Python for Cybersecurity

Edrick Goad

Python for Cybersecurity

Automated Cybersecurity for the beginner

Edrick Goad

This book is for sale at http://leanpub.com/pythonforcybersecurity

This version was published on 2021-09-10

I would like to thank my wife for loving and encouraging me every day. I would have never continued to try and make myself better without her love and affection to assist me.

Contents

Introduction

The goal of this book is to introduce you to Python with a focus on cybersecurity. This book will use a "learn by doing" approach to help teach about Python and how to use it by performing various cybersecurity related tasks. As new ideas and concepts are introduced, I will do my best to explain them in introductory terms, and then provide links to where you can find more information.

The assumption for this book is little to no experience with Python, Linux, or cybersecurity topics. These topics will be presented as introductory as possible, i.e a 100-level class. However, this book will not cover everything related to these topics. For various reasons (pacing of the labs, learning at a later point, unnecessarily deep details, and so on), many details will be glossed-over or skipped entirely.

Specifically, much of the background information for scripting and programming that would be found in development courses will be bypassed. While important to eventually learn, these details may be overly confusing and unhelpful at this point.

An example of information that will be excluded from this book is the various data types used in computes. While it is helpful to know the difference between a bit, byte, word, double-word, string, small-int, big-int and others, these concepts can be learned later.

All of the labs and activities in this book are designed to work with the Raspberry Pi. These are inexpensive computers that were initially designed to help train students in low-income environments. Since their initial development however, the Raspberry Pi has become a very popular platform for Internet of Things (IoT) projects, gaming, media servers, and also desktop replacements. A complete Raspberry Pi system, with everything needed except keyboard, mouse, and monitory, can be purchased for less than $100. The low price and versatility, make this platform ideal for learning environments.

Additional resources

It is suggested that you do additional research regarding the topics or concepts you find interesting or needing clarification. Throughout this book there will be several **Additional resources** sections that include links to information I have found that may be helpful. Along with these links, I suggest using a search engine to find additional information.

Two additional books I can suggest are found below.

https://automatetheboringstuff.com/[1]

https://inventwithpython.com/cracking/[2]

[1] https://automatetheboringstuff.com/
[2] https://inventwithpython.com/cracking/

Chapter 1: Introduction to Linux and Version Control

In this chapter we will cover the following:

- Introduction to the Raspberry Pi
- Introduction to Linux
- Common Linux commands to know
- Absolute vs. Relative paths
- Introduction to Version Control
- Introduction to Python

Introduction

This chapter covers many of the basics we will be using throughout this book. Many of the tasks, scripts, and examples can work without the foundations presented here, but it is suggested to at least briefly read through the contents to be familiar with them.

Introduction to the Raspberry Pi

The Raspberry Pi is a small single-board computer (SBC) that was designed to assist in teaching computer science in developing countries. The initial goal of the Raspberry Pi was to create a fully functional computer for $25, all that was needed to work with the device is a keyboard, mouse, and monitor.

Raspberry Pi 4

The name Raspberry Pi was selected based on the tradition of naming early computer companies after fruit (Apple, Blackberry, and so on), and the Python scripting language.

Since its initial release in 2012 the Raspberry Pi has undergone several revisions. Each version has been released with more power and capabilities, including built-in Wi-Fi, support for USB-C connectors, and dual-monitor support.

This book will be focused on using the Raspberry Pi 4 and Raspberry Pi 400. While specifically focused on this generation of hardware, the examples shown should work on prior generations as well.

Raspberry Pi 400

https://www.raspberrypi.org/[3]

Introduction to Linux

Linux is open-source operating system (OS), similar to Microsoft Windows or Unix. Initially developed by Linus Torvalds in 1991, the Linux OS has grown extensively in capabilities and use throughout the world. The term open-source means the source code of the operating system is open to anyone who wishes to view, and change, the operating system. This ability makes it extremely flexible and capable of running on both the lowest-end equipment, and highest-end super-computers.

The Raspberry Pi uses a customized distribution of Linux as "Raspberry Pi OS". This Raspberry Pi OS is based on the Debian distribution, which along with its derivatives, is one of the most popular Linux distributions today.

For this book we will be using the Raspberry Pi OS on the Raspberry Pi. If you don't have access to a Raspberry Pi, but are familiar with virtualization tools like Hyper-V, VMware, and VirtualBox, you can run virtualized copies of the Raspberry Pi Desktop or other Linux distributions.

Common Linux commands to know

There are several commands that we will be using regularly in our Raspberry Pi. These will mostly be fairly basic commands dealing with managing files and folders on our computer. We won't describe them in too much detail here, but the primary commands we will be using are:

[3] https://www.raspberrypi.org/

NOTE: Linux commands are case-sensitive as shown below. There is a difference between typing `ls` , `Ls` , `lS` , and `LS`.

- pwd: Print Working Directory - tells us what folder/directory we are in.
  - usage: `pwd`
- cd: Change Directory - changes the directory we are in.
  - usage: `cd Documents`
- nano: a terminal-based text editor available on most Linux distributions.
  - usage: `nano HelloWorld.py`
- cat: Concatenate - used to print the contents of a file to the screen, or combine files together.
  - usage: `cat HelloWorld.py`
- cp: Copy - copies files from one directory to another.
  - usage: `cp HelloWorld.py HelloWorld2.py`
- mv: Move - moves files from one directory to another.
  - usage: `mv HelloWorld2.py ~/Documents`
- grep: Global Regular Expression Print - searches files and returns patterns.
  - usage: `grep hello *.py`
- mkdir: Make Directory - creates directories/folders.
  - usage: `mkdir CH02`
- sudo: Super User Do - allows a normal, non-administrator, account to perform admin tasks.
  - usage: `sudo reboot`
- tail: returns the last several lines of a file.
  - usage: `tail /var/log/syslog`
- head: returns the first several lines of a file.
  - usage: `head /var/log/syslog`
- chmod: Change Modifier - changes permissions and "executable" flags on files.
  - usage: `chmod +x HelloWorld.py`
- ping: performs an ICMP ping to a remote host.
  - usage: `ping google.com`
- git: a version control system that tracks changes to source files.
  - usage: `git pull`
- useradd: creates a user on the Linux host.
  - usage: `useradd testuser1`
- userdel: deletes a user on the Linux host.
  - usage: `userdel testuser1`

See Also

This is a short list of common Linux command and there are many more commands, and many more ways to use these commands. Because of the nature of this book, we will describe the commands more as we encounter them.

For more information about these, and other, Linux commands, look online for the commands and how to best use them. Some starting pages would be:

35 Linux Basic Commands Every User Should Know[4] from hostinger.com.

Common Linux Commands[5] from dummies.com

Absolute vs. Relative paths

In most operating systems, files are referred to in two ways: via an absolute path, or a relative path. An absolute path always starts from a central location, before guiding to the destination. A relative path however starts from where you currently are.

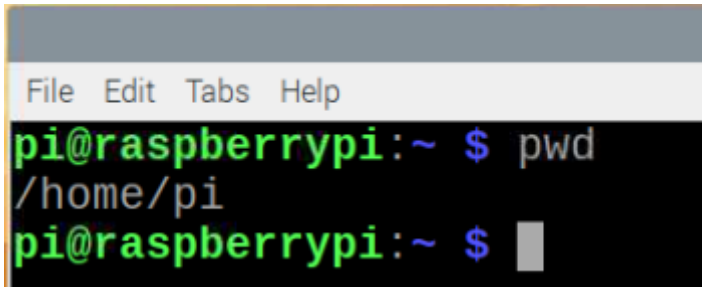
To highlight the differences, say you are driving someone to a park you have never been to before. As an example of an absolute path, your friend requires you to drive to their home before taking you to the park. This process may require extra time and effort to you because their home may be a long way away, and you may pass the park on your way to their home. However, the benefit of this method is that you can always get to the destination by going to the friend's home first.

An example of a relative path would be your friend taking you directly to the park. Often times this is quicker and easier because you don't need to travel to a central location first. However, the instructions your friend provides only work when you are at that specific starting point. If you try to get to the park from a different starting point, you need different directions to reach your destination.

Absolute paths in Linux

Absolute paths in Linux always start at the "root" of the operating system. The root of the OS is identified by a single forward slash (/), and all resources, files, and folders are referenced from there. For instance, to access the "home directory" of the **pi** user, the absolute path is `/home/pi`.

[4] https://www.hostinger.com/tutorials/linux-commands
[5] https://www.dummies.com/computers/operating-systems/linux/common-linux-commands/

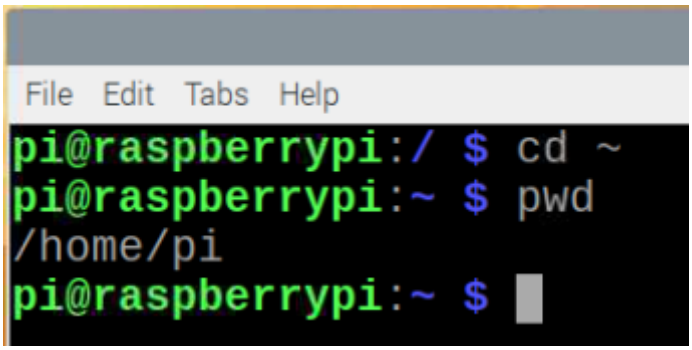


Reporting the absolute path with the pwd command

Relative paths in Linux

Relative paths in Linux always start from where you are, which can complicate tasks if you are unsure about your current location. Additionally, relative paths utilize a few unique characters to reference specific items.

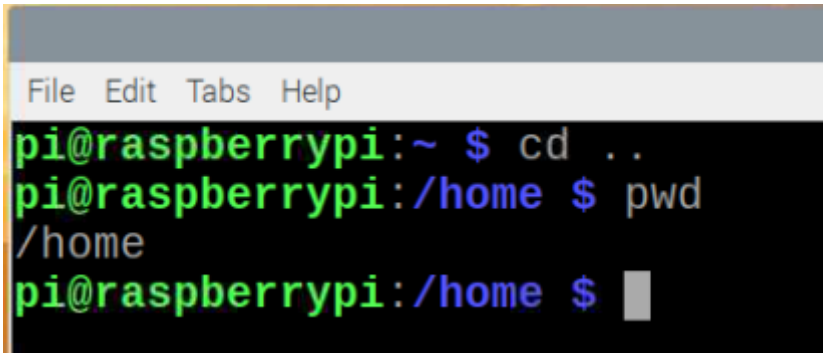
The first unique character used is the tilde (\). The \ character is frequently used to reference a user's home directory.



Using the tilde to relative path

The next unique character used is the single dot (.). The . character is used to reference the current directory. This is useful when executing scripts due to Linux security requiring the script path and name both be represented.

The last unique character used for relative paths is the double dot (..). The .. is used to reference the parent directory, or the directory above the current one. This can be used multiple times in the same command to reference multiple layers at the same time.



Moving up 1 level with the cd command

Introduction to Version Control

Version Control, sometimes referred to as "revision control", "source control", and "source code management", is a process of tracking changes to files over time. One of the main goals of version control is to track changes that occur so that they can be reviewed and if necessary, rolled-back.

A simple form of version control is simply naming files with a number appended to it. For instance, you may have a resume that you named Resume001.docx. As you make changes to the resume for various positions or organizations, you increase the number to Resume002.docx, Resume003.docx, and so on. If, for some reason, you decide you don't like the latest version anymore, you could review the prior versions and either "roll-back" to a prior version, or combine changes from prior versions with the current version and save it as Resume004.docx.

The idea of a Version Control System (VCS) is to automate this previously manual process. With a version control system you can edit files, and then "commit" changes to a repository. These tools enable you to keep the file name the same between edits, and then review changes based on version numbers.

There are a lot of version control systems in existence today. For this book we will be focusing on using Git, which is a "distributed version control" system. Meaning you can keep a local repository on your computer (including history), and synchronize it with a remote repository.

Working with Git and GitHub

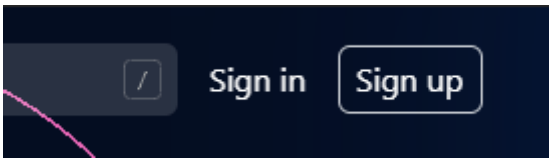
While there are many ways to work with Git, GitHub, and other repositories, we will be focusing on what I believe to be the easiest methods to get started. In this case, we will begin by creating an account on GitHub, "copying" a template repository, and then "cloning" the repository to the local computer. Once we have the repository local, we can make changes synchronize them among our devices.

> **Note:** These are very basic commands and more advanced users will quickly bypass the examples shown here.

Creating a GitHub account

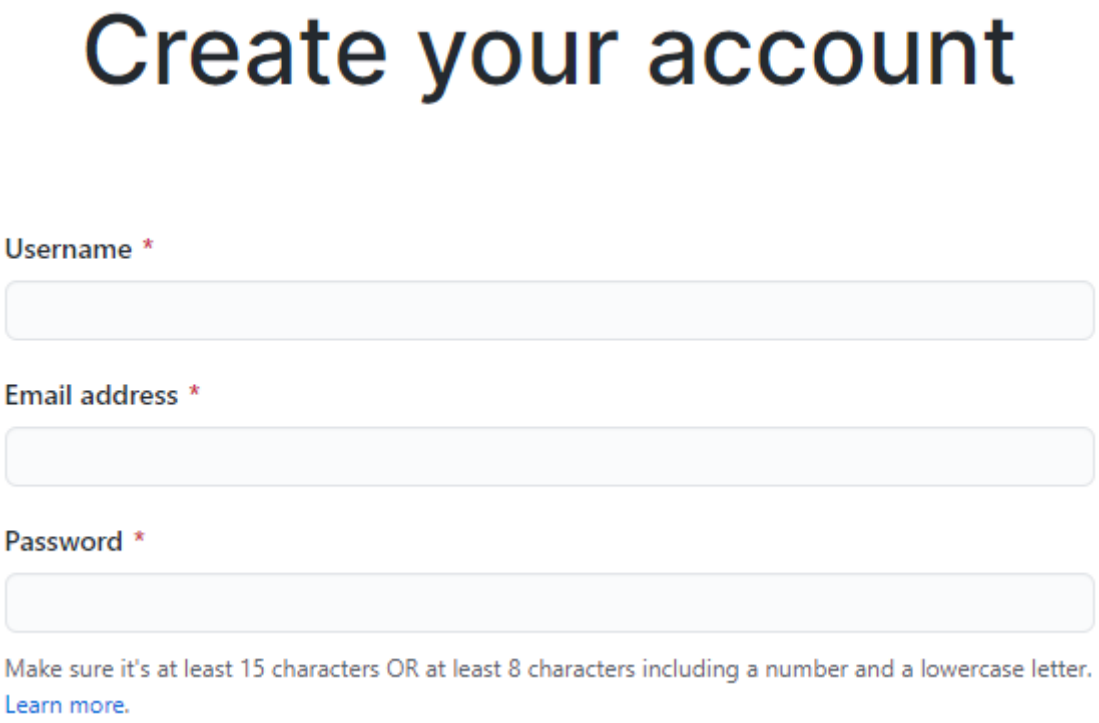
To begin we will create an account on GitHub. To start:

1. Open a web browser and go to https://github.com[6]
2. On the GitHub home page, click **Sign up**.



GitHub Sign Up

3. On the **Create your account** page, enter your desired **Username**, **Email address** and **Password**.



Creating GitHub account

4. Click to verify your account and solve the puzzle presented.

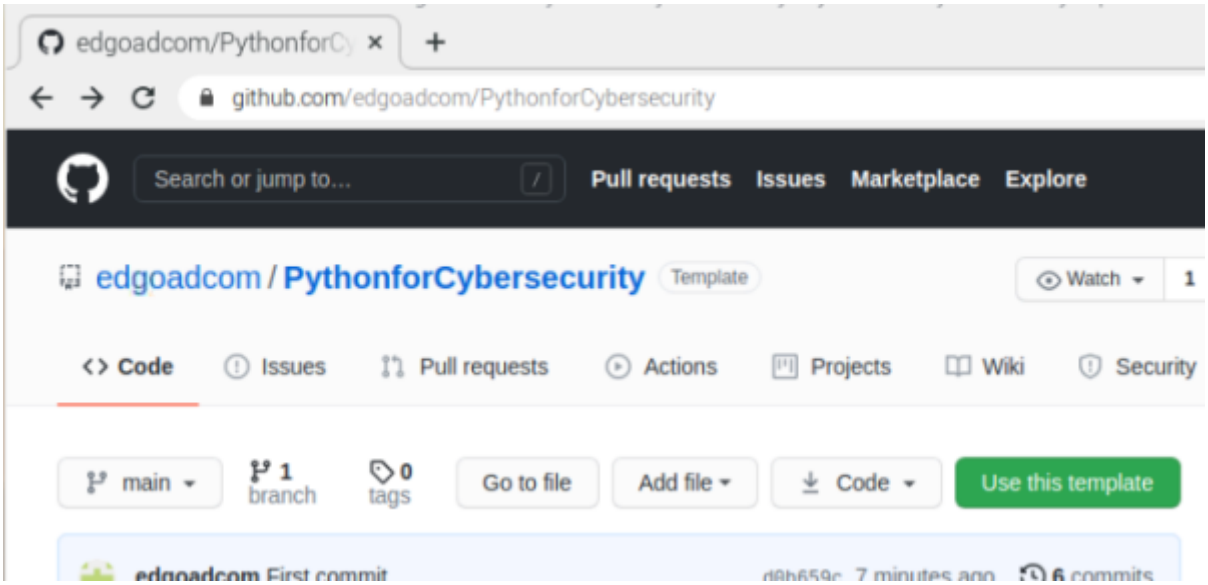
NOTE: The web page may have changed since these screenshots have been taken.

[6] https://github.com/

Copying the template repository

Now that we have an account on GitHub, we can start by copying a sample repository. A repository is where all of your files are stored and tracked. I have created a sample repository that can we can use to begin our process.

1. In the web browser, go to https://github.com/edgoadcom/PythonforCybersecurity[7], this is the repository for the sample files in this book.
2. Click **Use this template**.



GitHub template

3. Enter a new **Repository name**.
4. Set the repository to **Private**.
5. Click **Create repository from template**.

[7] https://github.com/edgoadcom/PythonforCybersecurity

Creating new repository

Your copy of the repository is now available for your use.

There are multiple ways for us to use and access the files in GitHub. The easiest method, using Visual Studio Code, will be covered in the next chapter. For more information about using the **Git** command line, look in the appendix.

Introduction to Python

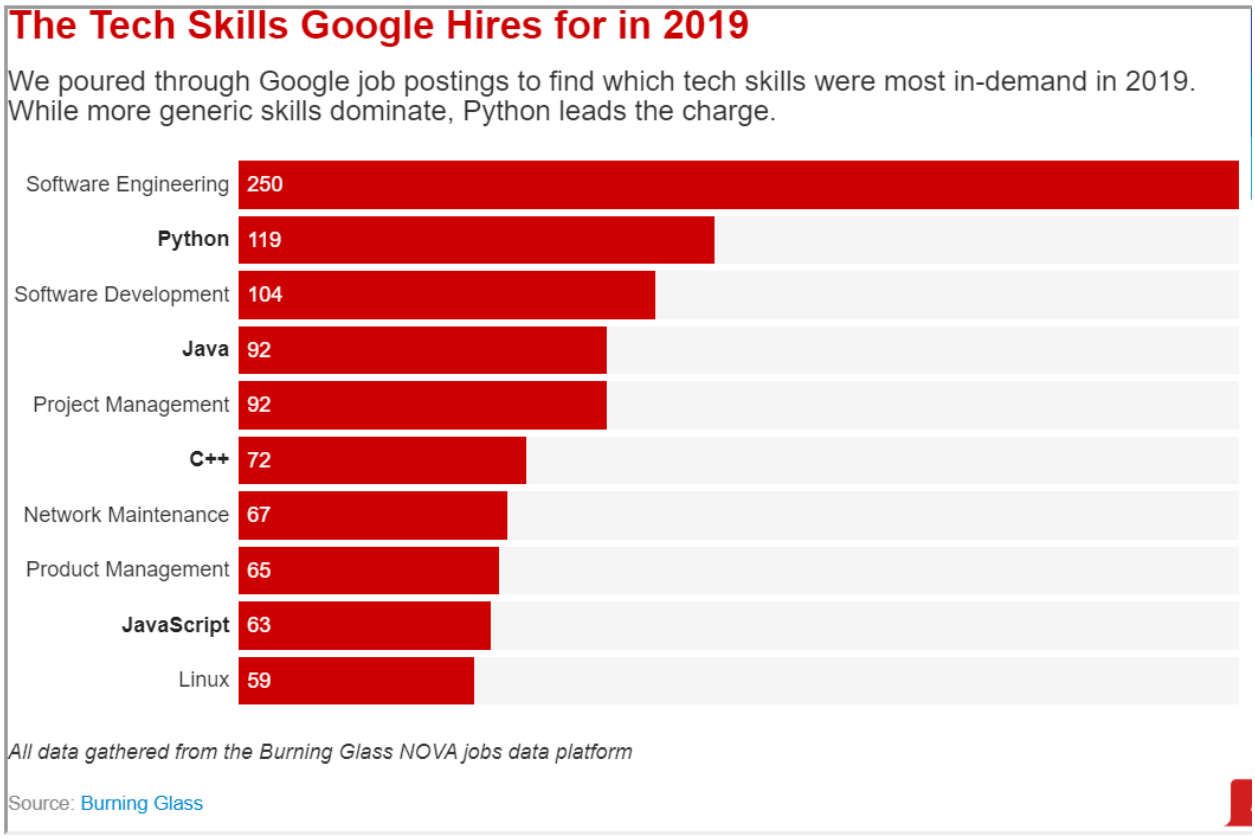
Python is a programming/scripting language initially developed in 1989 by Guido van Rossum. Many of main tenants for the Python programming language center around readability and simplicity, which has made it extremely accessible and easy to use. Python is meant to be "fun to use", which is reflected by its name-sake, Monty Python's Flying Circus.

For this book, and for most current projects today, we will be using Python version 3. Previously, Python 2 was the most popular version of the programming language, and several examples can be

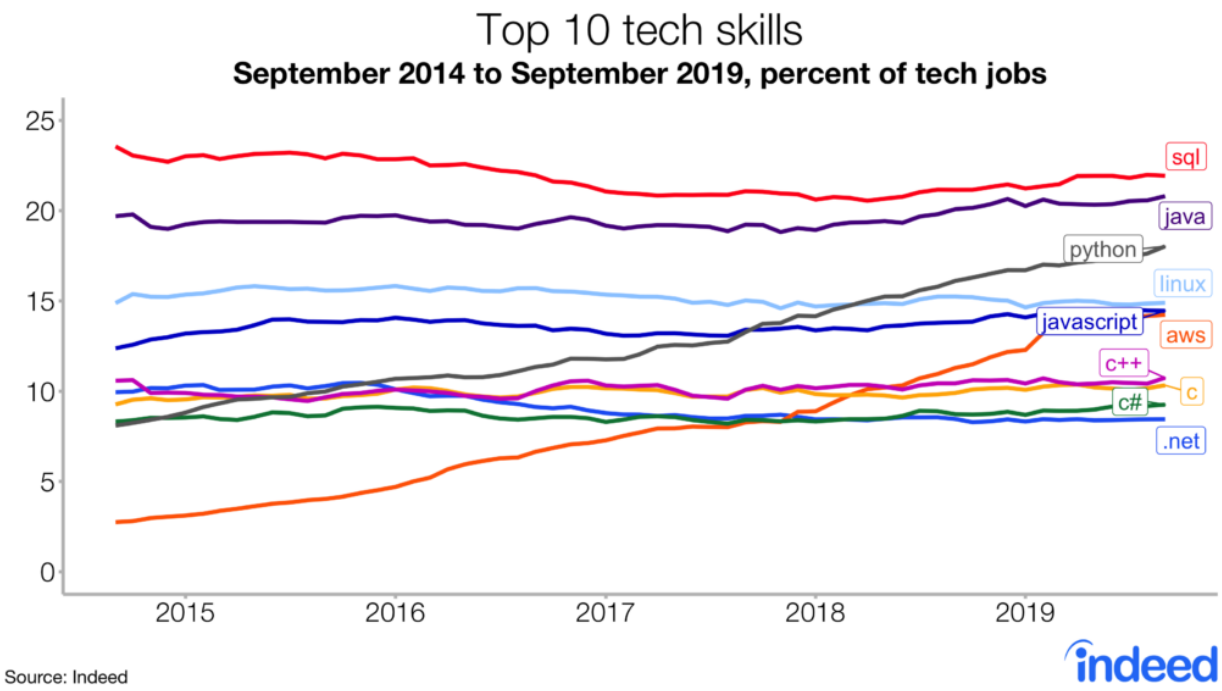
found online. There are some key differences in the versions that stop a script written in Python 2 from running in Python 3.

In recent years, the flexibility and usability of Python has resulted in its use growing rapidly. Python has become a critical skill to know in several realms including: networking, cyber-security, systems management, data scientists, artificial intelligence, and many more.

A few studies have occurred that show Python as one of the top tech skills looked by employers today.



Burning Glass survey



Indeed survey

These are just a few examples, a quick search online will find several other examples of how popular and in-demand Python scripting is.

top 10 most popular tech skills of 2020[8]

The 6 Most In-Demand Tech Jobs For 2020/[9]

7 Most In-Demand Programming Languages To Learn In 2020[10]

[8] https://www.cnbc.com/2019/11/19/these-will-be-the-top-10-most-popular-tech-skills-of-2020.html
[9] https://www.kivodaily.com/careers/the-6-most-in-demand-tech-jobs-for-2020/
[10] https://www.agiratech.com/most-in-demand-programming-languages-learn-2020/

Chapter 2: Python and IDEs

In this chapter we will cover the following:

- Installing Python
- Installing Visual Studio Code
- Visual Studio Code and GitHub
- Interactive Python
- Hello World
- Debugging
- Simple Calculator

Introduction

In this chapter we will begin our first steps with Python. Starting with interactive Python and moving toward our first Python script - the infamous "Hello World". We will also introduce the idea of an Integrated Development Environment (IDE) to show how useful it can be for creating and debugging scripts.

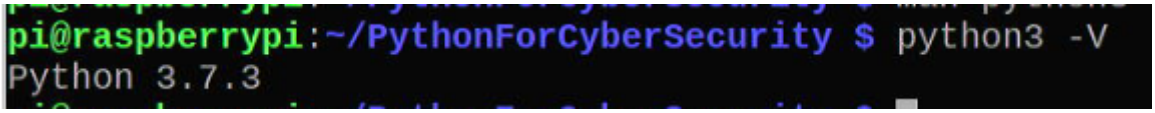
Throughout this book we will be using Python version 3, which is referenced as **python3**. Python versions 1 and 2 are both referenced as simply **python**. When we are referring to the Python language, and running Python scripts, note we will use the command **python3**.

Installing Python

By default, Python will be installed in most, if not all, Raspberry Pi environments.

Verify Python is installed

To very Python is installed on a system, open a terminal and type `python3 -V`.



Showing the Python version

NOTE: The upper-case "**V**"

This will return the version of Python installed. Most any Python version 3.x will work for this book.

Install Python on Raspberry Pi

If Python isn't installed on the Raspberry Pi, you can easily install it. Open a terminal and type the following commands:

```
sudo apt update
sudo apt -y install python3
```

> **NOTE:** After typing the first command you may be prompted to reenter your password. This is a security feature of Linux, simply type your password and hit enter.

When finished, type `python3 -V` again to return the version number and confirm it is installed correctly.

Install Python on other Operating Systems

Python works on most popular operating systems today. If it's not already installed, it can quickly be installed by downloading it from python.org.

1. Open a web browser and go to https://python.org[11]
2. Go to **Downloads** and select your operating system.
3. Click to download and install the **Latest Python 3 Release**.

> **NOTE:** Unless there is a specific reason to choose a different version, always choose the latest *stable* version.

Download from python.org

4. Walk through the installer accepting the defaults.

Installing Visual Studio Code

Visual Studio Code is an Integrated Development Environment (IDE) developed by Microsoft. This IDE is supported on most major operating systems and is free to install and use. While Python scripts can be created using simple text editors like Notepad or Nano, an IDE includes several features that makes writing, running, and debugging of scripts easier.

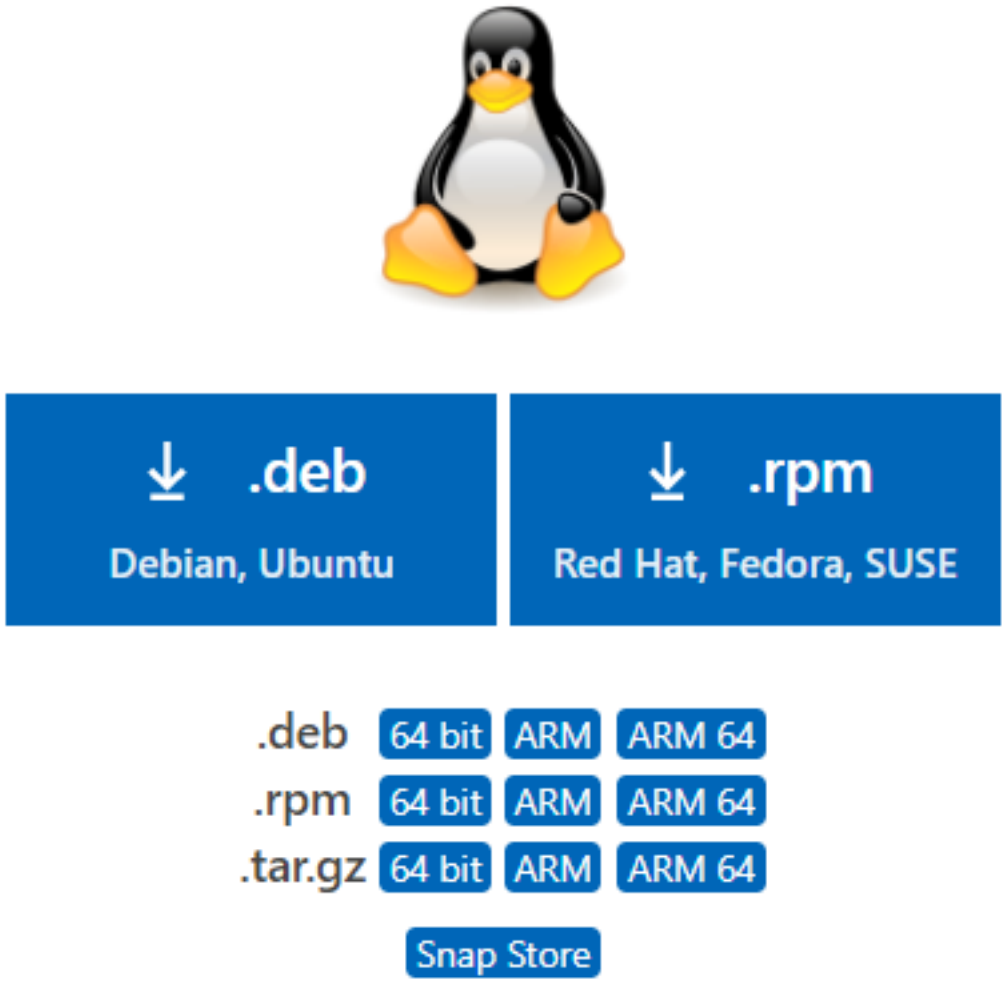
[11] https://python.org/

To install Visual Studio Code on the Raspberry Pi

1. Log into the Raspberry Pi.
2. Open the web browser and go to https://code.visualstudio.com[12]
3. Click **Other platforms**.
4. Under the Linux section (penguin icon), to the right of **.deb**, select **ARM**.

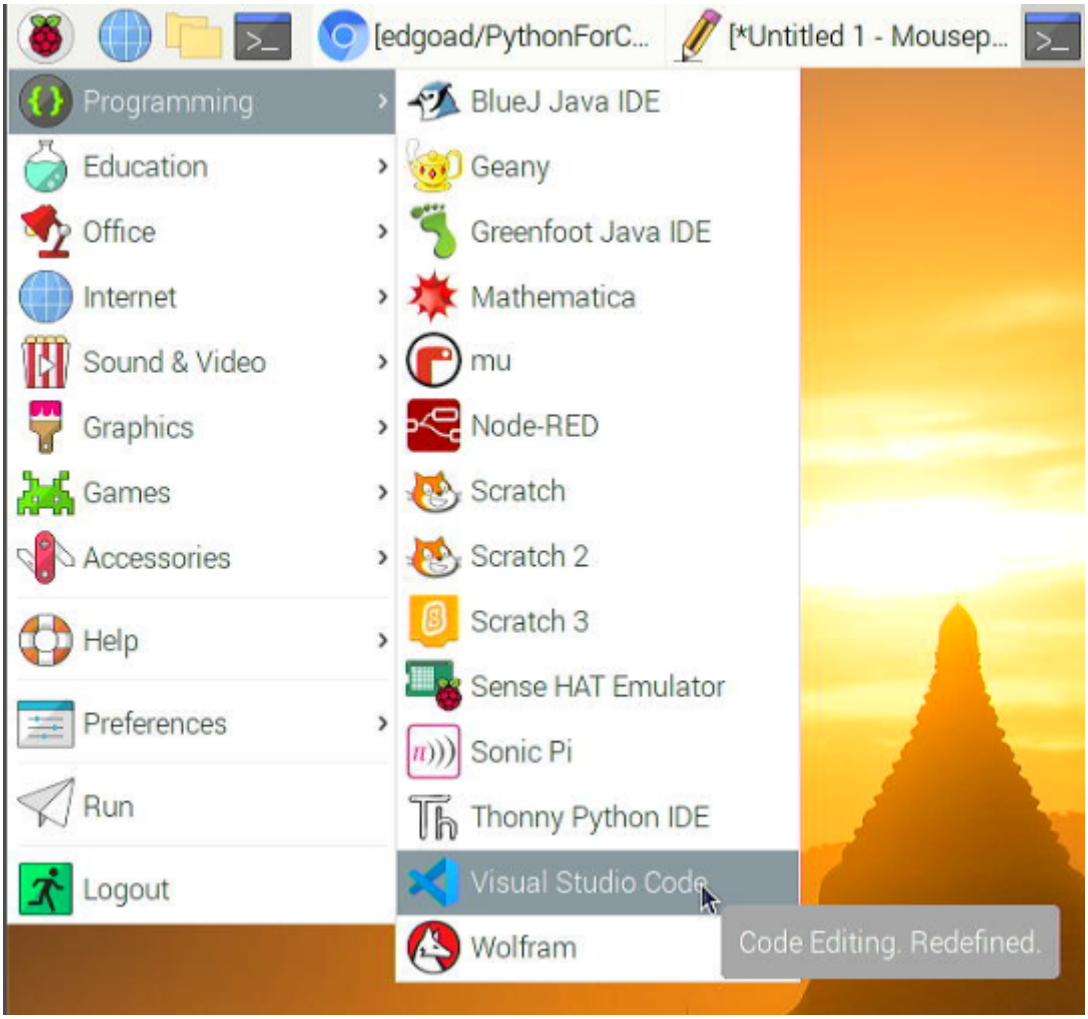
   NOTE: The other **.deb** versions are for 64-bit operating systems. The default Raspberry Pi image is 32-bit.

[12] https://code.visualstudio.com/



Visual Studio Code download

5. When the download completes, click on the downloaded file an select to install. If prompted, enter your password to confirm.
6. When installation is finished, go to the start menu, and select **Programming | Visual Studio Code**.



Launching Visual Studio Code

Installing Visual Studio Code from the command line

If you prefer to install Visual Studio Code using the command line instead, you can use the following commands. These commands are summarized from the instructions provided by Microsoft at https://code.visualstudio.com/docs/setup/linux[13]. If the commands change, look at Microsoft's site for the updated directions.

[13] https://code.visualstudio.com/docs/setup/linux

```
wget -qO- https://packages.microsoft.com/keys/microsoft.asc | \
 gpg --dearmor > packages.microsoft.gpg
sudo install -o root -g root -m 644 packages.microsoft.gpg \
 /etc/apt/trusted.gpg.d/
sudo sh -c 'echo "deb [arch=amd64,arm64,armhf \
signed-by=/etc/apt/trusted.gpg.d/packages.microsoft.gpg] \
https://packages.microsoft.com/repos/code stable main" > \
/etc/apt/sources.list.d/Visual Studio Code.list'
sudo apt install -y apt-transport-https
sudo apt update
sudo apt install -y code
```

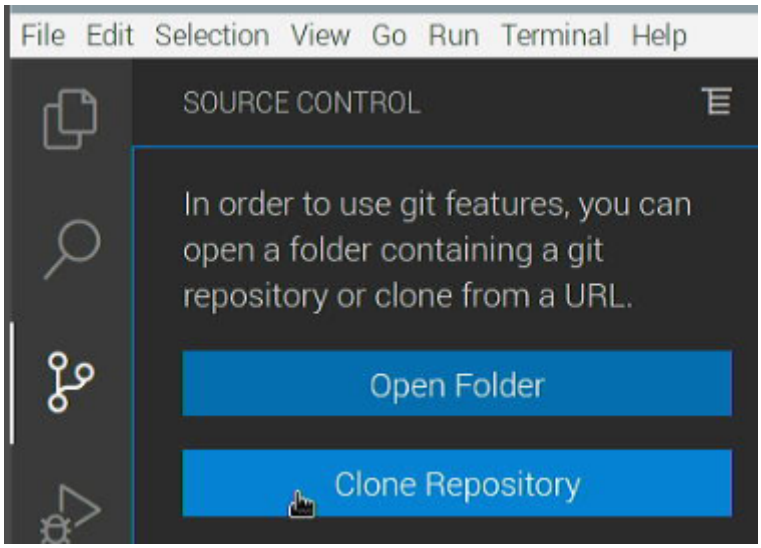
> **NOTE:** From personal experience I have found the need to run these commands more than once to complete properly.

Visual Studio Code and GitHub

Visual Studio Code allows us to access GitHub directly, streamlining our efforts to create and manage our scripts in a user-friendly form.

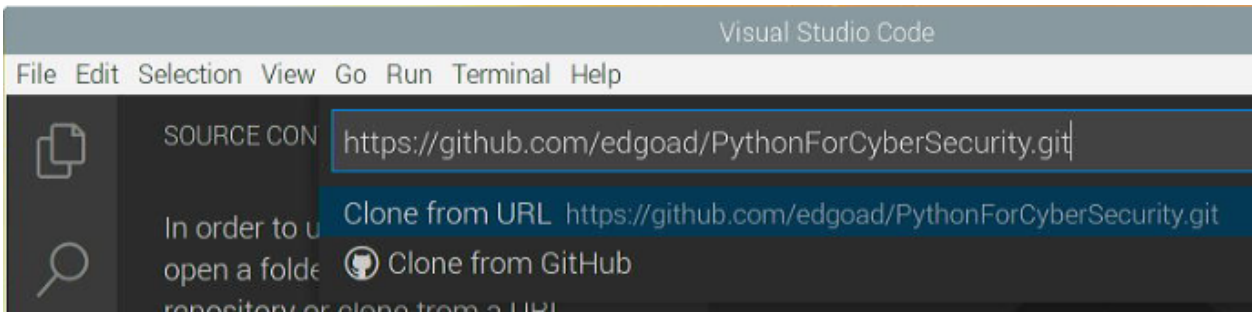
Configure Visual Studio Code for GitHub

1. Launch Visual Studio Code if it isn't already running.
2. On the **Activity Bar** (on the far left), click the **Source Control** icon (3 dots connected by lines).
3. On the **Source Control** side bar that appears, click **Clone Repository**.



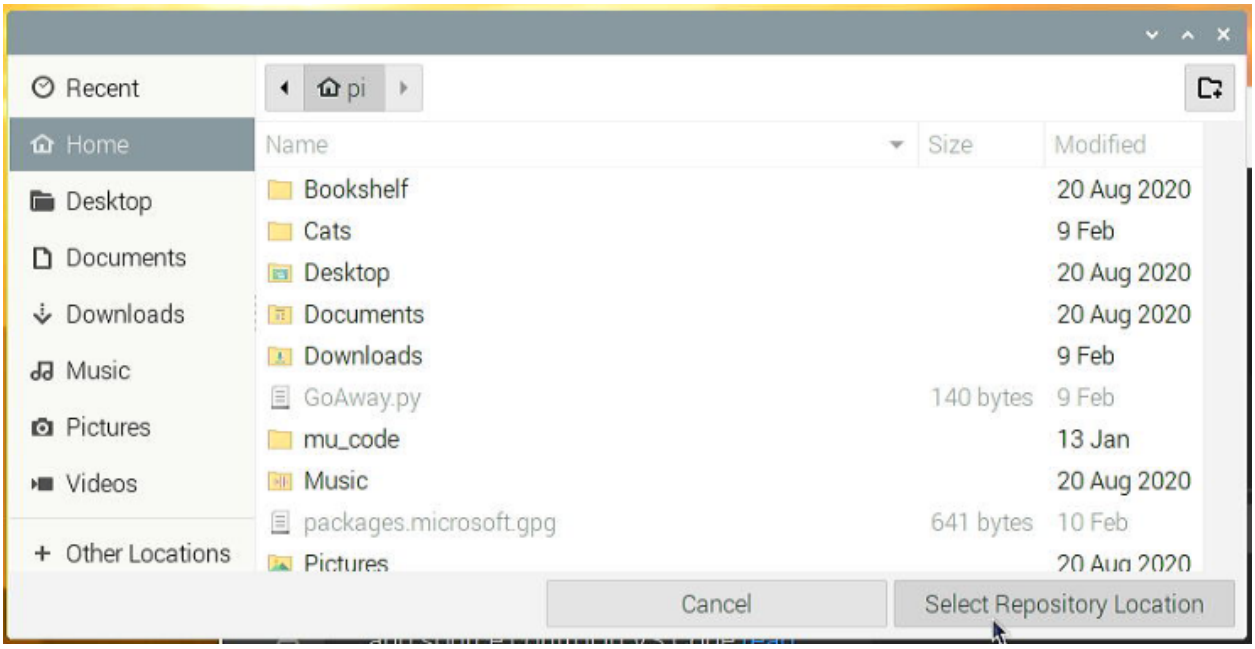
Clone Repository

4. In the text box at the top of the window, enter the GitHub URL for your repository.



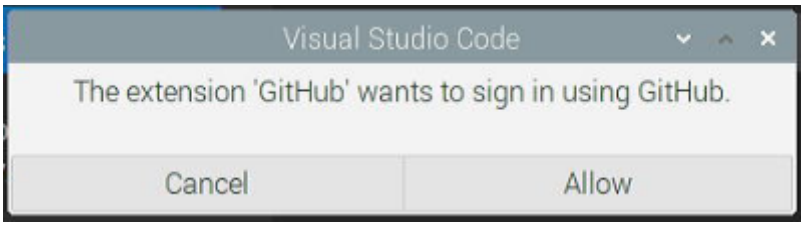
Entering GitHub URL

5. When prompted for where to keep the local repository, choose a location, or simply click **Select Repository Location**.



Repository Location

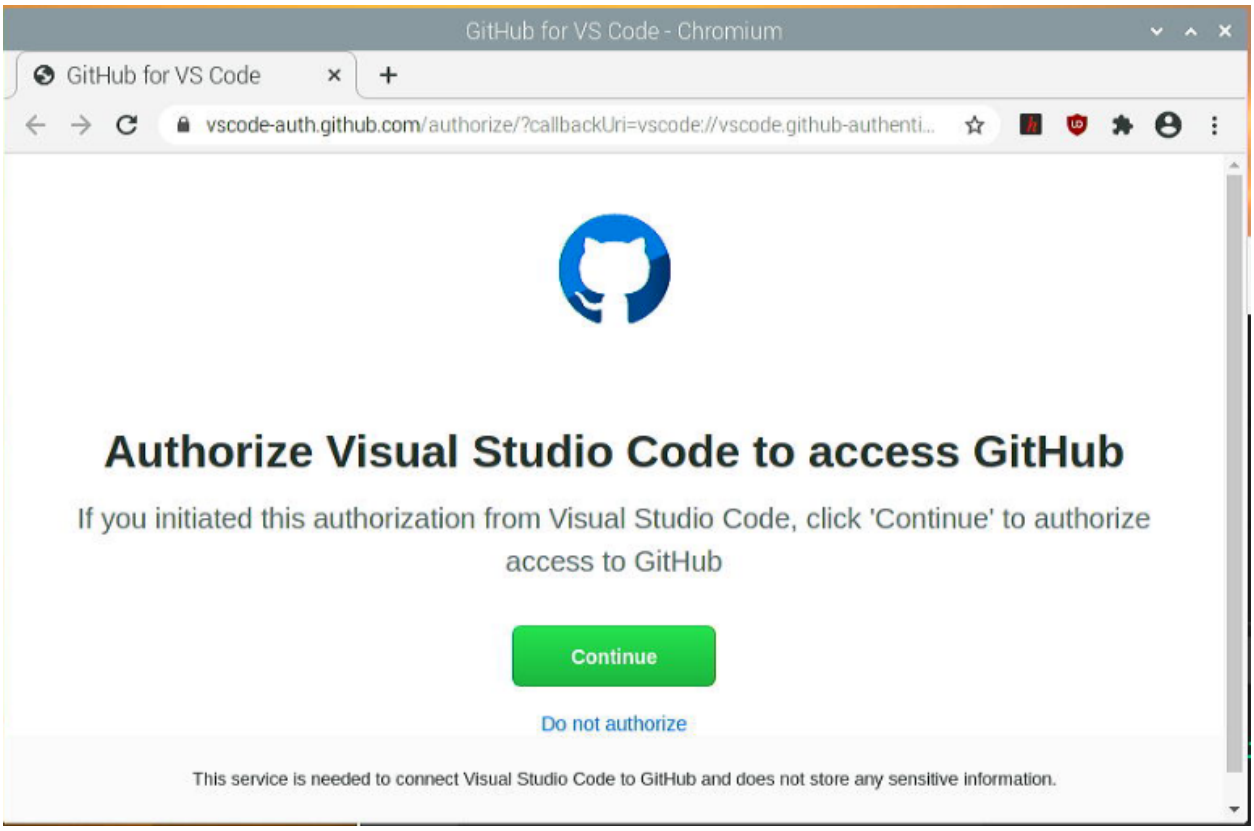
6. When prompted to allow GitHub, click **Allow**.



Allowing sign in

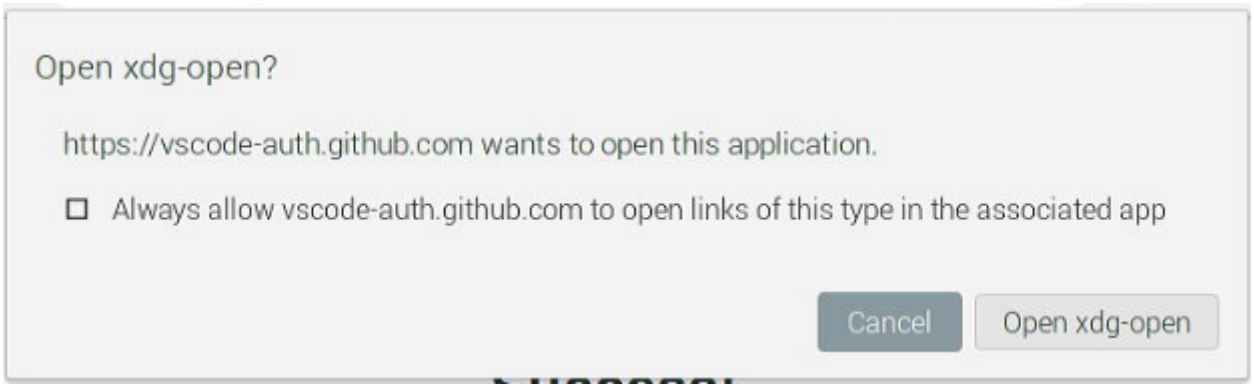
7. Visual Studio Code will open a web browser and log into GitHub.com. Once logged in, it will

prompt you to authorize Visual Studio Code to access GitHub. Click **Continue**.

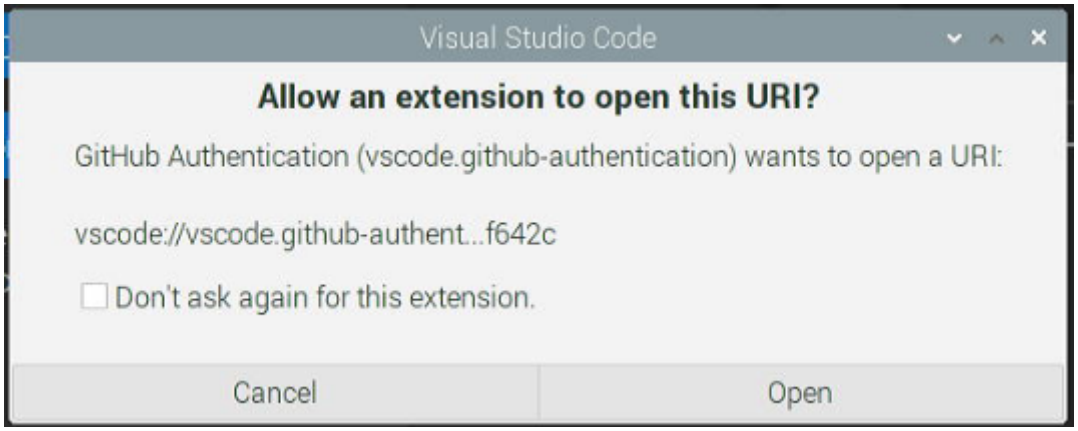


Authorizing Visual Studio Code

8. On the following pop-ups, click **Open**.

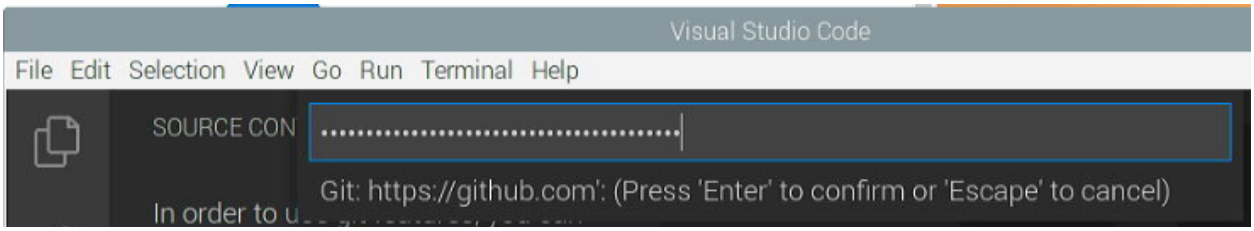


Opening link



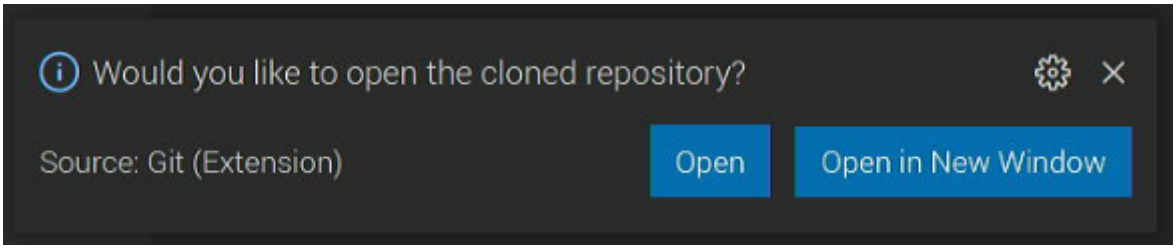
Allowing extension

9. If prompted by Visual Studio Code, re-enter your GitHub.com password and press **Enter**.



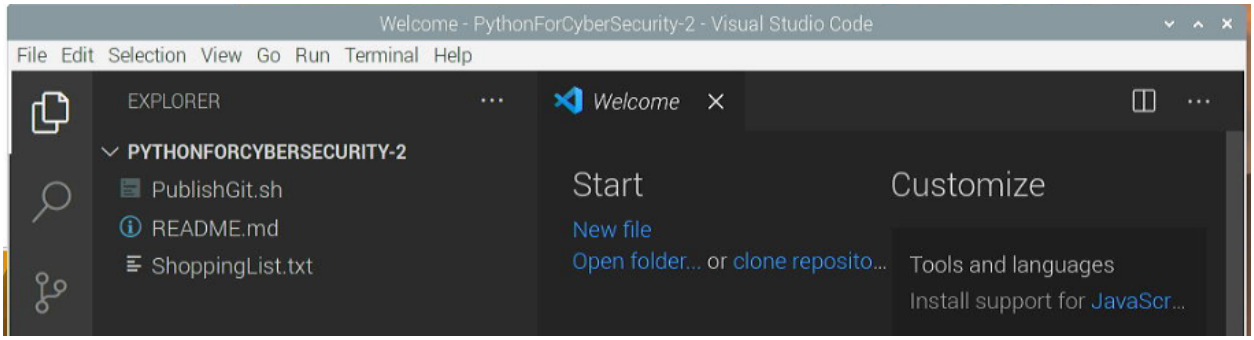
Entering GitHub password

10. At the bottom right of Visual Studio Code, a pop-up message will prompt to open the cloned respository. Click **Open**.



