecs.decode(b'68656c6c6f','hex').decode():print("YES")
...
YES
```

You can see in both of these solutions, it now finds the word 'hello' and prints 'YES' to the screen.



Issue 5: Fix Sorting Lists of Mixed Types for Comparison

- Python 2 will sort mixed types

```
>>> sorted(["1","2",1,2,[1,2],(1,2),None])
[None, 1, 2, [1, 2], '1', '2', (1, 2)]
```

- Why do numbers come before letters? Why are tuples after lists?
  - Why not? It's meaningless! It makes no sense. But it is consistent every time
- Python 3 wants you to be specific about how to sort these

```
>>> sorted(["1","2",1,2,[1,2],(1,2),None])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: int() < str()
```

The fix?
Make everything a string

- Python 3 uses any KEY function

```
>>> sorted(["1","2",1,2,[1,2],(1,2),None],key=str)
[(1, 2), '1', 1, '2', 2, None, [1, 2]]
```

- The results are equally meaningless, but they are equally consistent
- Similar sort:

```
>>> sorted(["1","2",1,2,[1,2],(1,2),None],key=lambda x:(str(type(x)),x))
[None, 1, 2, [1, 2], '1', '2', (1, 2)]
```

Unnecessary

SANS

SEC573 | Automating Information Security with Python 238

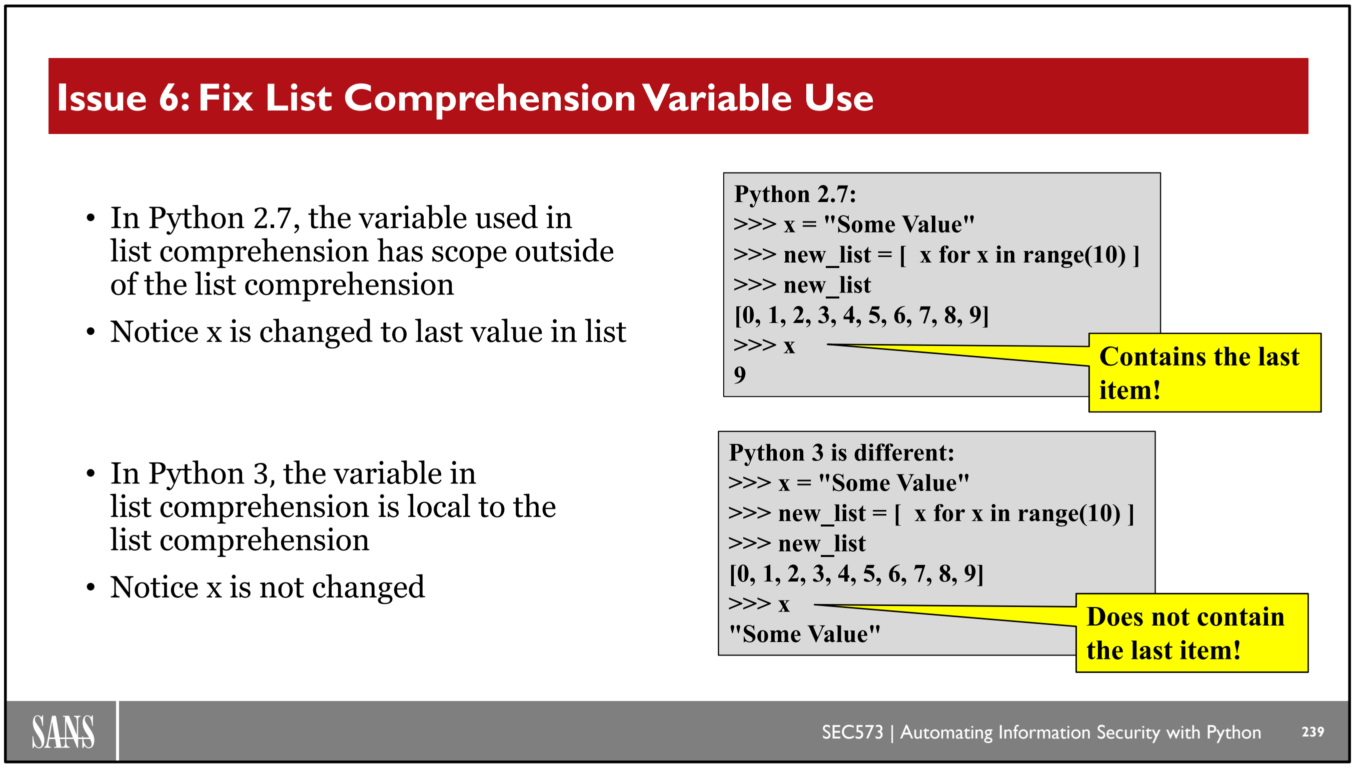
In Python 2, you could compare any two objects and Python would give you an answer. Oftentimes this answer was meaningless. For example:

```
student@573:~$ python2 -c "print(1 < '4', 6 < '4' )"
True True
```

In Python3, this same command generates an error.

```
student@573:~$ python3 -c "print(1 < '4', 6 < '4' )"
Traceback (most recent call last):
  File "<string>", line 1, in <module>
TypeError: '<' not supported between instances of 'int' and 'str'
```

Why is the integer one less than the character 4? No reason; Python 2 just says it is. As a result, in Python 2, you could sort a list of things with mixed types and it would not generate an error. In Python 3, an error is generated. But there are times when you want to sort lists of mixed types. For example, if I want to compare two lists of mixed items to see if they have the same items in them, I could sort them both and compare them. In that case, you could use any key function. For example, you can see that when we use "str" as our key function, the errors go away. The order of the items can be just as meaningless as it was before if we are only sorting for comparison reasons. But if you wanted to create a key function that is close to but not completely compatible with Python 2, I provide you one here. I don't use this myself. Instead, I just use a key of str and that makes my code forward compatible with both Python 2 and 3.



Issue 6: Fix List Comprehension Variable Use

- In Python 2.7, the variable used in list comprehension has scope outside of the list comprehension
- Notice x is changed to last value in list

Python 2.7:

```
>>> x = "Some Value"
>>> new_list = [  x for x in range(10) ]
>>> new_list
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x
9
```

Contains the last item!

- In Python 3, the variable in list comprehension is local to the list comprehension
- Notice x is not changed

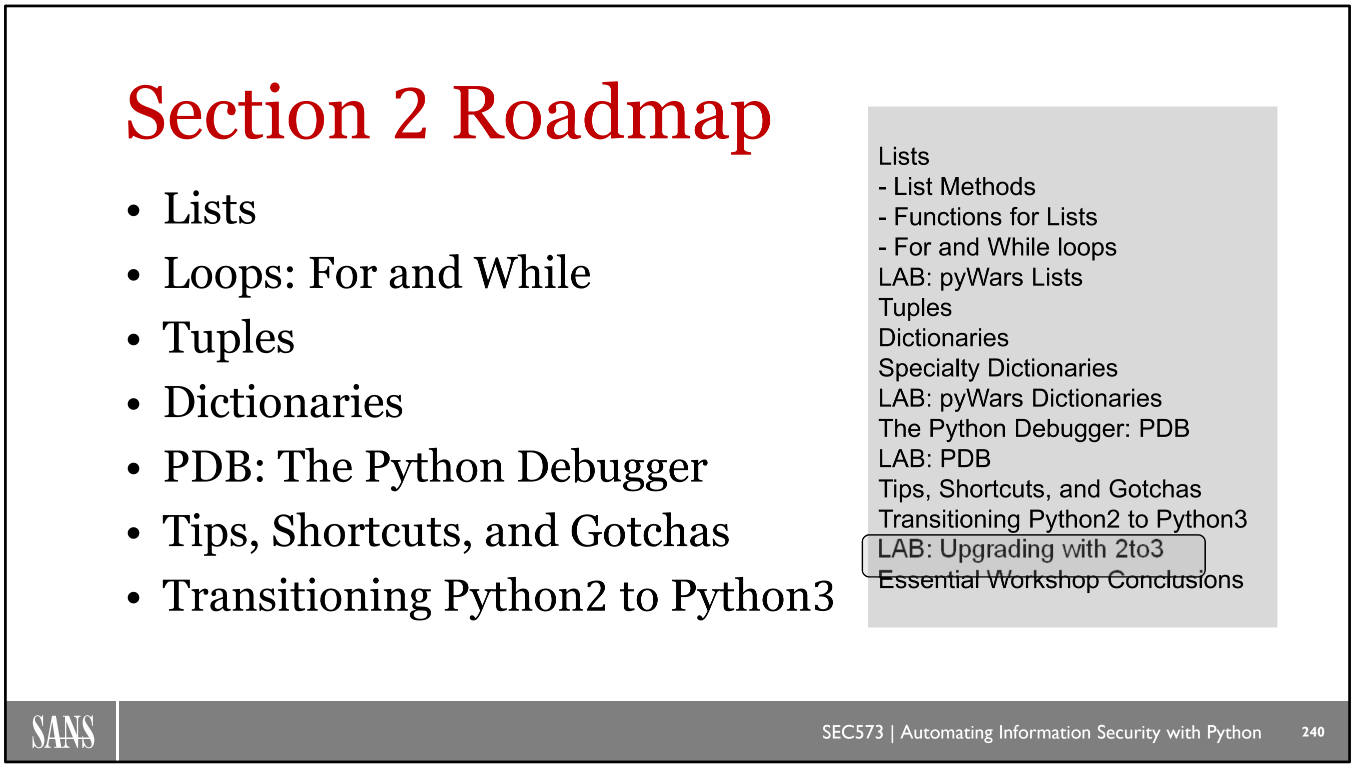
Python 3 is different:

```
>>> x = "Some Value"
>>> new_list = [  x for x in range(10) ]
>>> new_list
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x
"Some Value"
```

Does not contain the last item!