Open cloned repository

11. You now have a local copy of the repository accessible in Visual Studio Code. If not already selected, on the Activity bar (icons on the far left), click Explorer view - icon looks like 2 pieces of paper. The Side bar will now show a list of folders and files.

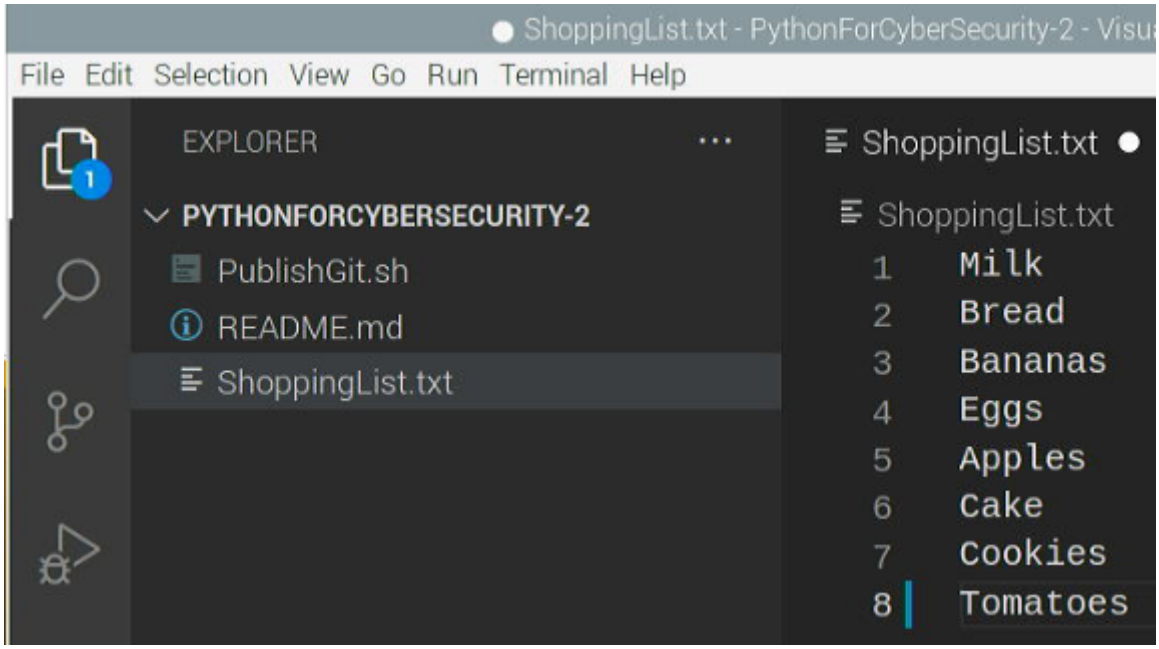


Viewing files

Committing Changes

By committing changes to Git and GitHub, we are creating different versions of our files. These versions can be reviewed to identify the changes that have occurred over time. It is a good practice to commit changes frequently.

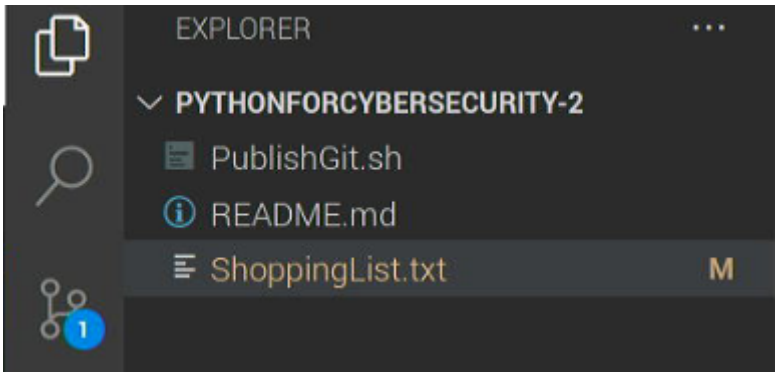
1. In Visual Studio Code, click on **ShoppingList.txt** to open it, or create a new file.
2. In the Editor pane on the right, you will see the contents of the file. Add one or two items to the shopping list.



Editing ShoppingList

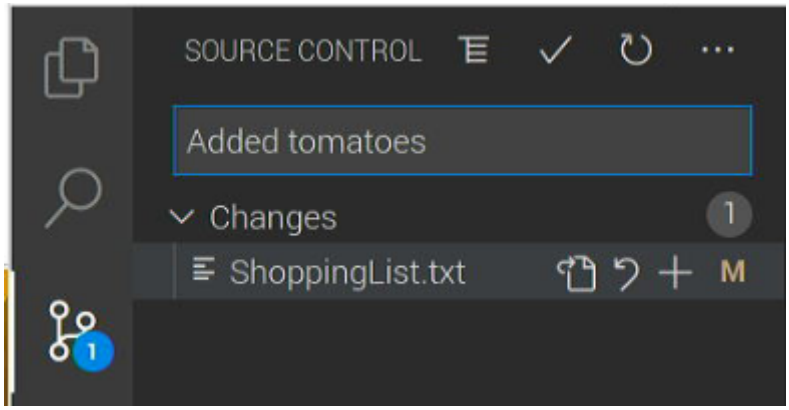
3. Save the file by selecting **File | Save**, or by pressing **CTRL+S**.

4. On the Activity bar, click the **Source Control** icon (3 small circles connected with lines). Here we will see the ShoppingList.txt is listed as having changes made.



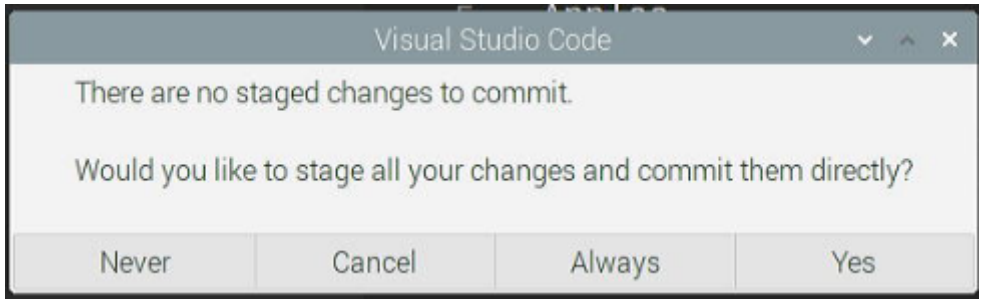
Source Control

5. In the **Message** field above the changes, enter a description of the changes made.



Adding description

6. Once the comment has been added, press **CTRL + Enter** to commit the changes. Optionally, press the 3 dots at the top of the Source Control side bar and select **Commit | Commit**.
7. If prompted by Visual Studio Code that there are no staged changes, click **Always** to always stage the changes without being prompted.



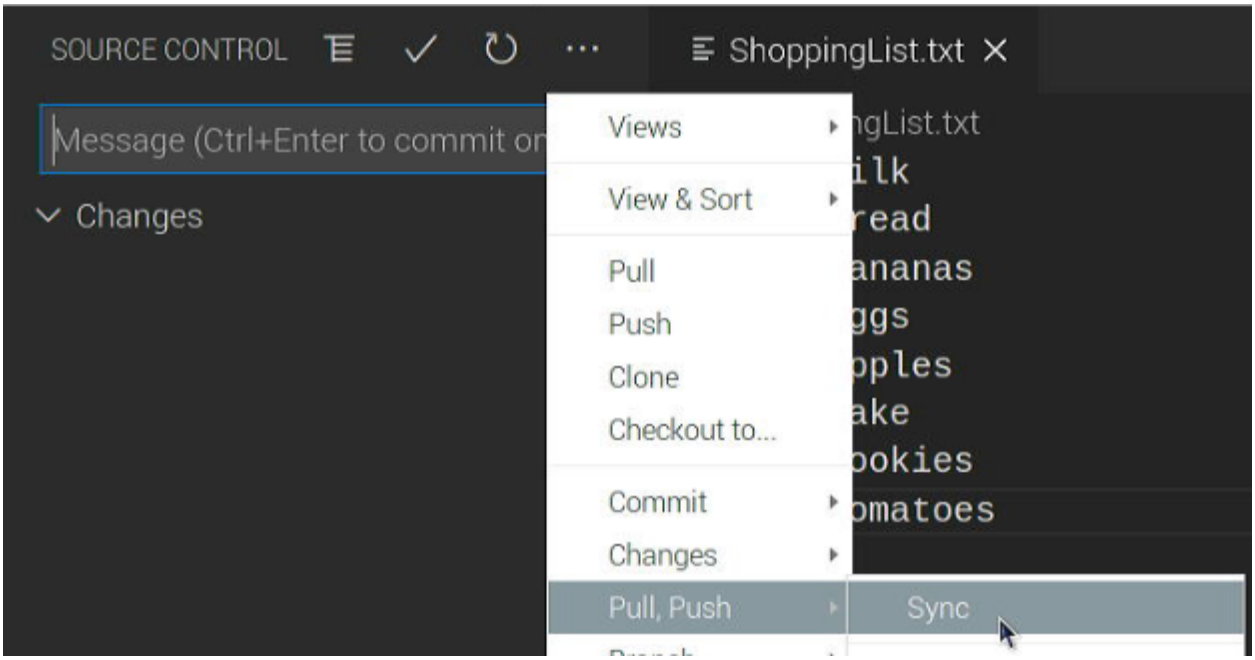
Staging changes

NOTE: As a general rule, commit comments should be descriptive and not include the word **AND**. If the word AND is needed, it suggests there should be 2 separate commits.

Synchronizing with GitHub

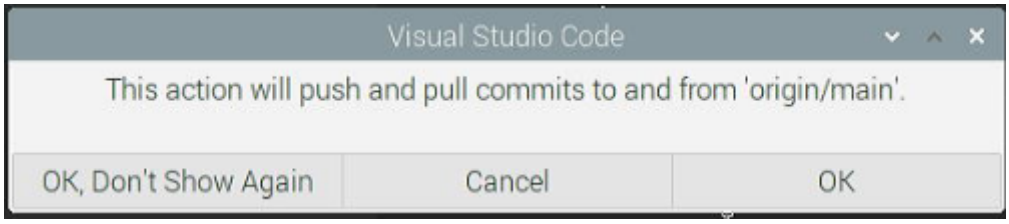
Once the changes have been stored in the **local** Git repository, we can then synchronize them to the **remote** repository on GitHub.

1. Still on the Source Control side bar in Visual Studio Code, click the 3 dots at the top of the pane. Select **Pull**, **Push | Sync**.



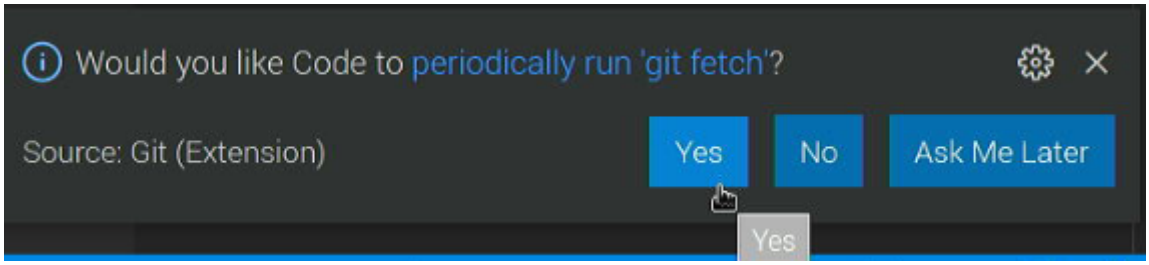
Synchronizing changes

2. If prompted by Visual Studio Code to confirm the action, click **OK**, **Don't Show Again**.



Don't show again

3. If prompted to periodically run git fetch, click **Yes**.



Periodic git fetch

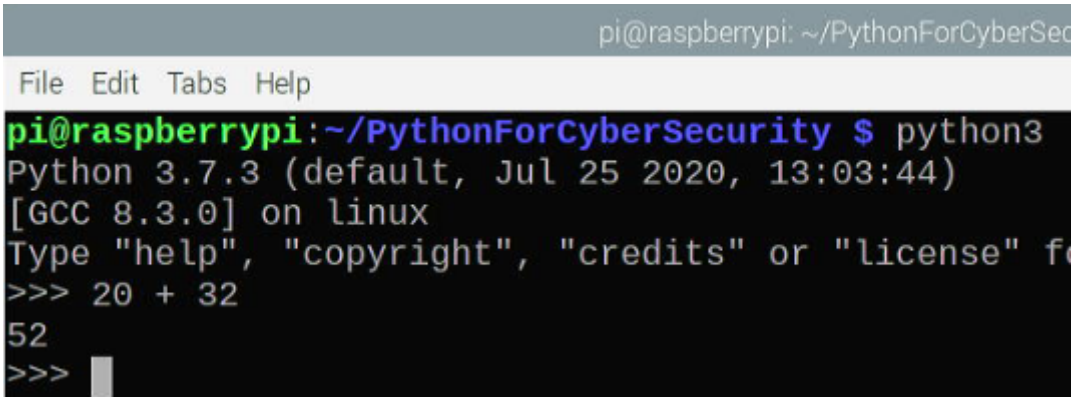
At this point all the change have saved, version controlled, and synchronized online. The changes can be viewed in Visual Studio Code, and on GitHub.com.

Interactive Python

Python works in both an interactive and scripted mode. The interactive mode is a quick and easy way to become familiar with some aspects of the Python language, as well as explore different aspects of Python.

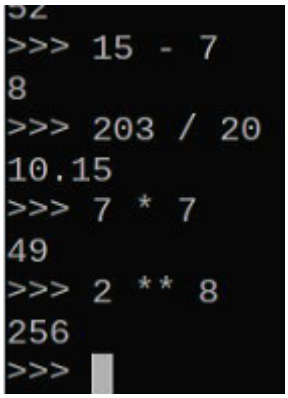
To open Python in an interactive mode, on the Raspberry Pi open a terminal window. From there, type `python3`, all lower case. The first few lines provide information about the version of Python running, as we can see in the screenshot below, I am running Python 3.7.3. Here however, we are primarily concerned with the major version of **Python 3**.

Once in interactive Python mode, we can begin typing various Python commands. The simplest commands to demonstrate are simple math. For this example, we can type `20 + 32` and hit **Enter**, Python then immediately returns the result.



Interactive python

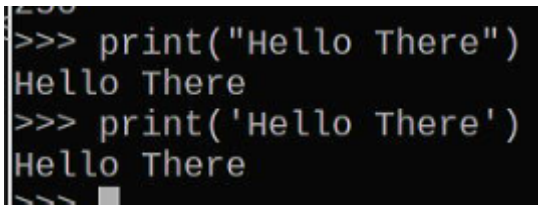
Along with addition, we can perform other math functions. For instance, we can type `15 - 7` (subtraction), `203 / 20` (division), `7 * 7` (mutltiplication), and `2 ** 8` (exponents). Once we hit enter after each of these, the results are immediately presented to the screen.



Math in interactive mode

Another common command in Python, which we will be using frequently, is the `print()` command. As its name suggests, the command will print to the screen any value we pass it.

```
1 print("Hello There")
2 print('Hello There')
```



Using different quotes

NOTE: For the `print()` statement to work properly, there are several key components that must be correct.

- **print** is all lower case.
- There must be a matching opening and closing parenthesis **(** and **)**. This becomes important as we may use multiple sets of parenthesis on the same line.
- There must be a matching opening and closing set of quotes ”“ or ’‘. Generally, it isn’t important if you use single or double quotes, but they need to match on the same line.

Our first Python Script

Python conventions

File Extension

By convention (meaning what most people follow), Python scripts always have a **.py** file extension. This helps the operating system and the user to quickly identify the file type, and how it should be used. There are many other common file extensions (for instance **.doc** and **.docx** for Microsoft Word).

Shebang

The first line in a Python script should be the shebang. The shebang is normally the first line in a script and starts with a pound sign (**#**), followed by an exclamation mark (**!**), and then the path of the interpreter for the script. The shebang is named as such by saying the words “hash” (another name for the pound sign) and “bang” (a colloquial name for exclamation mark) quickly: hash-bang -> shebang.

Different scripting languages such as bash, perl, python, php, and custom languages all have different shebangs. While not required to be included, a proper shebang will make scripts execute in a more reliable fashion. The preferred shebang for Python version 3 is:

```
1 #!/usr/bin/env python3
```

Comments

Comments in Python are identified by the use of the pound sign (**#**), and should be included throughout Python scripts. When the Python interpreter encounters a comment character, it will ignore any text following it. This is a great way to document what the script is and what it does.

Each script should start off including several comments describing what the script is, what it is supposed to do, who wrote it, and when. This information, along with other comments throughout the script, will make the script easier to work with in the future when you may not remember the purpose of the script and various commands.

```
1   #!/usr/bin/env python3
2   # Basic Hello World script
3   # By Ed Goad
4   # 2/24/2021
```

Python comments

Marking as executable

Lastly, once a script is created, we can mark it as executable so that we can run it from the command line. To mark the file as executable, on the terminal window in the same directory as the file, type `chmod +x <pythonfile.py>` (replacing the brackets with the file name). After this is completed, if you use the **ls** command, you will see the file is now a different color signifying it is executable.

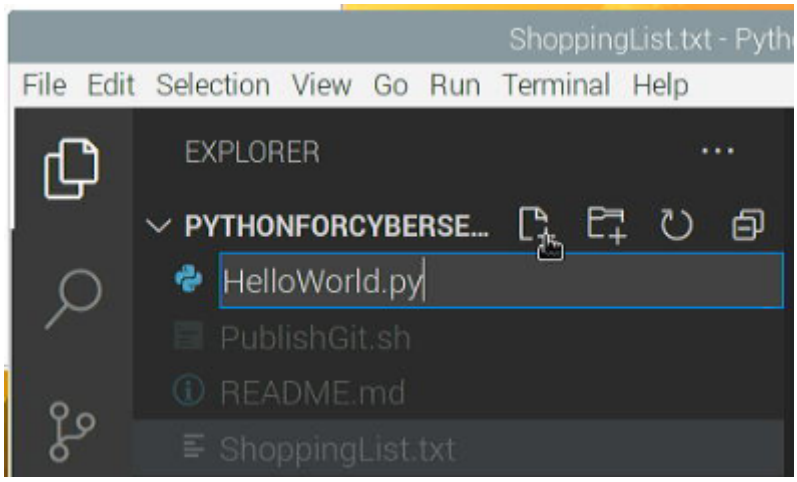
```
pi@raspberrypi:~/PythonForCyberSecurity $ chmod +x HelloWorld.py
pi@raspberrypi:~/PythonForCyberSecurity $ ls
HelloWorld.py  PublishGit.sh  README.md  ShoppingList.txt
```

Marking as executable

Getting Ready

In Visual Studio Code, select the Explorer icon on the Activity bar (looks like 2 pieces of paper on the far left). In the Explorer side bar, browse to **start | CH02 | HelloWorld.py**. This will open the file in the editor pane.

Alternatively, to create the file yourself, select the Explorer icon on the Activity bar (looks like 2 pieces of paper on the far left). In the Explorer side bar, next to where it says **PythonForCyberSecurity** click the **New File** icon - this looks like a piece of paper with a plus (**+**) on it. Name the new file **HelloWorld.py**.



Creating a new file

How to do it

Our first Python script will be the famous Hello World script. This is a simple script that simply prints to the screen the words "Hello World" and is used in most programming/scripting classes as the first example created.

Once the file is open, in the editor pane, type in the following information. Start with the shebang, add in the appropriate comments, and the final print statement.

```
#!/usr/bin/env python3
# A simple "Hello World" script in python
# Created by Ed Goad, 2/3/2021
print("Hello World")
```

Pay special attention to the line with the **print** statement. Watch to ensure the print command is lower-case, both opening and closing parenthesis **()** are included, and both quotes "" are included.

When finished, save the file by click **File | Save**, or press **CTRL + S**.

Committing changes to GitHub

Commit the changes to the local Git and GitHub by:

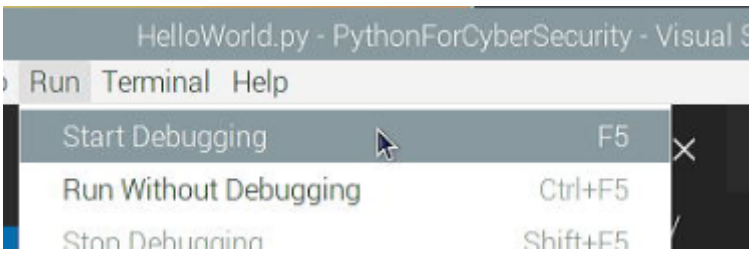
1. On the Activity bar, click the **Source Control** icon (3 small circles connected with lines).
2. In the **Message** field above the changes, enter a description of the changes made.
3. Once the comment has been added, press **CTRL + Enter** to commit the changes. Optionally, press the 3 dots at the top of the Source Control side bar and select **Commit | Commit**.
4. Still on the Source Control side bar in Visual Studio Code, click the 3 dots at the top of the pane. Select **Pull, Push | Sync**.

How it works

Running inside Visual Studio Code

To execute our script inside of Visual Studio Code

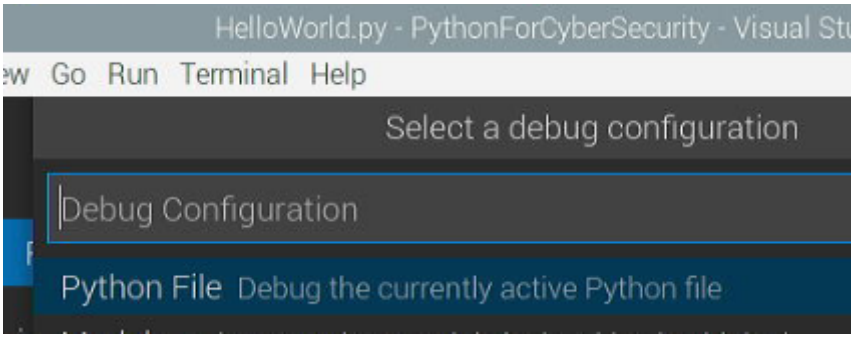
1. With the file still open, go to **Run | Start Debugging**



Start debugging

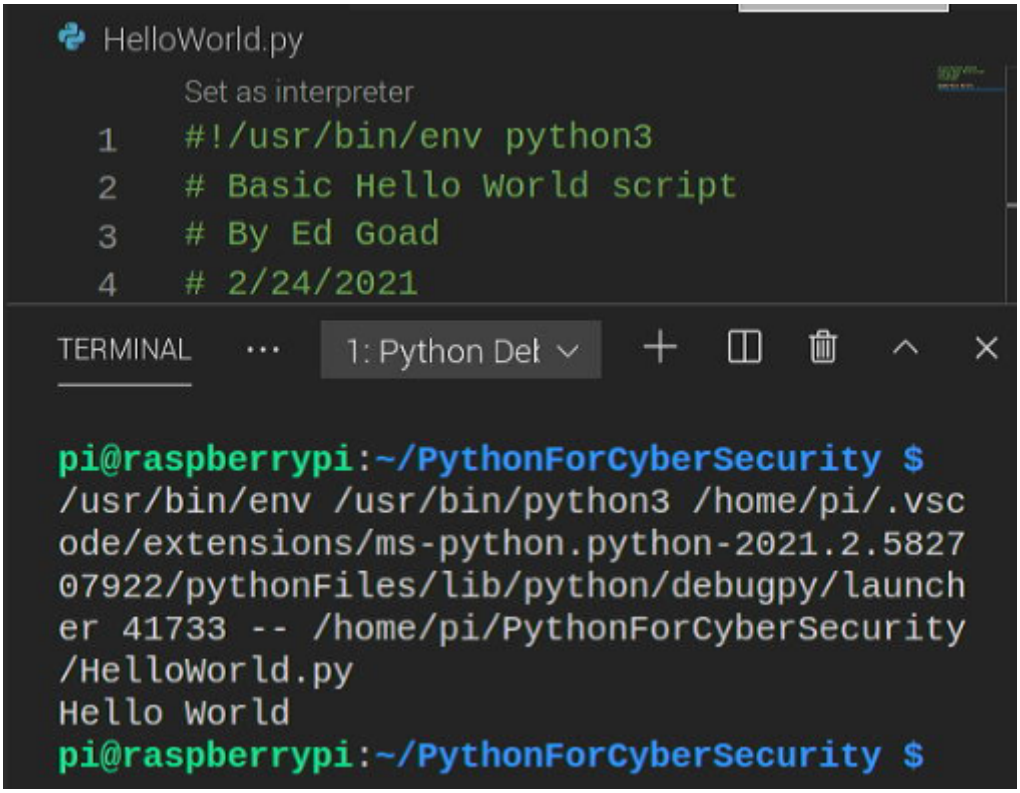
> **Note**: as shown on the **Start Debugging** line, you can also press **F5** on the keyboard to perform the same task.

2. In the pop-up window, select **Python File**.



Specify Python File

3. At the bottom of Visual Studio Code you will see a **TERMINAL** window open, and after a few seconds the script will complete, with "Hello World" printed.



Viewing the terminal

Running from the command line

Running Python scripts from the command line is a relatively simply process of typing the name of the script. As you can see in the screenshot below, we are in the PythonForCyberSecurity directory (shown in blue), and typed `./HelloWorld.py`. The output of the script is shown on the next line.

Running from command line

There are several important items to point out regarding the command typed above. The first being the need for the period and forward slash (./) before the script name. This is due to a security feature of Linux that requires scripts to be referenced by their location. The period and forward slash identify the current directory as the location of the HelloWorld.py script.

Instead of the **relative path** of ./ , we could also have used the **absolute path** of the script. Since the PythonForCyberSecurity folder is in the pi user's home directory (identified by the ~ in the

prompt), the file is in the /home/pi/PythonForCyberSecurity directory. In that case, we could also call the script by typing `/home/pi/PythonForCyberSecurity/HelloWorld.py`.

The next item to highlight was the previously mentioned using **chmod +x** to mark the script as executable. This step is needed to execute the script as shown, but the script can be called in different ways. For instance, if we didn't mark the script as executable, we could still run it by typing `python3 HelloWorld.py`. This command starts the Python interpreter directly, and includes HelloWorld.py as a parameter or option.

Variables

The term variables refers to something that can change. Even if you are not familiar with the idea of a variable, if you took algebra or trigonometry in school, you have used variables in the past. You would have seen them written similarly to **x = 3 + 5** , followed by the request **Find the value of x**.

> **Don't Panic!** This book won't get close to the difficulty of algebra or trigonometry.

The easiest way to describe a variable in a computer is to imagine a box. We can take a box and give it a name such as "Frank". We then can put almost anything we want, such as marbles, into "Frank". Then, instead of asking for the specific contents (i.e., the marbles we put in it), we can ask for "Frank" instead.

If we want to change what is in the box, like coins, we remove the current contents and put something else in it. Now, instead of marbles being in "Frank", we have coins in "Frank". It is still the same box, and we still refer to it by the same name, but it has different contents.

This idea of changing the contents of a container like a box, envelope, or variable, is extremely useful when performing repetitive tasks. For instance, if we have a container called **name**, we can put a person's name in it, and then do tasks for that person like make coffee, send birthday cards, or create user accounts. When we change the contents in the container, we then repeat the same tasks with different values.

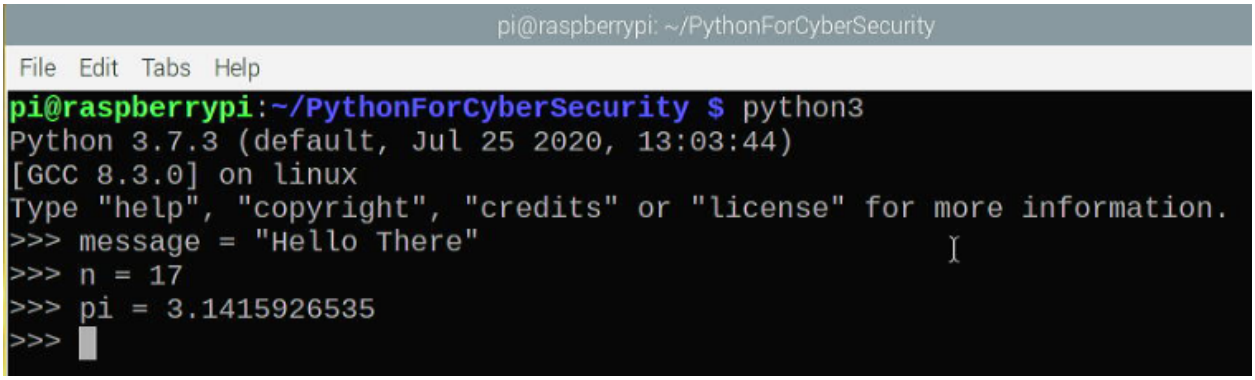
Working with Variables

Working with variables in Python is mostly simple; there are more complex variable types seen later will make it more difficult, but these are *generally* based on simple types.

We start with a name, such as `color`, and then *assign* a value to it using a single equal sign (`=`). When a single equal sign is used in Python, it assigns the value on the right of the equal sign to the variable on the left. For instance, if we were painting a car, we might assign `color = "Yellow"`

If we open Python in interactive mode, we can assign some variables as shown below.

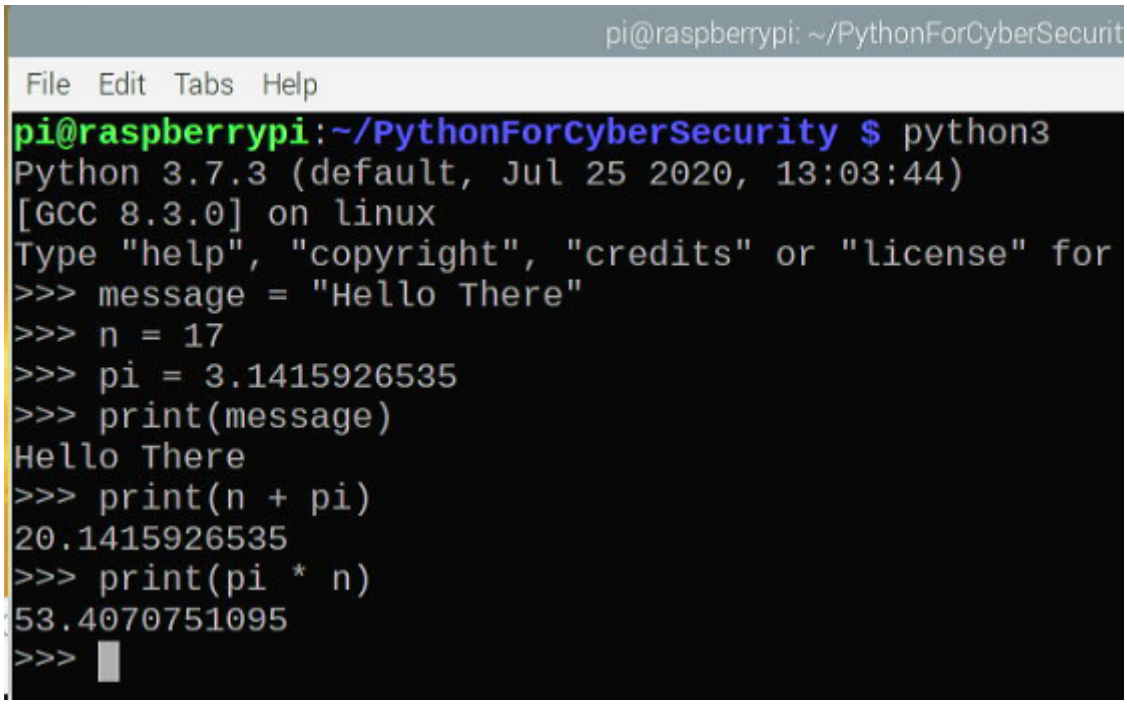
```
1  message = "Hello There"
2  n = 17
3  pi = 3.1415926535
```



Assigning variables

> **NOTE:** Variable names can consist of letters (a-z, A-Z), numbers (0-9), and underscore (_). The first character cannot be a number.

Once the variables are created, we can begin to use them as if we were using their contents. So instead of typing `print("Hello There")`, once we have assigned the message variable, we can type `print(message)`.



Printing variable contents

Variable Names

When working with Python, there several conventions, or suggestions, on how to name variables. These conventions help to encourage the "readability" focus when working with Python. More information regarding conventions can be found in the Python Style Guide[14] .

Make names meaningful

Variable names should be descriptive of what information, or type of information, they hold. For instance, names such as **student_id**, **first_name**, and **vehicle_type** help identify the content of the variable and where it would be used. Names such as **x**, **spam**, and **eggs** are not descriptive and can become confusing, especially if reused multipled times in a script.

Names should be lowercase

Wherever possible, Python variable should be all lower case. Different programmers have preferred variable naming styles such as camelCase, PascalCase, skewer-case, nocase, and even SCREAMING_SNAKE_CASE. Python suggests using what is known as snake_case, all lower case letters with underscores separating words.

Type	Example variable name
camelCase	exampleVariableName
PascalCase	ExampleVariableName
skewer-case	example-variable-name
nocase	examplevariablename
SCREAMING_SNAKE_CASE	EXAMPLE_VARIABLE_NAME
snake_case	example_variable_name

Global variables

Global variables, or variables whose values are accessible throughout the script, are often identified by including two underscores before and after the variable name. For instance, if you had a variable named **ip_address** that you wanted to be globally accessible, you would name it **__ip_address__**.

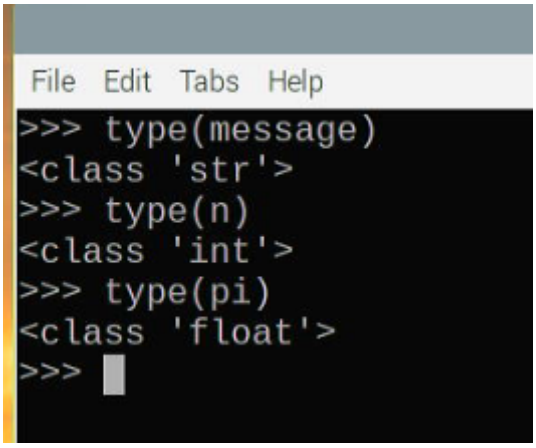
We won't be using global variables in this book, but you may see them referenced in other scripts in the future.

Variable Types

Just as there are different types of pets in people's homes, there are different types of variables in Python. It is important to recognize the different types of variables because some types work well together, while other types need special consideration to work together.

We can see more information about what our variable type is by using the `type()` built-in Python function. We can see an example of this below.

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



Viewing variable types

The three main types of variables used in Python:

- **String** (str) - can be a combination of text, numbers, and spaces, or most anything you can type on the keyboard.
- **Integer** (int) - is a simple number that doesn't include a period. Numbers such as 0, 1, 2, -5, and 42 are integers.
- **Float** (float) - is a number that includes a period. Numbers such as 3.14 and 1.234 are floats.

Our Second Python Script

Getting Ready

In Visual Studio Code, select the Explorer icon on the Activity bar (looks like 2 pieces of paper on the far left). In the Explorer side bar, browse to **start | CH02 | HelloWorld2.py**. This will open the file in the editor pane.

Another Hello World

Now that we know a bit more about variables, lets create a new Hello World script that uses a variable to return our name. Enter the following text:

```
#!/usr/bin/env python3
# A simple "Hello World" script in python with Inputs
# Created by Ed Goad, 2/3/2021

your_name = input("What is your name? ")
print("Hello {0}".format(your_name))
```


Don't forget to commit the changes to Git and GitHub

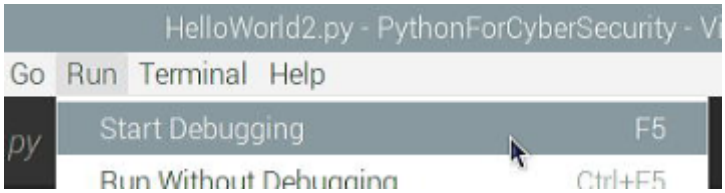
How it works

1. Starting on line **#5** on the right of the equal sign we are using the **input()** built-in Python function to prompt the user and record their input.
2. Once we have input from the user we create a variable called **your_name** and assign the value to the variable.
3. On line **#6** we then print **Hello** followed by the name provided.

   NOTE: We will discuss how the print statement works shortly.

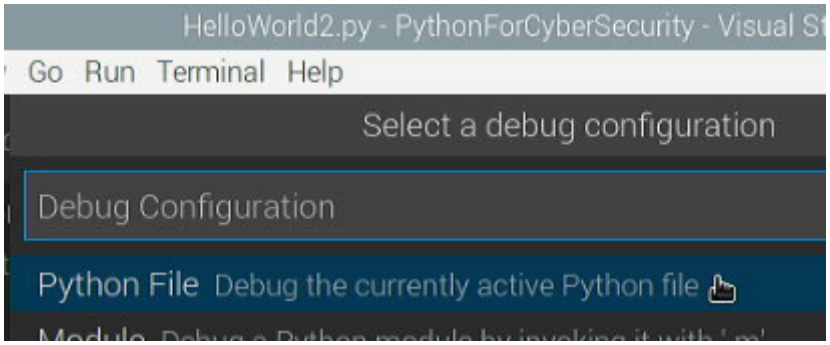
To run the script:

1. On the Visual Studio Code menu, select **Run | Start Debugging**.



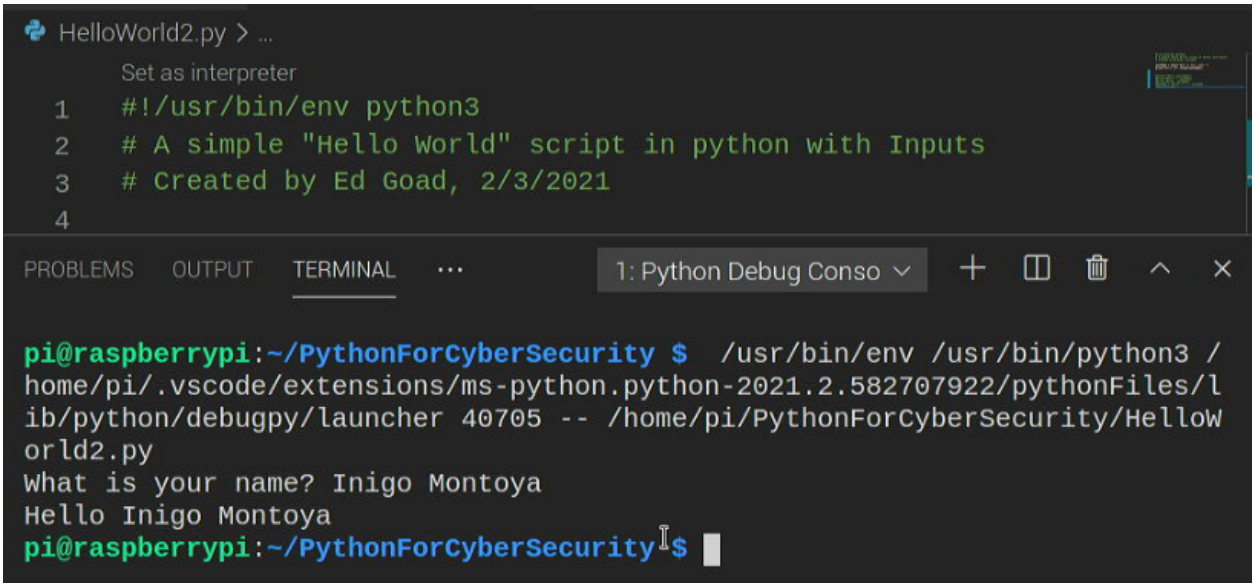
Start debugging

2. When prompted to **Select a debug configuration** choose **Python File**.



Python file

3. In the **TERMINAL** window, the script will prompt you for your name and wait for a response. When you press **Enter**, the script will finish by printing the final Hello statement.



Entering a name

There's More

Other ways to Print

In the prior example, we showed one way to use a print statement when combining text with a variable. As with all things in computers, there is always more than 1 way to complete the same task.

Below we have extended our script to **HelloWorld3.py** to highlight other common print statements. All of these forms print the same information.

```
#!/usr/bin/env python3
# A simple "Hello World" script in python with Inputs
# Created by Ed Goad, 2/3/2021

your_name = input("What is your name? ")
print("Hello {0}".format(your_name))
print(f"Hello {your_name}")
print("Hello " + your_name)
print("Hello", your_name)
message = "Hello" + " " + your_name
print(message)
```

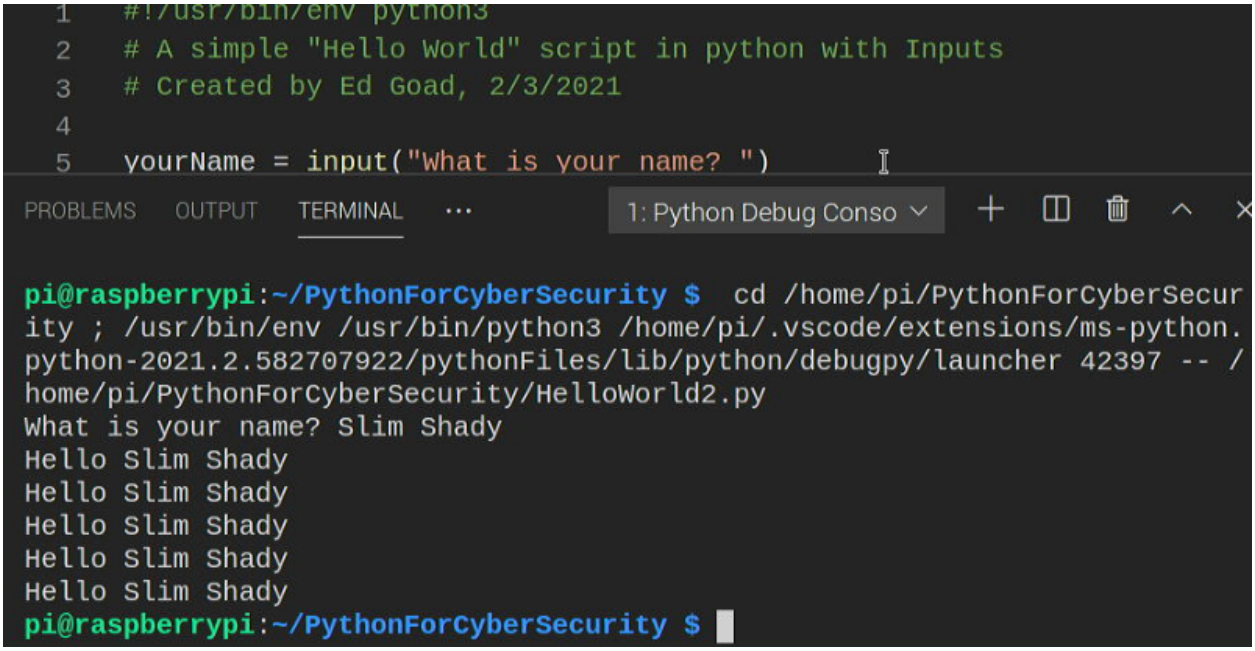
The first 2 examples on lines **#6**, **7** use Python's "string format" features. Formatting covers a large group of concepts, but here we are using it as placeholders to be filled in later. In line **#6** we create a placeholder using `{0}`, which is later filled with the contents of the variable named **your_name**. Line **#7** uses the same format option, but completes it without substitution.

Line **#8** works by "appending" the two strings together. By appending, it joins the strings "Hello " and the variable contents together. This method works well when only working with strings, but can cause problems if working with different variable types (string, integer, float, and other data types).

Line **#9** essentially is 2 different print statements on the same line. It first prints the string "Hello", and then prints the variable output after. This output may look slightly different due to spacing between the two print statements.

Line **#10** creates a new variable by joining the strings together and assigning it to the **message** variable. Once the message variable is assigned a value, it is then printed on line **#11**.

As we can see below, each of the print statements works identically.

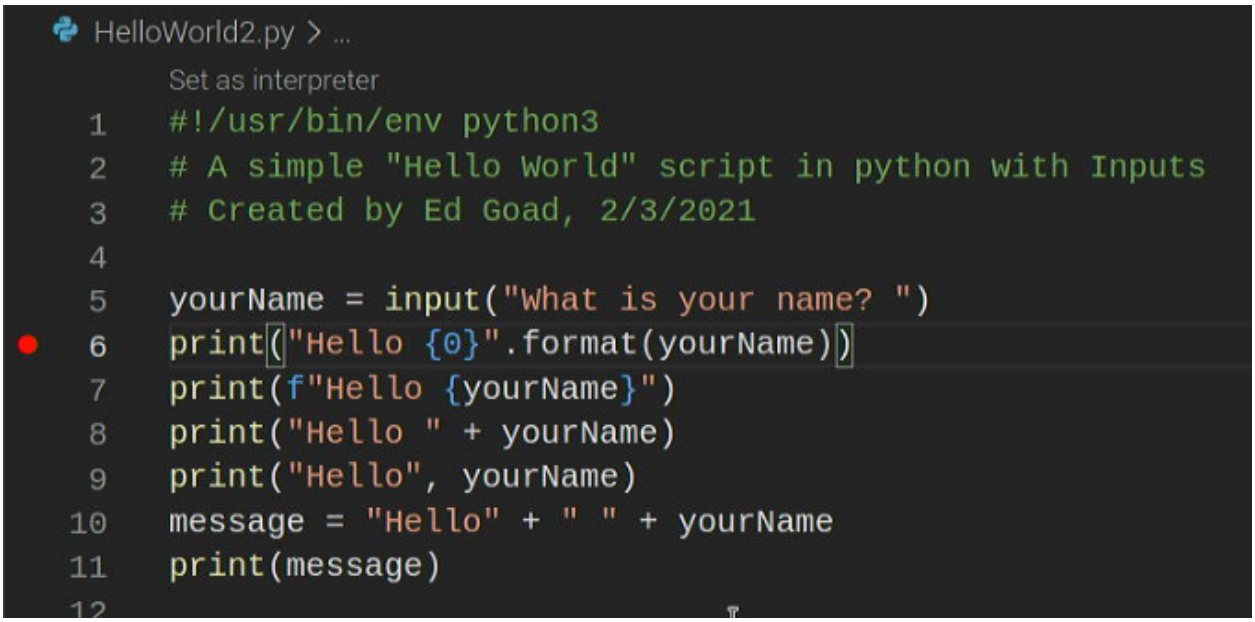


Script output

Debugging

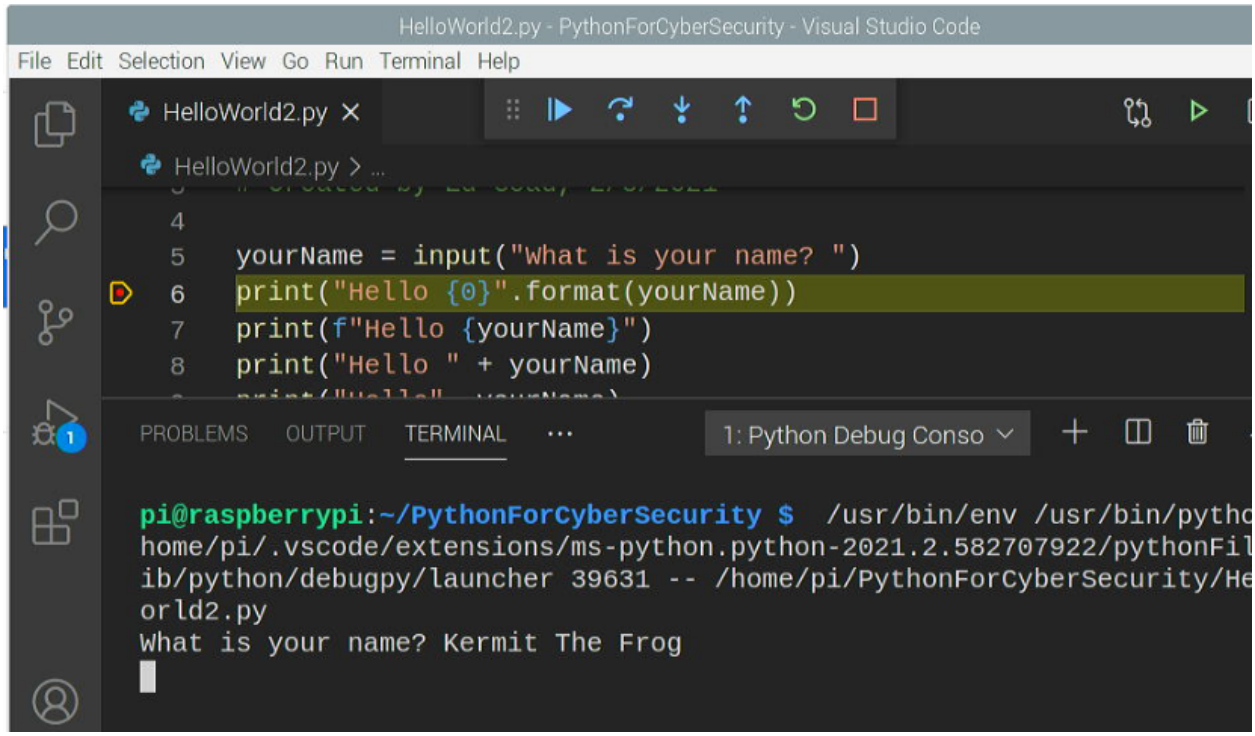
Another great benefit of using an IDE similar to Visual Studio Code is the ability to debug the code as it runs. Debugging is useful if the script is not working properly, if there are unexpected outcomes, or simply for demonstration and educational purposes. Here we will introduce the idea of debugging in Visual Studio Code.