SANS SEC573 | Automating Information Security with Python 239

In the C Python interpreter, the way that a variable is used inside a list comprehension changes between version 2 and version 3. In Python 2.7, variables used inside a list comprehension do not have their own scope and will overwrite variables in their scope. In Python 3, they do have their own scope and will not overwrite existing values.

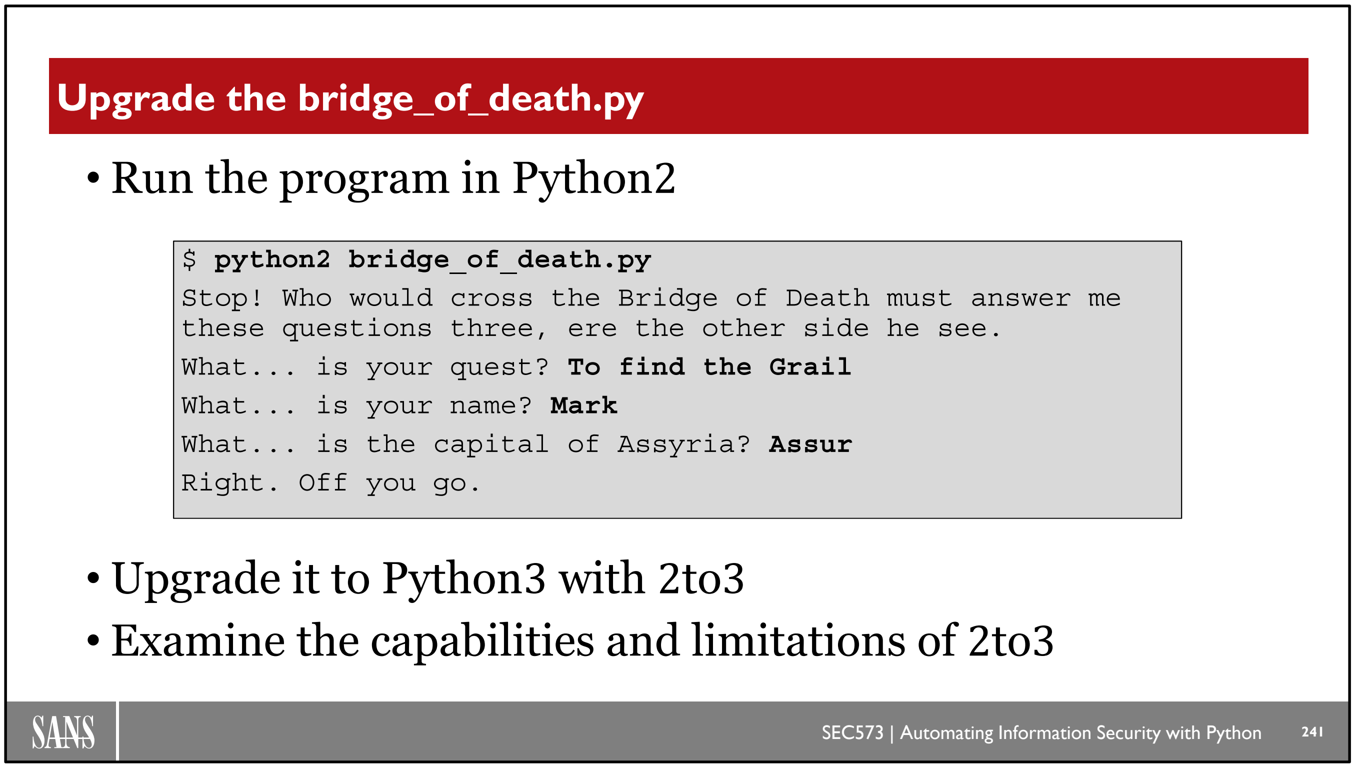


Section 2 Roadmap

- Lists
- Loops: For and While
- Tuples
- Dictionaries
- PDB: The Python Debugger
- Tips, Shortcuts, and Gotchas
- Transitioning Python2 to Python3

SANS

This is a Roadmap slide.



Upgrade the bridge_of_death.py

- Run the program in Python2

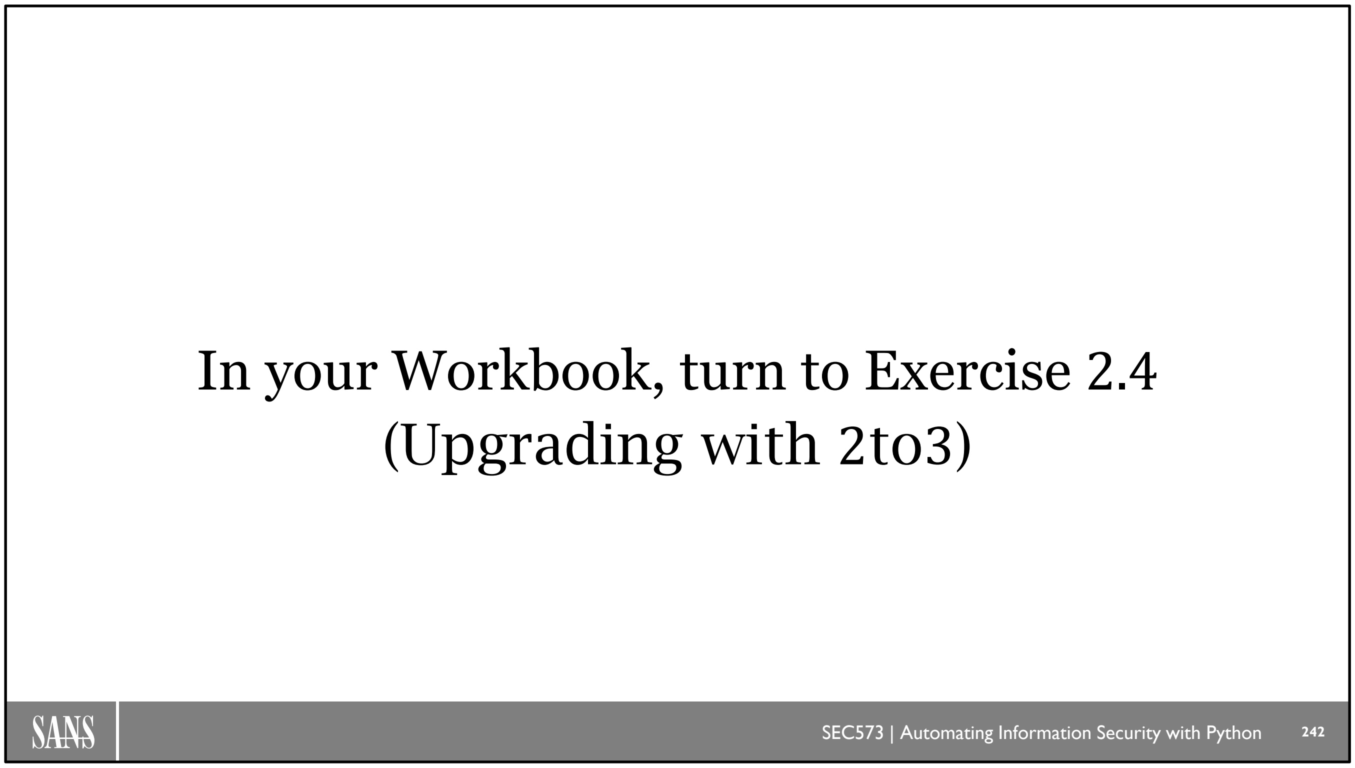
```
$ python2 bridge_of_death.py
Stop! Who would cross the Bridge of Death must answer me
these questions three, ere the other side he see.
What... is your quest? To find the Grail
What... is your name? Mark
What... is the capital of Assyria? Assur
Right. Off you go.
```

- Upgrade it to Python3 with 2to3
- Examine the capabilities and limitations of 2to3

SANS | SEC573 | Automating Information Security with Python 241

First, I want you to run the program "bridge_of_death.py" with Python2. Then you will use 2to3 to upgrade the program to Python3. This program contains several issues that 2to3 will find and automatically fix, but it also contains several others that it does not. You will have to find and fix the code yourself. Your goal is to get the program to work completely in Python3.