To demonstrate this, we can continue working with the HelloWorld3.py script. In the editor, to the left of the first print statement (line #6 in my example) when you click with the mouse you will see a red dot appear. If you click again, the red dot will disappear or toggle on/off. This red dot is known as a "breakpoint".



Setting breakpoint

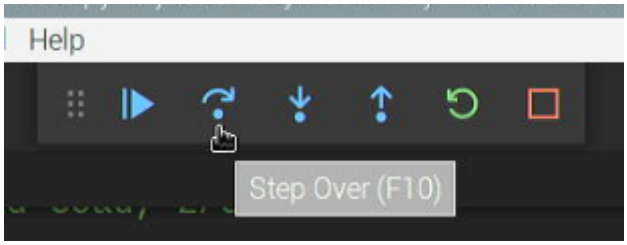
Once you have the breakpoint toggled, run the script by selecting **Run | Start Debugging**, and choosing **Python File**. The script will begin normally and prompt you for your name, and then pause.



Paused at breakpoint

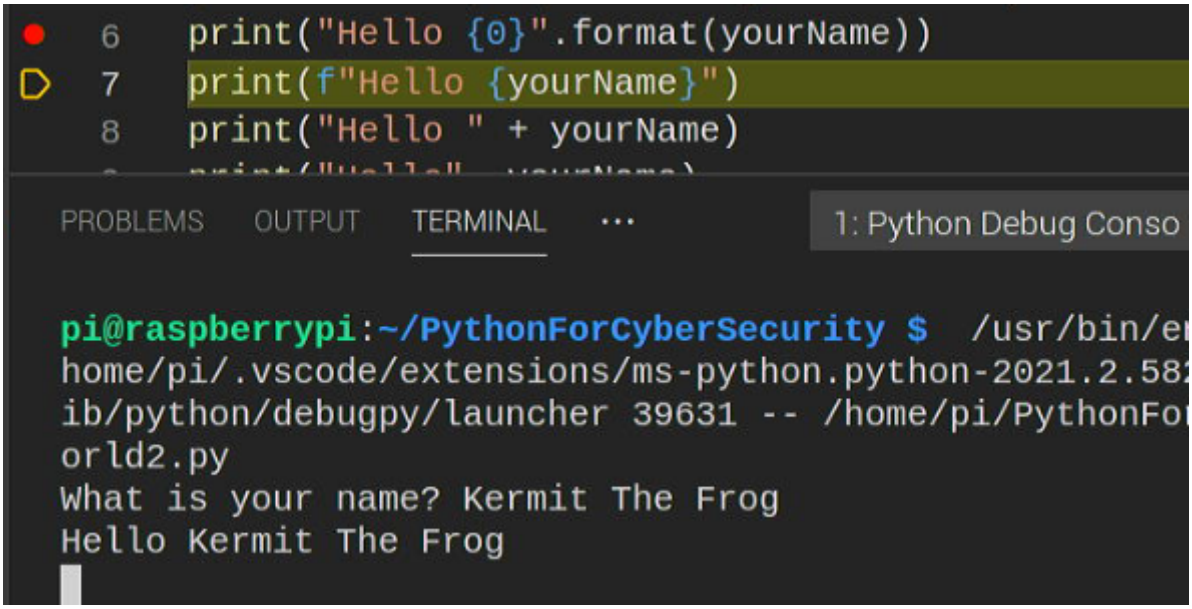
As you can see in the image above, line #6 is highlighted showing that the script is currently paused at this point. Additionally, there are several buttons at the top of Visual Studio Code, when you move your mouse over them they will show a description.

- Continue (F5)
- Step Over (F10)
- Step Into (F11)
- Step Out (Shift+F11)
- Restart (Ctrl+Shift+F5)
- Stop (Shift+F5)



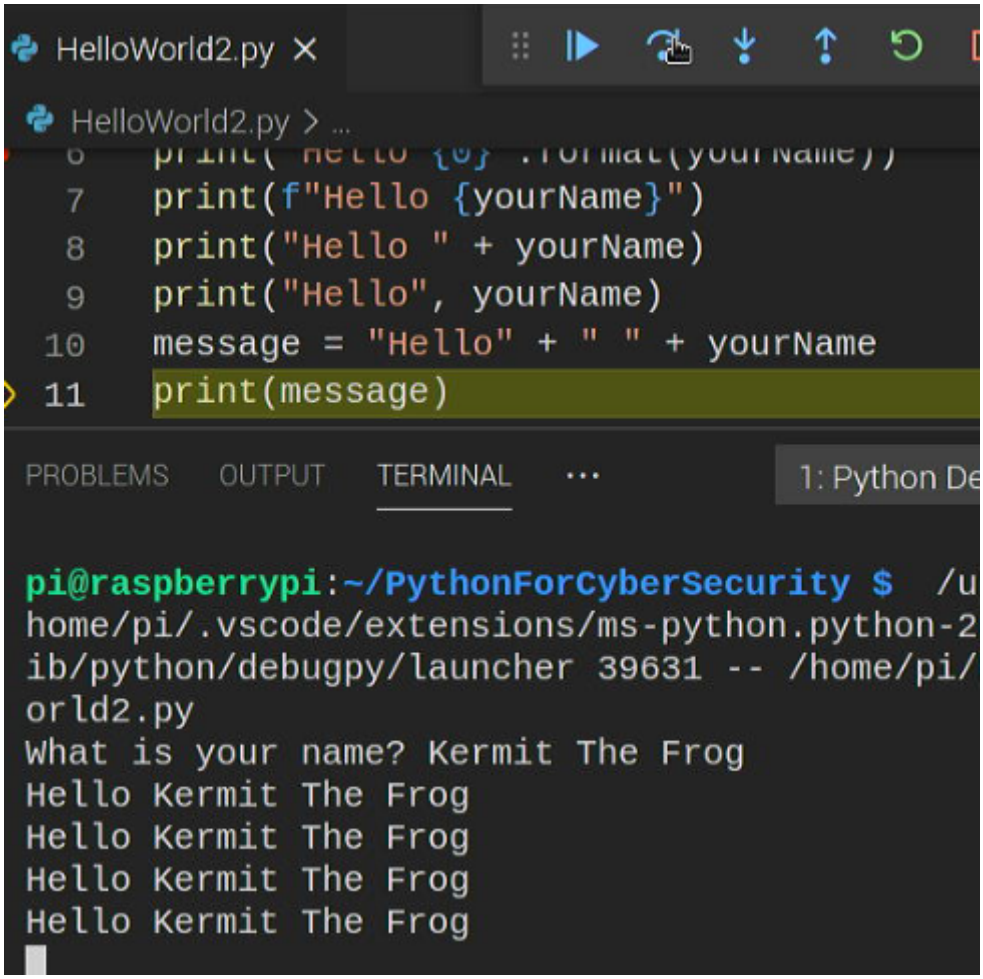
Debugging toolbar

With your mouse, click the **Step Over (F10)** button one time. When you click the button, you will see that the highlighted print statement is executed, and the output is shown in the TERMINAL window. Once the command completes, the script pauses again and the next line is highlighted.



Stepping through debugger

You can keep pressing the **Step Over (F10)** button to see each line executed one-by-one. When you get to the end of the script, the script will end, and the debugging tools will automatically be removed.



Final debugging line

One interesting aspect of the debugger is that we can view the contents of our variables while the script is running. While debugging is occuring, you can hover your mouse over any of the variables to view their current contents. This is useful when the contents of the variables may change frequently, so that you can confirm if the script is working as expected.

Simple Calculator

In most scripting and programming languages, there is an idea known as an **if/then** statement. We use these terms in every day human language, for instance we may say "**if** it is your birthday, **then** I will bake a cake." In Python, this works essentially the same way.

Getting Ready

In Visual Studio Code, select the Explorer icon on the Activity bar (looks like 2 pieces of paper on the far left). In the Explorer side bar, browse to **start | CH02 | SimpleCalculator.py**. This will open

the file in the editor pane.

How to do it

SimpleCalculator.py

```
1  #!/usr/bin/env python3
2  # A simple calculator to show math and conditionals
3  # Created by Ed Goad, 2/3/2021
4
5  # Get inputs first
6  # Note we are casting the numbers as "float", we could also do "int"
7  first_num = float(input("What is the first number: "))
8  activity = input("What activity? ( + - * / ) ")
9  second_num = float(input("What is the second number: "))
10
11 # depending on the selected activity, perform an action
12 if activity == "+":
13     print(first_num + second_num)
14 if activity == "-":
15     print(first_num - second_num)
16 if activity == "*":
17     print(first_num * second_num)
18 if activity == "/":
19     print(first_num / second_num)
```

 Don't forget to commit the changes to Git and GitHub

How it works

Lines **#7** and **#9** use the built-in Python function, `input()`, to collect the numbers used in our calculator. By default, the `input()` function returns **string** values which we can't use for math. To convert the string values into numbers, we wrap the `input()` function with the `float()` function. This converts the string into a float and it is assigned to the appropriate variables.

Line **#8** prompts the user for the activity to perform: add, subtract, multiply, or divide. This will be used to later choose what our script does.

The remaining lines are pairs of `if` and `print` statements with different values in each pair.

In the first pair of if/print statements starting on line **#12**, we have our first `if` statement. This if statement is comparing the value in **activity** to the string "**+**". If the comparison is True (i.e., activity contains "+"), then the indented lines below the if statement will be performed.

> **NOTE:** A single equal sign is an assignment. A double equal sign is an evaluation.

In line **#13** we start with indentation to identify this line to be executed as part of the if statement. In this case we only have 1 line, but multiple lines with the same indentation can be included. Here we are using a print statement to simply add the values together.

> **NOTE:** Indentation must be consistent throughout the script. Inconsistent indentation and mixed indentation types (spaces vs tabs) will cause the script to fail.

There's more

This simplified script makes use of 4 separate **if** statements to evaluate the various options. While this works, it is not the most efficient method, and in some cases can cause problems. More advanced Python scripts will use **if/elif/else** statements to combine these options.

Chapter 3: Ping

In this chapter we will cover the following:

- Importing modules in Python
- Creating scripts to ping devices in a network
- Python Conditionals
- Extending our script to work on multiple Operating Systems
- Python Loops
- Using loops to ping multiple hosts
- Python Functions
- Ping function

Introduction

This chapter will dig into more Python fundamentals while beginning our first cybersecurity related script. Here we will discuss extending our capabilities by importing modules, using conditionals, loops, functions, and pinging devices on our network.

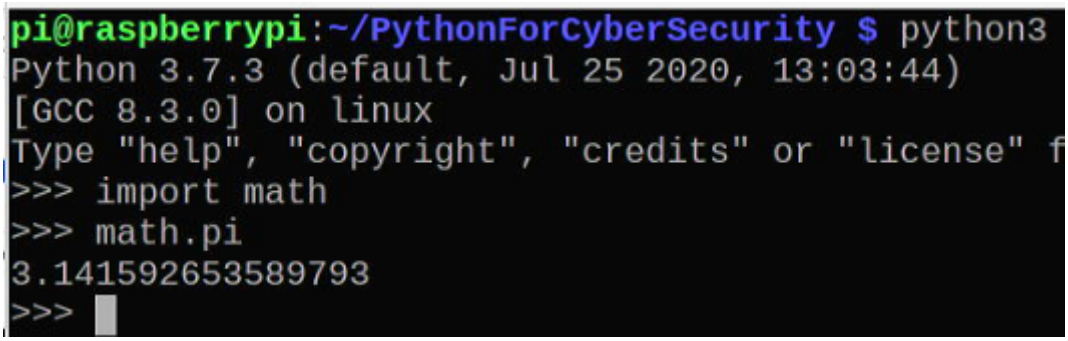
Importing modules into Python

Python, by default, has several capabilities built-in. Using these capabilities, many simple and complex tasks can be completed. However, there are times when there are more specific tasks needed, or you wish to utilize work that has already been completed. For this, we can import **modules** into Python.

One module / library we may want to import is the math library. As it's name suggests, the math library has several math related features and capabilities.

There are several different formats we can use when importing modules. The easiest is to simply **import** the module, and then use it by referencing the module each time we use it. For example, we can import and begin using the math library:

```
1 import math
2 math.pi
```

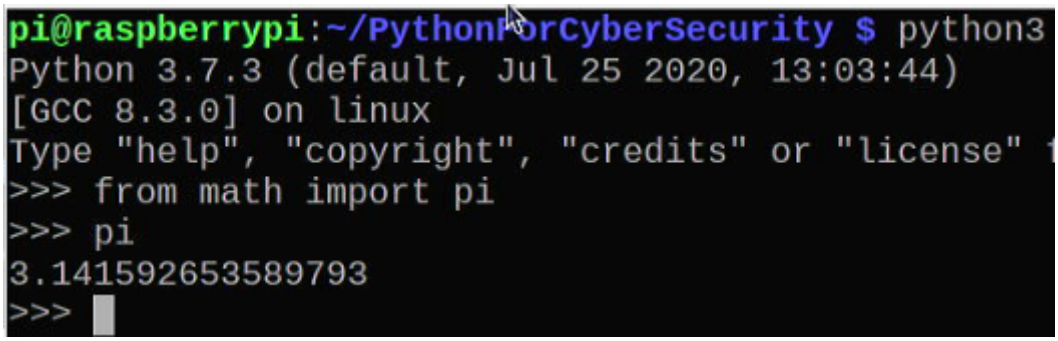


Importing math

Here we can see that by calling `math.pi` the math module prints the value of **pi** to 15 decimal points.

Sometimes, we don't need to import the whole module, but only a few features of the module. This can make our script smaller, easier to write, and faster. To import just a few features, we change our **import** statement:

```
1 from math import pi
2 pi
```

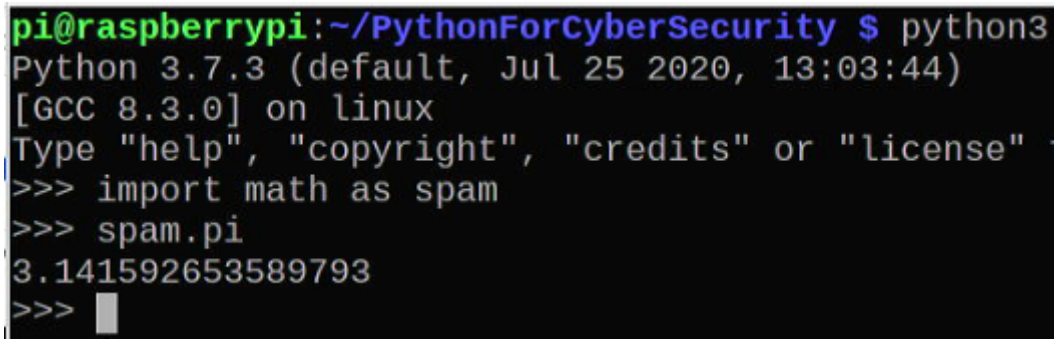


Importing pi from math

As we can see, we have the same result as before and the value of **pi** is returned. However, instead of needing to type the fully qualified name of `math.pi`, we can reference it as simply `pi`.

We can extend the prior 2 examples by providing different names for the modules as we import them. This can be useful when we there are multiple modules for us to choose from, but allows us to reference them under a common name. For instance, there are several math modules to choose from (math, numpy, scipy, and so on…), which may or may not be available on our system. To import and use one of the libraries with a common name of `spam`, we change the import statement:

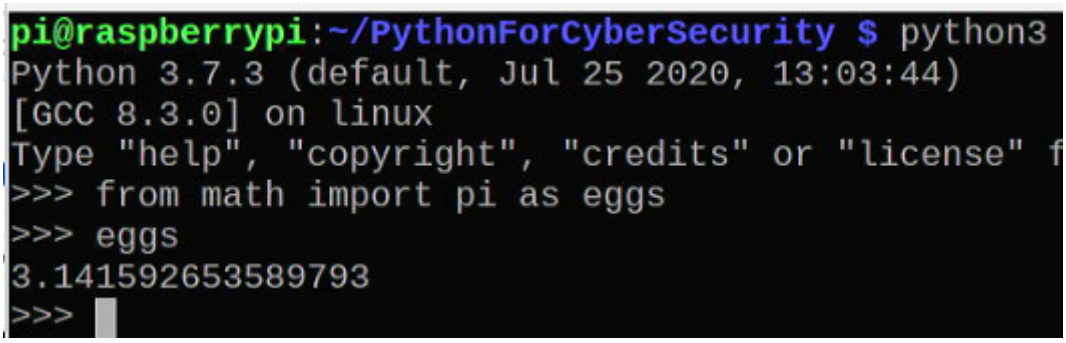
```
1 import math as spam
2 spam.pi
```



Importing math as spam

Additionally, we can import specific portions of the module, again with a reference/common name:

```
from math import pi as eggs
eggs
```



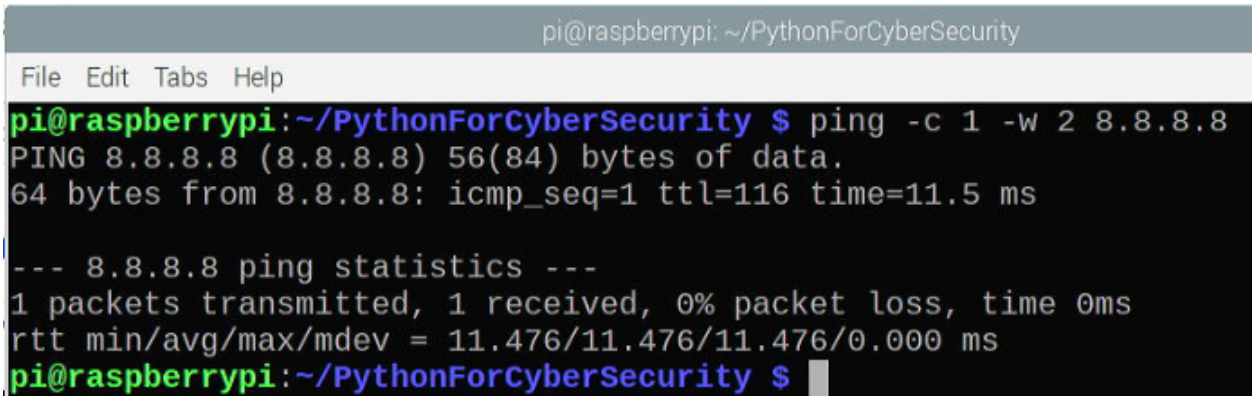
Importing math.pi as eggs

Pinging devices in a network

As with all things related to computers, there are several ways to perform the same task. One example is the process of **pinging** an address. If you are unfamiliar with ping, it is a method of sending a message to a remote computer, and then having them respond back. This is useful for security and network engineers to find out which resources are online.

On the Raspberry Pi, open a terminal window and type `ping -h` and review the returned options. Of particular interest to us for this example are the **-c** and **-w** options that will make automation much easier. The first option (**-c**) allows us to specify the number of packets, or attempts, to send to the target. The second option (**-w**) allows us to specify how long we should wait for a response.

We can combine these options together and use the command `ping -c 1 -w 2 8.8.8.8` as a test.



Output of ping command

As you can see from the screenshot above, the ping completed successfully.

NOTE: There may by several reasons why a ping would fail. If you attempted the same command and it didn't succeed, it is possible the network administrator is blocking ping for your network. Check for typos and try to ping other destinations such as `ping -c 1 -w 2 127.0.0.1`.

Pinging in Python

Getting ready

On the Raspberry Pi, open a terminal window and type `python3` to start Python in interactive mode.

How to do it

We begin by importing 2 Python modules for our ping command. The `os` module allows us to send commands to the underlying operating system, this will perform our **ping**. The `platform` module will be used later to customize our commands based on the operating system.

```
1  import platform
2  import os
```

Once the modules are imported, we can use the `os.system()` function to pass commands to the operating system.

```
1  os.system("ping -c 1 -w 2 8.8.8.8")
```

```
pi@raspberrypi:~/PythonForCyberSecurity $ python3
Python 3.7.3 (default, Jul 25 2020, 13:03:44)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more in
>>> import platform
>>> import os
>>> os.system("ping -c 1 -w 2 8.8.8.8")
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=116 time=15.3 ms

--- 8.8.8.8 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 15.325/15.325/15.325/0.000 ms
0
>>>
```

Output of ping command

Once we press enter, we will see the ping command execute in the same manner as when we typed the command on the terminal. This is our first glance at "automation" - i.e. having a script that runs commands for us.

In our case, we don't want all of the output provided. The **icmp_seq**, **ttl**, and **time** are all useful, but if we desire a simple up/down result, uneccessary. Instead, we simply want a **success** or **failure** result. This has been provided for us by the number on the final line, a **0** in the screenshot above.

To remove the useful, but currently unnecessary, information we can redirect it to **/dev/null**. This is a special location in Linux that acts as a garbage bin. We update our command to include the redirection as show:

```
1 os.system("ping -c 1 -w 2 8.8.8.8 > /dev/null 2>&1")
```

```
>>> os.system("ping -c 1 -w 2 8.8.8.8 > /dev/null 2>&1")
0
>>>
```

Ping succeeding

The final number returned by the command is the "exit code" which reports how the command finished. An exit code of 0 means success, in our case a successful ping. Any other number means the ping failed. We can demonstrate a failure by pinging an address that likely doesn't exist.

```
1 os.system("ping -c 1 -w 2 10.207.53.199 > /dev/null 2>&1")
```

```
>>> os.system("ping -c 1 -w 2 10.207.53.199 > /dev/null 2>&1")
256
>>>
```

Ping failing

There's more

In this example, we are looking for a simple success/failure status from the ping command. As such, we are interested in the value of the exit code being either a 0 (success), or non-zero (failure). However, more information can be learned by the different exit codes returned. Identifying the exit code and the cause for failure, can help build a more robust script.

See also

More information about the os module can be found on the Python documentation[15] site.

First Ping

In our first ping script, we will duplicate the activity we performed in interactive mode. Once we have this working, we can continue adding to it to make it more useful.

Getting Ready

In Visual Studio Code, browse to **start/CH03/pinger1.py**. This will open the file in the editor pane.

How to do it

```
1  #!/usr/bin/env python3
2  # First example of pinging from Python
3  # By Ed Goad
4  # 2/27/2021
5
6  # import necessary Python modules
7  import platform
8  import os
9
10 # Assign IP to ping to a variable
11 ip = "127.0.0.1"
```

[15] https://docs.python.org/3/library/os.html

```
12 # Build our ping command
13 ping_cmd = f"ping -c 1 -w 2 {ip} > /dev/null 2>&1"
14 # Execute command and capture exit code
15 exit_code = os.system(ping_cmd)
16 # Print results to console
17 print(exit_code)
```

Note the excessive use of comments in the script. While maybe more than necessary, it is helpful to get into a habit of documenting your work so it is easily understandable in the future when you review it.

 Don't forget to commit the changes to Git and GitHub

How it works

Line **#1** is the shebang, identifying this as a Python script.

The next few lines are comments identifying the script and what it does.

Lines **#7** and **#8** begin importing the Python modules needed for the script to function. These are the same modules we imported in interactive mode.

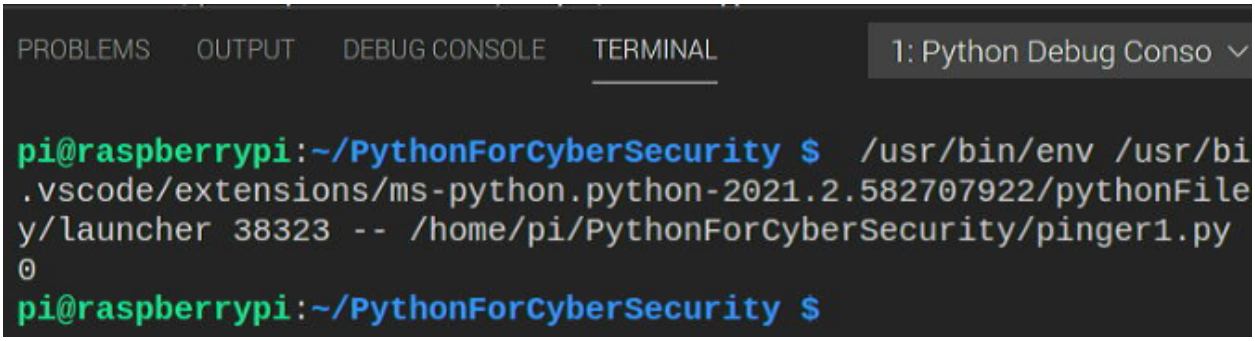
In line **#11** we assign our IP address to a variable named **ip**. The use of a variable here can be helpful so that if we wish to change it later, we only have to change 1 line of our code.

> **NOTE**: We are using the IP address of 127.0.01, which is known as the *loopback* address, which should succeed on almost all devices and networks.

Line **#13** crafts our ping command. Notice how we use the same command used from the command line, and we use substitution to add the IP address into the command. While there are more efficient ways to prepare our ping command, I have found this method the best for learning about how scripting works.

Line **#15** executes our ping command, stored in the **ping_cmd** variable, and saves the results in the **exit_code** variable.

Line **#17** prints the result of the ping to the console. You can see the result of 0 provided in the screenshot below.



Successful output of script

There's more

Using Visual Studio Code, put a breakpoint on line **#11** where the IP address is assigned. Start the debugging process and step through the script 1 line at a time watching the variables change. This can be especially interesting to watch **ping_cmd** as you pass line **#13**.

This script can be easily expanded to prompt the user for the IP address prior to performing the ping. This is left out here for simplicity and easy development.

Python Conditionals

In scripting, there is a concept known as a "conditional". As the name suggests, by evaluating various conditions, we can have our script perform different tasks. The simplest conditional to understand is the **if**...**then**. For example: when I walk into a room **if** the lights are off, **then** turn on the lights.

if...then

Below is an example of a simple **if**...**then** statement in Python:

```
1  if true_statement_here:
2      execute_this_code
```

> **NOTE**: This is known as pseudo-code, or code that doesn't work, but is used to describe an idea.

We start with the **if** statement, which is followed by some sort of evaluation, and then a colon (:). The evaluation returns either `True` or `False`. If the statement is true, then the **indented lines** following the colon are executed. If the statement is false, then the indented lines are skipped.

Below are 3 additional pseudo-code examples of the **if**...**then** conditional.

```
if lights_off:
    turn_on_lights
if not lights_on:
    turn_on_lights
if lights_on == False:
    turn_on_lights
```

The first example evaluates the variable **lights_off**. If the lights are off, then the value of **lights_off** would be `True`. Because the value of the variable is the only part being evaluated, and it is true, the command **turn_on_lights** is executed.

The second example evaluates the variable **lights_on**, which should be the opposite of the prior variable. In this case we are evaluating **not lights_on**, which tells Python to look at **lights_on** and then return the opposite. Therefore, if the lights are off, **lights_on** would be `False`, but the opposite would be `True`.

In the third example we are evaluating **lights_on** again, but this time determining if it is `False`. If the lights are off, then the **lights_on** variable is `False`. If we replace the variable with the value, we are then evaluating if `False == False`. Since false does equal false, it returns `True`.

if...else

In addition to the **if..then** statement shown above, we can add an **else** option. The idea being that if the initial comparison isn't true, then we will do a different task.

For instance, if we wanted to toggle the lights on/off, we could use the following statement:

```
if lights_off:
    turn_on_lights
else:
    turn_off_lights
```

The first line in the example checks the value of **lights_off**. If the lights are **off**, then the **turn_on_lights** command will be called. However, if the lights are on, then the **turn_off_lights** command will be called.

See also

There are many additional ways to use conditionals in Python. We will briefly cover these as we see them, but more information can be found online. One example is from w3schools.com[16]

[16] https://www.w3schools.com/python/python_conditions.asp

Second Ping

Now that we know a bit more about conditionals, lets use them to extend our pinger script. Specifically, we will use conditionals to customize the ping command based on the operating system we are using.

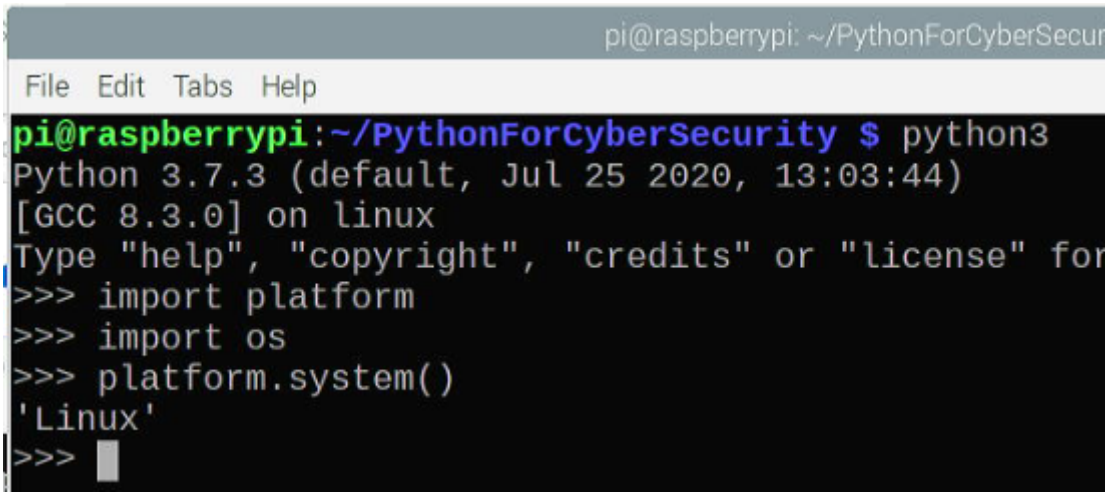
First - How to find our operating system

To find which operating system we are running the script on, we can use the following example code in Python.

```
1 import platform
2 import os
3 platform.system()
```

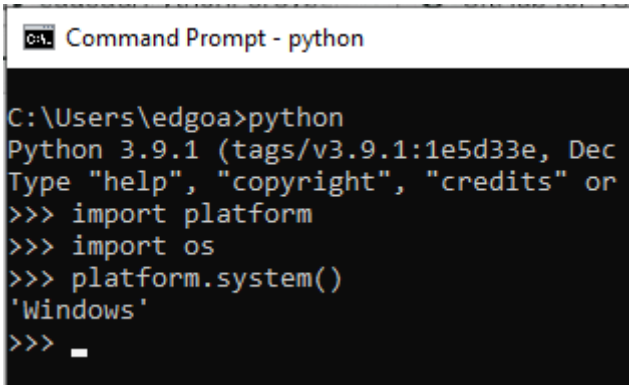
As we can see in the following screenshots, different operating systems return different values for `platform.system()`.

On Linux



platform.system() on Linux

On Windows



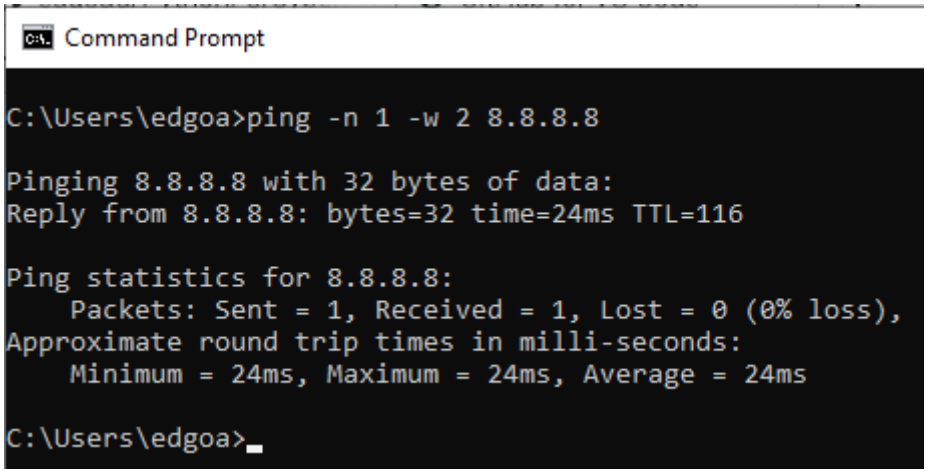
platform.system() on Windows

Second - Differences in ping

We have already seen the options for the ping command on Linux, now lets view the options for ping on Windows. On a Windows computer, open a command prompt (**Start Menu | Windows System | Command Prompt)**. Once the command prompt is open, type `ping /?` to view the options available to us.

Of particular interest to us are the **-n** and **-w** options, which give us the count and timeout. These are the equivalent of the **-c** and **-w** options from Linux.

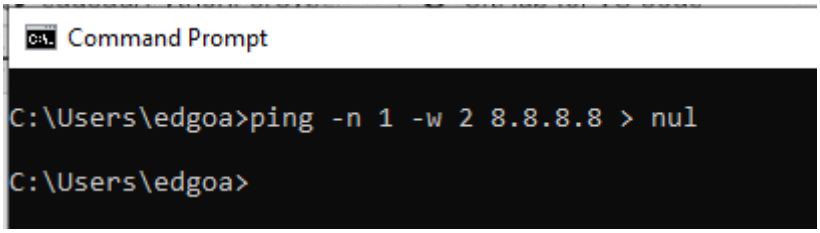
We can combine these options together and use the command `ping -n 1 -w 2 8.8.8.8` as a test.



ping command on Windows

The results are similar to that from our Linux command. We will want to further optimize the command to remove the unnecessary information like **time** and **TTL**. To do this, we redirect the output to **nul**, the Windows equivalent of a garbage bin.

```
ping -n 1 -w 2 8.8.8.8 > nul
```



Redirecting to nul

> **NOTE:** Even though it doesn't show, the command did return an exit code. We will use the exit code in the same manner as the Linux exit code to determine success or failure.

Getting started

In Visual Studio Code, browse to **start/CH03/pinger2.py**. This will open the file in the editor pane.

How to do it

```
#!/usr/bin/env python3
# Second example of pinging from Python
# By Ed Goad
# 2/27/2021

# import necessary Python modules
import platform
import os

# Assign IP to ping to a variable
ip = "127.0.0.1"
# Determine the currrent OS
currrent_os = platform.system().lower()
if currrent_os == "windows":
    # Build our ping command for Windows
    ping_cmd = f"ping -n 1 -w 2 {ip} > nul"
else:
    # Build our ping command for other OSs
    ping_cmd = f"ping -c 1 -w 2 {ip} > /dev/null 2>&1"

# Execute command and capture exit code
exit_code = os.system(ping_cmd)
# Print results to console
print(exit_code)
```

 Don't forget to commit the changes to Git and GitHub

How it works

After the shebang and several comments, the script begins on lines **#7**, **8** by importing the necessary Python modules.

At line **#13** we use the `platform.system()` command to determine the current operating system and save the value in the **current_os** variable. One thing to note about this line is the `.lower()` at the end. This makes the OS name all lower-case, meaning "Windows" becomes "windows", and "Linux" becomes "linux". This (extra) step can make our evaluations easier in the future.

On line **#14** we perform our first evaluation to see if **current_os** is equal to "windows". If we are running on a Windows computer, then this evaluation would be true, and the **ping_cmd** crafted on line **#16** will be used.

> **NOTE:** There are multiple operating systems in use today (Windows, Linux, IOS, Android, Unix, and so on…). Many of those OSs have a basis in Linux and therefore would likely use the same ping command as the Linux example. Because Windows is the exception to this, we look for that OS, and assume the rest will use the Linux command format.

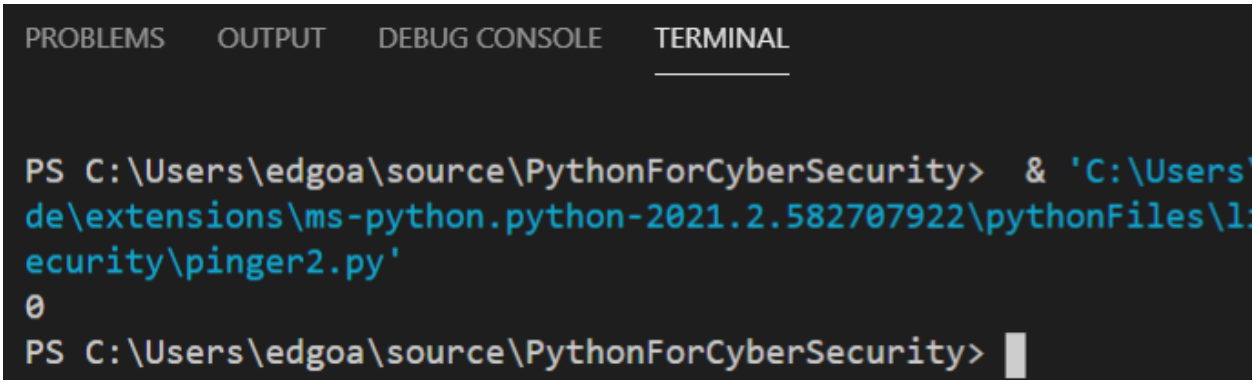
On line **#17**, if the operating system is not Windows (i.e., line **#14** evaluated as false), we prepare a different **ping_cmd**.

The remaining lines remain the same from **pinger1.py** , even though the line numbers have changed.

Here we can see the Python script executed from the Raspberry Pi:

Script result on Linux

The same Python script executed from Windows:



Script result on Windows

 Use the debugger to walk through the script for more details on how it works.

Introduction to Python Loops

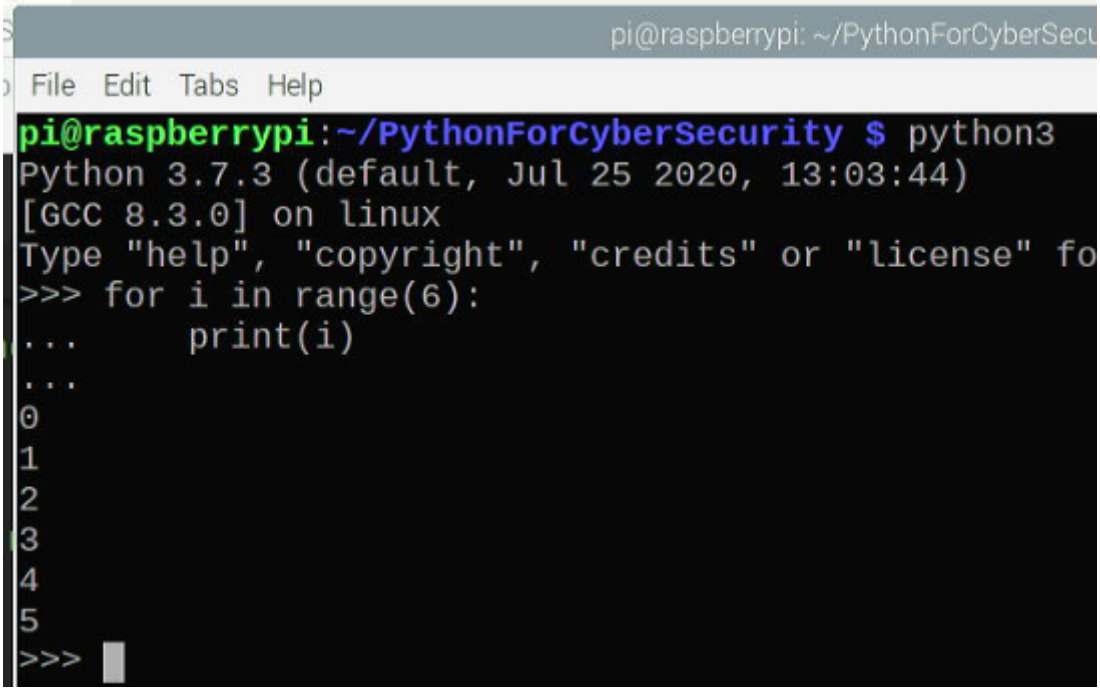
In scripting, a loop allows the computer to perform the same task over and over. These loops can be based on names in an employee roster, based on a number of cycles, or even never ending. One example of where a loop would be helpful is sending emails to a large number of people. The computer receives the list of people, sends an email to the first person on the list, then the second, and so on until the list is finished.

How to do it

From the Linux terminal, type `python3` to open the interactive Python. In the interactive Python, type the following and then press enter twice.

```
1  for i in range(6):
2      print(i)
```

NOTE: The `print(i)` line is indented by pressing the space key several times.



Example of range() command

How it works

Our interactive command starts with the `range(6)` function on the first line. This function starts counting from 0 until the number specified. Therefore, this gives us a list of the numbers 0, 1, 2, 3, 4, and 5.

The `for` statement at the beginning defines the loop. This takes the first item in the list returned by range(), the number **0**, and assigns it to the variable **i**. Once the variable is assigned, the indented command(s) are executed and then control is returned to the **for** statement. When control is returned, the statement takes the next item in the list returned by range, the number **1**, assigns it to **i** and repeats.

When there are no more items left in the list provided by range, the **for** loop exits.

See also

https://www.w3schools.com/python/python_for_loops.asp[17]

https://www.w3schools.com/python/ref_func_range.asp[18]

https://www.geeksforgeeks.org/python-range-function/[19]

[17] https://www.w3schools.com/python/python_for_loops.asp
[18] https://www.w3schools.com/python/ref_func_range.asp
[19] https://www.geeksforgeeks.org/python-range-function/

Third Ping

So far, our pinger scripts have only pinged 1 IP address. Using what we learned about loops in Python, we can extend our script to ping an entire network of addresses and return the status of all of them. Hackers do this to quickly find available targets to attack, and system administrators do this to find available addresses to use.

Getting ready

In this script, we will ping an entire Class C (/24) network. In a Class C network, the first three octets, or numbers, remain the same and the fourth octet ranges between 1 and 254. This means for the 192.168.0.0 Class C network, the IP address will range from 192.168.0.1 - 192.168.0.254.

> **NOTE**: I am using the 192.168.0.0/24 network because it matches **my** test environment. You may wish to use a different range to match your environment.

In Visual Studio Code, browse to **start/CH03/pinger3.py** and open it for editing.

How to do it

```
#!/usr/bin/env python3
# Third example of pinging from Python
# By Ed Goad
# 2/27/2021

# import necessary Python modules
import platform
import os

# Define the prefix to begin pinging
ip_prefix = "192.168.0."
# Determine the currrent OS
currrent_os = platform.system().lower()
# Loop from 0 - 254
for final_octet in range(254):
    # Assign IP to ping to a variable
    # Adding 1 to final_octet because loop starts at 0
    ip = ip_prefix + str(final_octet + 1)
    if currrent_os == "windows":
        # Build our ping command for Windows
        ping_cmd = f"ping -n 1 -w 2 {ip} > nul"
```

```
22      else:
23          # Build our ping command for other OSs
24          ping_cmd = f"ping -c 1 -w 2 {ip} > /dev/null 2>&1"
25
26      # Execute command and capture exit code
27      exit_code = os.system(ping_cmd)
28      # Print results to console only if successful
29      if exit_code == 0:
30          print("{0} is online".format(ip))
```

 Don't forget to commit the changes to Git and GitHub

How it works

The first few lines are the same between pinger2.py and pinger3.py, except for the updated comments.

Line **#11** is the first change we see. Here, we are identifying the first three octets (numbers) of the IP address, which in my case is my test network of **192.168.0.** and assigning it to the variable **ip_prefix**. Note the trailing period (.) in the **ip_prefix**. This is included here so that adding the final octet easier.

On line **#15** we start our `for` loop. We define a variable named **final_octet** to store values in, and use the `range(254)` function to provide a list of numbers from 0 - 253. For each pass of the loop, the next number from range is assigned to **final_octet**.

> **NOTE:** Remember `range()` starts counting at 0 and ends before the number provided.

In line **#18**, we append / join the **ip_prefix** and **final_octet** values together into the variable **ip**. There are 2 items of note here: First, we are adding 1 to the value of **final_octet**. We do this because range will provide 0 - 253, but we want 1 - 254; adding 1 to the **final_octet** will provide us with the correct values.

Secondly, **ip_prefix** is a variable of type **string**, however, **final_octet** is a variable of type **integer**, and we cannot directly join a string with an integer. To resolve this, we use the `str()` built-in Python function to convert **final_octet** into a type of **string**. We then join the strings together using the plus (**+**) sign.

> **Note:** The **if**...**else** statement is still indented when compared to the remaining code in the loop. Even though the entire code was indented to support the loop, the code inside the **if**...**else** statement still needs additional indenting to work properly.

```
pi@raspberrypi:~/PythonForCyberSecurity $  /usr/bin/env /
b/python/debugpy/launcher 46135 -- /home/pi/PythonForCybe
192.168.0.1 is online
192.168.0.10 is online
192.168.0.51 is online
```

Output of script


Use the debugger to walk through the script for more details on how it works.

There's more

Here, we are using a Class C IP range for simplicity. This script could be extended to scan networks of all sizes, but that is beyond the scope of this project.

Python Functions

A function is a block of code that can be called by name. Functions are frequently used to minimize duplication of code, simplifying scripts, and improve readability.

In Python, all functions start with the keyword `def`, meaning to **def**ine the function. On the same line is the function name, a pair of parentheses, and a colon. In the same manner of conditionals and loops, any indented code after the colon is considered part of the function.

Simple function

```
1  def say_hello():
2      print("Hello World")
3
4  say_hello()
```

Above is an example of a simple function named **say_hello**. This function only contains 1 command, which is a print statement. The function is called by its name, followed by parentheses, `say_hello()`.

If we then run `say_hello()` multiple times, we will see the contents of the function are repeated.

Parameters

Frequently it is necessary to send information into a function to make it more flexible and capable, this is known as a parameter. To include parameters in the function definition, we add variables inside the parentheses. These variables can then be used inside the function.

```
1  def say_hello(user_name):
2      print("Hello {0}".format(user_name))
3
4  say_hello("Ed")
```

Here we can see our extension of the **say_hello()** function now expects a value when called. When the function is called, the value to pass into the function is put inside the parentheses; in this case my name. The function assigns that value to the **user_name** variable.

If we repeat the call the the `say_hello()` function with different names, we can see the function continues to run with the new information added each time. While only 1 value/variable is shown in this example, multiple values and parameters can be passed to functions.

Returning values

In addition to passing values *into* functions, it is frequently needed to pass values *out of* functions. To get values out of a function, we use the **return** keyword.

```
1  def say_hello(user_name):
2      print("Hello {0}".format(user_name))
3      return True
4
5  if say_hello("Ed"):
6      print("Successfully said Hello")
```

Here we can see at the end of the function on line **#3** we added a **return** statement with the value of **True**. This return statement does 2 tasks: 1) it ends the execution of the function, and 2) it returns a value to location that called it.

Ending the function means that if there were more commands in the function, they wouldn't be executed. The return statement stops any additional execution of the function, and returns back to the location that called the function.

When the say_hello function finishes, it returns `True` to the **if** statement that called it on line **#5**. Because the statement is now **true**, the final print command is executed.

There's more

There is an idea known as variable scope that refers to where a variable exists and where it doesn't. When a variable is created inside a function, it only exists while that function is being executed. You can't access it outside of the function.

See also

https://www.w3schools.com/python/python_functions.asp[20]

https://www.tutorialspoint.com/python/python_functions.htm[21]

Fourth Ping

To make our ping script more useable for future changes, we can move the "pinging" portion into a function. The new function will take an IP address as a parameter, and return a success/failure message.

Creating a function

To prepare for creating our "ping" function, there are a few items we must identify before we begin:

1. What we want the function to do.
2. What existing code will be moved in to the function.
3. What parameters will be passed into the function.
4. What values will be returned from the function.

What the function will do

The goal of this function will be to ping a target, and return the status of the target. In order to separate the "ping" code from the "loop" code, this function should only ping 1 address each time it is called.

Code to move

Looking at our prior incarnations of the pinger script, we will want to move the code that does the actual pinging. This includes where the **ping_cmd** is constructed, and where the ping is executed.

Parameters to pass

Since the function will perform 1 ping at a time, we will need a parameter that holds the IP address to ping.

Values to return

So far, our script has been fairly simple in that all we want to know is if the IP is "up". In our prior versions of this script, we identified this by the value of **exit_code** being either a `0` or something else. Since this is the result we are after, we can simply use `return exit_code`.

[20] https://www.w3schools.com/python/python_functions.asp
[21] https://www.tutorialspoint.com/python/python_functions.htm

Getting ready

In Visual Studio Code, browse to **start/CH03/pinger4.py**. This will open the file in the editor pane.

How to do it

```
#!/usr/bin/env python3
# Fourth example of pinging from Python
# By Ed Goad
# 2/27/2021

# import necessary Python modules
import platform
import os

def ping_host(ip):
    # Determine the currrent OS
    currrent_os = platform.system().lower()
    if currrent_os == "windows":
        # Build our ping command for Windows
        ping_cmd = f"ping -n 1 -w 2 {ip} > nul"
    else:
        # Build our ping command for other OSs
        ping_cmd = f"ping -c 1 -w 2 {ip} > /dev/null 2>&1"
    # Execute command and capture exit code
    exit_code = os.system(ping_cmd)
    return exit_code

# Define the prefix to begin pinging
ip_prefix = "192.168.0."

# Loop from 0 - 254
for final_octet in range(254):
    # Assign IP to ping to a variable
    # Adding 1 to final_octet because loop starts at 0
    ip = ip_prefix + str(final_octet + 1)

    # Call ping_host function and capture the return value
    exit_code = ping_host(ip)

    # Print results to console only if successful
    if exit_code == 0:
        print("{0} is online".format(ip))
```

 Don't forget to commit the changes to Git and GitHub

How it works

This script **pinger4.py** is the same as **pinger3.py**, meaning it has the same commands and logic, but part of the script has been moved to a function called **ping_host**.

Function ping_host

Starting on line **#10**, we **def**ine our function named **ping_host**, and expect a value to be passed to it. The following lines, **#11 - #21** are all indented under the **def**, making them part of the function.

Comparing pinger3.py to pinger4.py, we can see the lines in the function were simply moved between the versions. The table below highlights the lines that moved between versions.

pinger3.py	pinger4.py
12, 13	11, 12
19 - 27	13 - 20

The final line in the function is a **return** statement. This is a special command that tells the function to end and return a value back to where it was called from with a result.