Do not be deceived by the difficulty of this small program. I have intentionally picked a program that has a lot of subtle errors. I would only expect to see this number of issues in a much larger program. It is not uncommon for 2to3 to fix much larger programs than this and not require any additional changes, but you should test your code thoroughly after the upgrade.

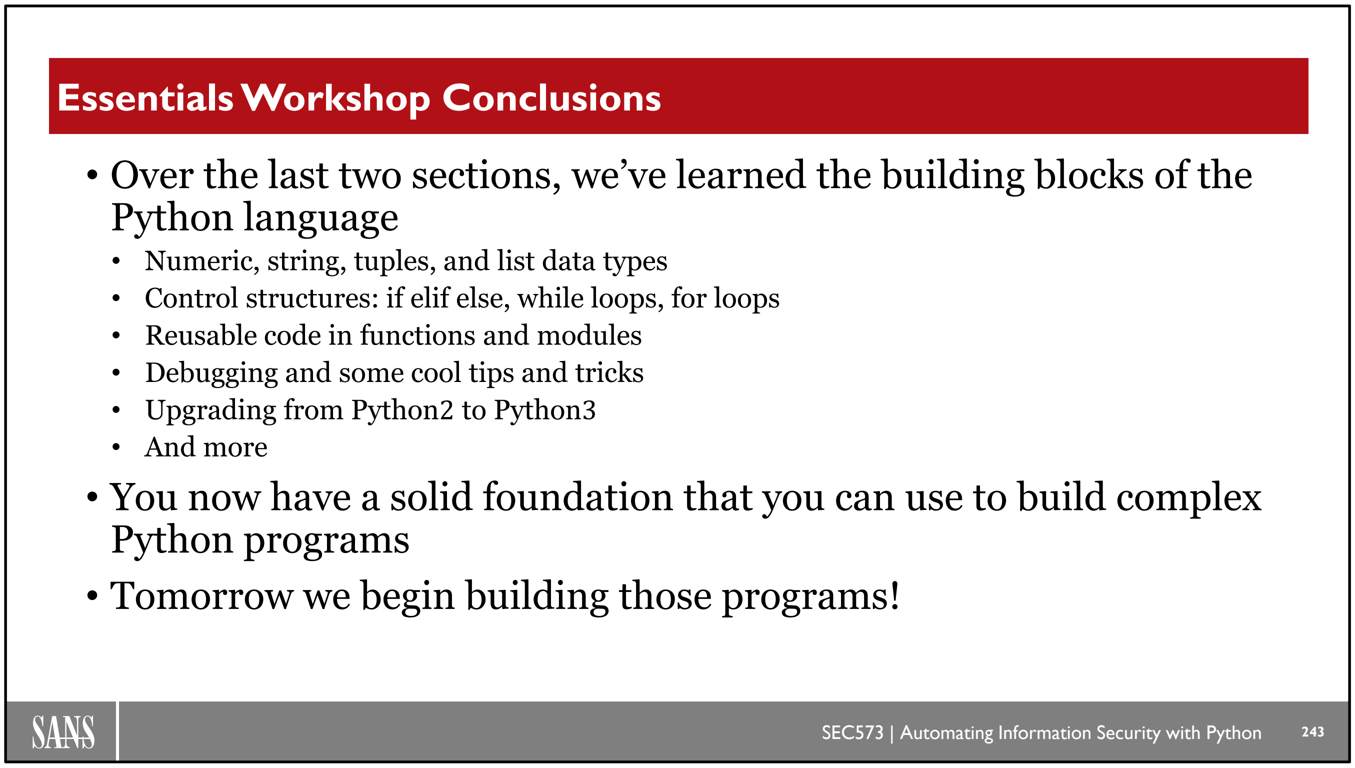


In your Workbook, turn to Exercise 2.4 (Upgrading with 2to3)

SANS

SEC573 | Automating Information Security with Python 242

Please complete the exercise in your Workbook.



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
  - Debugging and some cool tips and tricks
  - Upgrading from Python2 to Python3
  - And more
- You now have a solid foundation that you can use to build complex Python programs
- Tomorrow we begin building those programs!

SANS SEC573 | Automating Information Security with Python 243

We have covered a lot of ground in one day. Today 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. We learned how to debug Python and upgrade from Python 2 to Python 3.

With this basic set of skills in hand, we are ready to build more complex programs. Over the next three sections, we will apply these skills (and a few new ones) to build tools to perform basic information security tasks.