Loop

Other than the lines that were moved into the function, the only change in the loop is shown on line **#33**. This line calls the **ping_host** function and passes the IP address into it. When the value is returned from the function, it is assigned to **exit_code** and then printed.

```
pi@raspberrypi:~/PythonForCyberSecurity $  /usr/bin/env /usr/
b/python/debugpy/launcher 33011 -- /home/pi/PythonForCyberSec
192.168.0.1 is online
192.168.0.10 is online
192.168.0.51 is online
192.168.0.100 is online
192.168.0.118 is online
```

Output of script

 Use the debugger to walk through the script for more details on how it works.

There's more

This script can obviously be expanded to include many additional features and capabilities. For instance, this script currently pings a statically assigned Class C network. This could be expanded

to prompt the user for the range of IP addresses to ping. This enables the script to be more portable and work in multiple different networks that use different ranges and network sizes.

Chapter 4: More Ping

In this chapter we will cover the following:

- Reading Files
- Getting Ping targets from a file
- Writing Files
- Writing Ping results to a file
- Extending Ping with a Port Scan

Introduction

In this chapter we will continue to extend our Ping script to include more capabilities. First, we will start off discussing how to read files in Python, and then use that capability to get a list of IP addresses to ping. Then, we will cover writing to files in Python and use that to write the output of our Ping script for later review. Lastly, we will look at how we can extend our Ping script by include a Port-Scan to each host that is online.

Reading Files in Interactive mode

Getting ready

In Visual Studio Code, browse to **start/CH04/ips.txt**. Edit the file to include several IP addresses to ping. This can include addresses that you know will succeed AND addresses you know will fail.

How to do it

On the Raspberry Pi, open a terminal window and change directory (`cd`) into the **CH04** directory. Once there, start Python in interactive mode. Most likely, the commands will be:

```
cd PythonForCyberSecurity
cd start
cd CH04
python3
```

In the Python interactve window, open and view file by typing:

```
1  myfile = open("ips.txt", "r")
2  print(myfile.read())
```

Reading Files

Getting ready

In Visual Studio Code, browse to **start/CH04/ReadFile.py**. This will open the file in the editor pane.

How to do it

```
1  #!/usr/bin/env python3
2  # Sample script that reads from a file
3  # By Ed Goad
4  # 2/27/2021
5
6  # Open file for reading
7  ip_file = open("ips.txt", "r")
8
9  # Read the contents of the file and print to screen
10 ip_addresses = ip_file.read()
11 print(ip_addresses)
12
13 # Close the file
14 ip_file.close()
```

 Don't forget to commit the changes to Git and GitHub

How it works

The script starts on line **#7** and uses the built-in Python function `open()` to open the file **ips.txt** (remember Linux is case-sensitive). The file is opened in read-only mode, as signified by the "**r**" after the file name.

When the file is opened, it is put into a special type of variable known as an object. The object, named **ip_file**, has attributes (such as name, mode, softspace, and closed), and methods (such as read, write, rename, remove, close, and others). What is important to understand at this point is that while it *looks like a variable*, **ip_file** works slightly differently.

> **NOTE:** This last statement is an over-simplification, made so that we can focus on *what* Python is doing instead of details about *how* it is doing it.

On line **#10** we read the contents of the file using the `ip_file.read()` method. The contents of the file are read and stored in the **ip_addresses** variable. On line **#11** the contents of **ip_addresses** are printed to the screen.

On line **#14** the file is closed using the `close()` method.

```
PS C:\Users\edgoa\source\PythonForCyberSecurity>  & 'C:\Users\e
de\extensions\ms-python.python-2021.2.582707922\pythonFiles\lib
ecurity\ReadFile.py'
192.168.0.10
192.168.0.11
192.168.0.12
```

Output of script

 Use the debugger to walk through the script for more details on how it works.

There's more

File modes in Python

There are several modes for accessing files in Python. Below is a list of the various options, and some options can be combined together. For instance, in our example we used the **r** mode for reading, but we could have been more specific and stated **rt** to identify that we were reading a text file.

Mode	Description
r	Opens file for reading. This is a default mode, and is used if no other mode is specified.
w	Opens file for writing.
x	Creates a new file, and fails if the target already exists.
a	Opens file for appending.
t	Opens file in "text" mode. This is a default mode, and is used if no other mode is specified.
b	Opens file in "binary" mode.
+	Opens file for reading and writing.

Other ways to open files

There are several methods to open files for reading. For instance, in the script above we used:

```
# Open file for reading
ip_file = open("ips.txt", "r")

# Read the contents of the file and print to screen
ip_addresses = ip_file.read()
print(ip_addresses)

# Close the file
ip_file.close()
```

Another option is to open the file and read it line-by-line. This can be useful if looking for specific content in the line. In the example below we are looking for IP addresses that start with "192." only.

```
# Open file for reading
ip_file = open("ips.txt", "r")
# Use a loop to read the file line-by-line
for line in ip_file:
    # Perform evaluation
    if line.startswith("192."):
        print(line)
# Close the file
ip_file.close()
```

And we can also use a `with` statement to open the file. One of the main benefits here is that the file will automatically close when we finish reading it.

```
# Open file for reading
with open("ips.txt", "r") as ip_file:
    print(ip_file.read())
```

Fifth Ping

In this extension of our pinger script, we will get our target addresses from a text file. Instead of pinging an entire network, which may be slow and may alert an Intrusion Detection System, this allows for more specific targeting of addresses.

Getting ready

To begin pinging IP addresses from a file, we must have a file of IP addresses to ping. If you began by cloning my repository from GitHub, edit the **CH04\ips.txt** file and enter in several IP address, one on each line.

In Visual Studio Code, browse to **start/CH04/pinger5.py**. This will open the file in the editor pane.

How to do it

```
#!/usr/bin/env python3
# Fifth example of pinging from Python
# Reading IPs from a file
# By Ed Goad
# 2/27/2021

# import necessary Python modules
import platform
import os

def ping_host(ip):
    # Determine the currrent OS
    currrent_os = platform.system().lower()
    if currrent_os == "windows":
        # Build our ping command for Windows
        ping_cmd = f"ping -n 1 -w 2 {ip} > nul"
    else:
        # Build our ping command for other OSs
        ping_cmd = f"ping -c 1 -w 2 {ip} > /dev/null 2>&1"
    # Execute command and capture exit code
    exit_code = os.system(ping_cmd)
    return exit_code

def import_addresses():
    # Create empty list object
    lines = []
    # Open file and read line-by-line
    f = open("ips.txt", "r")
    for line in f:
        # Use strip() to remove spaces and carriage returns
        line = line.strip()
        # Add the line to the lines list object
        lines.append(line)
    # Return the list object to the main body
    return lines

# read IPs from file
ip_addresses = import_addresses()

for ip in ip_addresses:
    # Call ping_host function and capture the return value
```

```
42      exit_code = ping_host(ip)
43
44      # Print results to console only if successful
45      if exit_code == 0:
46          print("{0} is online".format(ip))
```

 Don't forget to commit the changes to Git and GitHub

How it works

Our new script utilizes much of the existing code written for our prior ping scripts. Most of the work is still being performed by the `ping_host()` function, which is still being called the same way. The updates in this script are centered around the `import_addresses()` function.

Starting on line **#26**, we create an empty list object named **lines**. Lists in Python are identified by the use of the square brackets **[]**, which are either empty like ours, or can have items between the brackets. Here, we are pre-creating an empty list so we can add to it later.

> **NOTE**: A list object, sometimes referred to as a 1-dimensional array, can be thought of as a simple list of items. Similar to a shopping list or a to-do list, there is 1 item per line, and we can work with the list 1 item at a time.

In lines **#28**, **29**, we open the file **ips.txt** for reading and use a **for** loop to begin reading the file line-by-line. The expectation is the text file will have 1 IP address per line, and nothing else.

In line **#31**, we use the `strip()` method to remove "whitespace" from the beginning and end of the line. In computers, whitespace refers to any horizontal or vertical spacing. Common whitespace characters are created by pressing the "space bar", the "tab key", and the "enter key". By using the `strip()` method, we are removing all of these, if they exist, from our line and leave only the address.

In line **#33**, we append, or add to the value our **line** variable (the line in the file) into the **lines** list. And then lastly, we return the list object on line **#35** to the main body of the script.

A **for** loop beginning on line **#40** then loops through each **ip_addresses**. Each address is then pinged, and the results printed to the console.

```
PS C:\Users\edgoa\source\PythonForCyberSecurity>
de\extensions\ms-python.python-2021.2.582707922\py
ecurity\pinger5.py'
192.168.0.10 is online
PS C:\Users\edgoa\source\PythonForCyberSecurity>
```

Output of script


Use the debugger to walk through the script for more details on how it works.

There's more

The `strip()` method is a method built into the **string** data type. We can see how it works in interactive Python by creating a string with whitespace in it, and then `print()` it without `strip()`, and again with `strip()`. In Python, to create a multi-line variable (one with carriage-returns in it), begin the assignment with 3 quotes together (`"""`). Everything typed after that will be part of the variable until another set of 3 quotes (`"""`) are presented. For example:

```
1  bar = """
2
3  lkjlkj
4
5  """
6  print(bar)
7  print(bar.strip())
```

```
>>> bar = """
...
... lkjlkj
...
... """
>>> print(bar)


lkjlkj


>>> print(bar.strip())
lkjlkj
>>>
```

Interactive python commands

As you can see here, we assign the variable 2 empty lines, some text, and then 2 more empty lines. When we use **print()** normally, the empty lines/whitespace are shown. When we use **print()** with the **strip()** method, only the text is shown.

Writing Files

In addition to reading files in Python, we can also write to files. This can be helpful if we want to save information, such as a log file, for review later.

Getting ready

In Visual Studio Code, browse to **start/CH04/WriteFile.py**. This will open the file in the editor pane.

How to do it

```
1  #!/usr/bin/env python3
2  # Sample script that writes to a file
3  # By Ed Goad
4  # 2/27/2021
5
6  # Open file for writing
7  test_file = open("testfile.txt", "w")
8
9  # Write lines to the file
10 test_file.write("Hello World\n")
11 test_file.write("My name is Ed\n")
12 test_file.write("I like rubber ducks\n")
13
14 # Close the file
15 test_file.close()
```

 Don't forget to commit the changes to Git and GitHub

How it works

The script starts on line **#7** by opening the file **testfile.txt** for **w**riting. When the file is opened, a reference to the file object is assigned to **test_file**.

In lines **#10 - #12**, we use the `test_file.write()` method to write to our file several times. Note the **\n** at the end of each string, this is the equivalent of pressing **Enter** on your keyboard. This is needed so the file will move to the next line.

In line **#15**, we call the `test_file.close()` method to close the file. While not strictly necessary because the file will close automatically when the script finishes, it is a good habit to have.

There's more

Additional things to try:

Change the file mode from **w** to **a** to append to the file. Test the script and see how the file is updated with each consecutive execution.

Try the `write()` method without the **\n** at the end of the string, or with it included multiple times. Run the script and see how the saved file changes.

Try modifying the file open to use a `with` statement instead. Review how the **with** statement was used to read files, and apply the same method to writing files.

Last Ping

Now that we know how to 1) ping an address in Python, 2) read a list of IPs from a file, and 3) write to files, we can now combine these to create our final pinger script.

Getting ready

In Visual Studio Code, browse to **start/CH04/pinger6.py**. This will open the file in the editor pane.

How to do it

```
#!/usr/bin/env python3
# Sixth example of pinging from Python
# Writing log messages to a file
# By Ed Goad
# 2/27/2021

# import necessary Python modules
import platform
import os
from datetime import datetime

def write_log(message):
    now = str(datetime.now()) + "\t"
    message = now + str(message) + "\n"
    f = open("pinger.log", "a")
    f.write(message)
    f.close()

def ping_host(ip):
    # Determine the currrent OS
    currrent_os = platform.system().lower()
    if currrent_os == "windows":
        # Build our ping command for Windows
        ping_cmd = f"ping -n 1 -w 2 {ip} > nul"
    else:
```

```
26          # Build our ping command for other OSs
27          ping_cmd = f"ping -c 1 -w 2 {ip} > /dev/null 2>&1"
28      # Execute command and capture exit code
29      exit_code = os.system(ping_cmd)
30      return exit_code
31
32  def import_addresses():
33      # Create empty list object
34      lines = []
35      # Open file and read line-by-line
36      f = open("ips.txt", "r")
37      for line in f:
38          # Use strip() to remove spaces and carriage returns
39          line = line.strip()
40          # Add the line to the lines list object
41          lines.append(line)
42      # Return the list object to the main body
43      return lines
44
45  # read IPs from file
46  write_log("Reading IPs from ips.txt")
47  ip_addresses = import_addresses()
48  write_log("Imported {0} IPs".format(len(ip_addresses)))
49
50  for ip in ip_addresses:
51      # Call ping_host function and capture the return value
52      exit_code = ping_host(ip)
53
54      # Print results to console only if successful
55      if exit_code == 0:
56          write_log("{0} is online".format(ip))
57          print("{0} is online".format(ip))
58      else:
59          write_log("{0} is offline".format(ip))
```

 Don't forget to commit the changes to Git and GitHub

How it works

We start our changes from the prior script on line **#10** where we import the **datetime** module. More specifically, we are importing the **datetime** object from the **datetime** module. This will allow us to

call `datetime.now()` to get a timestamp of our current date and time, which will be helpful when reviewing our logs.

> **NOTE:** It may appear confusing to import this way, but this is done to minimize confusion later. If we had instead used `import datetime`, to get the current timestamp we would then have to call `datetime.datetime.now()`.

Our main addition begins on line **#12** with our `write_log()` function. On line **#13** we create a variable named **now** and store the current date-time in it. In addition to the time, we append a **\t**, which is the equivalent of pressing the **TAB** key on the keyboard.

In line **#14** we combine our timestamp, our message, and a carriage return (**\n**) into the **message** variable. Note that we used `str(message)` to ensure our message was in a string format. This can be helpful in the future if we try to log other data types such as number, lists, or dictionary objects.

In the main body of pinger6.py, we add several calls to our new `write_log()` function. This starts on line **#46** with a comment that we are reading the IPs from a file. And another comment on line **#48** where we log how many addresses were imported.

NOTE: Here we are using `len(ip_addressess)` to find the number of addresses that are in the file. The `len()` function returns the number of items, or **len**gth, in an object.

In line **#58**, we added an **else:** clause to our **if** statement to capture when addresses are offline, as well as online. Both of the possible results (online/offline) are written to the log file, with the expectation that we may desire that information at a later date. Only the online status is still presented to the user.

Once the script has ran successfully, we can open the **pinger.log** file and view the output. As you can see in the screenshot below, the date and time are included to help simplify review of subsequent runs of the script.

pinger.log

```
2021-02-28 09:09:27.836169  Reading IPs from ips.txt
2021-02-28 09:09:27.837166  Imported 16 IPs
2021-02-28 09:09:27.872075  192.168.0.10 is online
2021-02-28 09:09:28.267016  192.168.0.11 is offline
2021-02-28 09:09:28.766680  192.168.0.12 is offline
2021-02-28 09:09:29.266345  192.168.0.13 is offline
2021-02-28 09:09:29.767006  192.168.0.14 is offline
2021-02-28 09:09:30.266671  192.168.0.15 is offline
2021-02-28 09:09:30.766335  192.168.0.16 is offline
2021-02-28 09:09:31.266997  192.168.0.17 is offline
2021-02-28 09:09:31.766661  192.168.0.18 is offline
2021-02-28 09:09:32.266325  192.168.0.19 is offline
2021-02-28 09:09:32.766988  192.168.0.110 is offline
2021-02-28 09:09:33.266651  192.168.0.111 is offline
2021-02-28 09:09:33.766315  192.168.0.112 is offline
2021-02-28 09:09:34.266977  192.168.0.113 is offline
2021-02-28 09:09:34.767639  192.168.0.114 is offline
2021-02-28 09:09:35.266305  192.168.0.115 is offline
```

Log file example


Use the debugger to walk through the script for more details on how it works.

Scanner Example

In addition to IP addresses identifying a machine, computers use port numbers to specify services and applications on the system. A security engineer will want to know what ports are open on a computer to determine what is running on the computer, and what needs to be secured. By performing a port scan on the computer, we can get a better view of the overall security posture.

Getting started

For this project we will be integrating with an existing cybersecurity tool known as Network Mapper (Nmap). To use Nmap, we must first install the tool on our system, and then install a **python-nmap** library to interact with it.

Installing Nmap

To install Nmap on the Raspberry Pi, open a terminal window and type:

```
sudo apt update
sudo apt install nmap
```

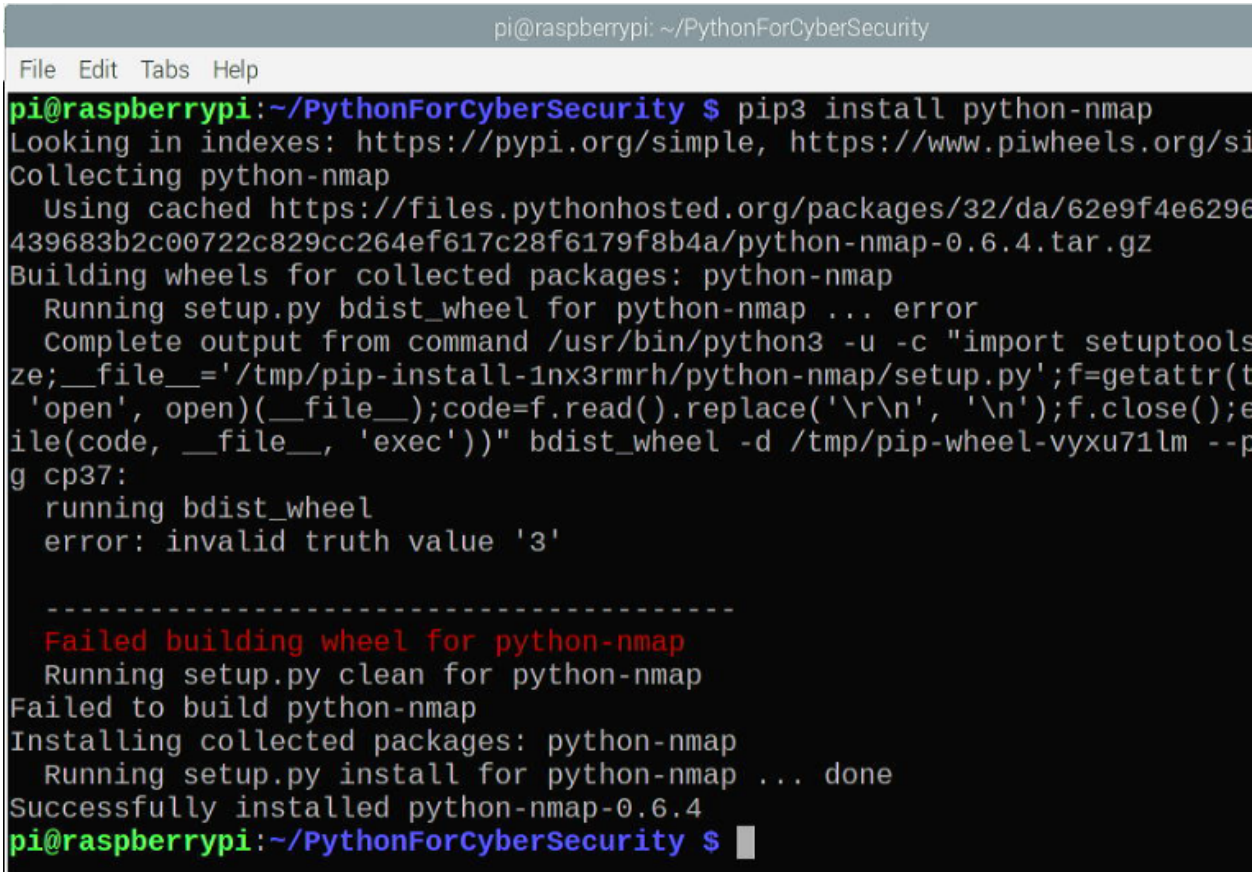
If prompted for a password, enter the user's password.

Installing python-nmap

Once the Nmap tool is installed, we can download and install a library specifically designed to allow Python to interact with Nmap. To do this, we will use a command-line tool named **pip3**. On the Raspberry Pi, open a terminal window and type:

```
pip3 install python-nmap
```

NOTE: You may see an error similar to below. The final statement however is "Successfully installed".



Installing python-nmap

How to do it

```
1  #!/usr/bin/env python3
2  # Port scanner example
3  # Use 'pip3 install python-nmap' to install modules
4  # Use 'sudo apt -y install nmap' to install nmap
5  # By Ed Goad
6  # 2/27/2021
7
8  # import necessary Python modules
9  import nmap
10
11 # Identify target address
12 target_address = "192.168.0.10"
13
14 # Identify start and stop port for the scan
15 port_start = 1
16 port_end = 100
17
18 # Create the scanner object
19 scanner = nmap.PortScanner()
20
21 print("Scanning {0}".format(target_address))
22 # Loop through each port and scan
23 for port in range(port_start, port_end + 1):
24     result = scanner.scan(target_address, str(port))
25     port_status = result['scan'][target_address]['tcp'][port]['state']
26     print("\tPort: {0} is {1}".format(port, port_status))
```

 Don't forget to commit the changes to Git and GitHub

How it works

The script starts on line **#9** by importing the **nmap** library. This library contains Python scripts and code from other languages that simplify our port scanner. Everything in this library *can be* created in our script, but that could be considered reinventing the wheel.

On lines **#12-16** we are assigning values to the **target_address**, **port_start**, and **port_end** variables. The address is the IP address we want to scan, and then the start and end variables define the range of ports to scan. By defining them here our script will be easier to modify in the future.

On line **#19** we create a new `nmap.PortScanner()` object and name it **scanner**. In computers, an object is like a car: it has **properties** like color and engine size, and has **methods** like start and go forward. Our new scanner object has both methods and properties we can use moving forward.

In line **#23** we create a loop beginning at **port_start** and ending at **port_end**. Because the range() function ends before the top number, we set the range to end at **port_end + 1** to include the final port.

In line **#24** we call the `scanner.scan()` method and pass the IP address and port. The method returns a large amount of information that we store in the **result** variable.

Lines **#25**, **26** filter through the contents of the result variable to find only the state of the port. This is then printed out to the screen.

 Use the debugger to walk through the script for more details on how it works.

There's more

This sample scanner script scans a small subset of ports on a single IP address. This can easily be extended to allow:

- Use a loop to target multiple IPs
- Put port scanning into a function. Pass in IP and list of ports to scan.
- Use Nmap to perform the IP scanning in addition to the port scanning
- Change the ports from 1-100 to be a list of most commonly used ports

See also

https://pypi.org/project/python-nmap/[22]

http://xael.org/pages/python-nmap-en.html[23]

[22] https://pypi.org/project/python-nmap/
[23] http://xael.org/pages/python-nmap-en.html

Chapter 5: Cryptography

In this chapter we will cover the following:

- Introduction to Cryptography
- Simple cryptography - Caesar Cipher / ROT13
- Pseudo-Encryption - Encoding Data
- Why you shouldn't create your own encryption scheme
- Using Encryption Libraries

Introduction

In this chapter we will introduce the idea of cryptography, or the ability to hide and secure data from unauthorized people. We will begin by creating a script using an old form of cryptography known as the Caesar Cipher, also known as ROT13. We will then cover the idea of pseudo-encryption, or encoding of data to make it look encrypted. And lastly, we will use some of the more popular Python libraries to encrypt data.

Introduction to Cryptography

Cryptography is the idea of keeping information private in the presence of adversaries. Essentially the ability to hide information that may be visible by someone else, or overheard by an eavesdropper.

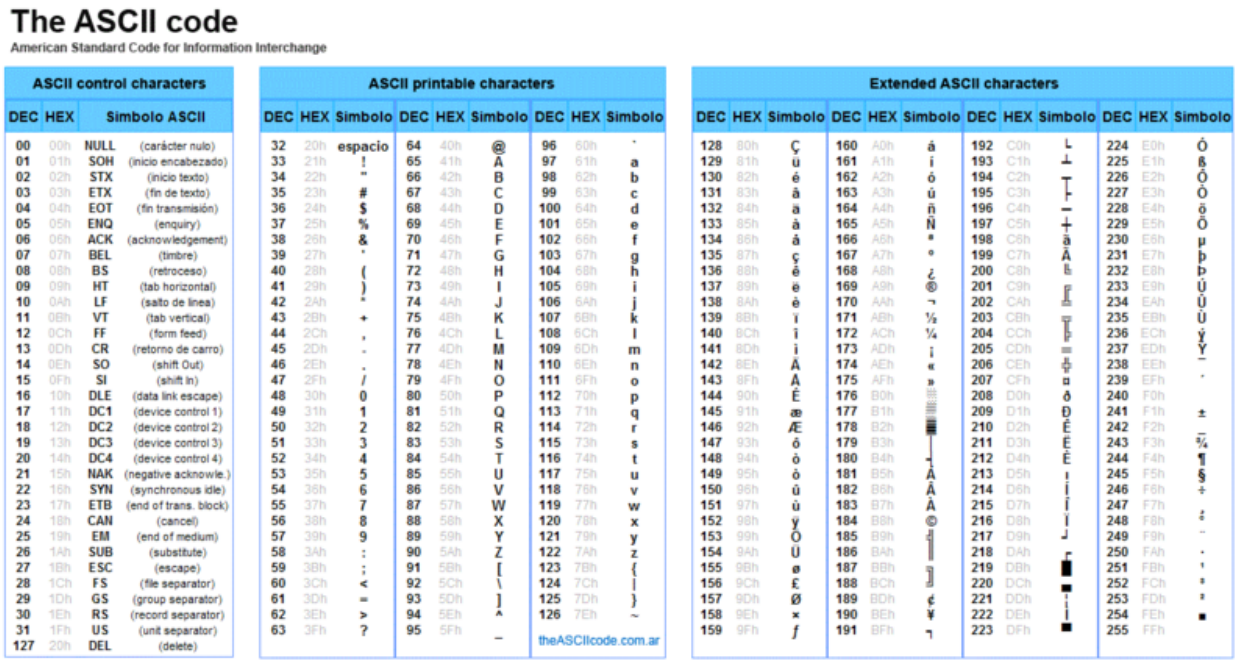
Cryptography has its roots with simple ciphers such as the transposition or substitution ciphers. These encryption methods would either jumble the letters of a word, or replace letters with other letter and symbols.

Simple cryptography - Caesar Cipher

The Caesar cipher is an early form of a substitution cipher, sometimes referred to as a shift cipher. The idea behind the Caesar cipher is to take a letter in a word, like "a" and shift it a predefined number of letters. In the case of a 1 letter shift, "a" becomes "b". In this case the string "Hello World" becomes "Ifmmp Xpsme".

Letters == Numbers

Computers ultimately only understand numbers and basic math. To make computers more human-friendly, some of those numbers are used to identify letters. One of the common methods to do this is the American Standard Code for Information Interchange (ASCII) table. As we can see below, we have the numbers from 0 - 255, and the associated letters they represent.



The ASCII code

American Standard Code for Information Interchange

ASCII control characters

DEC	HEX	Simbolo ASCII	
00	00h	NULL	(carácter nulo)
01	01h	SOH	(inicio encabezado)
02	02h	STX	(inicio texto)
03	03h	ETX	(fin de texto)
04	04h	EOT	(fin transmisión)
05	05h	ENQ	(enquiry)
06	06h	ACK	(acknowledgement)
07	07h	BEL	(timbre)
08	08h	BS	(retroceso)
09	09h	HT	(tab horizontal)
10	0Ah	LF	(salto de línea)
11	0Bh	VT	(tab vertical)
12	0Ch	FF	(form feed)
13	0Dh	CR	(retorno de carro)
14	0Eh	SO	(shift Out)
15	0Fh	SI	(shift In)
16	10h	DLE	(data link escape)
17	11h	DC1	(device control 1)
18	12h	DC2	(device control 2)
19	13h	DC3	(device control 3)
20	14h	DC4	(device control 4)
21	15h	NAK	(negative acknowle.)
22	16h	SYN	(synchronous idle)
23	17h	ETB	(end of trans. block)
24	18h	CAN	(cancel)
25	19h	EM	(end of medium)
26	1Ah	SUB	(substitute)
27	1Bh	ESC	(escape)
28	1Ch	FS	(file separator)
29	1Dh	GS	(group separator)
30	1Eh	RS	(record separator)
31	1Fh	US	(unit separator)
127	20h	DEL	(delete)

ASCII printable characters

DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo
32	20h	espacio	64	40h	@	96	60h	`
33	21h	!	65	41h	A	97	61h	a
34	22h	"	66	42h	B	98	62h	b
35	23h	#	67	43h	C	99	63h	c
36	24h	$	68	44h	D	100	64h	d
37	25h	%	69	45h	E	101	65h	e
38	26h	&	70	46h	F	102	66h	f
39	27h	'	71	47h	G	103	67h	g
40	28h	(	72	48h	H	104	68h	h
41	29h	)	73	49h	I	105	69h	i
42	2Ah	*	74	4Ah	J	106	6Ah	j
43	2Bh	+	75	4Bh	K	107	6Bh	k
44	2Ch	,	76	4Ch	L	108	6Ch	l
45	2Dh	-	77	4Dh	M	109	6Dh	m
46	2Eh	.	78	4Eh	N	110	6Eh	n
47	2Fh	/	79	4Fh	O	111	6Fh	o
48	30h	0	80	50h	P	112	70h	p
49	31h	1	81	51h	Q	113	71h	q
50	32h	2	82	52h	R	114	72h	r
51	33h	3	83	53h	S	115	73h	s
52	34h	4	84	54h	T	116	74h	t
53	35h	5	85	55h	U	117	75h	u
54	36h	6	86	56h	V	118	76h	v
55	37h	7	87	57h	W	119	77h	w
56	38h	8	88	58h	X	120	78h	x
57	39h	9	89	59h	Y	121	79h	y
58	3Ah	:	90	5Ah	Z	122	7Ah	z
59	3Bh	;	91	5Bh	[	123	7Bh	{
60	3Ch	<	92	5Ch	\	124	7Ch	|
61	3Dh	=	93	5Dh	]	125	7Dh	}
62	3Eh	>	94	5Eh	^	126	7Eh	~
63	3Fh	?	95	5Fh	_			

theASCIIcode.com.ar

Extended ASCII characters

DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo
128	80h	Ç	160	A0h	á	192	C0h	└	224	E0h	Ó
129	81h	ü	161	A1h	í	193	C1h	┴	225	E1h	ß
130	82h	é	162	A2h	ó	194	C2h	┬	226	E2h	Ô
131	83h	â	163	A3h	ú	195	C3h	├	227	E3h	Ò
132	84h	ä	164	A4h	ñ	196	C4h	─	228	E4h	õ
133	85h	à	165	A5h	Ñ	197	C5h	┼	229	E5h	Õ
134	86h	å	166	A6h	ª	198	C6h	ã	230	E6h	µ
135	87h	ç	167	A7h	º	199	C7h	Ã	231	E7h	þ
136	88h	ê	168	A8h	¿	200	C8h	╚	232	E8h	Þ
137	89h	ë	169	A9h	⌐	201	C9h	╔	233	E9h	Ú
138	8Ah	è	170	AAh	¬	202	CAh	╩	234	EAh	Û
139	8Bh	ï	171	ABh	½	203	CBh	╦	235	EBh	Ù
140	8Ch	î	172	ACh	¼	204	CCh	╠	236	ECh	ý
141	8Dh	ì	173	ADh	¡	205	CDh	═	237	EDh	Ý
142	8Eh	Ä	174	AEh	«	206	CEh	╬	238	EEh	¯
143	8Fh	Å	175	AFh	»	207	CFh	¤	239	EFh	´
144	90h	É	176	B0h	░	208	D0h	ð	240	F0h	
145	91h	æ	177	B1h	▒	209	D1h	Ð	241	F1h	±
146	92h	Æ	178	B2h	▓	210	D2h	Ê	242	F2h	‗
147	93h	ô	179	B3h	│	211	D3h	Ë	243	F3h	¾
148	94h	ö	180	B4h	┤	212	D4h	È	244	F4h	¶
149	95h	ò	181	B5h	Á	213	D5h	ı	245	F5h	§
150	96h	û	182	B6h	Â	214	D6h	Í	246	F6h	÷
151	97h	ù	183	B7h	À	215	D7h	Î	247	F7h	¸
152	98h	ÿ	184	B8h	©	216	D8h	Ï	248	F8h	°
153	99h	Ö	185	B9h	╣	217	D9h	┘	249	F9h	¨
154	9Ah	Ü	186	BAh	║	218	DAh	┌	250	FAh	·
155	9Bh	ø	187	BBh	╗	219	DBh	█	251	FBh	¹
156	9Ch	£	188	BCh	╝	220	DCh	▄	252	FCh	³
157	9Dh	Ø	189	BDh	¢	221	DDh	¦	253	FDh	²
158	9Eh	×	190	BEh	¥	222	DEh	Ì	254	FEh	■
159	9Fh	ƒ	191	BFh	┐	223	DFh	▀	255	FFh	

ASCII table

Source: https://commons.wikimedia.org/wiki/File:Ascii-codes-table.png[24]

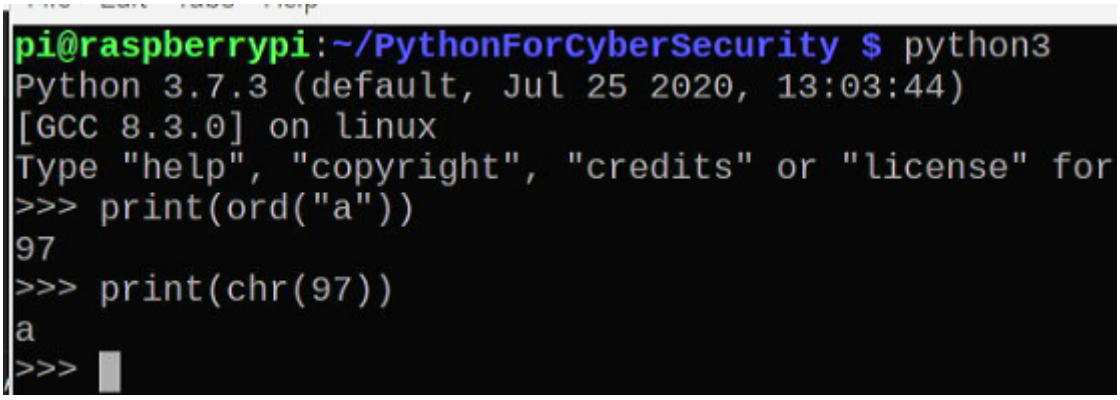
Converting numbers and letters

In Python, we can use the `ord()` and `chr()` built-in functions to convert between numbers and letters. The `ord()` function takes a letter as input, and then outputs the ASCII equivalent number. The `chr()` function converts numbers into letters. We can quickly view an example of this in interactive Python by typing the following.

```
1 print(ord("a"))
2 print(chr(97))
```

We can see from the screenshot that we converted the letter **a** into the number **97**. We then converted the number **97** back into the letter **a**.

[24] https://commons.wikimedia.org/wiki/File:Ascii-codes-table.png



Using ord() and chr()

Creating your own ASCII table

We can futher view the relationship between numbers and letters in computers by creating our own ASCII table.

How to do it

ASCIIgen.py

```
1  #!/usr/bin/env python3
2  # ASCII generator
3  # Uses chr() to create ASCII characters
4  # By Ed Goad
5  # 2/27/2021
6
7  for i in range(127):
8      print("{0}\t'{1}'".format(i,chr(i)))
```

 Don't forget to commit the changes to Git and GitHub

How it works

This simple script starts on line **#7** with a for loop. First the **range()** function returns a list of numbers between 0 and 126. The first number, **0**, is taken from the list and assigned to the variable **i**.

On line **#8**, the print statement first prints the value of **i**, and then using the **chr()** function, also prints the ASCII character associated with that number. When finished, the loop repeats with the next number in the range.

```
pi@raspberrypi:~/PythonForCyberSec $ /usr/bin/python3 /home/pi/PythonForCyb
erSec/end/CH05/ASCIIgen.py
0       ''
1       ''
2       ''
3       ''
4       ''
5       ''
6       ''
7       ''
8       '
9       '       '
10      '
'
11      '
         '
```

ASCIIgen output


Use the debugger to walk through the script for more details on how it works.

There's more

If you look through the results of the script, you will notice that several characters return unexpected results. The first example of this is for ASCII number 8, which only shows a single quote. This is because this is a special character used by the operating to perform formatting, in this case a backspace. Other special characters you may see include tab, linefeed, and carriage return.

If desired, this script can be modified to display the "Extended ASCII" codes that continue up to 255. Additionally, it can be extended to include Unicode Characters, though that is out of scope for this book.

Rot13.py

One common example of the Caesar Cipher is known as ROT13. Because there are 26 letters in the alphabet, we can "encrypt" them by rotating them 13 spaces. When we do this, the string "Hello World" becomes "Uryyb Jbeyq".

The benefit of ROT13 is the ease of then "decrypting" the string by rotating the letters by 13 space again. By doing this several times we start with "Hello World", then transition to "Uryyb Jbeyq", and finally back to "Hello World". Both the encryption and decryption processes are the same.

In Visual Studio Code, browse to **start/CH05/Rot13.py** and open it for editing.

How to do it

Rot13.py

```
1  #!/usr/bin/env python3
2  # Script that encrypts/decrypts text using ROT13
3  # By Ed Goad
4  # date: 2/5/2021
5
6  # Prompt for the source message
7  source_message = input("What is the message to encrypt/decrypt? ")
8  # Convert message to lower-case for simplicity
9  source_message = source_message.lower()
10 final_message = ""
11
12 # Loop through each letter in the source message
13 for letter in source_message:
14     # Convert the letter to the ASCII equivalent
15     ascii_num = ord(letter)
16     # Check to see if an alphabetic (a-z) character,
17     # if not, skip
18     if ascii_num >= 97 and ascii_num <= 122:
19         # Add 13 to ascii_num to "shift" it by 13
20         new_ascii = ascii_num + 13
21         # Confirm new character will be alphabetic
22         if new_ascii > 122:
23             # If not, wrap around
24             new_ascii = new_ascii - 26
25         final_message = final_message + chr(new_ascii)
26     else:
27         final_message = final_message + chr(ascii_num)
28
29 # Print converted message
30 print("Message has been converted:")
31 print(final_message)
```

 Don't forget to commit the changes to Git and GitHub

How it works

The script starts on line **#7** by prompting the user for a message to either encrypt or decrypt. This message is then converted to lower-case on line **#9** to simplify our script.

On line **#13** we begin a **for** loop to break our message into individual letters. These letters will be converted one at a time.

On line **#15** the letter is converted to the ASCII equivalent number using the **ord()** built-in Python function. If we look at the ASCII table, this means the letter "a" will be converted to the number 97.

On line **#17** the number is evaluated to see if it is between 97 and 122, the ASCII values for the letters **a** and **z** respectively. If the number is out of that range, we can assume it isn't a letter and instead is punctuation such as :",'#; or other similar characters that we will ignore. If that is the case, we skip to line **#26** where the ASCII value is converted back into the character and appended to the **final_message** string.

If the ascii_num is found to contain a valid letter, on line **#19 new_ascii** is assigned ascii_num + 13 to "shift" it.

At this point it is possible we "shifted" the letter too far, beyond the letter z and into non-printable characters. To resolve this, on line **#21** we check the value of new_ascii to see if it has moved past the letter z. If we find we have gone too far, we "wrap-around" to the beginning of the alphabet by subtracting 26 from the value.

Lastly, the **final_message** is printed to the console for the user.

 Use the debugger to walk through the script for more details on how it works.

There's more

This is a simplified script used primarily to demonstrate how encryption *can* work. Several simplifications of the process have been taken to demonstrate the idea of encryption. If you so desire, the script can be extended as a learning exercise. Some suggestions on possible extensions:

- Upper and lower case - Update the script to work with both upper-case and lower-case letters.
- Files - Update the script to encrypt/decrypt files instead of requiring the message to be typed.
- Variable shifting - Update the script to support "shifting" from 1 to 25 letters. Note: This will require separate encryption and decryption processes.
- Punctuation - Update the script to shift punctuation as well as letters.
- Changing rotation - Update the script so that each letter rotates based on the value of the prior letter or letters.
- Rotate based on a one-time pad - Using a separate file filled with random numbers, rotate each character using these numbers. To decrypt the data, the recipient must have the same one-time pad. More details can be found at https://en.wikipedia.org/wiki/One-time_pad[25]

See also

https://en.wikipedia.org/wiki/ROT13[26]

https://en.wikipedia.org/wiki/Substitution_cipher[27]

[25] https://en.wikipedia.org/wiki/One-time_pad
[26] https://en.wikipedia.org/wiki/ROT13
[27] https://en.wikipedia.org/wiki/Substitution_cipher

Pseudo-Encryption - Encoding Data

Encoding data to keep it hidden is not true encryption, but that doesn't stop it from being used to "obfuscate" the information. One of the easiest ways to describe the idea of encoding data is having a conversation with someone in a different language, possibly to stop from being overheard. Whether the language is Spanish, Russian, or Pig-Latin, if the eves-dropper can't understand what you are saying, you have effectively "hidden" the contents of the message.

> **NOTE:** This isn't real encryption and can be easily overcome by the eves-dropper learning the new language, enlisting the help of someone who knows the language, or recording the message for in-depth analysis later.

Encoding of data in computers is used for 2 primary purposes: the first is to hide or obfuscate information, and the second is to convert the information in a way that works better for computers or transportation across the internet.

Getting ready

Base64.py

Here we will create an example that performs base64 encoding and decoding. Base64 encoding is commonly used on the Internet to convert binary data, like a picture or file, into ASCII characters. Once converted, the information can be sent using tools and protocols initially designed for text only, such as email.

This example below is closer to the idea of real computer encryption. In this example we can see that there are similar, but different, encoding and decoding processes. Additionally, the encoding and decoding occur in functions that could be external to the script. The consumer of the script doesn't need to know how the functions work, except how to call them, and theoretically the function could be updated without the consumer noticing.

In Visual Studio Code, browse to **start/CH05/Base64.py** and open it for editing.

How to do it

```
#!/usr/bin/env python3
# Script that "encrypts"/"decrypts" text using base64 encoding
# By Ed Goad
# 2/5/2021

# import necessary Python modules
import base64

def encode_data(plain_text):
    # Convert plain_text string to bytes
    plain_text = plain_text.encode()
    # Encode the plain_text
    cipher_text = base64.b64encode(plain_text)
    # Convert the encoded bytes back to string
    cipher_text = cipher_text.decode()
    return cipher_text

def decode_data(cipher_text):
    # Decode the cipher_text
    plain_text = base64.b64decode(cipher_text)
    # Convert the decoded bytes to string
    plain_text = plain_text.decode()
    return plain_text

# Prompt the user for method and message
method = input("Do you wish to Encode or Decode (e/d)? ").lower()
message = input("What is the message? ")

# Using first letter in variable,
# call the encode or decode function
if method[0] == "e":
    print(encode_data(message))
elif method[0] == "d":
    print(decode_data(message))
else:
    # if method wasn't "e" or "d", print error message and quit
    print("Wrong method selected. Choose Encode or Decode")
```

Don’t forget to commit the changes to Git and GitHub

How it works

When the script is executed, it begins on line **#7** importing the base64 module. At **#26,27** the script prompts the user for the action to perform and the message contents.

Lines **#31 - #37** look at the first letter of the **method** variable to call the `encode_data()` function, `decode_data()` function, or return an error message.

Encode

If the method was "encode", our `encode_data()` function is called, and the **message** content is assigned to the **plain_text** variable.

On line **#11**, the **plain_text** string variable uses the `string.encode()` method to convert the contents from a string to a list of bytes, or simply binary. This is necessary because base64 encoding expects binary data.

On line **#13** we use the `base64.b64encode()` method to encode the **plain_text** data into **cipher_text**.

Finally on **#15**, the `string.decode()` method converts the binary data back into text, and then return the message.

Decode

If method was "decode", the `decode_data()` function is called, and the **message** content is assigned to the **cipher_text** variable.

On line **#20**, we use the `base64.b64decode()` method to decode the **cipher_text** data into **plain_text**.

Once the data is decoded, the **plain_text** string variable uses the `string.decode()` method to convert the contents to text from bytes to a string, and then returns the message.

Lastly, if neither encode or decode were specified, a warning message is printed to the screen on line **#36** and the script exits.

There's more

Base64 encoding is frequently used to transport information across the internet. The internet was primarily designed to transfer "text", which causes problems when pictures, videos, or other content is needed. So base64 is used to encode/convert binary information like email attachments into text, which can then be sent across the internet.

Our script can be extended to encode not just text data, but whole files as well.

Why you shouldn't create your own encryption scheme

It is very tempting to create and use a custom encryption scheme, or way encrypting and decrypting data. If we envisioned this as a door lock, by creating a custom locking mechanism and key, we might

assume it is more secure because it doesn't follow a commonly known standard. These custom locks would only be known by the creator, and therefore considerably difficult for the average lock-pick to bypass.

While it is possible that your custom encryption scheme may be more secure than other methods, it is very likely that it is less secure. Custom encryption methods and schemes primarily rely on a lack of knowledge by attackers about the encryption scheme to keep them secure. Once the inner workings are identified via various means, the environment will be vulnerable.

Additionally, while breaking an encryption scheme may be difficult for you and I, that doesn't mean it will be difficult for everyone. More experienced professionals may look at your process and easily overcome your encryption using standard cryptanalytic techniques.

Lastly, even with the strongest encryption model possible, you may make a mistake when creating your own encryption scheme. A simple typo in your code or a mishandled calculation could make your encryption scheme vulnerable.

By using publicly known and trusted encryption schemes, you can be generally assured the encryption is robust and secure. The currently trusted encryption schemes have been code-reviewed and tested by analysts.

There's more

In addition to the security risks of custom encryption schemes, it is often times considered "reinventing the wheel". Existing schemes have already been developed and tested, and are often easier to implement than creating your own. For example, the below code can be used to replace our Rot13 example above, resulting in an easier implementation that is more robust than we created.

```
1 import codecs
2 codecs.encode('Hello World', 'rot_13')
```

See Also

More information about the idea of using a common encryption method, but private keys, can be found by researching Kerchoffs's principle[28]. This principle, along with Shannon's maxim, (essentially) states that you should assume your enemy knows how your system works. Instead of protecting how the system functions, you should protect the secret keys to keep your information safe.

Using Encryption Libraries

Because we don't want to create our own encryption scheme, we can provided libraries. These libraries have been designed and reviewed by security experts, so we can generally trust that they are secure. A proper review of the library is important to ensure it uses current encryption methods.

[28] https://en.wikipedia.org/wiki/Kerckhoffs%27s_principle

Getting ready

In Visual Studio Code, browse to **start/CH05/CryptExample.py** and open it for editing.

How to do it

```
#!/usr/bin/env python3
# Script that encrypts/decrypts text using cryptography module
# By Ed Goad
# date: 2/5/2021

# May need 'pip3 install cryptography' or
# 'pip3 install cryptography -U' prior to running
# Import necessary Python modules
from cryptography.fernet import Fernet

def create_key():
    # Generate an encryption key
    # Keep this key secret and store in a secure location
    key = Fernet.generate_key()
    print("Key:", key.decode())

def encrypt(plain_text, key):
    # Convert plain_text and key into bytes for encryption
    plain_text = plain_text.encode()
    key = key.encode()
    # Encrypt the data using the provided key
    cipher_text = Fernet(key).encrypt(plain_text)
    # Convert the cipher_text back to a string
    cipher_text = cipher_text.decode()
    return cipher_text

def decrypt(cipher_text, key):
    # Convert cipher_text and key into bytes
    cipher_text = cipher_text.encode()
    key = key.encode()
    # Decrypt the data using the provided key
    plain_text = Fernet(key).decrypt(cipher_text)
    # Convert plain_text back to a string
    plain_text = plain_text.decode()
    return plain_text

```

```
37 encKey = ""
38 # Prompt the user for the method to use
39 print("Which would you like to do: ")
40 method = input("Create key, Encrypt, Decrypt (c/e/d)? ")
41 method = method[0].lower()
42 # Using the first letter of method, call the correct functions
43 if method == "c":
44     create_key()
45 elif method == "e":
46     # Prompt user for plain_text message and encryption key
47     plain_text = input("Message to encrypt: ")
48     encKey = input("Encryption key: ")
49     # Call the encryption function and print the results
50     cipher_text = encrypt(plain_text, encKey)
51     print(cipher_text)
52 elif method == "d":
53     # Prompt user for cipher_text message and encryption key
54     cipher_text = input("Message to decrypt: ")
55     encKey = input("Encryption key: ")
56     # Call the decryption function and print the results
57     plain_text = decrypt(cipher_text, encKey)
58     print(plain_text)
59 else:
60     # Invalid choice was selected, print error and quit
61     print("Wrong selection, choose C, E, or D")
```

 Don't forget to commit the changes to Git and GitHub

How it works

The script begins on line **#9** by importing the cryptography.fernet module.

On lines **#39**,**40**, the user is prompted for the method they want to perform: creating a key, encrypting a message, or decrypting a message. We trim the user response to just the first letter on line **#41** to simplify our comparison.

Create

```
PS C:\Users\edgoa\source\PythonForCyberSecurity>  & 'C:\Users\edgoa\Ap
582707922\pythonFiles\lib\python\debugpy\launcher' '59995' '--' 'c:\Us
Which would you like to do: Create key, Encrypt, Decrypt (c/e/d)? c
Key: XX1HTW3OEBXXeBR3uQynhLzrNoSnKJ8GjF08HQDmcF4=
PS C:\Users\edgoa\source\PythonForCyberSecurity>
```

Creating encryption key

If create key was chosen, our `create_key()` function is called. On line **#14** a random key is generated, and then printed to the screen on line **#15**.

Encrypt

```
PS C:\Users\edgoa\source\PythonForCyberSecurity>  & 'C:\Users\edgoa\AppData\Local\Programs\Python\Python3
582707922\pythonFiles\lib\python\debugpy\launcher' '60060' '--' 'c:\Users\edgoa\source\PythonForCyberSecu
Which would you like to do: Create key, Encrypt, Decrypt (c/e/d)? e
Message to encrypt: Hello World!
Encryption key: XX1HTW3OEBXXeBR3uQynhLzrNoSnKJ8GjF08HQDmcF4=
gAAAAABgPlpGacS5w2kWYXk35AE74bOrswov7kILwOHdyL1-bzi2WtYLl_jS2ZH2yKLeiI52_jU66Ib3YNa1kW1vXWvtdX05Tg==
PS C:\Users\edgoa\source\PythonForCyberSecurity>
```

Encrypting data

If encrypt was chosen, our `encrypt()` function is called. On lines **#19**,**20** the plain text message and encryption key are encoded from a string into bytes.

On line **#22**, the plain text message is encrypted with the key. This is then decoded from bytes into a string and returned.

Decrypt

```
PS C:\Users\edgoa\source\PythonForCyberSecurity>  & 'C:\Users\edgoa\AppData\Local\Programs\Python\Python39\python.exe' 'c:\
582707922\pythonFiles\lib\python\debugpy\launcher' '60129' '--' 'c:\Users\edgoa\source\PythonForCyberSecurity\Ch05\CryptExa
Which would you like to do: Create key, Encrypt, Decrypt (c/e/d)? d
Message to decrypt: gAAAAABgPlpGacS5w2kWYXk35AE74bOrswov7kILwOHdyL1-bzi2WtYLl_jS2ZH2yKLeiI52_jU66Ib3YNa1kW1vXWvtdX05Tg==
Encryption key: XX1HTW3OEBXXeBR3uQynhLzrNoSnKJ8GjF08HQDmcF4=
Hello World!
PS C:\Users\edgoa\source\PythonForCyberSecurity>
```

Decrypting data

If decrypt was chosen, then our `decrypt()` function is called. On lines **#29**,**30** the cipher text and encryption key are encoded from a string into bytes.

On line **#32**, the cipher text is decrypted using the key. This is then decoded from bytes into a string, and returned.

Lastly, if neither create key, encrypt, or decrypt were specified, an error message is returned to the user.

Use the debugger to walk through the script for more details on how it works.

There's more

Using the wrong key

If the encryption key isn't entered perfectly, the message doesn't decrypt. Below is an example using the same encrypted message and a slightly modified encryption key. In this case the first letter "X" has been replace with a lower-case "x".

```
PS C:\Users\edgoa\source\PythonForCyberSecurity>  & 'C:\Users\edgoa\AppData\Local\Programs\Python\Python39\python.exe' 'c:
582707922\pythonFiles\lib\python\debugpy\launcher' '60277' '--' 'c:\Users\edgoa\source\PythonForCyberSecurity\Ch05\CryptEx
Which would you like to do: Create key, Encrypt, Decrypt (c/e/d)? d
Message to decrypt: gAAAAABgPlpGacS5w2kWYXk35AE74bOrswov7kILwOHdyL1-bzi2WtYLl_jS2ZH2yKLeiI52_jU66Ib3YNa1kW1vXWvtdX05Tg==
Encryption key: xX1HTW3OEBXXeBR3uQynhLzrNoSnKJ8GjF08HQDmcF4=
```

Entering wrong key

```
30      # Decrypt the data using the provided key
31      plain_text = Fernet(key).decrypt(cipher_text)

Exception has occurred: InvalidToken
  File "C:\Users\edgoa\source\PythonForCyberSecurity\Ch05\Cr
    plain_text = Fernet(key).decrypt(cipher_text)
  File "C:\Users\edgoa\source\PythonForCyberSecurity\Ch05\Cr
    plain_text = decrypt(cipher_text, encKey)
```

InvalidToken error

When the script runs, it throws an error stating that the token, or encryption key, is invalid. In a more finalized version of this script, we should watch for errors and handle them in a more friendly way. We would do this using a **try**…**except** block to catch the error codes and report them back to the user.

See Also

In addition to Fernet encryption included in the cryptography library there are several other options available. Below are several options for use in Python.

NOTE: These modules are not endorsed or encouraged for use. They are provided as examples only. When dealing with sensitive information, it is your responsibility to research the available technology and ensure you choose the best option.

- More advanced encryption options in the cryptography library, including asymmetric encryption.
- Fernet encryption: https://pypi.org/project/cryptography/[29]
- Simple-crypt: https://pypi.org/project/simple-crypt/[30]

[29] https://pypi.org/project/cryptography/
[30] https://pypi.org/project/simple-crypt/

- pyAesCrypt: https://pypi.org/project/pyAesCrypt/[31]
- NaCl: https://nacl.cr.yp.to/[32]
- PyNaCl: https://pynacl.readthedocs.io/en/stable/[33]
- Tink: https://github.com/google/tink[34]
- RSA: https://pypi.org/project/rsa/[35]

[31] https://pypi.org/project/pyAesCrypt/
[32] https://nacl.cr.yp.to/
[33] https://pynacl.readthedocs.io/en/stable/
[34] https://github.com/google/tink
[35] https://pypi.org/project/rsa/

Chapter 6: Hacking Passwords

In this chapter we will cover the following:

- How Linux passwords work
- Hashing passwords for Linux
- Creating hashes
- Dictionary Attacks
- Brute-Force Attacks

Introduction

In this chapter we will begin discussing how passwords work in most environments, but specifically focusing on Linux passwords. Once we understand how they are formed, we will create a script that allows us to "hash" our own passwords. Once we have a way to hash passwords, we will review various methods to attack the passwords, enabling us to break into systems.

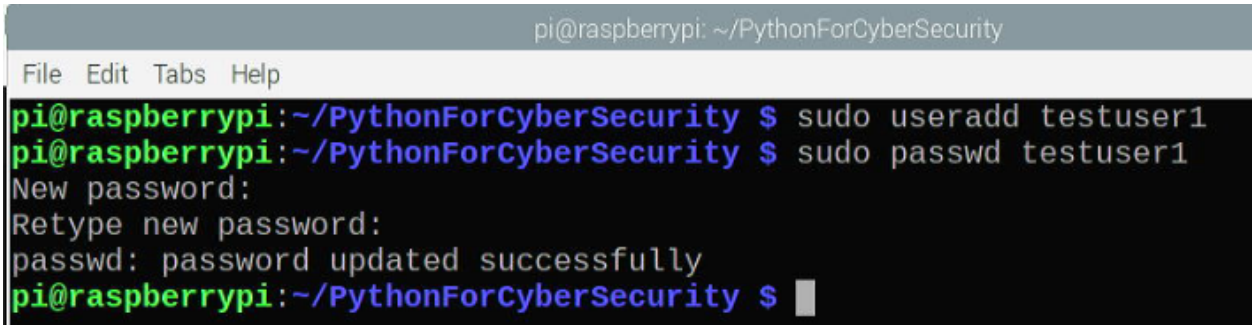
How Linux passwords work

One rule for computer security is to never store a password anywhere. Regardless of how secure your environment is, passwords can be stolen. Therefore, most computer systems store what is known as a "hash" of the password. A hash is a one-way function that takes an input, and creates a hashed output. The hashed output cannot be reversed to identify the original password.

Creating users

In the Raspberry Pi, open a terminal and create a new user by typing `sudo useradd testuser1` (you may be prompted to reenter your password). This creates the user, but doesn't set a password.

To set the password type `sudo passwd testuser1`. When prompted, enter the password **qwerty**, and then again when prompted to retype the password.



Creating test user

So that we have multiple user accounts to attack, repeat the process to create another user, this time named **testuser2**. Once created, provide **testuser2** with the same password **qwerty**.

> **NOTE:** We are intentionally creating users with **BAD PASSWORDS**. We will see shortly just how bad this password is, and why it should never be used. Do not use these passwords on public facing systems.

Files used for authentication in Linux

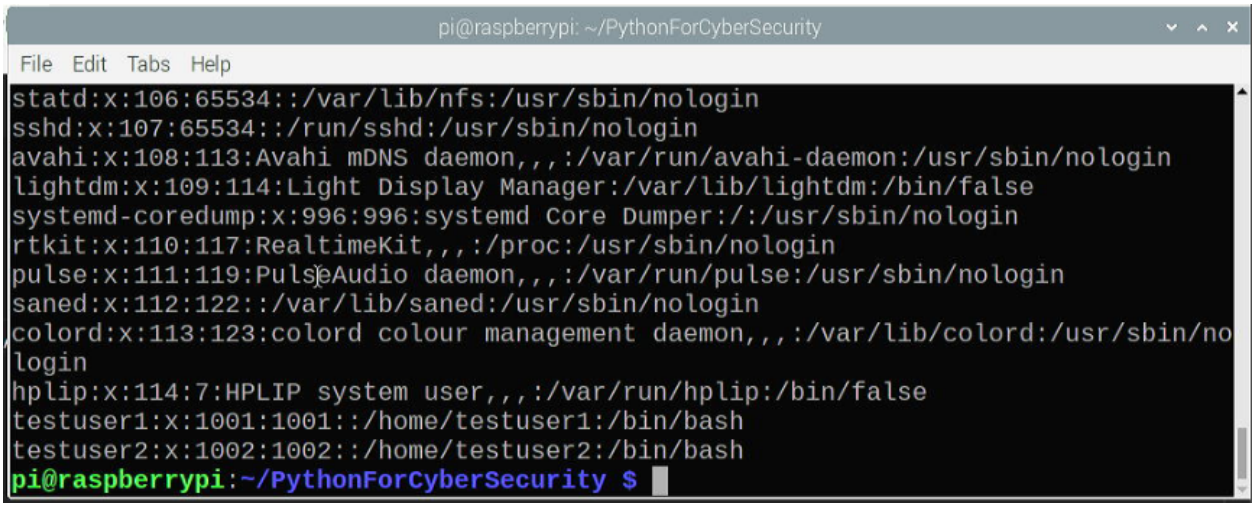
There are 2 files in Linux used for authenticating users: **/etc/passwd** and **/etc/shadow**. There is a third file **/etc/group** that is used for permissions, but not authentication.

/etc/passwd

In the terminal window, type `cat /etc/passwd`

The file is opened and printed to the screen. At the end of the file, we see our 2 new users **testuser1** and **testuser2**. Each line in the **passwd** file is a different user and is composed of 7 fields, separated by colons. As we can see for testuser1:

1. **testuser1** - login name.
2. **x** - password hash (if present).
3. **1001** - User ID number.
4. **1001** - Group ID number.
5. <empty> - user description field.
6. **/home/testuser1** - User's home directory.
7. **/bin/bash** - User shell.



/etc/passwd file

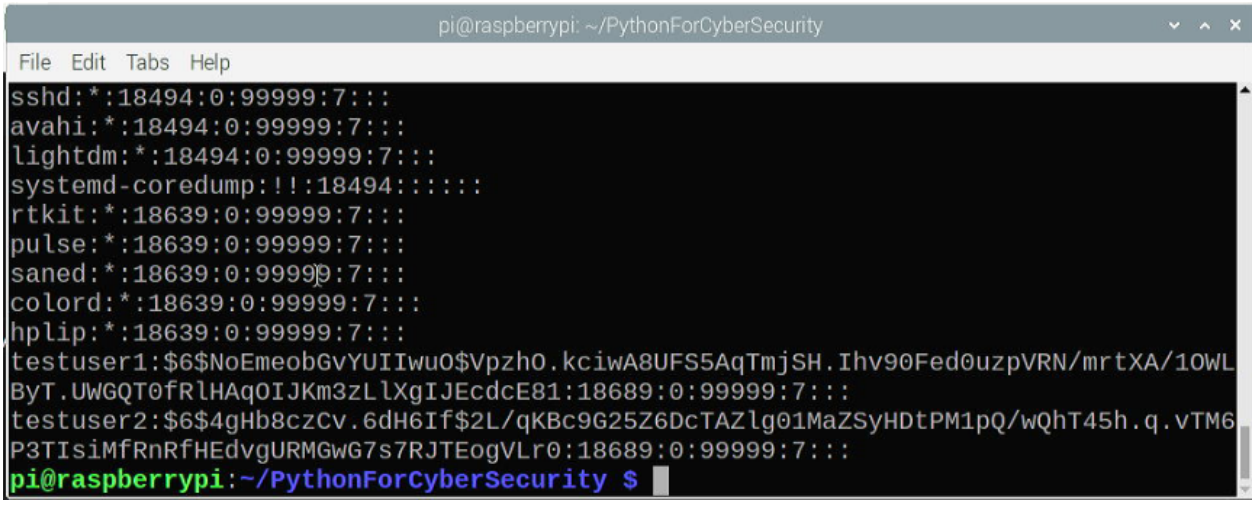
/etc/shadow

Of particular note, in the **/etc/passwd** file is that the **password** field is empty. When originally designed, this field included password hashes, but were eventually moved to the **/etc/shadow** file because of security issues (all users need to have read-access to **/etc/passwd**). This new file is known as the "shadow password" file, and is readable by only the **root** user and the **shadow** group.

To view the shadow password file, in the terminal window type **sudo cat /etc/shadow**

Similar to the passwd file, the shadow file is composed of multiple fields, each separated by colons. For **testuser1** we see the following information:

1. **Testuser1** - login name.
2. **6noEm**... - Salted and Hashed password, or other symbol.
3. **18689** - days since epoch of last password change.
4. **0** - days until password change allowed.
5. **99999** - days until password change is required.
6. **7** - Days warning before password expire.
7. <empty> - Lock after n days of no login.
8. <empty> - Days since epoch when account expired.
9. <empty> - Reserved.



/etc/shadow file

Salting our passwords

One of the most interesting aspects of the **/etc/shadow** file that we can see is that **testuser1** and **testuser2** have different password hashes. When we created these users, we created them with the same password of **qwerty**, which suggests these should have the same hash. This change is due to what is known as "salting" the passwords.

In simple terms, a "salt" is random data added to a password to make it return a unique hash. We can see in our example how our 2 test users have the same password, but the hashes are unique. If this file was available to our users, there would be no (easy) way for them to know they have the same password.

The salted password is composed of 3 main components, separated by dollar signs (**$**):

1. The first few characters define the hashing algorithm: (**$6** in the example above)

- $1=MD5
- $2a=Blowfish
- $2y=eksblowfish
- $5=SHA-256
- $6=SHA-512

2. A SALT which is between the 2 dollar signs ($ **$NoEmeobGvYUIIwuO** $ in the example above).
3. The hash of the salted password (**VpzhO.kciwA8**……).

NOTE: Not all hashing algorithms are supported by all machines.

There's more

We created our users with the **useradd** and **passwd** commands. However, the /etc/passwd and /etc/shadow files can be edited directly if the right permissions are available. By the right person, this can be used to maliciously reset passwords, create users, or "clone" users.

See also

https://en.wikipedia.org/wiki/Passwd[36]

https://en.wikipedia.org/wiki/Hash_function[37]

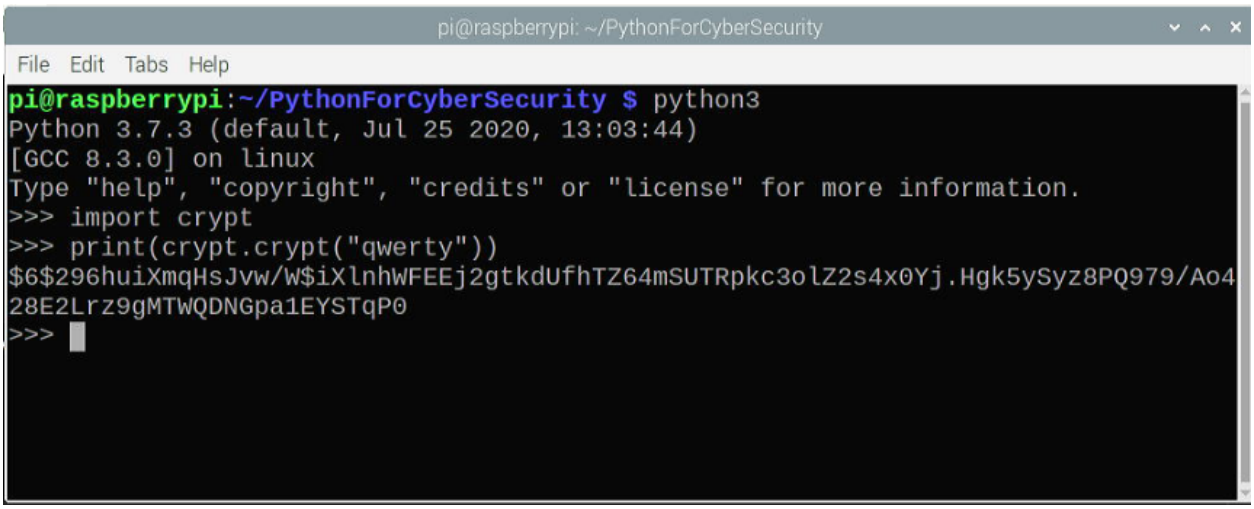
Hashing passwords for Linux

Now that we have some understanding about how passwords are hased and stored in Linux, we can now look at how to generate those hashes ourselves.

Getting ready

We can easily create password hashes in Python using the **crypt** module. We will start by using interactive Python to view how this works.

```
1 import crypt
2 print(crypt.crypt("qwerty"))
```



Hashing our password

[36] https://en.wikipedia.org/wiki/Passwd
[37] https://en.wikipedia.org/wiki/Hash_function

Here we import the crypt Python module and then call the `crypt.crypt()` method to hash our password. The hash is returned below, and by the **$6** at the beginning, we can see that the module defaults to the more secure SHA-512 hashing algorithm.

> **NOTE**: The Python crypt module is dependent on the Linux operating system. If you wish to do this process under Windows, you will need to use a different module such as passlib.hash. You can find more information about this by searching online.

By pressing the up arrow on the keyboard, we see the prior print statement is shown. If we hit enter, the command will be executed again. We can do this several times to execute the same command multiple times.

Hashing a password with different results

As we can see, consecutive calls to the **crypt.crypt()** method using the same password results in different hashes. This is because the **crypt()** method automatically uses different salts for each hash.

We will start by creating our first hashing script that shows the various hashing methods available in the **/etc/shadow** file.

In Visual Studio Code, browse to **start/CH06/CreateHash1.py** and open it for editing.

How to do it

CreateHash1.py

```
1  #!/usr/bin/env python3
2  # Script that hashes a password
3  # By Ed Goad
4  # date: 2/5/2021
5
6  # Import Python modules
7  import crypt
8
9  # Prompt user for plain-text password
10 plain_pass = input("What is the password? ")
11
12 # Print out hashes
13 print("MD5       : {0}".format(crypt.crypt(plain_pass,"$1$")))
14 print("Blowfish  : {0}".format(crypt.crypt(plain_pass,"$2a$")))
15 print("eksblofish: {0}".format(crypt.crypt(plain_pass,"$2y$")))
16 print("SHA-256   : {0}".format(crypt.crypt(plain_pass,"$5$")))
17 print("SHA-512   : {0}".format(crypt.crypt(plain_pass,"$6$")))
```

> **NOTE:** The spaces in the print statements are added for readability only. They aren't required to be perfect for the script to function properly.

 Don't forget to commit the changes to Git and GitHub

How it works

The script starts on line **#7** by importing the Python crypt library. On line **#10**, the user is prompted for the password to hash.

Lines **#13 - 17** use the `crypt.crypt()` method to hash the password. The second parameter of the method includes the hash type (1, 2, 2a, and so on). These hashes are printed to the screen.

As you can see from the output, not all hash types are supported on all systems. Additionally, the increased length of some hash types, in general, makes them more secure than others.

```
pi@raspberrypi:~/PythonForCyberSec $  /usr/bin/env /usr/bin/python3 /h
ome/pi/.vscode/extensions/ms-python.python-2021.2.633441544/pythonFile
s/lib/python/debugpy/launcher 36411 -- /home/pi/PythonForCyberSec/end/
CH06/CreateHash1.py
What is the password? qwerty
MD5       : $1$$hRSiYXuj/e1Pz5uyuOt3w0
Blowfish  : None
eksblofish: None
SHA-256   : $5$$3S3G2JRYR6TXhF3MXbONJzeKzwUvIucaktXObQRQM08
SHA-512   : $6$$1zOyLGD8fsSEsjb3npfpnEW2O/bLC5Ksb8FfvubaVDiy/qUrCCPAq1
ZHKaXeCCrfyKdez4WZX9dn7m5OWiYIz0
pi@raspberrypi:~/PythonForCyberSec $
```

Output of script

See also

More information about the **/etc/passwd** and **/etc/shadow** files can be found on Wikipedia.

https://en.wikipedia.org/wiki/Passwd[38]

Creating the same hash twice

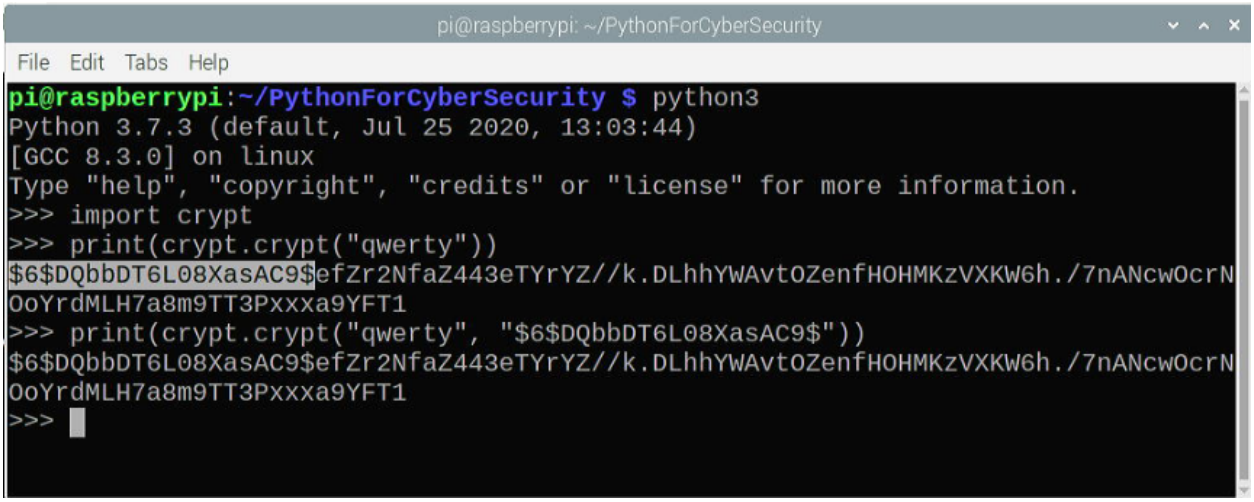
Because the salt is adding unique and random data to the password when hashing, the salt is responsible for having different hashes. If we can use the same password and salt, then we can create the same hash.

Getting ready

In interactive Python, we will use the same `crypt.crypt()` method as did previously. Then, we can copy the hash type and salt to run the `crypt.crypt()` method again.

```
1 import crypt
2 print(crypt.crypt("qwerty"))
3 print(crypt.crypt("qwerty","$6$DQbbDT6L08XasAC9$"))
```

[38] https://en.wikipedia.org/wiki/Passwd



Hashing with specified salt

Here we can see using the `crypt.crypt()` method to hash the password **qwerty**. Once the hash is returned, we can copy the hash type and salt that is shown and include that as a parameter to another call to the `crypt.crypt()` method. Because the second call uses the same password and salt, the resultant hash is the same.

Now that we can include a known salt in our hashing process, we can extend our CreateHash1.py script to use the known hash.

In Visual Studio Code, browse to **start/CH06/CreateHash2.py** and open it for editing.

How to do it

CreateHash2.py

```
#!/usr/bin/env python3
# Script that hashes a password with provided salt
# By Ed Goad
# date: 2/5/2021

# Import Python modules
import crypt

# Prompt user for plain-text password
plain_pass = input("What is the password? ")
salt = input("What is the salt? ")

# Print out hashes
print("MD5       : {0}".format(crypt.crypt( \
    plain_pass,"$1$" + salt)))
```

```
16 print("Blowfish  : {0}".format( \
17     crypt.crypt(plain_pass,"$2$" + salt)))
18 print("eksblofish: {0}".format( \
19     crypt.crypt(plain_pass,"$2a$" + salt)))
20 print("SHA-256   : {0}".format( \
21     crypt.crypt(plain_pass,"$5$" + salt)))
22 print("SHA-512   : {0}".format( \
23     crypt.crypt(plain_pass,"$6$" + salt)))
```

> Note the use of the backslash (\ ) on several lines. This is used in Python to continue a line onto the next line in the editor. This can be used to simplify the writing of a script by including it on several lines. This is not necessary, but can improve the readability of the code.

 Don't forget to commit the changes to Git and GitHub

How it works

This script is the same as the prior, except for the inclusion of the salt. The user is prompted for the salt on line **#11**, and it is then added to the `crypt.crypt()` method. When the salt is provided, that is used for hashing the passwords.

If we use this script with the password **qwerty** and one of the salts found in our **/etc/shadow** file, we will see that we can create the same hash. The SHA-512 output can be matched against the shadow file to confirm.

```
pi@raspberrypi:~/PythonForCyberSec $  /usr/bin/env /usr/bin/python3 /home/p
i/.vscode/extensions/ms-python.python-2021.2.633441544/pythonFiles/lib/pyth
on/debugpy/launcher 41919 -- /home/pi/PythonForCyberSec/end/CH06/CreateHash
2.py
What is the password? qwerty
What is the salt? NoEmeobGvYUIIwuO
MD5       : $1$NoEmeobG$bvS9U5RyA5UaHfIhHqmdk0
Blowfish  : None
eksblofish: None
SHA-256   : $5$NoEmeobGvYUIIwuO$WbLze7xd/s9lktkBuqdjrEB0kAW6hRFUtZ0GzWC3Gh.
SHA-512   : $6$NoEmeobGvYUIIwuO$VpzhO.kciwA8UFS5AqTmjSH.Ihv90Fed0uzpVRN/mrt
XA/1OWLByT.UWGQT0fRlHAqOIJKm3zLlXgIJEcdcE81
pi@raspberrypi:~/PythonForCyberSec $
```

Hashing results with salt

There's more

This is similar to the way some network-based authentication services work. However, instead of sending the password, or the password hash over the network, a salt is sent instead.

1. When the password is first saved on the authentication server, it is hashed using a known/-common salt and the hash is saved.
2. When a user logs on to a remote PC, the authentication server re-hashes the hashed password with a new salt.
3. The new salt is sent across the network to the remote PC.
4. The remote PC uses the typed password and hashes it using the common salt, and then hashes it again using the salt received across the network.
5. The final hash is sent from the remote PC to the authentication server. If both resultant hashes are the same, then the password was correct.

See also

Salting hashes in Wikipedia[39]

Dictionary Attacks

At this point, if we review what we have learned, we will see that we now know:

- Where passwords are stored in Linux
- How passwords are stored in Linux (hash type and salt included)
- How passwords are salted and hashed (technically)
- How to create our own salted hash
- How to create the same hash as found in the **/etc/shadow** file

Getting ready

Because Linux ultimately authenticates using the hash, instead of the password, we need a method to recreate the hash using the same salt and hash type. Assuming we have access to the **/etc/shadow** file, we have 3 critical items to being our attack: 1) the hash type, 2) the salt, and 3) the final hash. All that remains is to identify some input that can result in the same final hash. There are a few ways to achieve the same final hash:

1. Guessing passwords - Using the identified hash type and salt, hashing thousands or millions of passwords to see if they match. This is also known as a Dictionary Attack.
2. Brute force - Hashing every possible character combination and comparing the results to the final hash.
3. Hashing collision - A unlikely scenario where 2 different inputs generate the same hash.

[39] https://en.wikipedia.org/wiki/Salt_(cryptography)

In this example, we will see how option #1 works. For this option, we will need a list of common passwords, often known as a "dictionary file". With only a bit of searching you can find several dictionary files online. One example dictionary file can be found on GitHub[40].

This GitHub repository contains a curated collection of the most common passwords found to be in use, ordered by their frequency. If we look at the first file, **10-million-password-list-top-100.txt**, we can see that it has the top 100 most commonly used passwords found online.

Online password files

From this list we can see the #1 most commonly used password is **123456**. Of all the known computer attacks and leaked passwords, the most commonly used was 123456, and the #2 most commonly used password is **password**. These password lists, or dictionary files, are frequently used by security professionals and hackers to attack user accounts because these are known to be the most likely used passwords. This particular file is only the top 100 passwords, but you can see from the repository, we can easily get the top 10,000,000 passwords.

> **NOTE**: Looking at the list, note where the password for our **testuser1** and **testuser2** lies. This is why these passwords should never be used.

The **top10.txt** and **top1000.txt** files are already included in the cloned repository from GitHub.

In Visual Studio Code, browse to **start/CH06/DictionaryAttack.py** and open it for editing.

How to do it

DictionaryAttack.py

[40] https://github.com/danielmiessler/SecLists/tree/master/Passwords/Common-Credentials

```
#!/usr/bin/env python3
# Script that performs a dictionary attack
# against known password hashes
# Needs a dictionary file to run. Suggested to use
# https://github.com/danielmiessler/SecLists
# /tree/master/Passwords/Common-Credentials
# By Ed Goad
# date: 2/5/2021

# Import necessary Python modules
import crypt

def test_password(hashed_password, \
    algorithm_salt, plaintext_password):
    # Using the provided algorithm/salt and
    # plaintext password, create a hash
    crypted_password = crypt.crypt( \
        plaintext_password, algorithm_salt)
    # Compare hashed_password with the just created hash
    if hashed_password == crypted_password:
        return True
    return False


def read_dictionary(dictionary_file):
    # Open provided dictionary file
    # and read contents into variable
    f = open(dictionary_file, "r")
    message = f.read()
    return message

# Load dictionary file and prompt for hash and algorithm/salt
password_dictionary = read_dictionary("top10.txt")
hashed_password = input("What is the hashed password? ")
algorithm_salt = input("What is the algorithm and salt? ")

# For each password in dictionary file,
# test against hashed_password
for password in password_dictionary.splitlines():
    result = test_password(hashed_password, \
        algorithm_salt, password)
    if result:
        # If a match is found, print it and quit
```

```
44          print("Match found: {0}".format(password))
45          break
```

 Don't forget to commit the changes to Git and GitHub

How it works

The script starts on line **#33** by calling our `read_dictionary()` function with the specified dictionary file. This function, starting on **#25**, opens the file, reads the contents, and then returns the contents. This is stored in the **password_dictionary** list variable.

On lines **#34**,**35** the user is prompted for a hashed password, the algorithm, and the salt.

On line **#39** we begin a for loop to cycle through each of the passwords in **password_dictionary**. The `splitlines()` method is used to separate the contents of the password file on separate lines.

On line **#40** we call our `test_password()` function with the hashed password, algorithm and salt, and plain-text password. On line **#17**, the `crypt.crypt()` method is called to hash the plain-text password using the same algorithm and salt.

On line **#20**, the original **hashed_password** is compared against the newly hashed **crypted_password**. If they match, we return from the function with a **True** value. Otherwise, we return from the function with a **False** value.

On line **#42**, if the value returned from `test_password()` is true, then the match is printed to the screen, and the script exist.

> **NOTE:** For testing you may want to limit the dictionary to the **top10.txt** password file. Once the script is confirmed functional, you can use the larger files against more complex passwords.

There's more

Extend guessing script to include:

- Prompt user for location of shadow file.
- Create a function that separates lines into their separate component: name, hash type, salt, hash.
- Automatically attack all users in the file.

See also

Brute-Force Attacks

A brute-force attack is similar to the dictionary attack, except instead of using a dictionary file, it attempts every possible combination of letters, numbers, and symbols.

As a simplified example of a brute-force attack, we can imagine a combination bicycle lock. The lock has 4 slots for numbers, each ranging from 0 through 9.

Bicycle lock

To perform a brute-force attack against this lock, we can start by attempting 0000. Assuming that combination didn't work, then we increment by 1 and attempt 0001. We continue this incrementing of values until we reach 9999, or the correct answer.

In the bicycle lock example, we know there are 4 slots total, meaning there are a total or 10,000 possible combinations. If however, we don't know the length of the combination, we start at 0 and keep going until 9,999,999,999,999 or whatever maximum value we decide.

Getting ready

In this attack we will target the following hash. This hash is included in the script for ease of access.

```
$6$G.DTW7g9s5U7KYf5$QFcHx0/J88HV/Q0ab653gfYQ1KyNGx5HRhDQYyai2ZUy7Aw4tyfJ6/kI6kllfXl0\
DyS.LuaUJvqnlIn2fVM5F0
```

We know this PIN is somewhere between 0 and 100,000, but we don't know how many digits it is.

In Visual Studio Code, browse to **start/CH06/BruteForceAttack.py** and open it for editing.

How to do it

BruteForceAttack.py

```
#!/usr/bin/env python3
# Example brute-force attacker
#By Ed Goad
# date: 2/5/2021

# NOTE: this example is limited to numbers.
# There are no letters or symbols in this example
# Suggested to start by debugging to show how
# brute force walks through all available options

import crypt

def test_password(hashed_password, algorithm_salt, \
    plaintext_password):
    # Using the provided algorithm/salt
    # and plaintext password, create a hash
    crypted_password = crypt.crypt(plaintext_password, \
        algorithm_salt)
    # Compare hashed_password with the just created hash
    if hashed_password == crypted_password:
        return True
    return False

hashed_password = "$6$G.DTW7g9s5U7KYf5$QFcHx0/J88HV/Q0ab653"
hashed_password += "gfYQ1KyNGx5HRhDQYyai2ZUy7Aw4tyfJ6/kI6kl"
hashed_password += "lfXl0DyS.LuaUJvqnlIn2fVM5F0"
algorithm_salt = "$6$G.DTW7g9s5U7KYf5$"

for password in range(100000):
    result = test_password(hashed_password, \
        algorithm_salt, str(password))
    if result:
        print("Match found: {0}".format(i))
        break
```


Don’t forget to commit the changes to Git and GitHub

How it works

This example is similar to the Dictionary attack and uses same `test_password()` function. Instead of reading from a file, we do a `for i in range(100000)`. This example is simplified to only numbers. Use debugger if needed to highlight how each number in turn is being tested.

 Use the debugger to walk through the script for more details on how it works.

Theres more

More complex brute forcing includes larger character sets such as those shown below. This obviously complicates and slows the process, explaining why this may be the last choice in an attack.

- abcdefghijklmnopqrstuvwxyz
- ABCDEFGHIJKLMNOPQRSTUVWXYZ
- 0123456789
- 0123456789abcdef
- 0123456789ABCDEF
- «space»!”#$%&’()*+,-./:;<⇒?@[]^_‘~
- Unicode: 0x00 - 0xff

There are preexisting tools capable of performing brute-force attacks for specific services and applications. These tools can be faster than a Python script due to the use of optimized code, but possibly are not as flexible for all situations.

See also

https://en.wikipedia.org/wiki/Brute-force_attack[41]

https://hashcat.net/wiki/doku.php?id=mask_attack[42]

[41] https://en.wikipedia.org/wiki/Brute-force_attack
[42] https://hashcat.net/wiki/doku.php?id=mask_attack

Chapter 7: Log Files

In this chapter we will cover the following:

- Creating a web server
- Log file formats
- Reading log files
- Intro to Regular Expressions
- Finding interesting items in Apache Web Server logs
- Finding potential hacking attempts

Introduction

This chapter discusses the basics of working with log files. Utilizing Python and Regular Expressions, we will search log files for interesting information that we can use in the future.

Local web server

To being our discussion about working with log files, we can start by creating files. One popular log file format is from the Apache web server.

How to do it

On the Raspberry Pi, open a terminal and enter the following commands to install and start a web server.

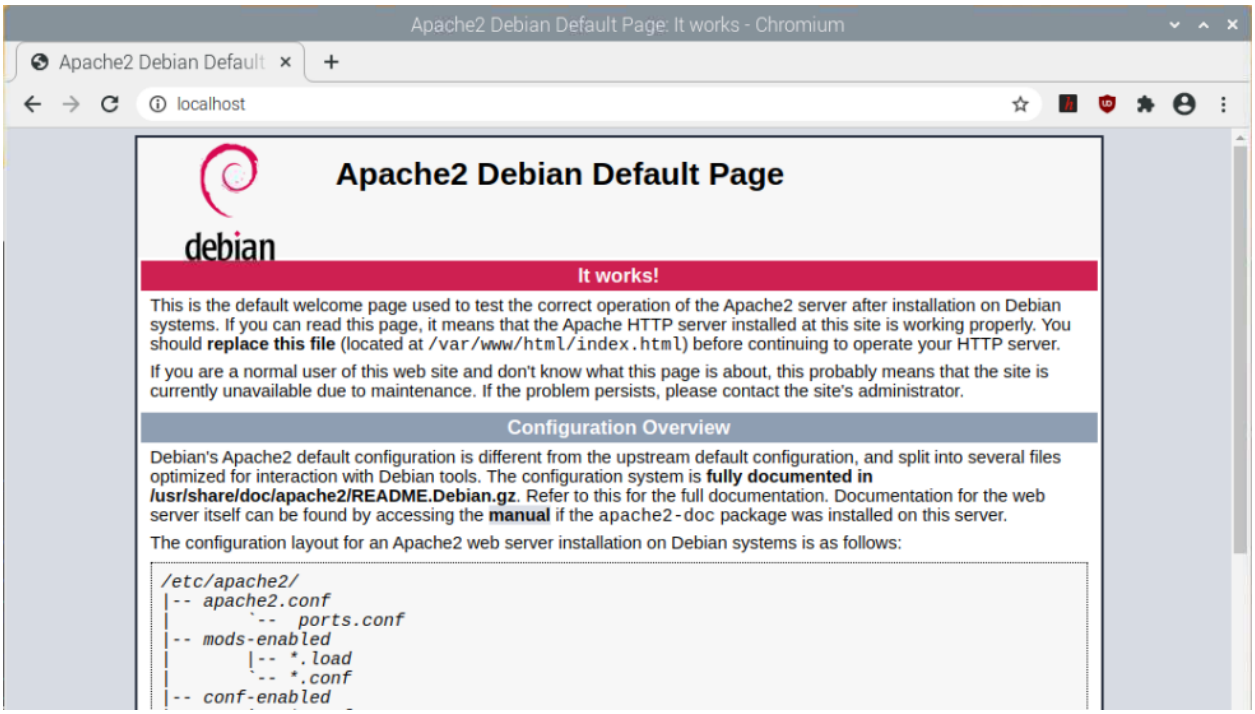
```
sudo apt update
sudo apt -y install apache2
sudo systemctl restart apach2
sudo systemctl enable apache2
```

> **NOTE:** After the first command, you may be prompted to reenter your password

Once the web server has been installed, open browser on Raspberry Pi and enter http://localhost[43] in the address bar.

[43] http://localhost/

NOTE: This address will only work on the Raspberry Pi. To access it from a different computer, you need the IP address or DNS name.

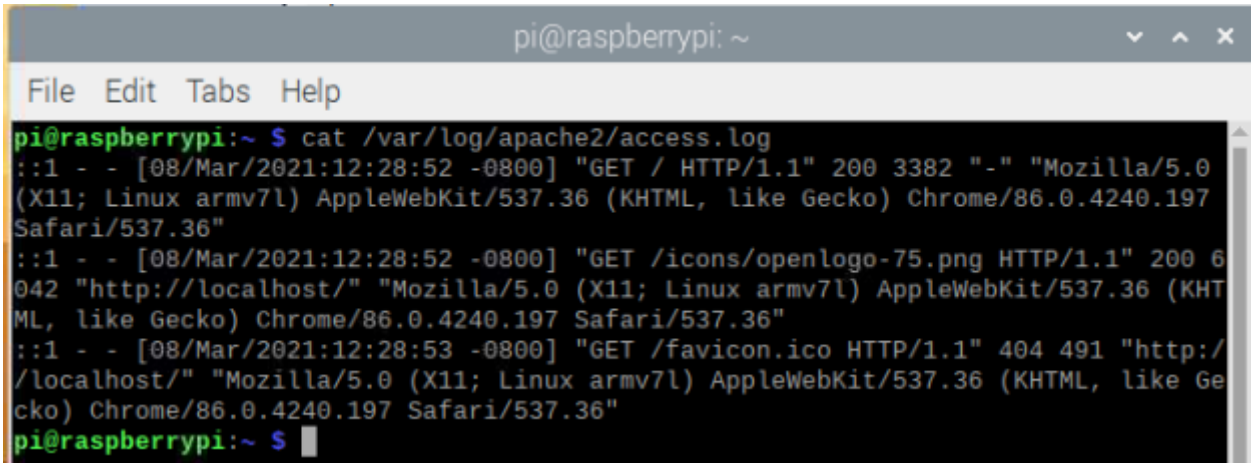


Default Debian Apache website

Apache log files are stored at **/var/log/apache2** for the Raspberry PI. This location may vary if you are using a different operating system.

Once the web server has been accesses at least once, we can view the contents of the log file. To view the log file, type the following command

```
cat /var/log/apache2/access.log
```



Viewing the access.log

You will probably only a few lines in there. Each line is a request from a client and is organized using a common format described at the apache web site[44]. For example, one line from my log file says:

::1 - - [08/Mar/2021:12:28:52 -0800] "GET /icons/openlogo-75.png HTTP/1.1" 200 6042 "http://localhost/" "Mozilla/5.0 (X11; Linux armv7l) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/86.0.4240.197 Safari/537.36"

According to the Common and Combined Log Formats, this can be described as:

Value	Description
::1	Client IP address
-	Identity of client machine
-	Identity of client user
[08/Mar/2021:12:28:52 -0800]	Time the request was received
"GET /icons/openlogo-75.png HTTP/1.1"	The request from the client
200	Status code sent to client
6042	Bytes sent to client
"http://localhost/"	Referrer page or site
"Mozilla/5.0 (X11; Linux armv7l) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/86.0.4240.197 Safari/537.36"	User Agent of client accessing page, normally including browser and OS

As we may imagine, these 9 fields can hold a wealth of information. Using this information, an administrator can identify commonly requested pages, where errors are occurring, details about who is visiting the site, and where they are coming from. A security engineer can use the same information to identify hacking attempts, access from undesirable locations, and other potentially malicious activity.

[44] https://httpd.apache.org/docs/2.4/logs.html#accesslog

There's more

More information about the Apache log file format can be found at:

https://httpd.apache.org/docs/2.4/logs.html[45]

Download sample files from Internet

While we can create our own log files for analysis, using previously created log files can simplify our testing and learning process. If you cloned my original GitHub repository, one anonymized file is already available to you in **start/CH07/access.log**.

Additionally, many people and organizations have published their logs online for public view, including NASA and other governmental agencies. The following searches may be able to help us find these files online.

https://www.sec.gov/dera/data/edgar-log-file-data-set.html[46]

https://www.google.com/search?q=inurl%3Aaccess.log+filetype%3Alog[47]

Simple evaluations - read line-by-line

Getting ready

When reviewing web server log files, one of the first things an administrator or security engineer may look for is **Error 404 - Page Not Found**. An HTTP 404 error means someone was looking for a resource, but didn't find it. This could be due to typos, misconfigured links, deleted/moved content, desired content, or attempts to find hidden resources or possible server attacks.

In Visual Studio Code, browse to **start/CH07/Find404.py** and open it for editing.

How to do it

Find404.py

[45] https://httpd.apache.org/docs/2.4/logs.html
[46] https://www.sec.gov/dera/data/edgar-log-file-data-set.html
[47] https://www.google.com/search?q=inurl%3Aaccess.log+filetype%3Alog

```
1  #!/usr/bin/env python3
2  # Script that scans web server logs for 404 errors
3  # By Ed Goad
4  # date: 2/5/2021
5
6  # Prompt for file to analyze
7  log_file = input("Which file to analyze? ")
8
9  # Open file
10 f = open(log_file, "r")
11
12 # Read file line by line
13 while True:
14     line = f.readline()
15     if not line:
16         break
17     # Check for 404
18     if "404" in line:
19         print(line.strip())
20
21 # Close file
22 f.close()
```

 Don't forget to commit the changes to Git and GitHub

How it works

This script begins on line **#7** by prompting the user for the log file to analyze. On line **#10** the file is then opened for reading.

On line **#13**, a while loop is started. The `while True:` line creates a never-ending loop that will continue unless otherwise terminated.

On line **#14** `f.readline()` method is called to read a single line of text from the log file. This is useful if the log files are too large to read into memory all at once. On line **#15** we check the value of **line** to confirm it has been populated, and if not then **break** out of the loop. This is one way to validate if we have reached the end of the file.

On line **#18**, we use the **in** statement to see if our line contains the string "404". If a match is made, the line is printed to the screen.

Lastly, when finished reading the file, `f.close()` method is used to close the file.

 Use the debugger to walk through the script for more details on how it works.

Intro to Regular Expressions

Regular Expressions, sometimes referred to as regex, is an extremely powerful tool to find and extract information. Making use of complicated patterns, regex can be used to automatically identify content. You have likely seen regex in use on the internet when filling out forms that alert to an improper email addresses, incomplete phone numbers, bad credit card numbers and more.

Regular expressions can be extremely complicated and go beyond the scope of this book, indeed, entire books have been written dealing with regex. Here, we will instead introduce the concept of regex, and show a few examples of how they can be useful in Python.

Regex examples

For our first regex example, we can use one of the many online regular expression testers. Using a search engine for "online regex tester" will yield several results such as https://regex101.com/[48], https://regexr.com/[49], and https://www.regexpal.com/[50]. For these examples, I will be using https://regexr.com/[51].

> **Note:** The example expressions given below are overly simplified. More research should be used before using them in production.

Email

As mentioned previously, one of the common uses of regular expressions is to validate email addresses. There are multiple patterns that can be used (try searching online for "regular expression email"), and unfortunately only the most complicated will match 100% of the possible email addresses.

One simplified email regular expression is `[\w-]+@([\w-]+\.)+[\w-]+`

How it works

The first section in the expression, `[\w-]+`, matches the "username" portion of the email. For the sample email of **testuser@example.com**, this expression would match **testuser**.

Regex	Description
`[]`	**Character set** - match any character inside the brackets
`\w`	**Word** - Any alphanumeric character (letter, number, underscore)
-	The dash "-" character
+	**Quantifier** - match 1 or more of the preceding

[48] https://regex101.com/
[49] https://regexr.com/
[50] https://www.regexpal.com/
[51] https://regexr.com/

The next character, `@` , is the literal @ sign. This character is required in all email addresses, so the regular expression is checking for the existence of this character.

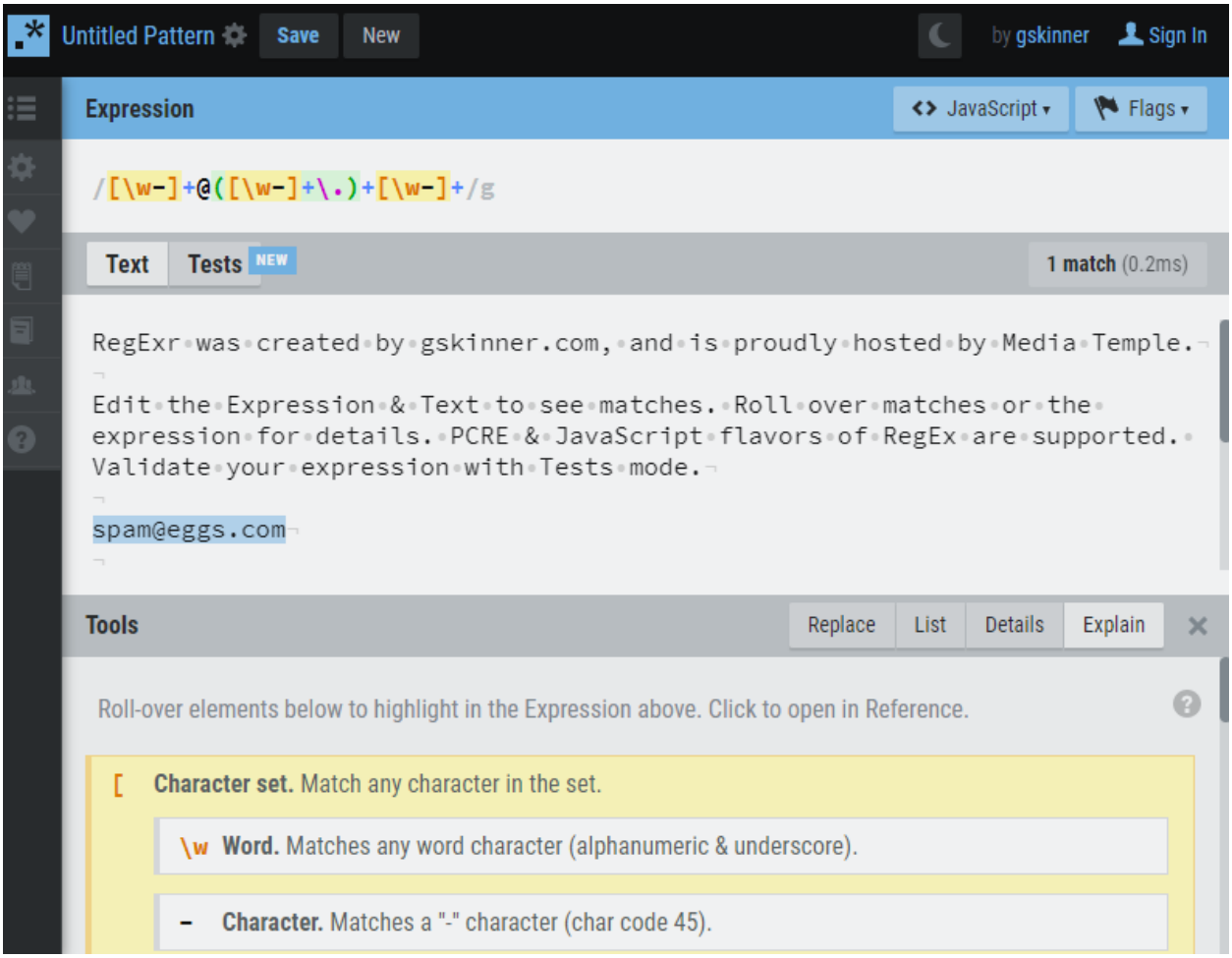
The next section in the expression, `([\w-]+\.)+`, matches the "domain" portion of the email. For the sample email of **testuser@example.com**, this expression would match **example**.

Regex	Description
`()`	**Capture group** - Groups part of the regular expression together
`[]`	**Character set** - match any character inside the brackets
`\w`	**Word** - Any alphanumeric character (letter, number, underscore)
-	The dash "-" character
+	**Quantifier** - match 1 or more of the preceding
`\.`	The period "." character. The preceding backslash () is an "escape character" that instructs regex to treat the character as a period.
+	**Quantifier** - match 1 or more of the preceding

The final section in the expression, `[\w-]+`, matches the "top-level domain" portion of the email. This expression is exactly the same as the "username" portion. For the sample email of **testuser@example.com**, this expression would match **com**.

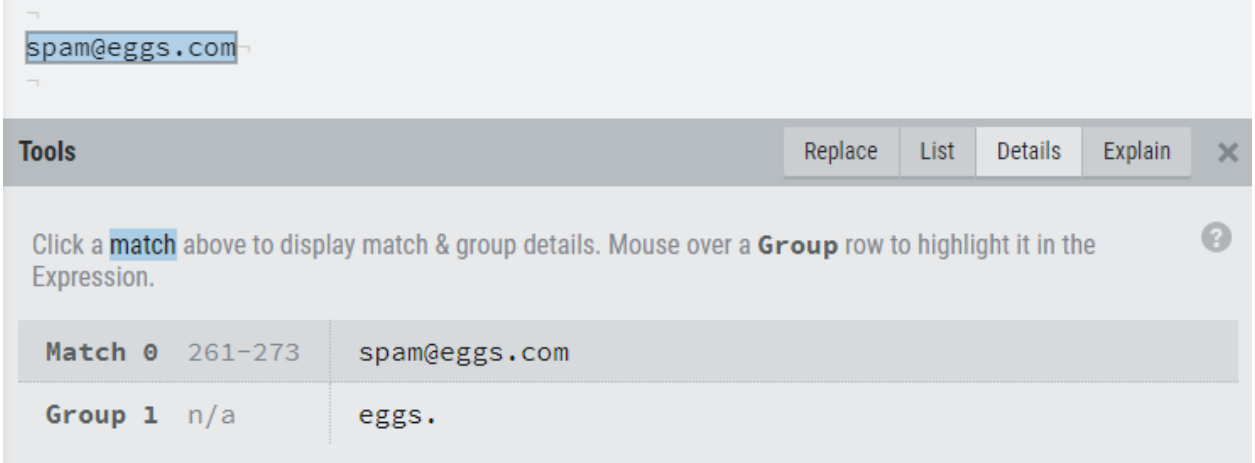
To test our email regex expression, open a web browser to https://regexr.com/[52]. In the **Expression** section, enter in the expression from above. In the **Text** section, along with the existing text, enter an email address. As the email is completed, the web page will automatically highlight the match.

[52] https://regexr.com/



regexr.com with email

More details about the match can be found on the **Explain** tab toward the bottom of the page. Each character, or group of characters in the expression are described. As you move your mouse over the descriptions, the character or group is highlighted.



regexr.com matches

If you click the **Details** tab at the bottom of the page and select the email address above, we can see the match and group identified. While not immediately useful for this example, this can be helpful as we have more complicated patterns and processes in the future.

Credit Card

Another common use for regular expressions is to quickly validate credit card numbers. Web sites will frequently use this to confirm a customer has entered the card number entirely and correctly.

One simplified regular expression for a Visa is `4[0-9]{3}(-[0-9]{4}){3}`

How it works

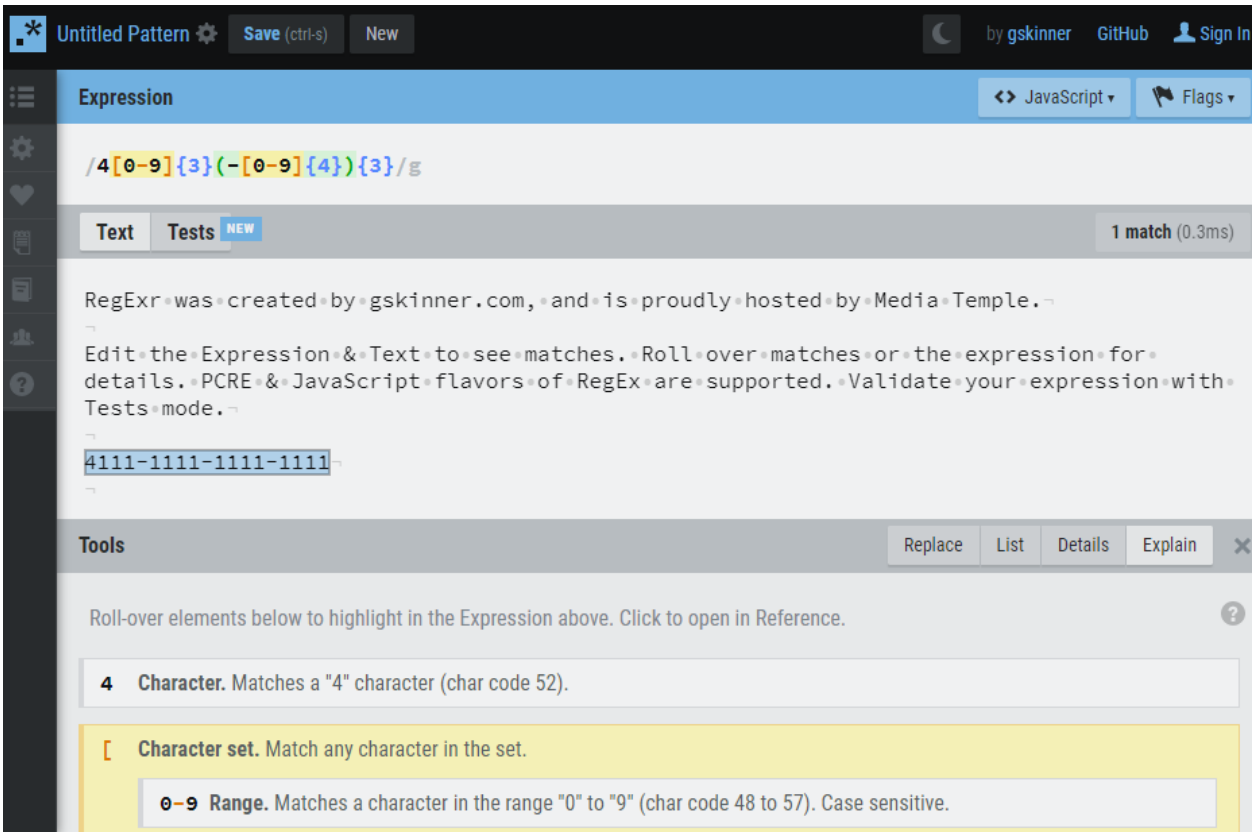
The first section, `4[0-9]{3}`, matches the first 4 digits of the Credit Card number.

Regex	Description
`4`	The number 4 (all Visa cards start with 4)
`[]`	**Character set** - match any character inside the brackets
`0-9`	**Range** - matches any number from 0 through 9
`{3}`	**Quantifier** - requires 3 of the preceding

The remaining section, `(-[0-9]{4}){3}`, matches the dashes and remaining digits.

Regex	Description
`()`	**Capture group** - Groups part of the regular expression together
-	The dash "-" character
`[]`	**Character set** - match any character inside the brackets
`0-9`	**Range** - matches any number from 0 through 9
`{4}`	**Quantifier** - requires 4 of the preceding
`{3}`	**Quantifier** - requires 3 of the preceding

To test our expression, open a web browser to https://regexr.com/[53]. In the **Expression** section, enter in the expression from above. In the **Text** section, enter a fake Visa number with dashes (suggested **4111-1111-1111-1111**) along with the existing text. As the number is completed, the web page will automatically highlight the match.



regexr.com matching credit card

Social Security Number

Another common use of regular expressions is to review large data sets for Personally Identifiable Information (PII). An example of this would be viewing logs for mail servers, web servers, or firewalls for Social Security Numbers (SSN).

One simplified regular expression for SSN is `\d{3}-\d{2}-\d{4}`

How it works

[53] https://regexr.com/

Regex	Description
`\d`	**Digit** - any number from 0 through 9
`{3}`	**Quantifier** - requires 3 of the preceding
-	The dash "-" character
`\d`	**Digit** - any number from 0 through 9
`{2}`	**Quantifier** - requires 2 of the preceding
-	The dash "-" character
`\d`	**Digit** - any number from 0 through 9
`{4}`	**Quantifier** - requires 4 of the preceding

To test our expression, open a web browser to https://regexr.com/[54]. In the **Expression** section, enter in the expression from above. In the **Text** section, enter a fake SSN with dashes (suggested **000-00-0000**) along with the existing text. As the number is completed, the web page will automatically highlight the match.

There's more

While the examples given work in our scenarios, there is still several ways that they fall short. For instance, we assumed the credit card and social security numbers had dashes included, would they still match if those are missing?

More detailed regex patterns can be found online. A few that I have found for email, CC# and SSN are:

https://www.regular-expressions.info/email.html[55]

https://www.regular-expressions.info/creditcard.html[56]

https://regexlib.com/Search.aspx?k=ssn[57]

Using Regex to filter Apache logs

Now that we have become familiar with the layout of Apache log files, and how to use regular expressions, we can combine these together to search our files for information.

Finding Method, URI, and Protocol manually

One great use of regex is to parse log files for necessary information. Web server administrators are often very interested to see what type of activity is occurring on their server, who is accessing it, how they are accessing it, and frequently accessed resources. Regex allows us to find this information quickly and easily.

One simple regular expression that allows identification of the request Method, URI, and Protocol is `\"(\S+)\s(\S+)\s\*(\S\*)\"`

[54] https://regexr.com/
[55] https://www.regular-expressions.info/email.html
[56] https://www.regular-expressions.info/creditcard.html
[57] https://regexlib.com/Search.aspx?k=ssn

How it works

The first section, `\"`, matches a double-quote (") character. The preceding backslash (\ ) is an "escape character" that instructs regex to treat the character as a double-quote.

The second section, `(\S+)\s`, finds the "method" used by the client. This is normally GET or POST, but several other methods can be used as well.

Regex	Description
`()`	**Capture group** - Groups part of the regular expression together
`\S`	**Not whitespace** - Matches to numbers, letters, or symbols. Anything except space, tab, carriage return. (Note the upper-case "S")
`+`	**Quantifier** - match 1 or more of the preceding
`\s`	**Whitespace** - Matches space, tab, or carriage return. (Note the lower-case "s")

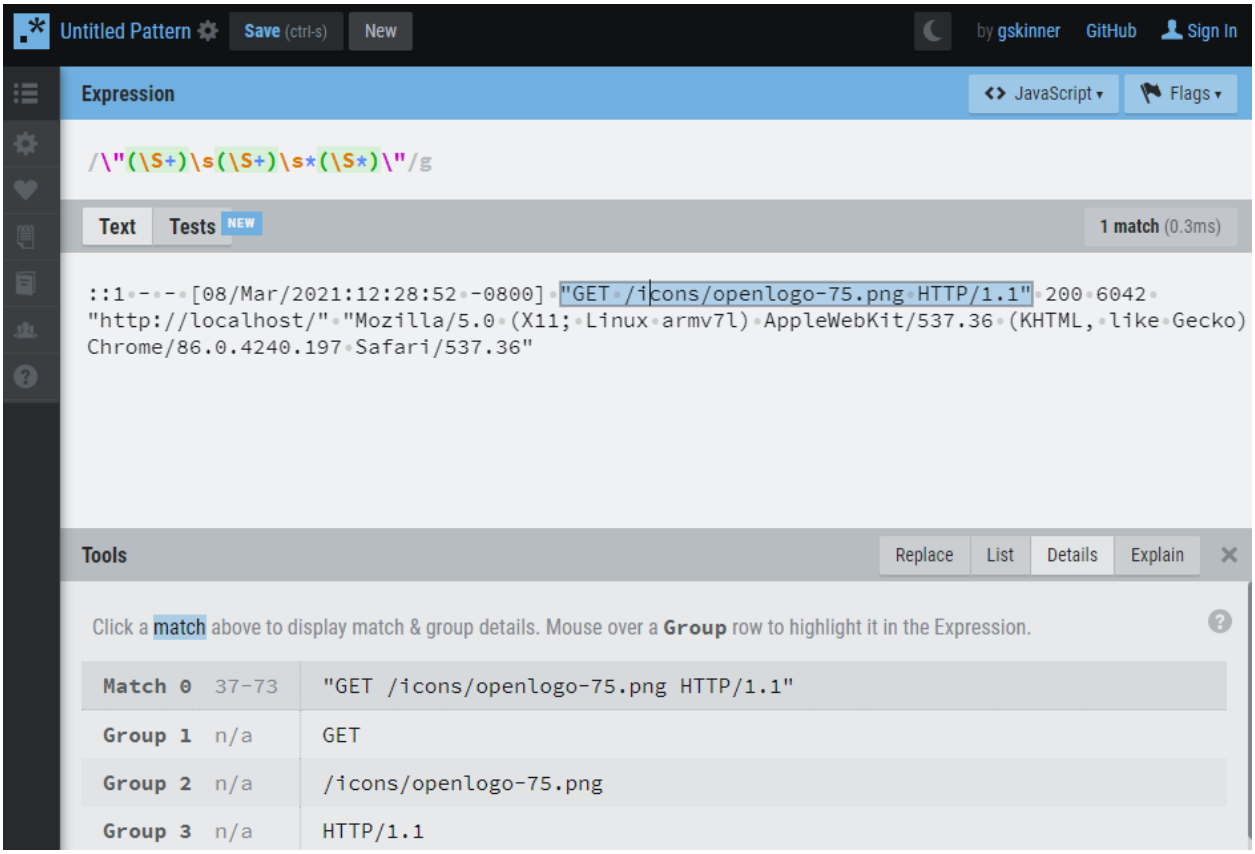
The next section, `(\S+)\s\*`, finds the "URI" requested by the client. This is the same pattern as used for the "method", with the exception of the final `\*`. This is quantifier (similar to the **+** and **{}**), but matches 0 or more of the preceding.

The final section, `(\S\*)\"`, finds the "protocol" used by the client. Again, this is the same pattern as used for the "method", with the exception of the final `\"`. This matches the closing double-quote character.

To test our expression, open a web browser to https://regexr.com/[58]. In the **Expression** section, enter in the expression from above. In the **Text** section, enter 1 or more lines from the **access.log** file. As the lines are entered, the web page will automatically highlight the matches.

If you select the **Details** tab at the bottom and select the highlighted section above, we can see how regex assists with complicated patterns.

[58] https://regexr.com/



regexr.com matching access.log

Here we can see we have a single **Match** which captures the entire method, URI, and protocol together. We can also see that we have three **Group** items, the first is the method, the second the URI, and the third is the protocol. Each of these groups corresponds to the parenthesis **()** in the original expression.

Using Python, we can utilize the entire **Match**, or any of the **Group** items in our scripts. This can simplify our processing of log files as a single match can return multiple items of interest.

Find most frequent client

One item that web administrators may be interested in knowing, is who the most frequent client is. Which person, or IP address is accessing my web site the most?

Getting ready

In Visual Studio Code, browse to **start/CH07/FindClients.py** and open it for editing.

How to do it

FindClients.py

```
#!/usr/bin/env python3
# Script that scans web server logs for client addresses
# Use RegEx to find and report on most frequent users
# By Ed Goad
# date: 2/5/2021

#Import Python modules
import re

# Prompt for file to analyze
log_file = input("Which file to analyze? ")

# Open file and load into memory
with open(log_file, "r") as f:
    sample_logs = f.readlines()

# Setup regex pattern and empty dictionary
client_pattern = r'(^\S+\.[\S+\.]+\S+)\s'
clientdict = {}

# Find match and store in dictionary
for line in sample_logs:
    # Search for pattern, and if found move forward
    m = re.search(client_pattern, line)
    if m:
        client = m.group()
        # Put access frequency in dictionary
        if client in clientdict.keys():
            clientdict[client] += 1
        else:
            clientdict[client] = 1


# Sort by most frequently accessed
for w in sorted(clientdict, key=clientdict.get, reverse=False):
    print(w, clientdict[w])
```

 Don’t forget to commit the changes to Git and GitHub

How it works

The script begins on line **#8** by importing the **re** library. This is the regular expression library used by Python and will enable us to perform regular expressions as we move forward.

On line **#11** the user is prompted for the file to open and analyze. Lines **#14,15** then open the file and read the contents into the **sample_logs** variable.

On line **#18** we define the pattern to use in the regular expression. This expression is looking for a space, followed by 3 digits (such as 404), and another space.

Regex	Description
`()`	**Capture group** - Groups part of the regular expression together
`^`	**Beginning** - Matches the beginning of the string or line
`\S`	**Not whitespace** - Matches to numbers, letters, or symbols. Anything except space, tab, carriage return. (Note the upper-case "S")
`+`	**Quantifier** - match 1 or more of the preceding
`\.`	The period "." character. The preceding backslash (\) is an "escape character" that instructs regex to treat the character as a period.
`[]`	**Character set** - match any character inside the brackets
`\S`	**Not whitespace** - Matches to numbers, letters, or symbols. Anything except space, tab, carriage return. (Note the upper-case "S")
`\.`	The period "." character. The preceding backslash (\) is an "escape character" that instructs regex to treat the character as a period.
`\s`	**Whitespace** - Matches space, tab, or carriage return. (Note the lower-case "s")

On line **#22** we begin a for loop to go through each line of **sample_logs** one at a time.

Line **#24** searches the line variable for our regex pattern. If it is found, the address of the client is assigned to the variable **client**.

On line **#28 - #31**, if the client address already exists in the **clientdict**.**keys** dictionary, the value of the key is increased by 1. If the client address doesn't exist in the dictionary, then it is added with a value of 1.

On line **#35**, the **clientdict** dictionary is sorted by values. The client address and number of times found are then printed to the screen.

 Use the debugger to walk through the script for more details on how it works.

Find status codes

Web site status codes, such as 200, 301, 404, and 500 mean differnet things and help describe how well the site is running. Administrators will look for these codes to see if common errors are occuring on their site to be proactive in resolving problems.

Getting ready

In Visual Studio Code, browse to **start/CH07/FindStatusCodes.py** and open it for editing.

How to do it

FindStatusCodes.py

```
#!/usr/bin/env python3
# Script that scans web server logs for status codes
# Use RegEx to find and report on most frequent status messages
# By Ed Goad
# date: 2/5/2021

#Import Python modules
import re

# Prompt for file to analyze
log_file = input("Which file to analyze? ")

# Open file and load into memory
with open(log_file, "r") as f:
    sample_logs = f.readlines()

# Setup regex pattern and empty dictionary
status_pattern = r'\s(\d{3})\s'
statusdict = {}

# Find match and store in dictionary
for line in sample_logs:
    # Search for pattern, and if found move forward
    m = re.search(status_pattern, line)
    if m:
        client = m.group()
        # Put access frequency in dictionary
        if client in statusdict.keys():
            statusdict[client] += 1
        else:
            statusdict[client] = 1


# Sort by most frequently accessed
```

```
35 for w in sorted(statusdict, key=statusdict.get, reverse=False):
36     print(w, statusdict[w])
```

 Don't forget to commit the changes to Git and GitHub

How it works

The script begins on line **#8** by importing the **re** library. This is the regular expression library used by Python and will enable us to perform regular expressions as we move forward.

On line **#11** the user is prompted for the file to open and analyze. Lines **#14**,**15** then open the file and read the contents into the **sample_logs** variable.

On line **#18** we define the pattern to use in the regular expression. This expression is looking for a space, followed by 3 digits (such as 404), and another space.

Regex	Description
`\s`	Whitespace - Matches space, tab, or carriage return. (Note the lower-case "s")
`()`	**Capture group** - Groups part of the regular expression together
`\d`	**Digit** - any number from 0 through 9
`{3}`	**Quantifier** - requires 3 of the preceding
`\s`	**Whitespace** - Matches space, tab, or carriage return. (Note the lower-case "s")

On line **#22** we begin a for loop to go through each line of **sample_logs** one at a time.

Line **#24** searches the line variable for our regex pattern. If it is found, the status code is assigned to the variable **client**.

On lines **#28** - **#31**, if the status code already exists in the **statusdict.keys** dictionary, the value of the key is increased by 1. If the status code doesn't exist in the dictionary, then it is added with a value of 1.

On line **#35**, the **statusdict** dictionary is sorted by values. The status codes and number of times found are then printed to the screen.

 Use the debugger to walk through the script for more details on how it works.

Finding potential hacking

Now that know how to efficiently read an Apache log file, and how to use Regular Expressions to search the file, we can do even more security related tasks. By searching the log files for suspicious traffic, we can be alerted to any questionable activity and then further secure the environment.

Getting ready

In Visual Studio Code, browse to **start/CH07/FindPotentialHacking.py** and open it for editing.

How to do it

FindPotentialHacking.py

```
#!/usr/bin/env python3
# Script that scans web server logs for possible hacking
# Use RegEx to find and report on common hacking types
# Based on https://www.cgisecurity.com/fingerprinting-port80
# -attacks-a-look-into-web-server-and-web-application-attack
# -signatures-part-two.html
# By Ed Goad
# date: 2/5/2021

#Import Python modules
import re

# Prompt for file to analyze
log_file = input("Which file to analyze? ")

# Open file and load into memory
with open(log_file, "r") as f:
    sample_logs = f.readlines()

# Setup regex pattens
find_wildcard_uri_pattern = r'\"(\S+)\s(\S*\*\S*)\s*(\S+)\"'
find_backtick_uri_pattern = r'\"(\S+)\s(\S*`\S*)\s*(\S+)\"'
find_code_uri_pattern = r']\s\"(\S+)\s(\S*[\^\[\]#{}\"]\S*)\s*(\S+)\"'
find_css_uri_pattern = r']\s\"(\S+)\s(\S*[\(\)]\S*)\s*(\S+)\"'
find_foldertraversal_uri_pattern = r']\s\"(\S+)\s(\S*///\S*)\s*(\S+)\"'

# Find match and report
for line in sample_logs:
    # Search for pattern, and if found move forward
    m = re.search(find_wildcard_uri_pattern, line)
    if m:
        print("Possible attack: Wildcard in URI")
        print("\t{0}".format(line.strip()))
    m = re.search(find_backtick_uri_pattern, line)
```

```
35      if m:
36          print("Possible attack: Backtick (`) in URI")
37          print("\t{0}".format(line.strip()))
38      m = re.search(find_code_uri_pattern, line)
39      if m:
40          print("Possible attack: Code in URI")
41          print("\t{0}".format(line.strip()))
42      m = re.search(find_css_uri_pattern, line)
43      if m:
44          print("Possible attack: Potential CSS in URI")
45          print("\t{0}".format(line.strip()))
46      m = re.search(find_foldertraversal_uri_pattern, line)
47      if m:
48          print("Possible attack: Folder Traversal in URI")
49          print("\t{0}".format(line.strip()))
```

 Don't forget to commit the changes to Git and GitHub

How it works

An extension of the prior script, this version looks for potential hacking using multiple regular expressions.

On line **#21-25**, 5 different regular expression patters are assigned to variables. It is encouraged for each of the patterns to be reviewed using regexr.com with sample log files to understand how they work and what they are looking for.

Beginning on line **#28**, each line in the log file is searched for each of the 5 regex patterns. If a pattern is found, a message stating a possible attack printed to the screen along with the suspicious line.

 Use the debugger to walk through the script for more details on how it works.

There's more

While a good starting location for scanning Apache log files, these 5 regular expressions are just a beginning. Several additional expressions can be added, such as testing for backdoor requests, command execution, SQL injection, using Unicode to hide attacks and more.

It is suggested that when adding or customizing the patterns, only 1 pattern be updated at a time. This way you can make a single change and test the changes, possibly with known effected log files.

See also

This example was based on information found at Fingerprinting Port80 Attacks: A look into web server, and web application attack signatures: Part Two[59]. Several other fingerprinting options are included on the web page and can be used to extend the script capabilities.

[59]https://www.cgisecurity.com/fingerprinting-port80-attacks-a-look-into-web-server-and-web-application-attack-signatures-part-two.html

Chapter 8: Intro to APIs

In this chapter we will cover the following:

- Introduction to APIs
- Using PostMan
- Introduction to JSON
- Finding how many people are in space
- Checking passwords for compromise
- Authenticating to APIs

Introduction

In this chapter we will begin using the wide world of Application Programming Interfaces (APIs). We will briefly cover the basics of what an API is, common data formats for APIs, and introduce an easy-to-use tool to help test APIs.

Once the groundwork is completed, we will jump into using different online APIs to perform various security and administrative type functions.

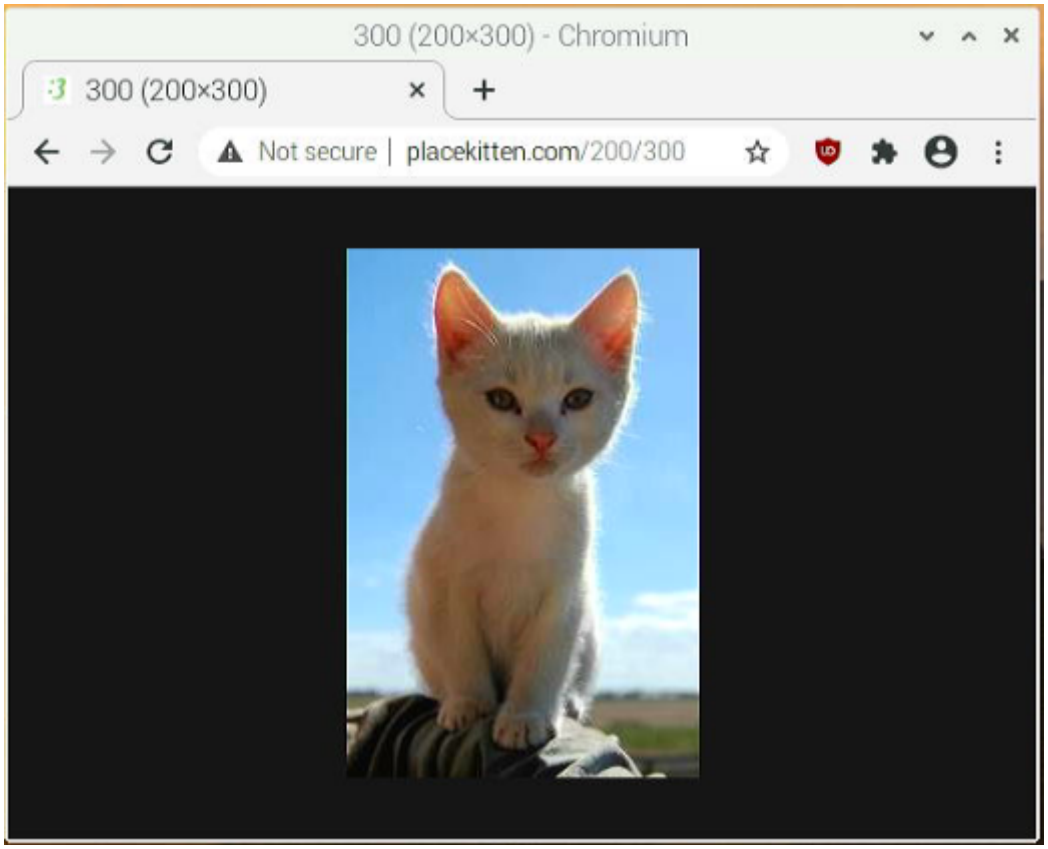
Intro to APIs

APIs are Application Programming Interfaces, or ways to work with other programs. To some degree, we have already used APIs even though we didn't call them that.

The first API we will begin using is an extremely simple one hosted by https://placekitten.com/[60]. The goal of this API is to provide images of whatever size is needed to act as placeholders. Designers would use these instead of empty blocks and would eventually replace them with their final images.

When visiting the web page, the use of the API is fairly straight-forward. We start with the site name https://placekitten.com/[61], followed by 2 numbers such as "200/300" signifying the width and height of the desired image. If we follow the suggested link, https://placekitten.com/200/300[62], we get the following in our web browser.

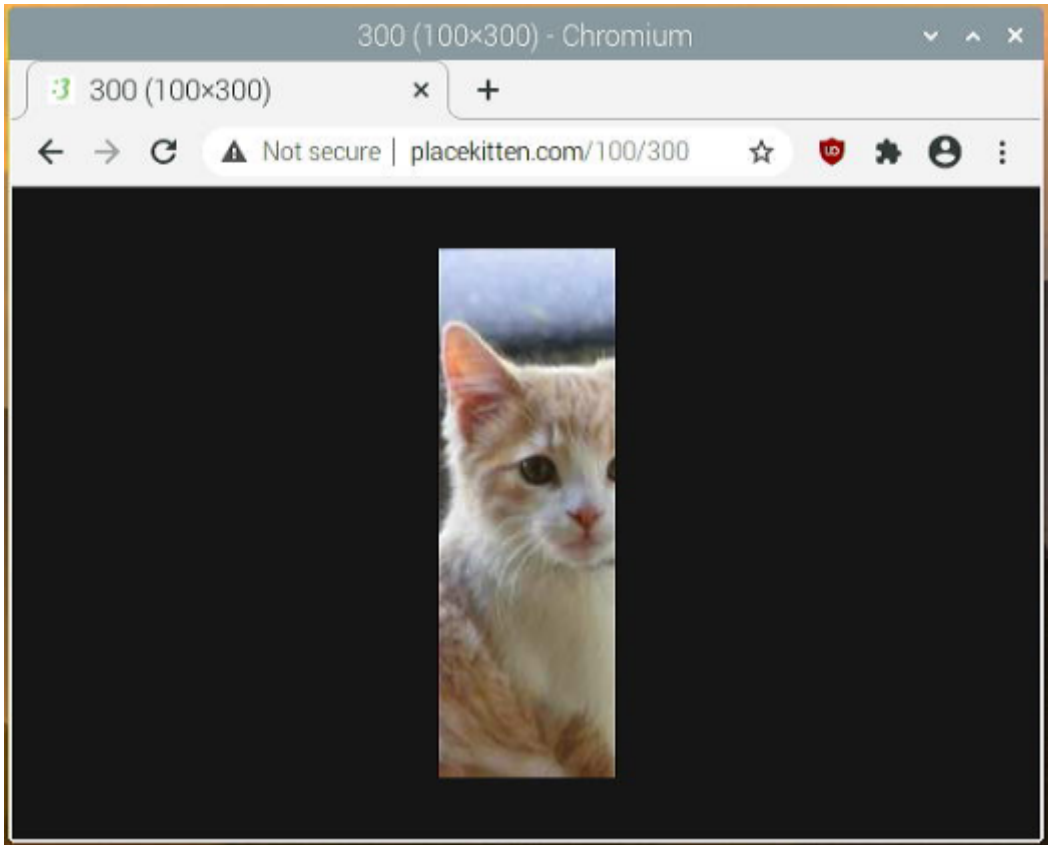
[60] https://placekitten.com/
[61] https://placekitten.com/
[62] https://placekitten.com/200/300



https://placekitten.com/200/300

At this point, we can explore how the API works by changing the numbers in the address bar and seeing how the image changes. For instance, if we change the address to https://placekitten.com/100/300[63], we are changing the request width, while keeping the height the same.

[63] https://placekitten.com/100/300

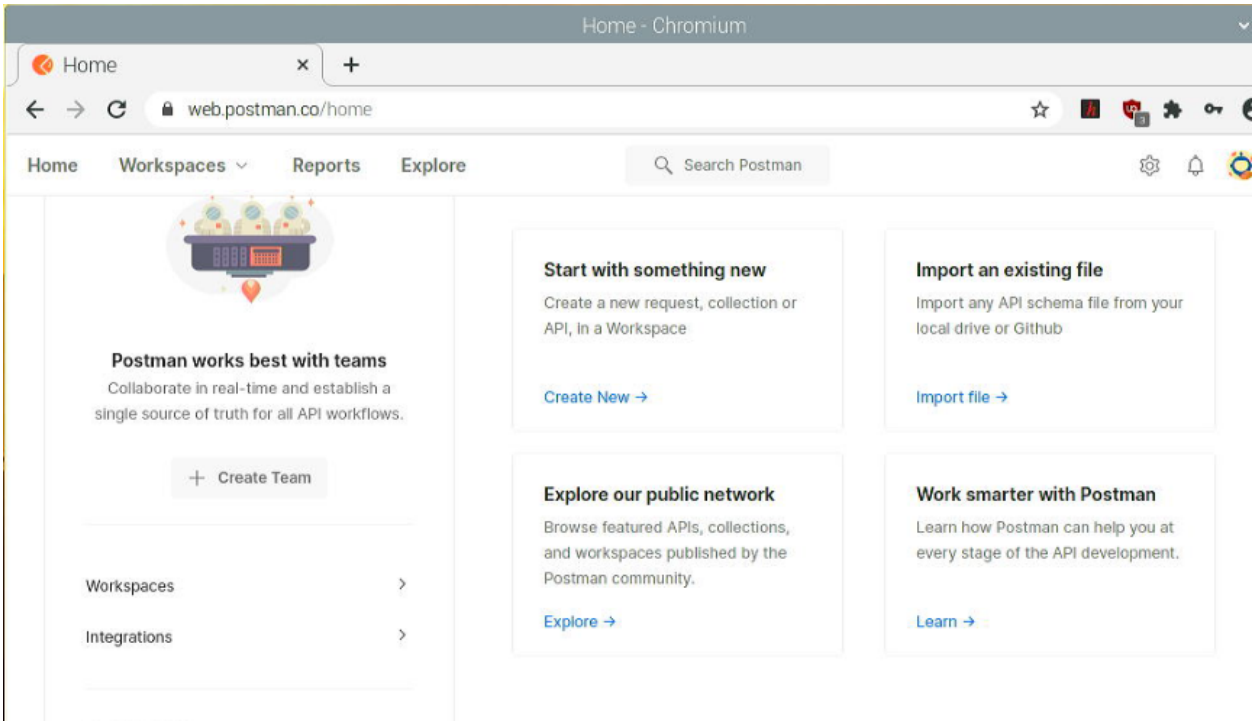


https://placekitten.com/100/300

Using PostMan

One of the common tools used when learning about a new API is called PostMan (https://www.postman.com/[64]). This is an incredibly flexible tool that allows for quick access to online APIs without coding. There is a free installable version as well as a web-based version. Because the installable version doesn't support the Raspberry Pi (yet), I will demonstrate using the web-based version.

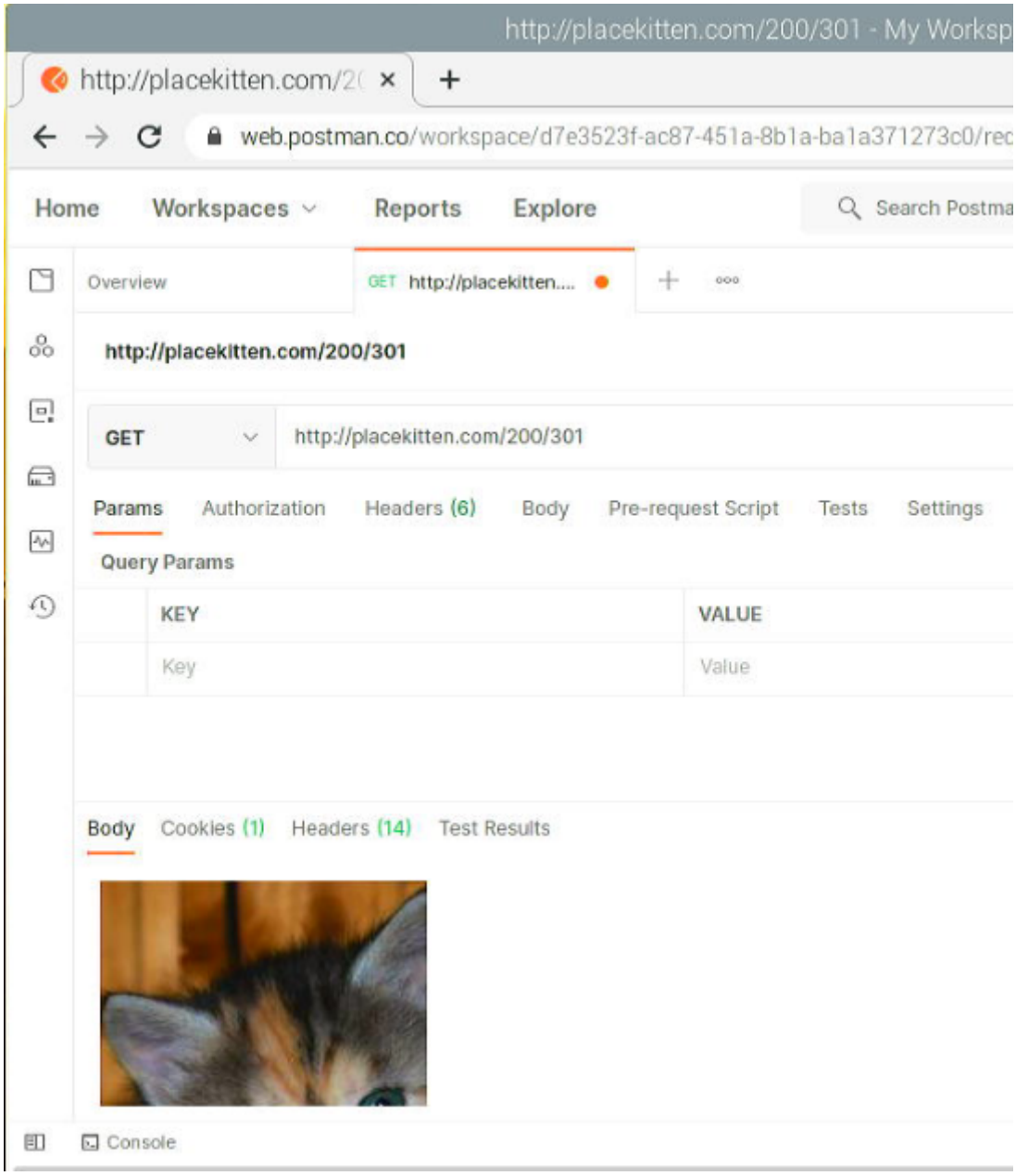
[64] https://www.postman.com/



Postman web console

Once you have created and confirmed your account, you will see a page similar to the image above. Let's start with our first API request.

1. Begin by clicking **Start with something new**.
2. In the window, next to the **Overview** tab, click the plus sign (**+**) to open a tab.
3. Next to where it says **GET**, paste in one of the URLs from placekitten.com.
4. Click **Send** to send the request .



Using Postman

5. The result should be shown in the **Body** at the bottom of PostMan.
6. Try changing the URL next to **GET** and note how the image changes as you click **Send**.

In this example we setup PostMan and confirmed it was working, but didn't use any of the more advanced features available to us. Just a few of the available options are to send credentials for authentication, include headers, upload data to the remote server, and receive and decode complex data formats. Even more options are available, but are out of scope for this book.

Introduction to JSON

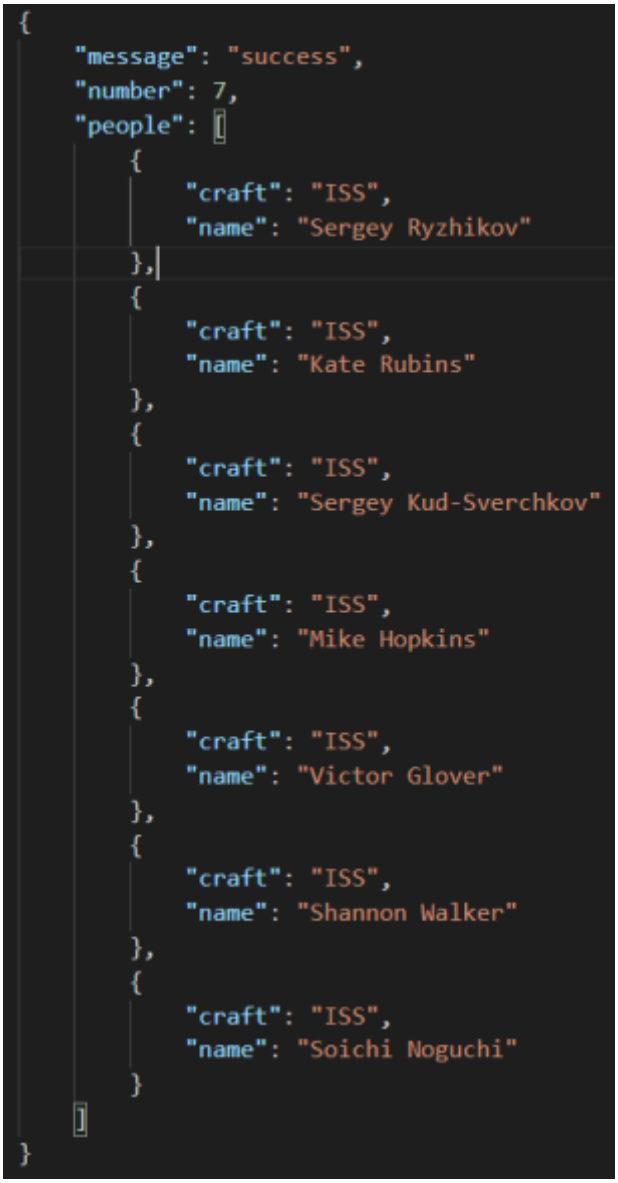
JSON stands for **J**ava **S**cript **O**bject **N**otation, and as its name suggests, was originally developed for working with JavaScript. JSON is designed to be a lightweight, yet human readable format that can be consumed by any application. Because JSON was originally designed to be used over the Internet, it has become a primary method for APIs to transmit data of the Internet.

As we can see from the below JSON file, the data is human readable, but possibly difficult to parse and understand immediately.


```
{"message": "success", "number": 7, "people": [{"craft": "ISS",
"name": "Sergey Ryzhikov"}, {"craft": "ISS", "name": "Kate
Rubins"}, {"craft": "ISS", "name": "Sergey Kud-Sverchkov"},
{"craft": "ISS", "name": "Mike Hopkins"}, {"craft": "ISS", "name":
"Victor Glover"}, {"craft": "ISS", "name": "Shannon Walker"},
{"craft": "ISS", "name": "Soichi Noguchi"}]}
```


Raw JSON information

Here we have the same information, but it has been "prettyfied" by including carriage returns and tabs to make it easier to read. Both formats (compressed and prettyfied) can be read by the computer.



Prettyfied JSON data

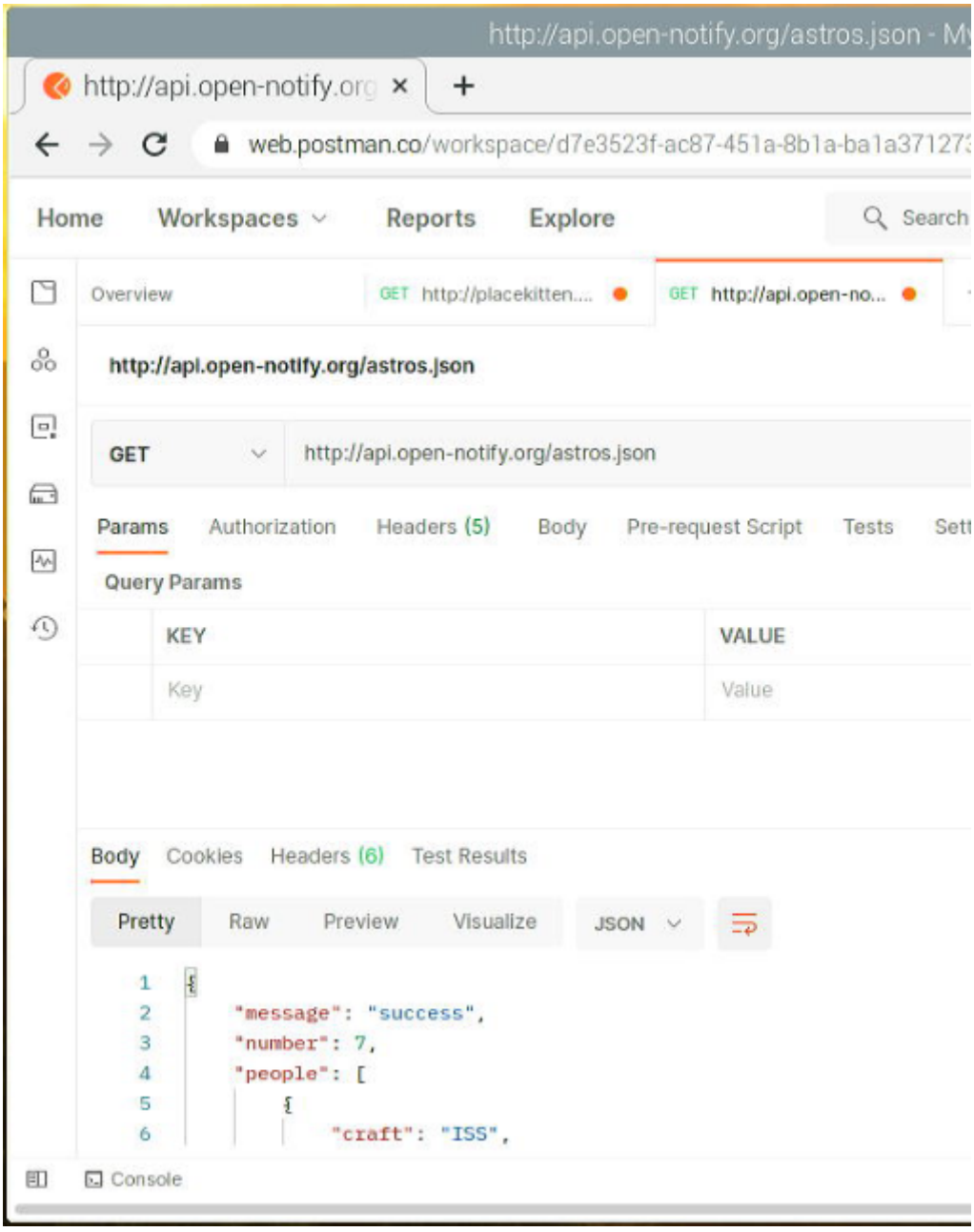
First API script

Getting ready

For our first Python script calling an API, we will find out how many people are in space. We will start this in PostMan to quickly and easily execute the API and view the results. Once we have confirmed functionality in Postman, then we will continue into Python.

1. In **PostMan**, click the plus sign (**+**) to open a new tab.

2. Next to **GET**, where it says **Enter request URL**, enter **http://api.open-notify.org/astros.json**
3. Click **Send** and the results should appear below.



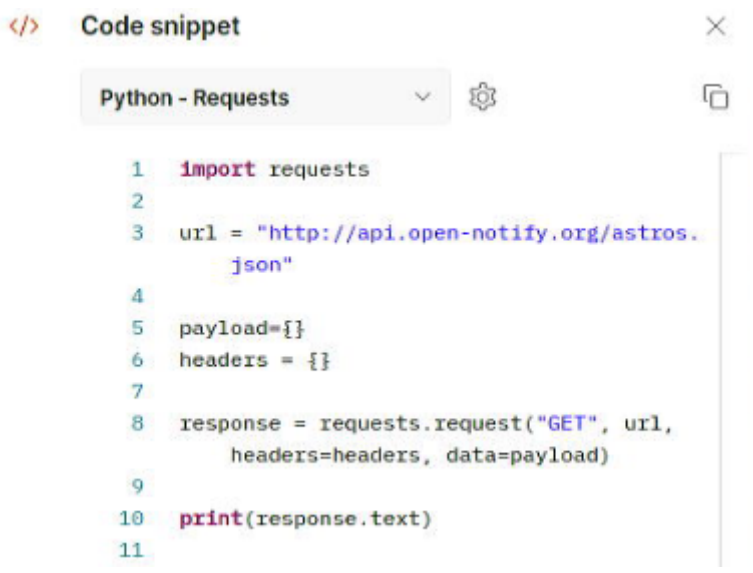
People in Space

NOTE: PostMan **prettyfied** the returned JSON. In the response pane you can see the original response by clicking **Raw**.

When using PostMan to test APIs, the tool can also generate sample code for use to use. To view the

sample code:

1. To the right of the **Send** button, click the </> icon to view the generated code.
2. Change the language to **Python - Requests**,



Postman Python code

3. In Visual Studio Code, browse to **start/CH08/PeopleInSpace1.py** and open it for editing.
4. Copy the sample code and past it in Visual Studio Code.

How to do it

PeopleInSpace1.py

```
import requests

url = "http://api.open-notify.org/astros.json"

payload={}
headers = {}

response = requests.request("GET", url, headers=headers, data=payload)

print(response.text)
```

 Don’t forget to commit the changes to Git and GitHub

How it works

The script begins on line **#1** loading the **requests** Python library. This is one of the main modules used in Python when working with web-based APIs.

Line **#3** we create a variable named **url** with the address of the API we are using.

Lines **#5** and **#6** prepare variables **payload** and **headers** as empty dictionary objects. If any payload or headers were needed, they would be set here.

Line **#8** performs the actual API call using the `requests.request()` method. The HTTP method is specified as "GET", followed by the address, headers, and data.

Finally, the **response.text** is printed on line **#10**. As we see from the output, this is in JSON format.

```
pi@raspberrypi:~/PythonForCyberSecurity $  /usr/bin/env /usr/bin/python3 /home/pi/
.vscode/extensions/ms-python.python-2021.2.582707922/pythonFiles/lib/python/debugp
y/launcher 44775 -- /home/pi/PythonForCyberSecurity/Ch08/PeopleInSpace1.py
{"message": "success", "number": 7, "people": [{"craft": "ISS", "name": "Sergey Ry
zhikov"}, {"craft": "ISS", "name": "Kate Rubins"}, {"craft": "ISS", "name": "Serge
y Kud-Sverchkov"}, {"craft": "ISS", "name": "Mike Hopkins"}, {"craft": "ISS", "nam
e": "Victor Glover"}, {"craft": "ISS", "name": "Shannon Walker"}, {"craft": "ISS",
 "name": "Soichi Noguchi"}]}
pi@raspberrypi:~/PythonForCyberSecurity $
```

Python response


Use the debugger to walk through the script for more details on how it works.

Second API script

Now that have called our first API from Python and received JSON output, our goal should be to make the JSON more friendly. In the response from our first script, we received more information than necessary, since we are only interested in the number of people in space, not the names of them.

Getting ready

In Visual Studio Code, browse to **start/CH08/PeopleInSpace2.py** and open it for editing.

How to do it

PeopleInSpace2.py

```
1  #!/usr/bin/env python3
2  # Script that tells us how many people there are in space
3  #By Ed Goad
4  # date: 2/5/2021
5
6  # Import Python modules
7  import requests
8  import json
9
10 def get_people_in_space():
11     request_uri = "http://api.open-notify.org/astros.json"
12     r = requests.get(request_uri)
13     items = r.json()
14     return items
15
16 astronauts = get_people_in_space()
17
18 # print basic details
19 print(astronauts)
20
21 # print "pretty" json
22 print(json.dumps(astronauts, indent=2))
23
24 # search through data to return specific information
25 print("There are {0} people in space right now".format( \
26     astronauts["number"]))
27 print("The first astronaut is {0} aboard the {1}".format( \
28     astronauts["people"][0]["name"], \
29     astronauts["people"][0]["craft"]))
30
31 # Loop through all the people
32 print("Full list of peole in space")
33 for person in astronauts["people"]:
34     print("{0} is aboard the {1}".format(person["name"], \
35     person["craft"]))
```

 Don't forget to commit the changes to Git and GitHub

How it works

The script begins on line **#16** by calling our `get_people_in_space()` function. This function performs the `requests.get()` method on line **#12** against our API.

The results from the API are temporarily stored in the variable **r**, and then converted from JSON into a Python dictionary object on line **#13**. Finally, the results are returned from the function.

On lines **#19 - #22**, the output from the API call are printed to the screen, first as the dictionary object, and then as a "pretty" JSON file.

On lines **#25 - #29** we print out specific information from the API results. First we report the number of astronauts, and then print the first name in the list.

On line **#33** we start a loop for each person in space. The loop prints the name and craft for each person currently in space.

See also

Accessing elements of Python dictionary[65]

Python Dictionary[66]

HaveIBeenPwned

The website HaveIBeenPwned[67] was created by the security researcher Troy Hunt. On this website, information from several data breaches is collected into a database that can be easily researched. Using this site, you can find out if your information was involved in a data breach and if your passwords are secure.

One of the APIs provided from this site is the ability to search for hacked passwords. Several data breaches included usernames and passwords, and this information has been collected for us.

Information about the API can be found at https://haveibeenpwned.com/API/v3[68]. Specifically, we are looking to validate our passwords are safe, which starts at https://haveibeenpwned.com/API/v3#PwnedPasswords[69].

Reading the documentation tells us a few important items:

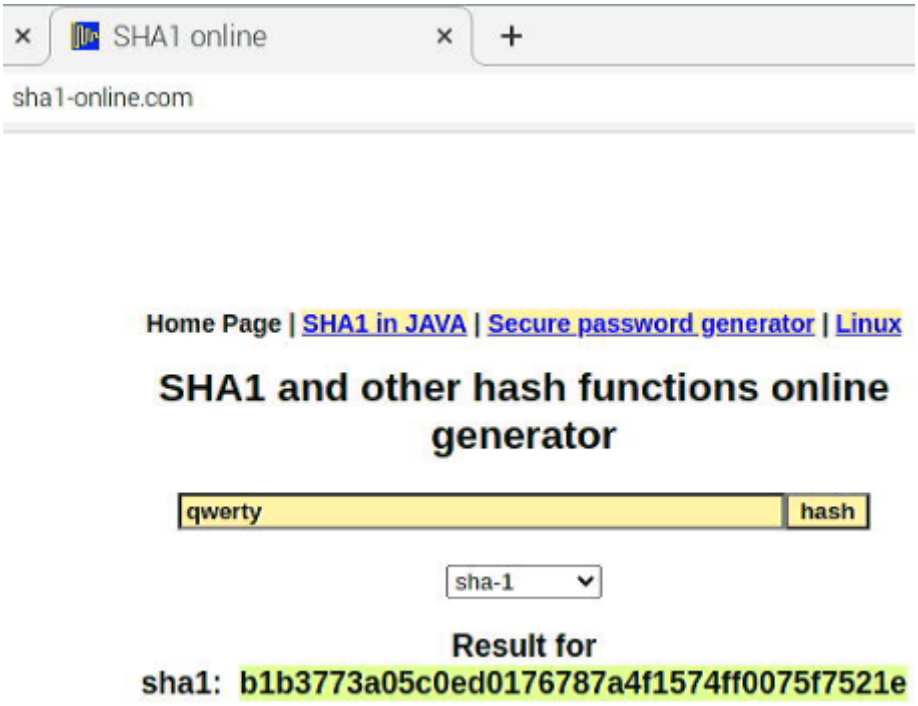
- All passwords are hashed using SHA-1.
- Instead of sending the entire hash across the internet, we only send the first 5 characters.
  - This is a security feature that limits the ability of anyone monitoring our traffic from determining the password we are using.
- We send a **GET** request to **https://api.pwnedpasswords.com/range/<first 5 hash chars>** to receive a list of hashes.
- We need to review the results to match to our hash.

[65] https://www.tutorialspoint.com/python/python_dictionary.htm
[66] https://www.programiz.com/python-programming/dictionary
[67] https://haveibeenpwned.com/
[68] https://haveibeenpwned.com/API/v3
[69] https://haveibeenpwned.com/API/v3#PwnedPasswords

How to do it

To test this prior to writing any code

1. To create a SHA-1 hash online, go to http://www.sha1-online.com/[70]
2. In the field, type in **qwerty** (all lower case) and click **hash**.



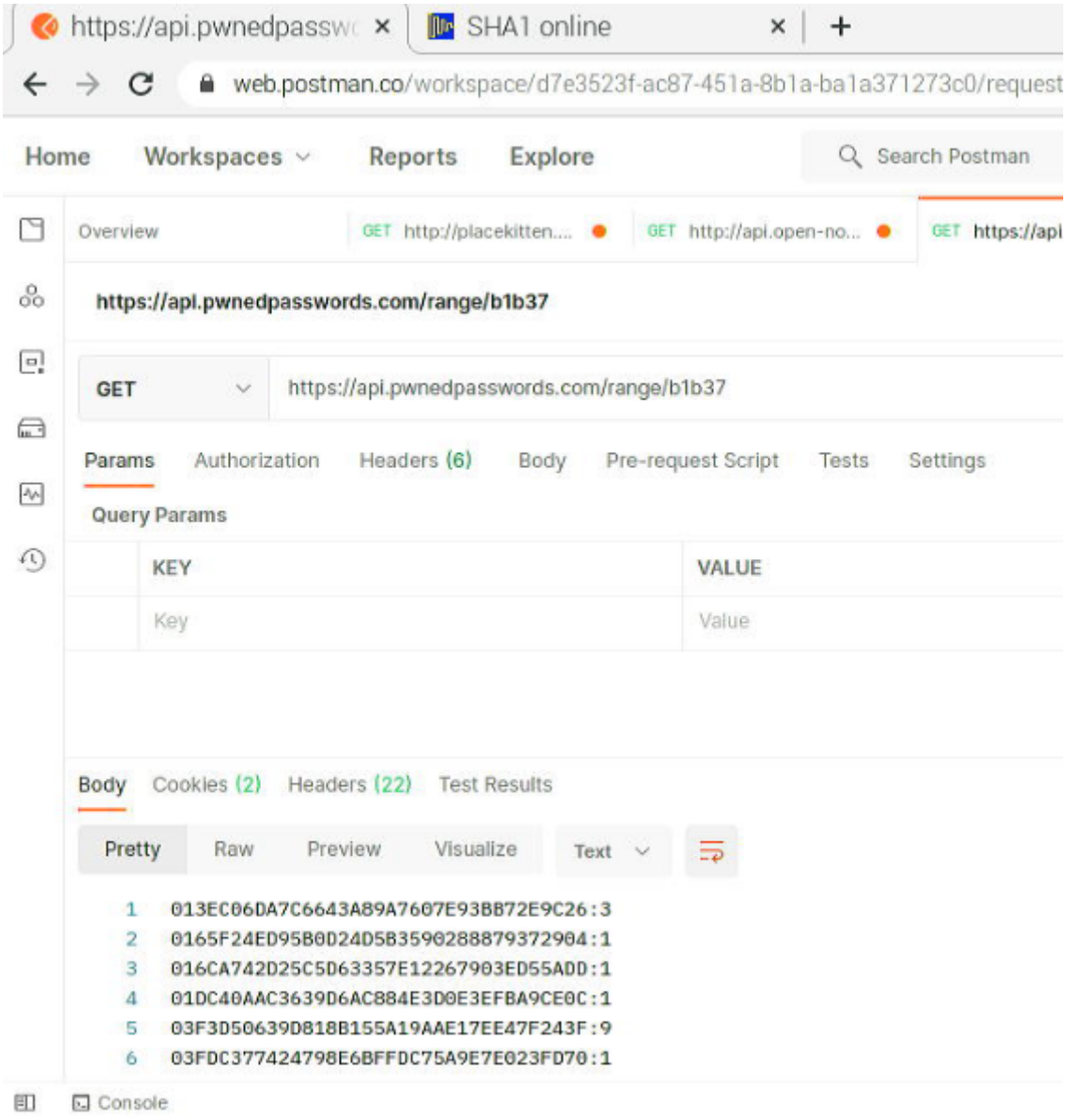
Generating SHA1 online

3. Copy the first 5 characters (**b1b37**).
4. Open **PostMan** and click the plus sign (**+**) to open a new tab.
5. Next to **Get** , where it says **Enter request URL** , enter

https://api.pwnedpasswords.com/range/b1b37 (the API base URL + the first 5 characters of the hash)

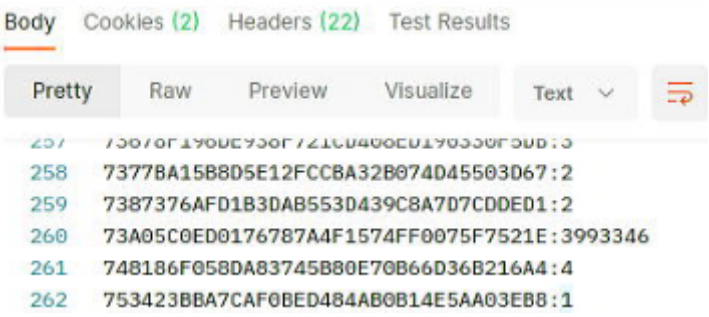
6. Click **Send**.

[70] http://www.sha1-online.com/



pwnedpasswords API request

7. Scan through the results for the rest of the hash **73a05c0ed0176787a4f1574ff0075f7521e**
8. If the hash is not found, this means the password has not been included in any leaks of compromised accounts.
9. If the hash is found in the results, the number following the colon tells us how many times it has been found in compromised accounts.



Hash results

> **NOTE:** We can see this password has been found in **3,993,346** compromised accounts and should not be used under any circumstances.

There's more

Try repeating the steps with other passwords and observing the results. You will find that some passwords (like **12345**) have been reported in millions of compromises, while other passwords (like **4:'yV2R3eWZF#<W5**) haven't been reported at all.

Automating Passwords

Now that we know how the HaveIBeenPwned API works, and the results we should be expecting, we can begin our first script using it.

Getting ready

In Visual Studio Code, browse to **start/CH08/HaveIBeenPwned.py** and open it for editing.

How to do it

HaveIBeenPwned.py

```
1  #!/usr/bin/env python3
2  # Script that checks passwords against haveibeenpwned.com API
3  # https://haveibeenpwned.com/API/v3#PwnedPasswords
4  # By Ed Goad
5  # date: 2/5/2021
6
7  # Import Python modules
8  import requests
9  import hashlib
10
11 def check_haveibeenpwned(sha_prefix):
12     pwnd_dict = {}
13     request_uri = "https://api.pwnedpasswords.com/range/" + \
14         sha_prefix
15     r = requests.get(request_uri)
16     pwnd_list = r.text.split("\r\n")
17     for pwnd_pass in pwnd_list:
18         pwnd_hash = pwnd_pass.split(":")
19         pwnd_dict[pwnd_hash[0]] = pwnd_hash[1]
20     return pwnd_dict
21
22 password = input("What password needs to be checked? ")
23 sha_password = hashlib.sha1(password.encode()).hexdigest()
24 sha_prefix = sha_password[0:5]
25 sha_postfix = sha_password[5:].upper()
26 pwnd_dict = check_haveibeenpwned(sha_prefix)
27
28 if sha_postfix in pwnd_dict.keys():
29     print("Password has been compromised {0} times".format( \
30         pwnd_dict[sha_postfix]))
31 else:
32     print("Password has not been compromised. It is safe to use!")
```

 Don't forget to commit the changes to Git and GitHub

How it works

The script begins on line **#22** by prompting the user to input a password. The `hashlib.sha1()` method is then used to generate a SHA-1 hash of the password. Lines **#25,25** split the hash into a prefix (the first 5 characters of the hash) and a postfix (the remainder of the hash). These pieces are converted to upper-case for easier matching later.

On line **#26**, we call our `check_haveibeenpwned()` function with the hash prefix.

On line **#12**, we start our function by creating an empty dictionary object named **pwnd_dict** , this will hold the results from the API. On lines **#13-15**, the URI is constructed with the base URL for the API and the first 5 characters of the hash. The API is then called, and results stored in our dictionary object.

On line **#16** we split the response text on the carriage return/newline (**\r\n**). This converts the large amount of text into a list (**pwnd_list**) of individual items that can be processed in the for loop beginning on line **#17**.

On line **#18**, the separate items returned from the API (**pwned_pass**) are split based on the colon (:) and stored in **pwnd_hash**. The two components of **pwnd_hash** (the hash, and number of times compromised), are then added to the **pwnd_dict** dictionary.

When loop has finished with the information returned from the API, the dictionary object is returned to the main code of the script.

On line **#28**, we check if **sha_postfix** is in **pwnd_dict.keys()**. If the hash exists in the dictionary, the number of times compromised is returned. Otherwise, a "safe to use" message is returned.

There's more

The script we created can easily be expanded to become a central part of a password management tool. This script could be joined with a "password generator" to first generate a random password, and then test the password against known bad lists.

This could also be part of a password change utility for users to change their passwords. Before permitting a change, the script can validate it online to determine if it is a secure password, or if it is a known bad password. Only if the password is secure is the password change allowed.

See also

Python dictionary object

https://www.w3schools.com/python/python_dictionaries.asp[71]

https://docs.python.org/3/tutorial/datastructures.html#dictionaries[72]

https://realpython.com/python-dicts/[73]

https://www.programiz.com/python-programming/dictionary[74]

[71] https://www.w3schools.com/python/python_dictionaries.asp
[72] https://docs.python.org/3/tutorial/datastructures.html#dictionaries
[73] https://realpython.com/python-dicts/
[74] https://www.programiz.com/python-programming/dictionary

Authenticating to APIs

So far, we have used public and free APIs, however that is rarely the case. Most APIs are closed for various reasons and require authentication to utilize them. One such API, that we already have access to, is the GitHub Rest API (https://docs.github.com/en/rest[75]).

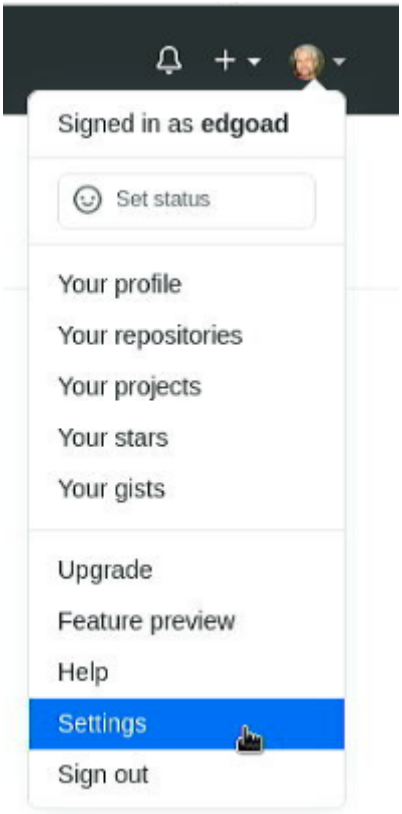
Getting GitHub API key

In Chapter 1 of this book, we walked through the process of creating a **Personal access token** for GitHub. This same token is what we will use for accessing the GitHub API. The benefit of using tokens/API keys is that we can create a new token specifically for our script here without affecting anything else. Creating new tokens/keys for each script is a proper and suggested method for working with APIs.

To create a GitHub Personal access token

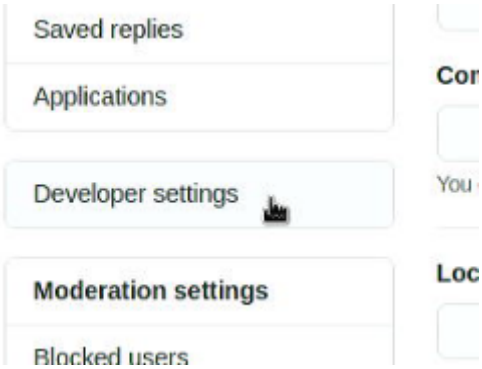
1. If not already, log into **GitHub**.
2. Click on your name / Avatar in the upper right corner and select **Settings**.

[75] https://docs.github.com/en/rest



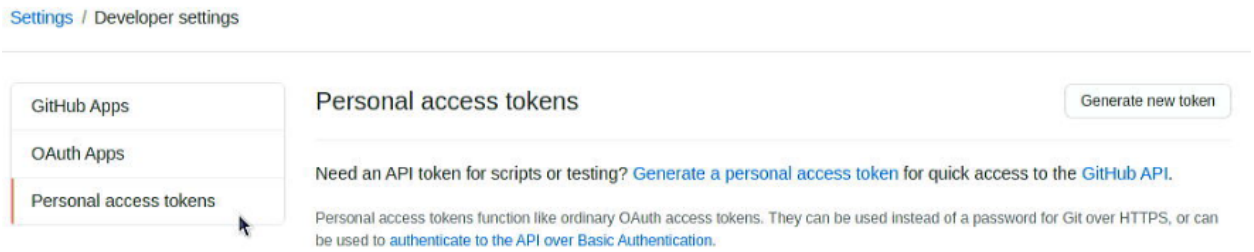
GitHub settings

3. On the left, click **Developer settings**.



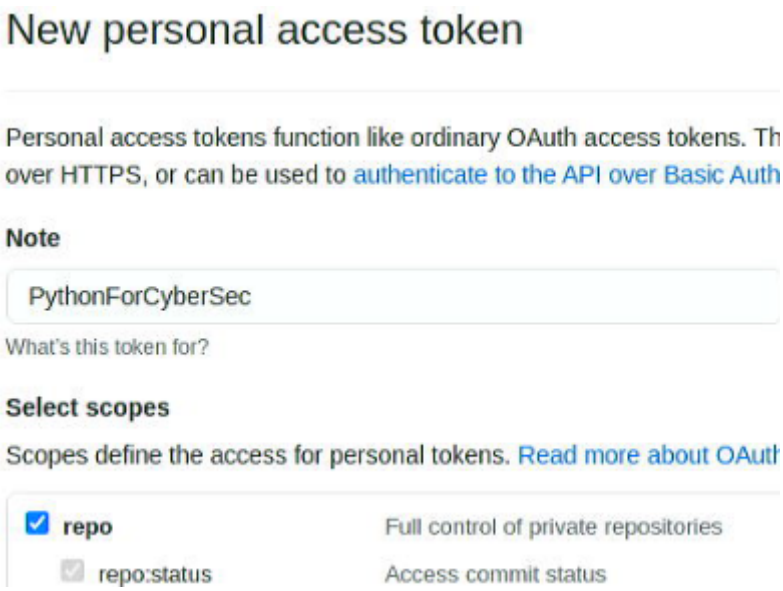
GitHub developer settings

4. Select **Personal access tokens** and click **Generate new token**.



Generating token

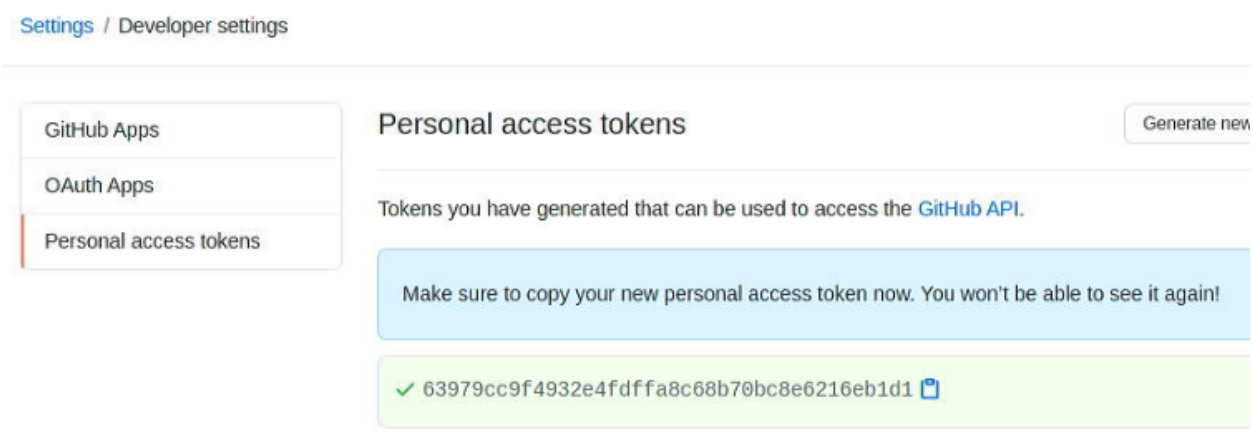
5. Give the token a description/name and select the scope of the token.



Token scope

- I selected **repo** only to facilitate pull, push, clone, and commit actions.
- Click the link **Read more about OAuth scopes** for details about the permission sets.

6. Click **Generate token**.
7. Copy the token.



Copying token

- **NOTE**: The token is only available at this point. If you lose the token, you will need to regenerate it.

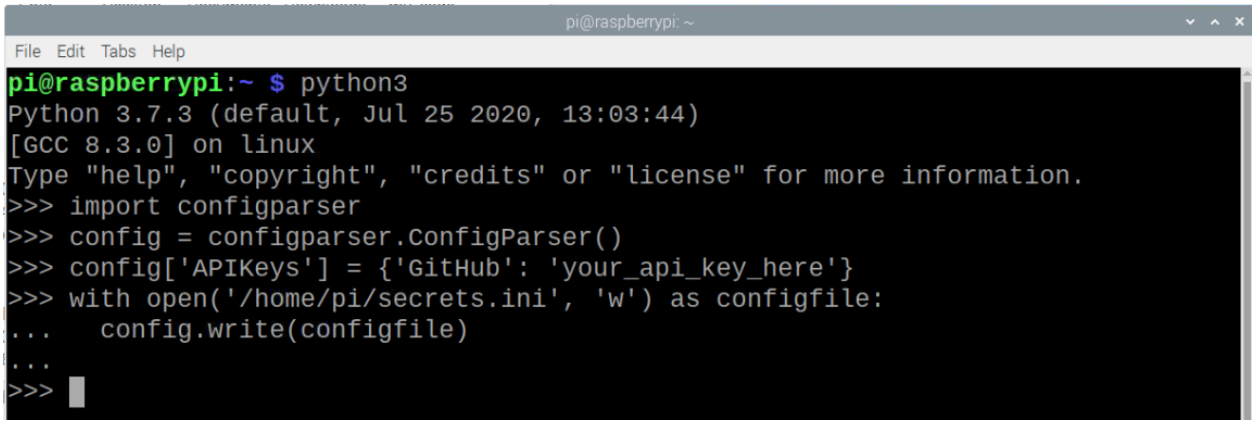
Storing API keys

API keys are the equivalent of both username **and** password for accessing APIs. Because these keys potentially have lots of permission to your accounts, services, and private information, they should be kept secret. Keeping keys secret is especially true when working with Git and GitHub.

Rule of thumb - **Do not store API keys where they may be published!**

The best method I have found to store API keys for Python is using the **configparser** library. To start using this library, open Python3 from the console and enter the following (enter your API key instead).

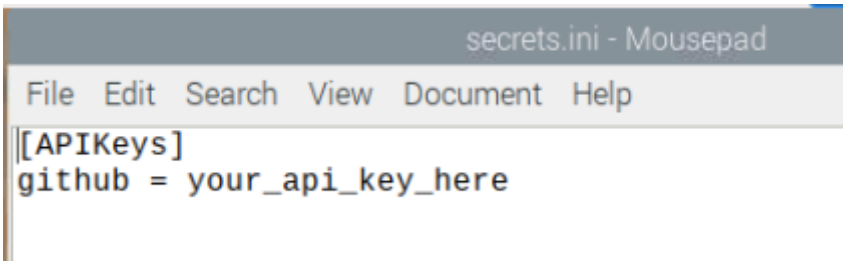
```
1  import configparser
2  config = configparser.ConfigParser()
3  config['APIKeys'] = {'GitHub': 'your_api_key_here'}
4  with open('/home/pi/secrets.ini', 'w') as configfile:
5    config.write(configfile)
```



Using configparser

This will create a file named **secrets.ini** in the **/home/pi** folder. This is important because the **/home/pi** folder won't be part of the Git repository, and therefore won't be accidentally uploaded to the internet.

If you open the file using a text editor, you can see how the information is stored, and what format to use to add additional keys in the future. If you need to update an API key, you can edit the text file with the new key.



Secrets file

See also

https://docs.python.org/3/library/configparser.html[76]

Viewing GitHub repositories

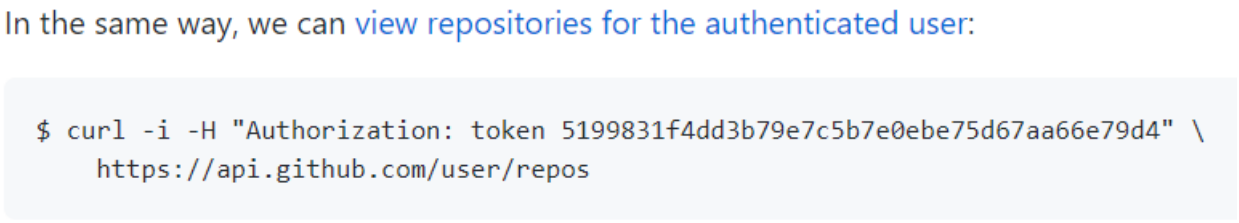
Now that we have our GitHub API key ready, we can use it for authentication.

Getting ready

There is a lot we can do with this API, such as creating users, creating repositories, managing organizations, changing permissions and so on. As an example of authenticating to APIs, we will

[76] https://docs.python.org/3/library/configparser.html

skip directly to the documentation at https://docs.github.com/en/rest/guides/getting-started-with-the-rest-api#repositories[77].



GitHub API example

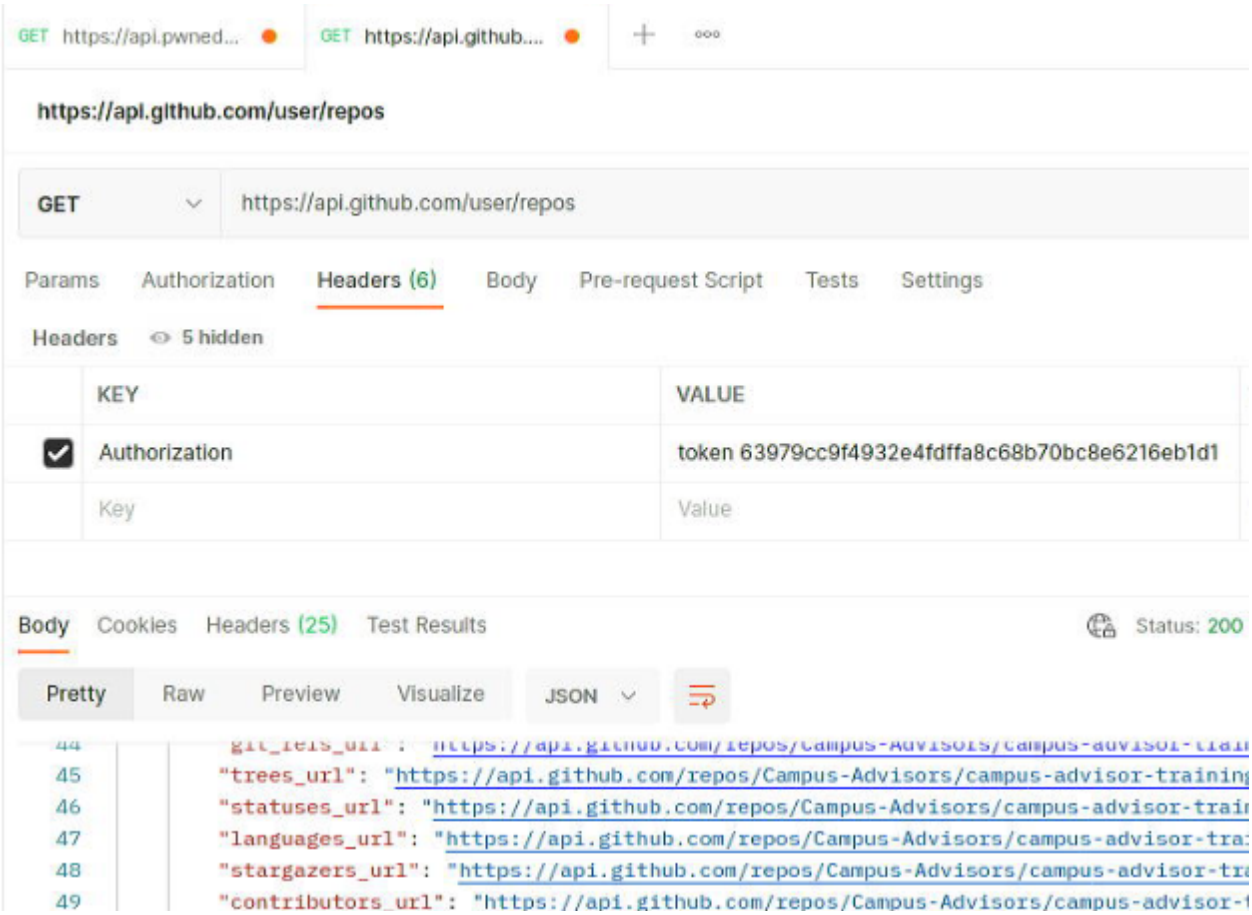
The second example from the API documentation describes how to use `curl` (essentially a command line web browser) to return a list of repositories owned by the requestor. The requestor is identified by the token provided as part of the authorization header (the -H in curl). This token is the **Personal access token** we created earlier. We can use PostMan to demo how to use this token to request a list of repositories.

1. Open **PostMan** and click the plus sign (**+**) to open a new tab.
2. Next to **GET** where it says **Enter request URL**, enter **https://api.github.com/user/repos**
3. Under the URL, click the tab labeled **Headers**
4. In an empty row, under **KEY**, type **Authorization**
5. Next to **Authorization**, under **VALUE**, enter **token <your personal access token>**

   NOTE: The space between the word **token** and the beginning of the access token.

6. Click **Send**.

[77] https://docs.github.com/en/rest/guides/getting-started-with-the-rest-api#repositories



GitHub API in Postman

7. In the result section we can see many details about our repositories

How to do it

ListRepositories.py

```
#!/usr/bin/env python3
# Script that lists repositories in GitHub
# Requires a Personal Access Token to run
# By Ed Goad
# date: 2/5/2021

# Import Python modules
import requests
import json
import configparser

```

```
12 def get_api_key(key_name):
13     # Create the ConfigParser and load the file
14     config = configparser.ConfigParser()
15     config.read("/home/pi/secrets.ini")
16     # Get the API key and return
17     api_key = config["APIKeys"][key_name]
18     return api_key
19 
20 def list_respositores(token):
21     # Setup the base URL and Authorization header
22     url = "https://api.github.com/user/repos"
23     headers = { 'Authorization' : "token " + token }
24     # Perform the request
25     response = requests.get(
26         url,
27         headers=headers
28         )
29     # Convert the JSON to Python objects
30     items = response.json()
31     return items
32 
33 # Get API key from file
34 token = get_api_key("GitHub")
35 
36 # Get repositories
37 repositories = list_respositores(token)
38 # For each repo, print out the name
39 for repository in repositories:
40     print(repository["name"])
```

 Don't forget to commit the changes to Git and GitHub

How it works

The script starts on line **#34** where we call our `get_api_key()` function. This function creates a `configparser.ConfigParser()` object named config. Once created, we use the `read()` method to open the secrets.ini file. Lastly, the file is filtered for a section named **APIKeys** and a key named **GitHub**.

Once we have our API key, we then call our `list_repositories()` function. Lines **#22-23** prepare the URL and Header information in the same way as we did with PostMan. Line **#25** performs the `requests.get()` method and then returns the information.

Once the information is returned, on line **#39** we begin to loop through the list of repositories. For each repository, we print the repository name and then quit.

There's more

In this example, we simply reported on the names of the repositories available to us as a user. Using the GitHub API, we can extend this to report various details about the repositories, update permissions and security, and even create/delete repositories.

If you work in a larger environment with multiple users, the GitHub API can be used to manage an entire organization. Creating users, granting permissions, creating teams, organizing repositories, and more.

Chapter 9: Cybersecurity APIs

In this chapter we will cover the following:

- VirusTotal APIs
- Automating VirusTotal
- Google SafeBrowsing API
- Automating URL scanning

Introduction

In this chapter we will jump into using APIs to show how we can protect ourselves on the internet. We will be focusing on 2 specific APIs, one provided by VirusTotal and the other by Google.

Because we aren't experts on using APIs yet, and because each API works slightly differently, we will slowly work into using these APIs. We will start by reviewing the available documentation, showing how to manually use the APIs, and then finally creating our Python scripts. By slowly introducing ourselves to these APIs, we can test that things are working properly before jumping into code.

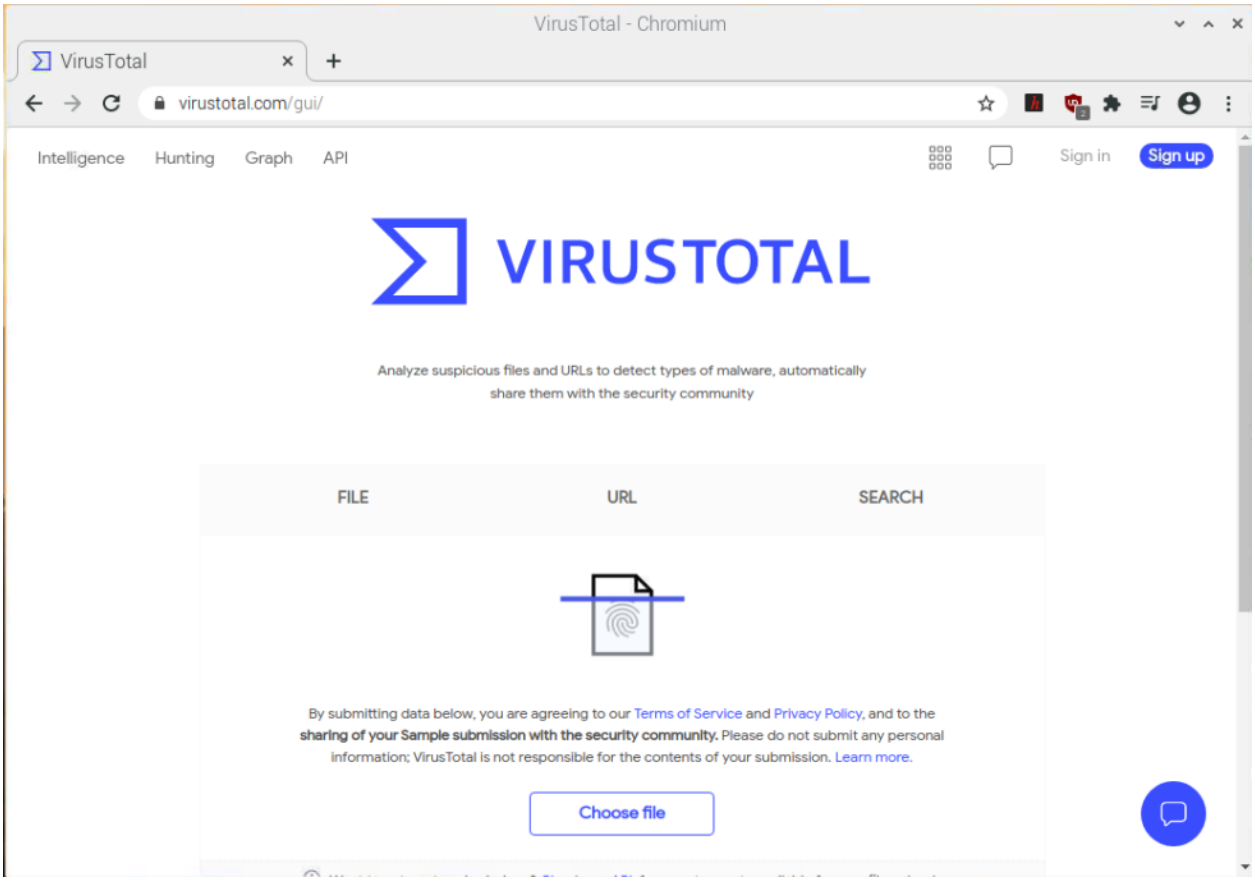
VirusTotal

VirusTotal (https://www.virustotal.com/[78]) is an online tool that allows for uploading of files to be scanned by multiple antivirus engines. Because no single virus scanner is perfect, it is common to scan suspicious files with multiple AV scanner. The VirusTotal site scans uploaded files against dozens of different scanners and reports the status returned by all of them.

To see how VirusTotal works:

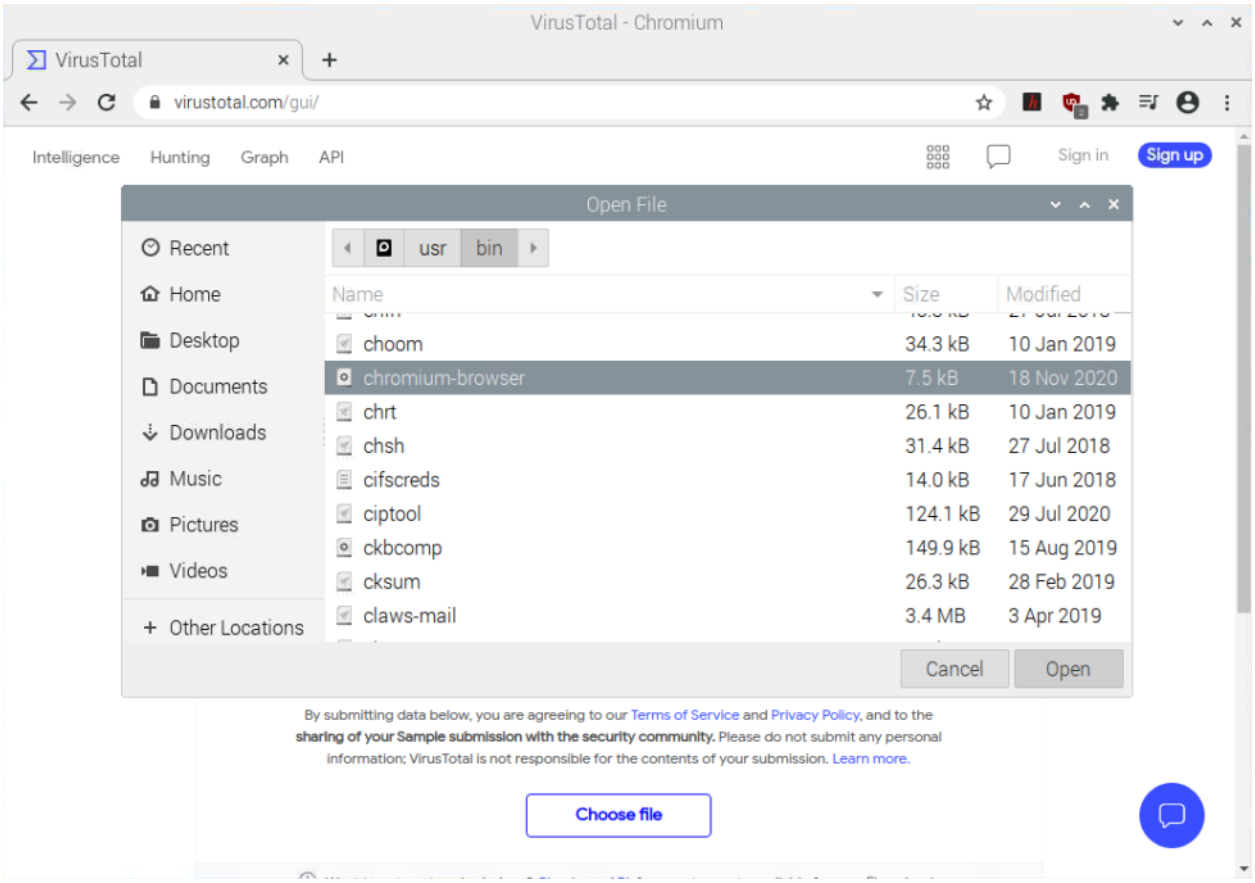
1. Open a web browser and go to https://www.virustotal.com[79]

[78] https://www.virustotal.com/
[79] https://www.virustotal.com/



virustotal.com

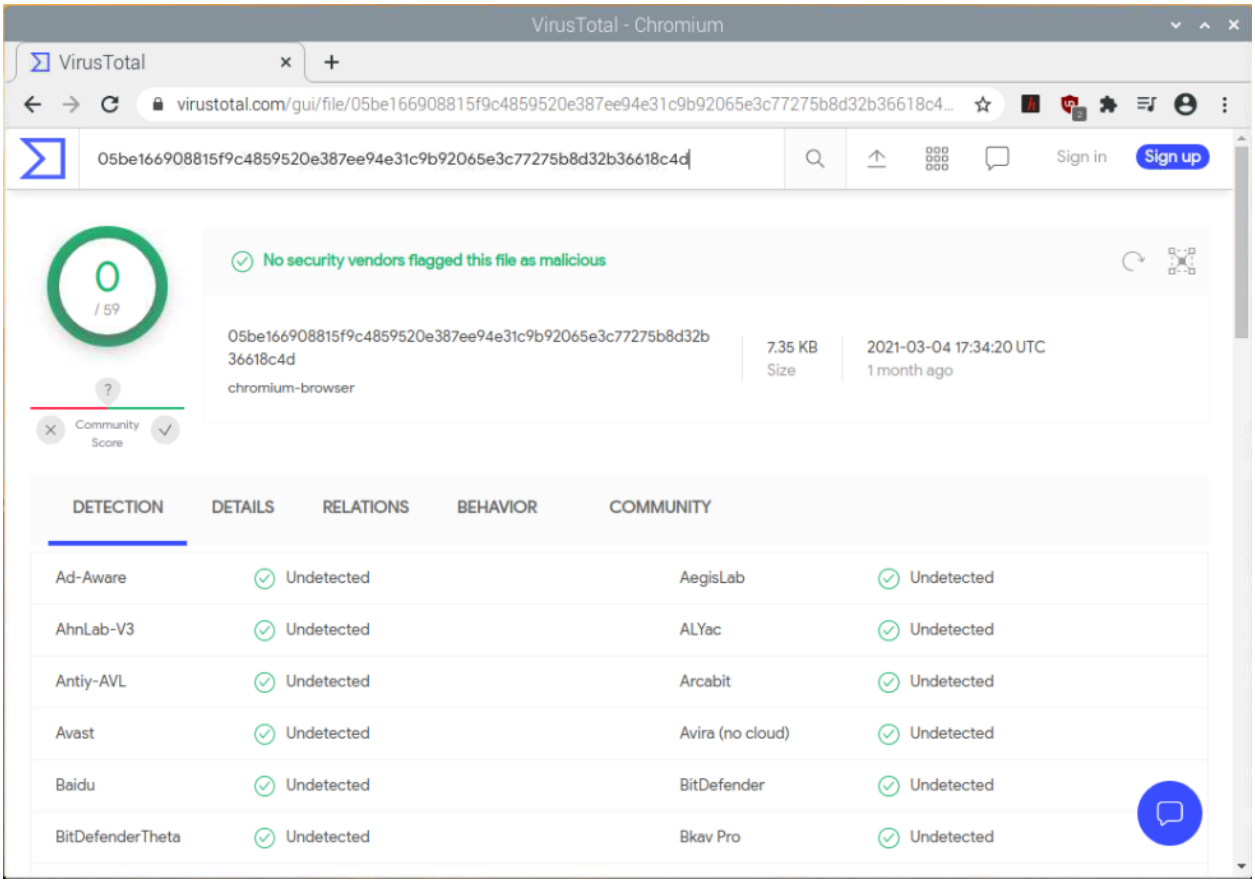
2. On the home page, click **Choose file** and then select a file to upload.



Uploading file for scanning

I chose **/usr/bin/chromium-browser** from the Raspberry Pi, a known good file.

3. If prompted to confirm the upload, click **yes**.
4. On the results page, review the status from each of the scanning engines.



Scan results

Manually Scanning for Viruses

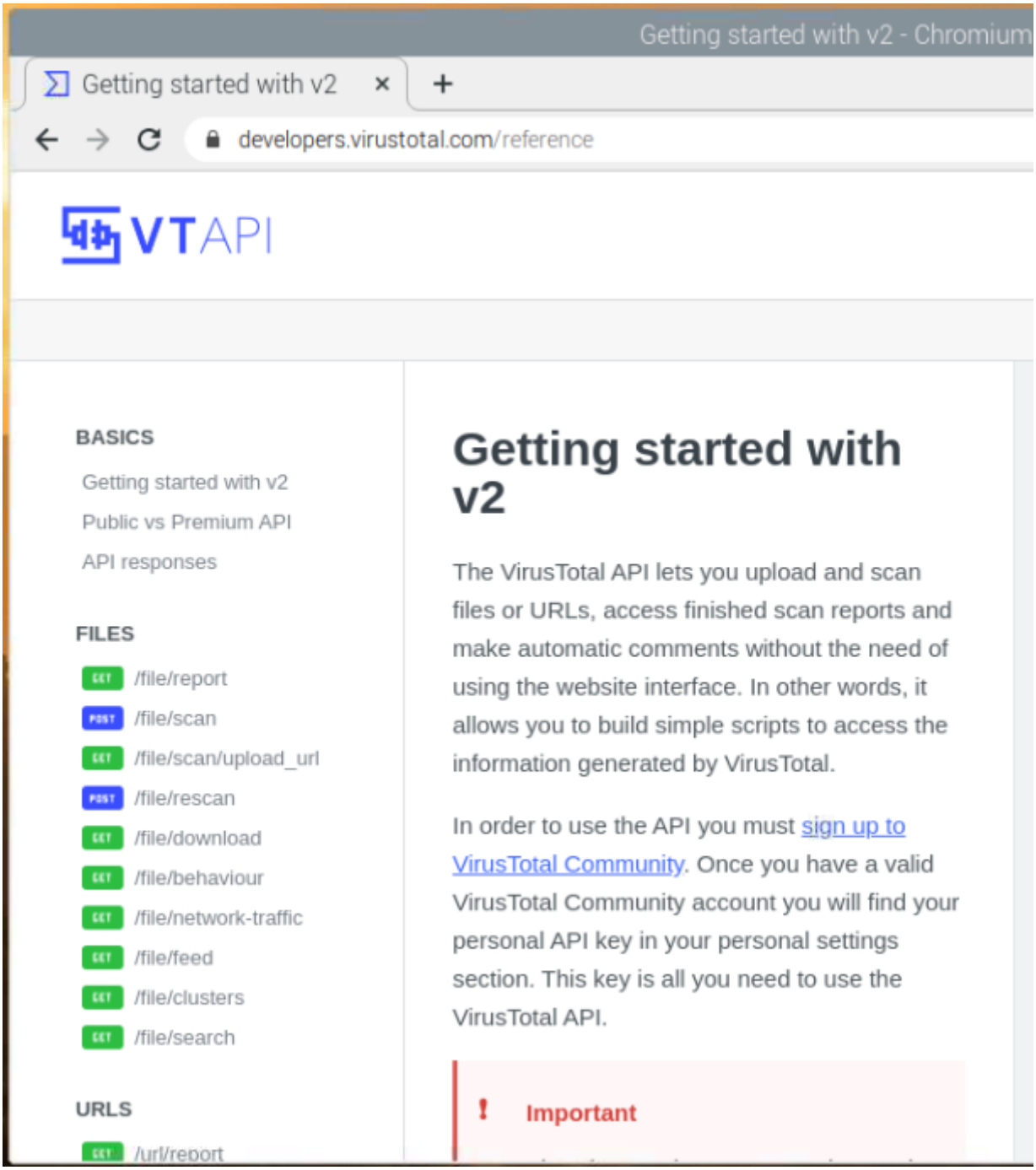
Now that we see what the tool can do, we can turn our attention to understanding the API.

Getting ready

API Reference

To use the VirusTotal API, we must first review the reference material available to us. The API documentation can be found at https://developers.virustotal.com/reference[80]. On this site, we can review details about the API, how to use the API, and what requirements exist for each API.

[80] https://developers.virustotal.com/reference



API reference

There are 2 important notes about this API:

1. We are using the Public API. Details about the difference between the Public and Premium API can be found at https://developers.virustotal.com/reference#public-vs-private-api[81]. These

[81] https://developers.virustotal.com/reference#public-vs-private-api

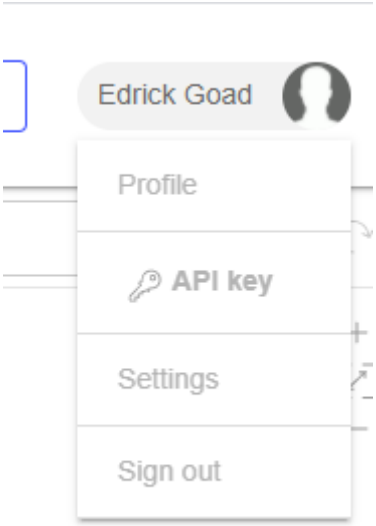
differences won't affect us now, but may be important to know in the future.
2. We are using version 2.0 of the API. There is a version 3.0 API[82] available, but 2.0 is still supported. We are using the older version specifically because it is easier to explain and use for educational purposes.

Getting a Key

To access the API, we need an account with VirusTotal. As we saw in prior APIs, accounts are used to restrict access to APIs, restrict access to private data, and for possible billing purposes. Once we have an account, we can request an API key to access the API.

To create an account and retrieve an API key:

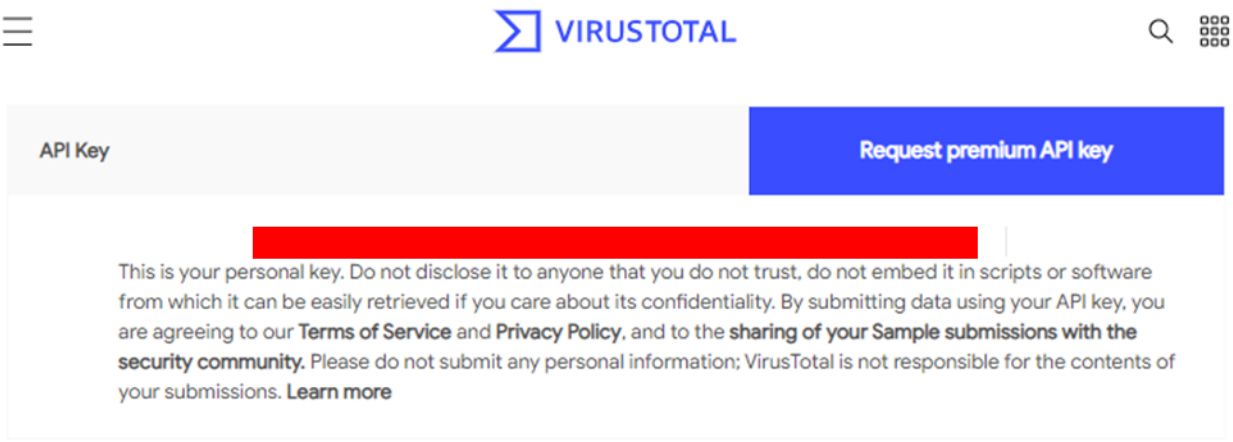
1. Open a web browser and browse to https://www.virustotal.com/gui/join-us[83]
2. Sign up for account with VirusTotal and activate the account via email.
3. Click the **API link** in the email, or login to VirusTotal, click your avatar in the upper right-hand corner and select **API key**.



User profile

4. Copy the API key to somewhere safe.

[82] https://developers.virustotal.com/v3.0/reference
[83] https://www.virustotal.com/gui/join-us



API key

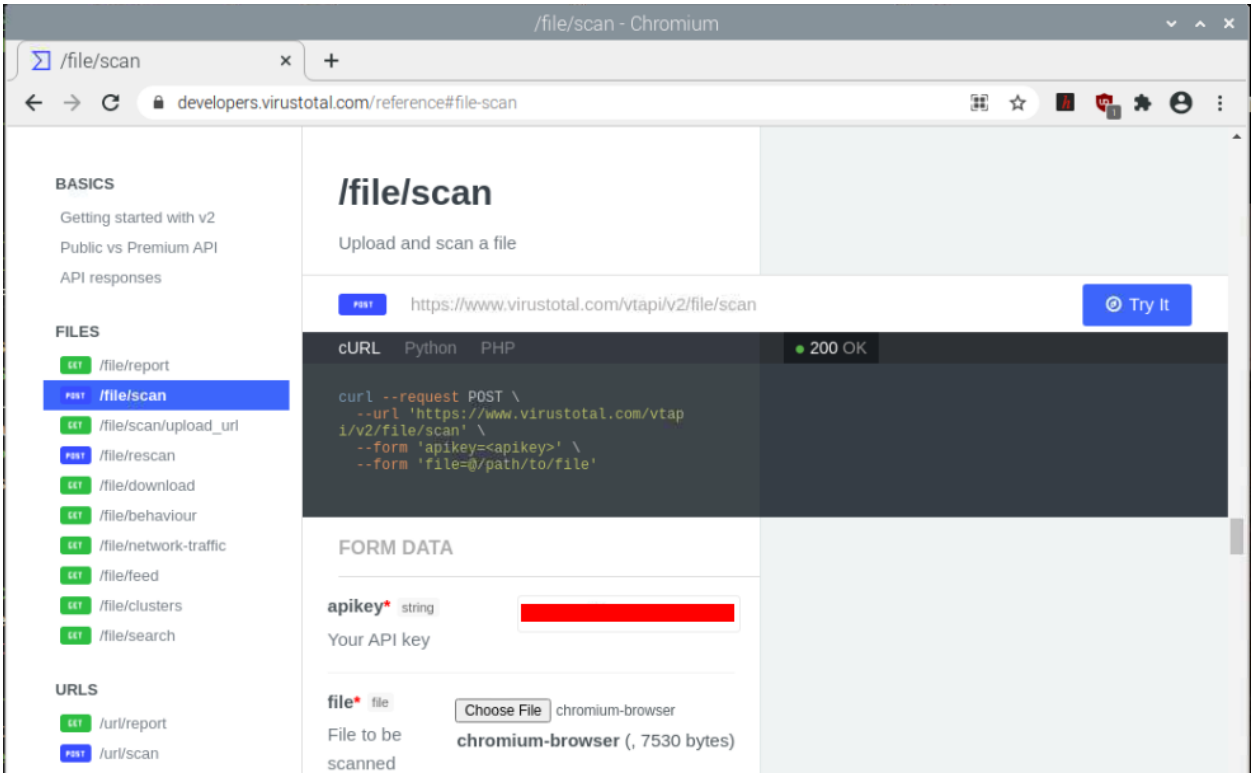
> **NOTE**: Your API key is private so do no share it, or save it somewhere that may be synchronized to GitHub.

Upload a file for scan

We will be using the VirusTotal API by manually uploading a file to scan. To make this easier, we will be using the sample code provided by the API reference page.

1. Browse to https://developers.virustotal.com/reference#file-scan[84]
2. In the **/file/scan** section, enter your **apikey**.
3. Click **Choose file** and select a file on your computer.

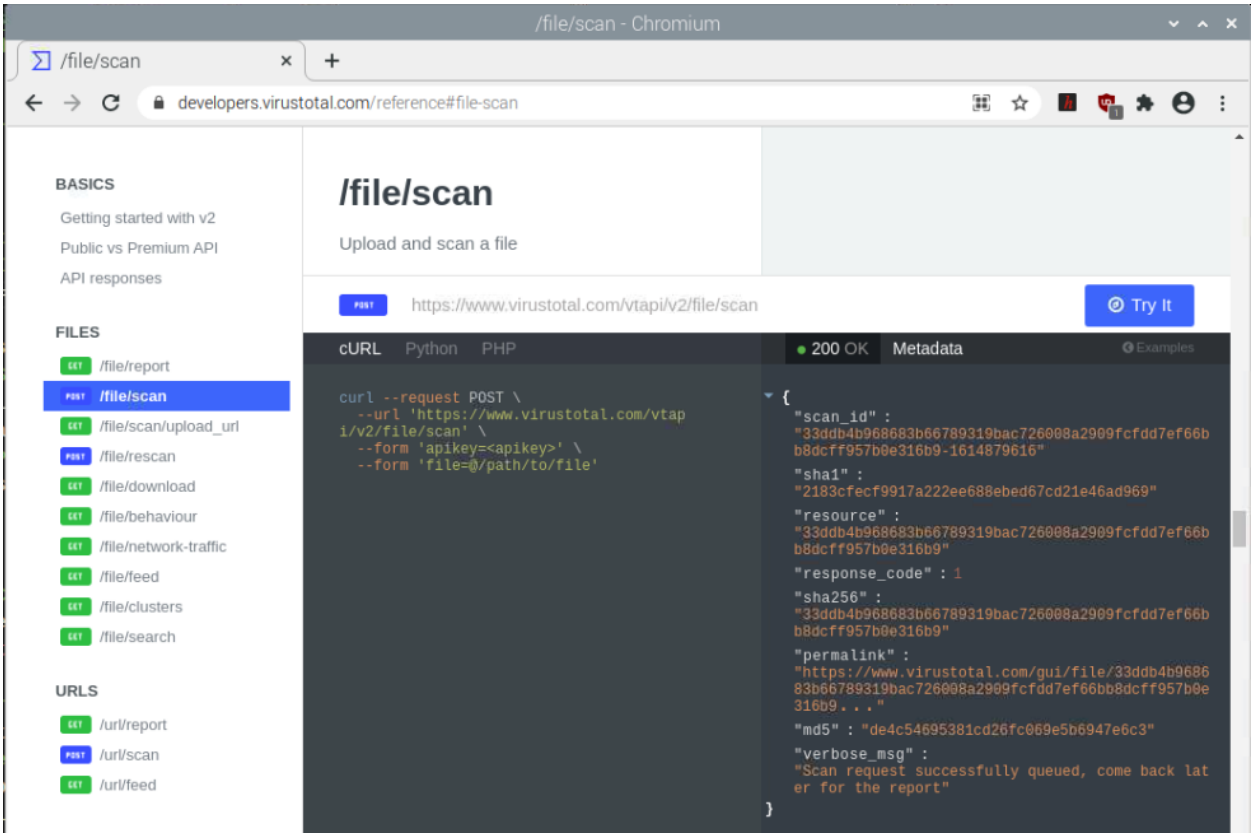
[84] https://developers.virustotal.com/reference#file-scan



/file/scan API reference

I chose **/usr/bin/chromium-browser** from the Raspberry Pi, a known good file.

4. Click **Try it**.
5. After a few seconds, the results should be displayed under the Try it button.



/file/scan results

6. In the response on the right, we can see the **verbose_msg** state that the file has been queued for scan, and that we can check later for the status of the scan.
7. Make note of the **scan_id**, **sha1**, **sha256**, **md5**, and/or **resource** responses. These will be used in the next step.

   NOTE: The API refence page also includes sample code for **cURL**, **Python**, and **PHP**. These can be used as a starting point for writing our own scripts.

Get results of scan

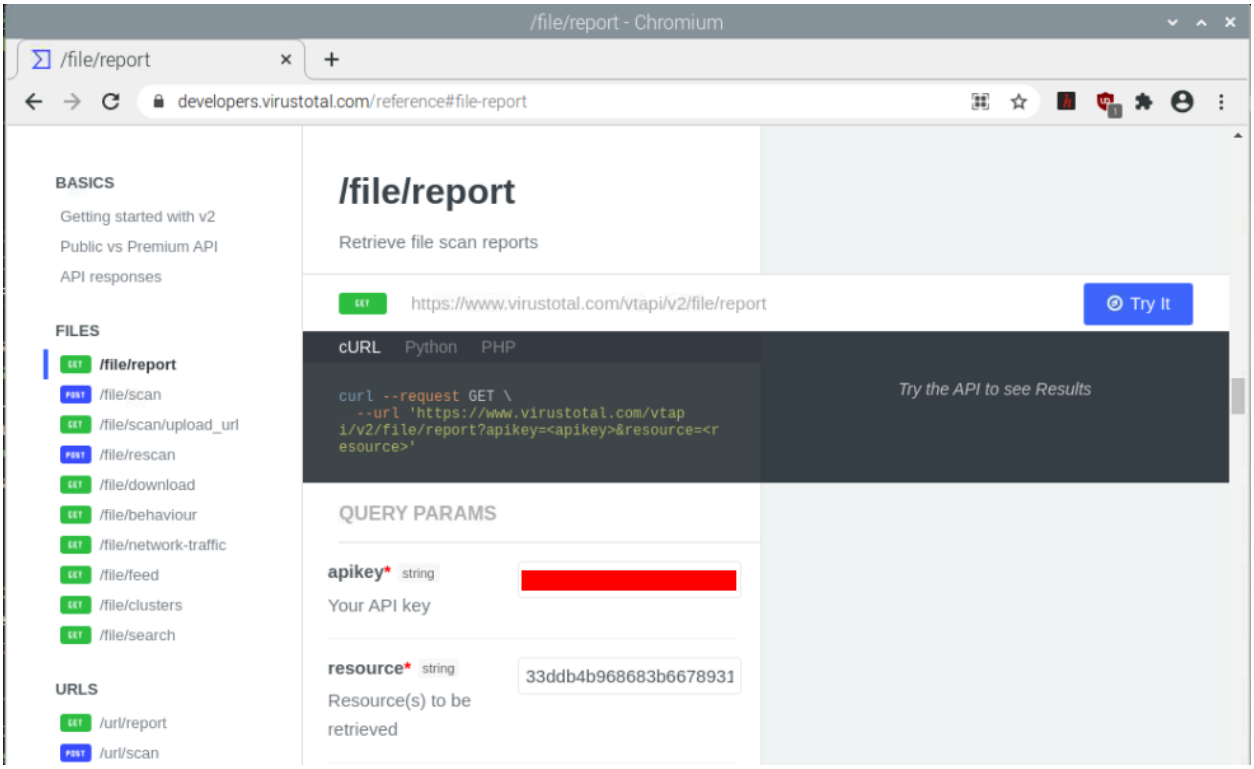
VirusTotal scans are performed in an asynchronous manner. First files are upload and then queued for scanning. Because the scanning may take several minutes, the requester must return at a later time to review the results.

To get the results of the scan

1. Browse to https://developers.virustotal.com/reference#file-report[85]

[85] https://developers.virustotal.com/reference#file-report

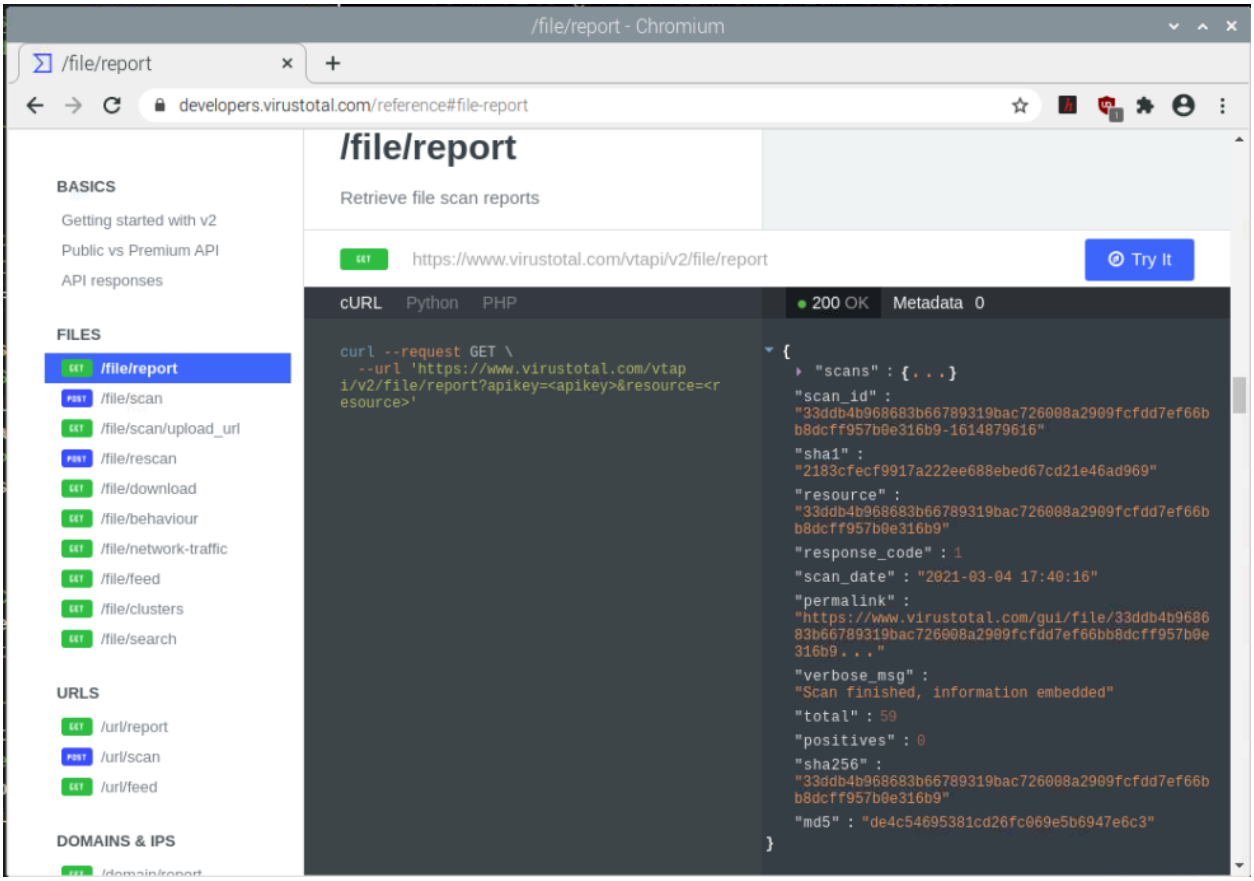
2. Enter the **apikey** and the **resource** to retrieve.



/file/report API reference

Note the comments below the **QUERY PARAMS** section. The resource can be the **MD5**, **SHA-1**, **SHA-256**, or **scan_id** of the file.

3. Click **Try it**.
4. After a few seconds, the results should be displayed on the right under the Try it button.



/file/report results

5. Note the numbers **total** and **positives**. Total is the number of scans performed against the file (**59**), positives is the number of scans that identified a virus (**0**).

   Note that the **/file/report** option can be called using the hash of a file. This means that instead of submitting the file, and then checking the report, you can first check the hash to see if the file has already been scanned. This can speed up your own scanning process by only uploading files that are new to VirusTotal.

There's more

In addition to the simple upload and scan performed here, VirusTotal provides several other useful features. For instance, once a file is scanned, VirusTotal can provide information about the contents of the file, what network activity the file attempts, and even community driven comments.

Scanning for Viruses with Python

Now that we have reviewed the documentation, know how VirusTotal works, created and confirmed our API key, and viewed sample Python code, we can begin our script to use the API.

Getting ready

Edit the **/home/pi/secrets.ini** file and add the **VirusTotal** API key to the file.

In Visual Studio Code, browse to **start/CH09/ScanFile.py** and open it for editing.

How to do it

```
#!/usr/bin/env python3
# Script that scans files using VirusTotal
# https://developers.virustotal.com/reference
# By Ed Goad
# date: 2/5/2021

# Import Python modules
import requests
import json
import hashlib
import configparser
import time
import os

def get_api_key(key_name):
    # Create the ConfigParser and load the file
    config = configparser.ConfigParser()
    config.read("/home/pi/secrets.ini")
    # Get the API key and return
    api_key = config["APIKeys"][key_name]
    return api_key

def check_virustotal_hash(token, hash):
    # Setup the base URL and Authorization header
    url = api_base + "/file/report"
    # Configure API key and file hash
    params = { 'apikey': token,
                'resource': hash
            }
    # Perform the request
    response = requests.get(
        url,
        params=params
        )
    # Convert the JSON to Python objects
```

```
36      items = response.json()
37      return items
38
39  def upload_virustotal_file(token, file_path):
40      # Separate file_name from file_path
41      file_name = os.path.basename(file_path)
42      # Setup the base URL and Authorization header
43      url = api_base + "/file/scan"
44      # Configure API key
45      params = { 'apikey': token }
46      # Setup file for upload
47      files = { 'file': (file_name, open(file_path, 'rb')) }
48      # Perform the request
49      response = requests.post(
50          url,
51          files=files,
52          params=params
53          )
54      # Convert the JSON to Python objects
55      items = response.json()
56      return items
57
58  def get_virustotal_comments(hash):
59      # here
60      return True
61
62  def hash_file(file_path):
63      # Setup buffer size and sha1 variables
64      buff_size = 65535
65      md5 = hashlib.md5()
66      sha256 = hashlib.sha256()
67      sha1 = hashlib.sha1()
68      # Open the file for reading in binary
69      with open(file_path, 'rb') as f:
70          while True:
71              # Read a chunk of data
72              data = f.read(buff_size)
73              # If we reached the end, quit while loop
74              if not data:
75                  break
76              # Update sha1 hash with data chunk
77              md5.update(data)
78              sha256.update(data)
```

```
            sha1.update(data)
    # Return hash
    #print("MD5   : {0}".format(md5.hexdigest()))
    #print("SHA256: {0}".format(sha256.hexdigest()))
    #print("SHA1  : {0}".format(sha1.hexdigest()))

    return sha256.hexdigest()


# Get API key from file
token = get_api_key("VirusTotal")
api_base = "https://www.virustotal.com/vtapi/v2"

# Prompt user for file to check
target_file = input("What file do you wish to scan? ")

# Generate SHA hash
file_hash = hash_file(target_file)

# Check /file/report using SHA
file_check = check_virustotal_hash(token, file_hash)

# Check if a scan has already been performed
if file_check['response_code'] == 1:
    print("Scan found")
    print("   {0} scans tested, {1} positive".format( \
        file_check['total'], file_check['positives']))
    print("   More details can be found at: {0}".format( \
        file_check['permalink']))
else:
    print("File not previously scanned, submitting")
    # Upload file to /file/scan
    upload_virustotal_file(token, target_file)
    print("File uploaded, sleeping for a maximum of 2 minutes")
    # Perform max of 6 loops, with a pause
    for i in range(3):
        # Sleep for 21 seconds
        # 21 is needed to due to API limit of 4/minute
        time.sleep(21)
        # Check for completed scan
        file_check = check_virustotal_hash(token, file_hash)
        if file_check['response_code'] == 1:
            # Successful scan found, print results and quit loop
```

```
122             print("Results returned")
123             print("   {0} scans tested, {1} positive".format( \
124                 file_check['total'], file_check['positives']))
125             print("   More details can be found at: {0}".format( \
126                 file_check['permalink']))
127             break
128
129 # If no results found, quit and suggest trying later
130 if file_check['response_code'] != 1:
131     print("Scans not finished, try again in a few minutes")
```

 Don't forget to commit the changes to Git and GitHub

How it works

This script starts on lines **#89 - #93** where the token and base URL (the location the API starts) are defined. The user is prompted for the file to scan and saves the path to the variable **target_file**.

Once the file name is identified, the `hash_file()` function is called on line **#96**. This function reads the file in chunks of 64kb and generates a the MD5, SHA1, and SHA256 hash of the file. When the hash is complete, the SHA256 is returned.

> **NOTE:** The file is read in 64kb chunks to facilitate working with large files. This "chunking" will slow down processing of smaller files which can be loaded entirely into memory. However, this process is required for larger files that may be bigger than the amount of available RAM.

Once the **file_hash** is determined, on line **#99** the `check_virustotal_hash()` function is called. This function uses the API token and hash to query the **/file/report** API. This checking of the hash is the first API call performed because it is possible that our file has already been scanned by VirusTotal. If already scanned, lines **#102-107** return the results and the script exits.

If VirusTotal hasn't already scanned the questionable file, we submit the file for scanning on line **#111**. This line calls our `upload_virustotal_file()` function along with our token and the file path. This function calls the **/file/scan** API and uploads the suspect file using an http.post on line **#49**. Because we already calculated the hash of this file, we don't need track any of the response information.

On line **#114** we begin a loop to check VirusTotal multiple time, with a 21 second sleep between each check. The `time.sleep()` function pauses our script to allow time for VirusTotal to scan our file before checking again. The time period, 21 seconds, is specifically chosen because the VirusTotal Public API allows only 4 calls per minute, and we have just performed 2 (the initial **check_virustotal_hash** and then **upload_virustotal_file**).

Every 21 seconds, the `check_virustotal_hash()` function is called on line **#119** using our previously determined file hash. If no result is found after several attempts, the script ends at line **#130** and reports to check back in several minutes.

There's more

Now that we have a working script to both submit and review the status of a single file, this can be expanded to support multiple files at once. This would allow for scanning of an entire system, which may or may not be compromised.

One possible way to accomplish this is to convert our script into 2 separate scripts that use a text file containing file names for input. The first script would loop through the lines of the text file to perform the hash and then lookup. If the file has previously been scanned, it can be removed from the text file (or otherwise commented out). If the status for the file was not safe, the results can be printed or stored for review.

When the first script is finished, the only items remaining in the text file should be previously unscanned files. The second script would upload these files for analysis. Later, the first script would be re-run to review the results.

See also

Hashing files in Python https://stackoverflow.com/questions/22058048/hashing-a-file-in-python[86]

Safe browsing on the internet

An API you may have already used, but didn't know it, Google's Safe Browsing API checks web URLs against a constantly updated list from Google. This helps to identify and warn users of unsafe websites that may contain malware, phishing, social engineering, or other unsafe content.

Details about the Safe Browsing API can be found at https://developers.google.com/safe-browsing/v4[87]

Getting stated

As we can see from https://developers.google.com/safe-browsing/v4/get-started[88], there are 4 steps necessary to get started: 1) Create an account, 2) Create a project, 3) Setup API key, and 4) Activate API key.

[86] https://stackoverflow.com/questions/22058048/hashing-a-file-in-python
[87] https://developers.google.com/safe-browsing/v4
[88] https://developers.google.com/safe-browsing/v4/get-started

Create account

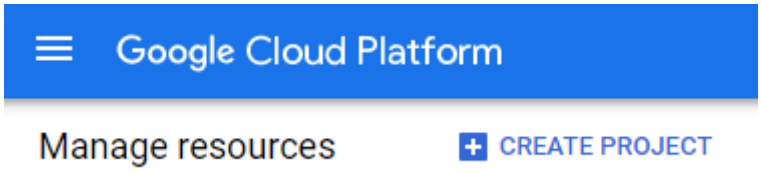
If you don't already have a Google account (frequently used for Google Mail), then you need to create one. Go to https://accounts.google.com/SignUp[89] and create an account.

Create project

1. Once you have logged into your Google account, browse to

https://console.cloud.google.com/cloud-resource-manager[90]

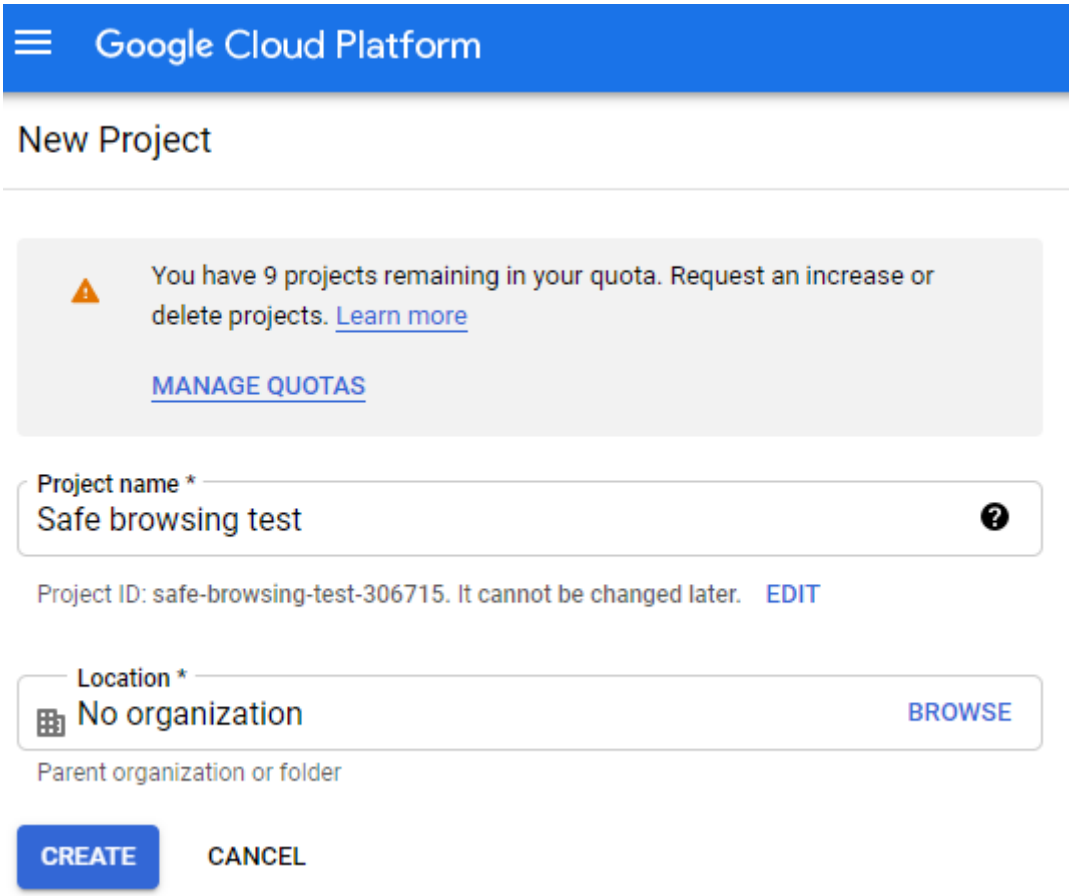
2. On the top of the screen, click **CREATE PROJECT** to begin creating a project.



Starting GCP project

3. On the **New Project** page, give your project a name and click **Create**.

[89] https://accounts.google.com/SignUp
[90] https://console.cloud.google.com/cloud-resource-manager

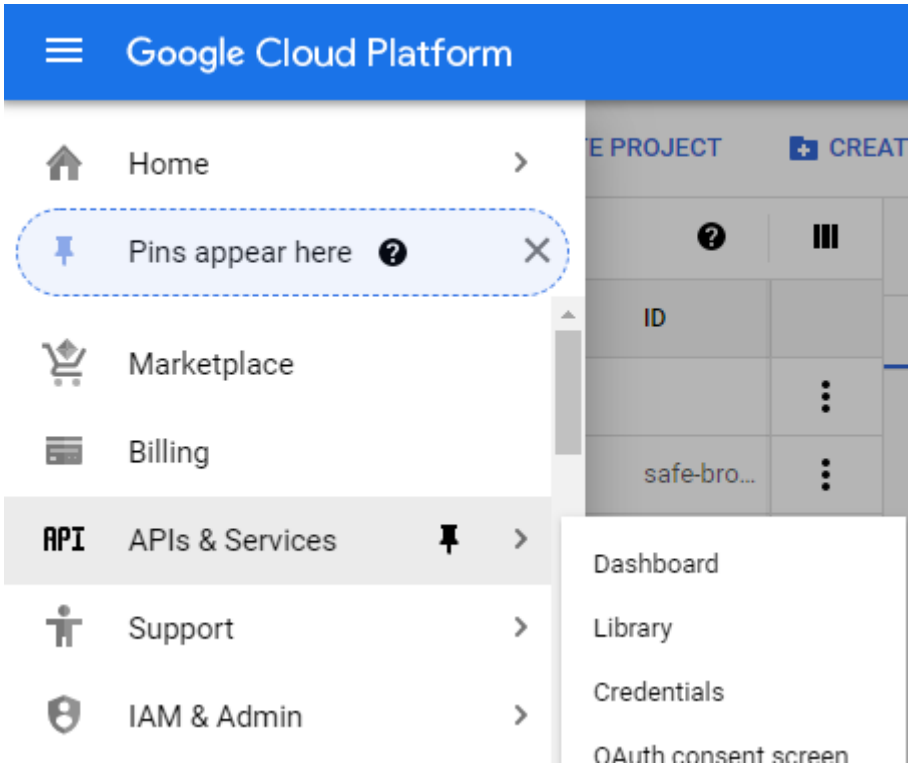


Creating GCP project

Setup API key

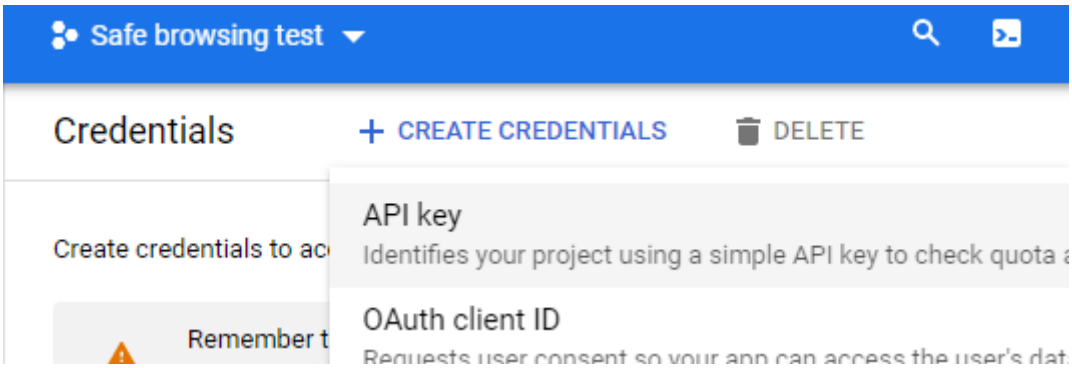
Once the project is created, we need to create an API key for the project.

1. In the **Google Cloud Platform** page, click the 3 horizontal lines on the top bar and select **APIs & Services | Credentials**.



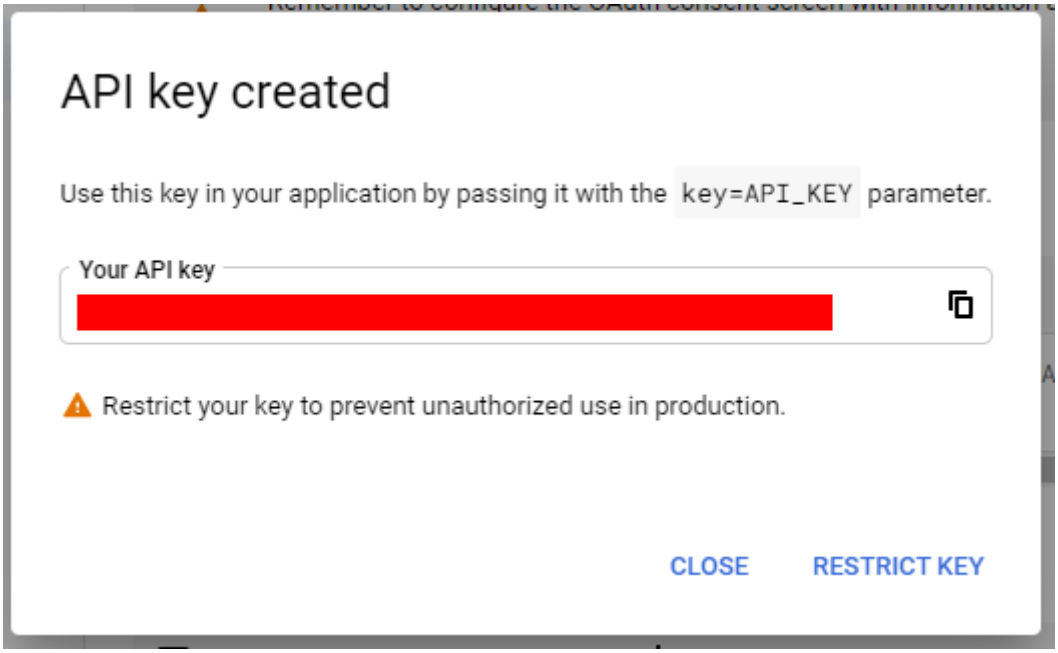
Accessing API credentials

2. At the top of the page, ensure the recently created project is selected.
    - Use the drop down to change projects if necessary.
3. Click **CREATE CREDENTIALS | API key**.



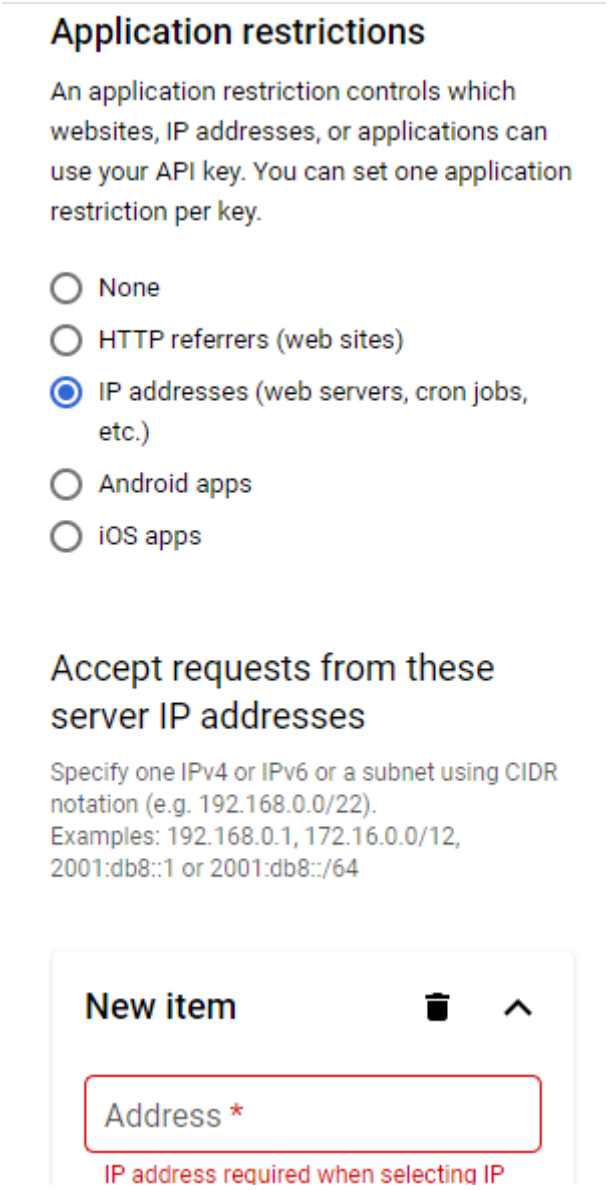
Generate API key

4. Copy the API key.



GCP API key

5. To prevent the key from being used maliciously, it is best practice to put restrictions on your API key. Click **RESTRICT KEY**.
6. In the **Restrict and rename API key** page, give the key a friendly name.
7. Under **Application restrictions** select **IP address**.
8. Enter your public IP address in the **Accept requests from these server IP addresses** section.



Application restrictions

NOTE, if you need to find your Public address, you can go to https://www.whatismyip.com/[91] to get your IPv4 and IPv6 address

9. When finished, click SAVE at the bottom

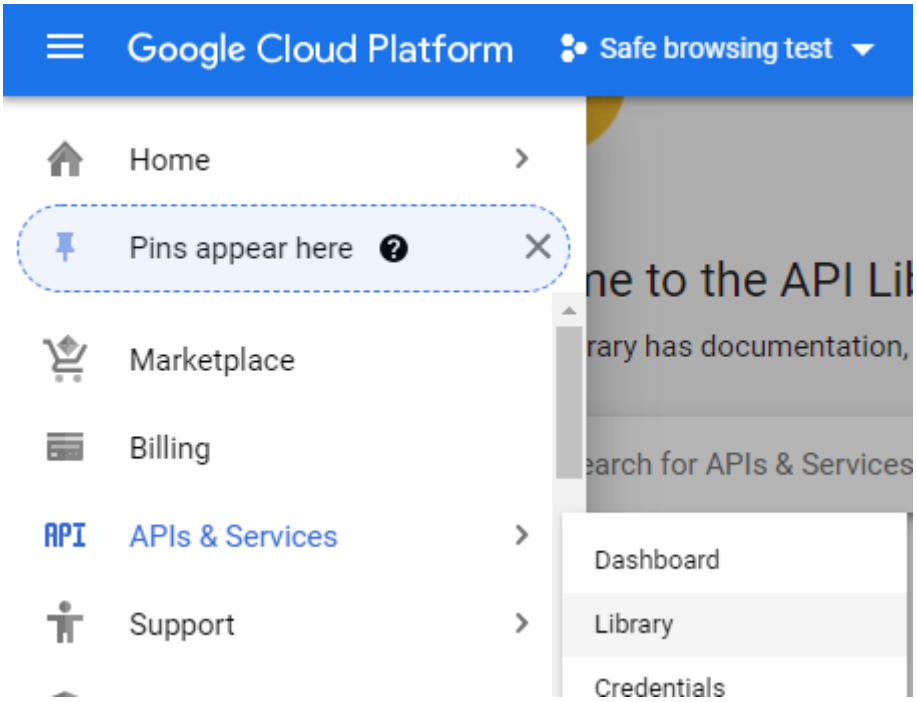
NOTE: We are restricting access to the API to our current IP address which works for testing. This will restrict the API from being used in a different location and may cause issues if your address changes or you use the online PostMan service. For production usage, other choices should be used instead.

[91] https://www.whatismyip.com/

Activate API

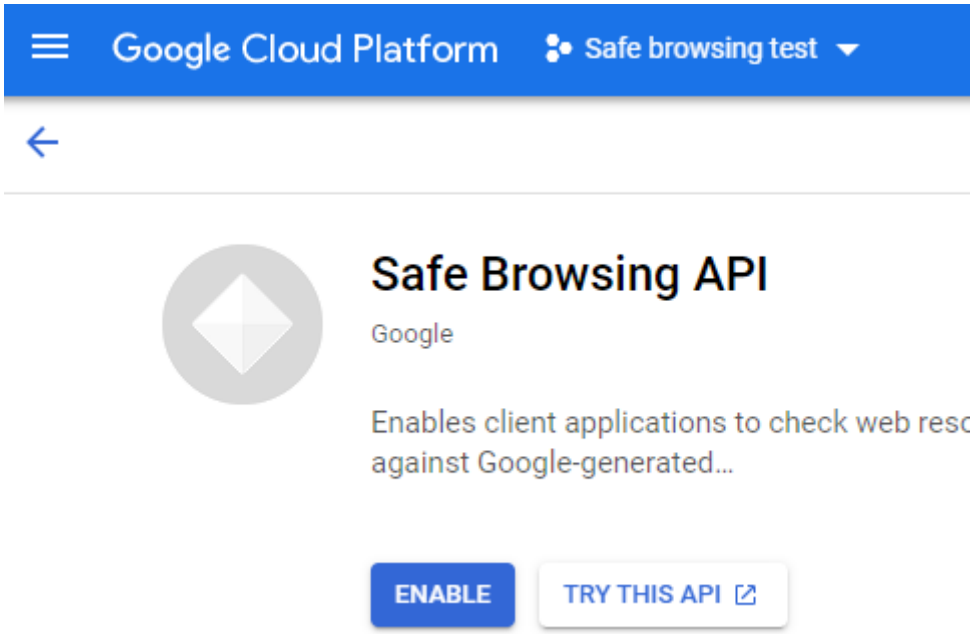
Lastly, we need to activate the key to work with the Safe Browsing API.

1. Click the 3 dashes at the top of the page and select **APIs & Services | Library**.



API Library

2. At the top of the page, ensure the recently created project is selected.
    - Use the drop down to change projects if necessary.
3. In the **Search for APIs & Services** box, begin typing Safe Browsing API and select **Safe Browsing API**.
4. On the **Safe Browsing API** page, click **ENABLE**.



Enabling API

PostMan 1

To generate our first API lookups, we will be using the sample information provided by the API at https://developers.google.com/safe-browsing/v4/lookup-api[92]. In the documentation, we will use the **Request header** and **Request body** information to begin our first lookup

A few details about the API to note before moving forward:

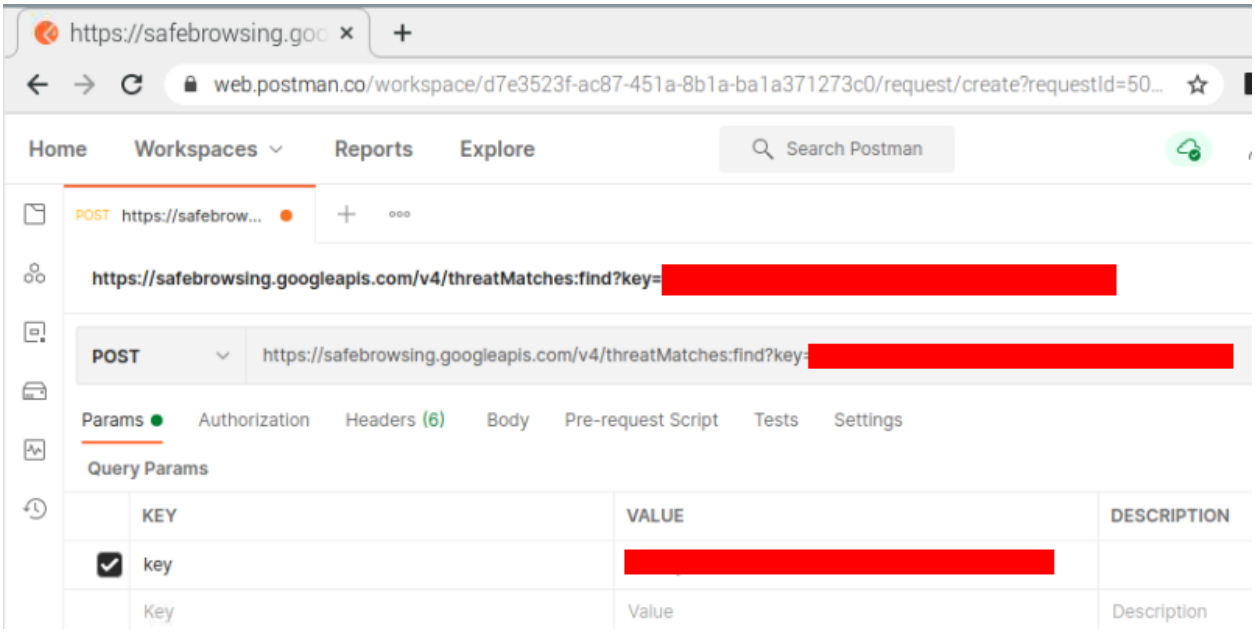
- In the **Request header** section, it identifies this as a **POST** request.
- The URL **https://safebrow**…. ends in **key=API_KEY**. Your API key will go here.
- **HTTP/1.1** is listed, most tools will use this without specifying.
- A **Content-Type** needs to be specified in our request.
- In the **Request body** section, there are several areas to fill in details.

To try our first lookup:

1. Open **PostMan** and click the plus sign (**+**) to open a new tab.
2. Click where it says **GET** and change the method to **POST**.
3. Next to **POST**, where it says **Enter request URL**, enter the URL from the **Request header** section in the API documentation.
4. Below the URL, select the **Params** tab if not already selected.

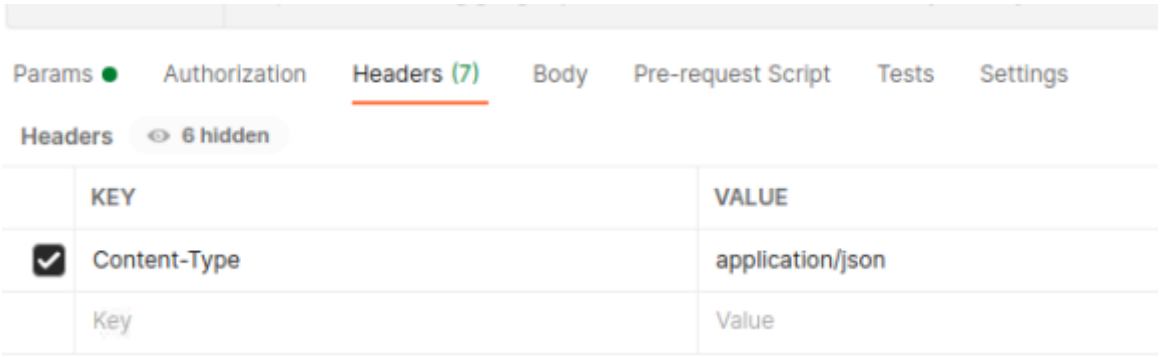
[92] https://developers.google.com/safe-browsing/v4/lookup-api

5. In the **Query Params** section, ensure there is a **KEY** named **key**, and enter your API key in the **VALUE** section. You will notice the URL grows to include the key.



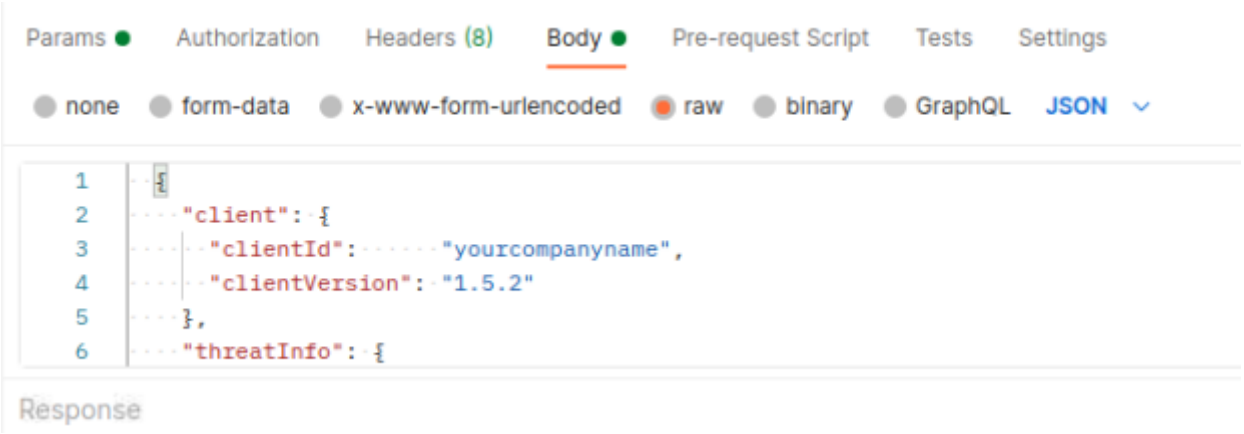
Postman API key

6. Click the **Headers** tab.
7. Under **Headers** , add a new **KEY** named **Content-Type** , and a **VALUE** of **application/json**.



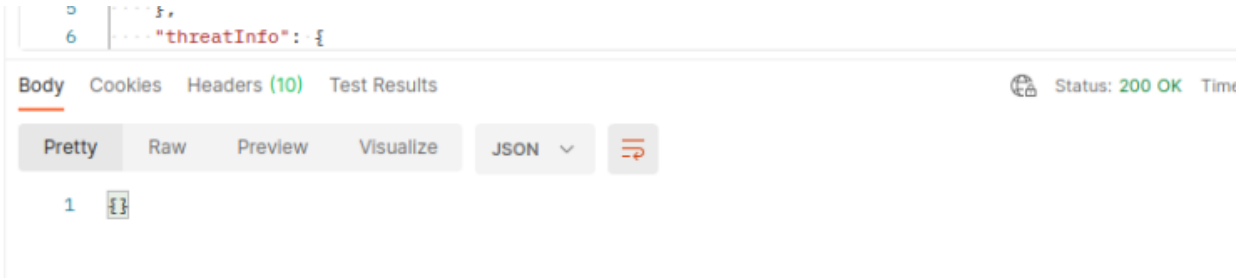
Setting API headers

8. Click the **Body** tab.
9. Under **Body**, select **raw** and ensure **JSON** is selected in the drop-down on the right.
10. In the **Body** section, paste the **Request body** sample the Google documentation.



Entering body content

11. Click **Send**.
12. After a few moments, the **Response** will be returned. If an empty result is returned (as shown below), then no issues were reported.



Reviewing results

> **NOTE:** Confirm the **Status: 200 OK** is shown. If a non 2xx status is returned, something is not working properly.

PostMan 2

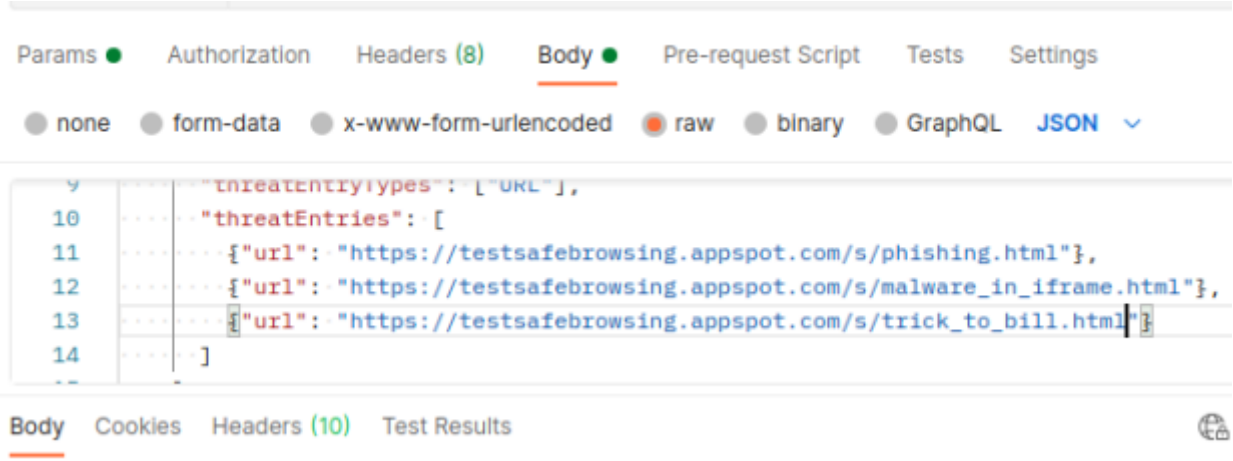
Now that we have PostMan setup and working, we can change the content of our request to scan actual URLs. To test the API is working we will be using links from https://testsafebrowsing.appspot.com/[93] to help generate alerts.

> **NOTE:** I am not associated with this site and cannot vouch for its safety. Caution is suggested if you directly browsing to this site.

1. Return to PostMan and select the **Body** tab for the request if not already selected.

[93] https://testsafebrowsing.appspot.com/

2. Scroll through the content of the **Body** section until we see the 3 URLs previously pasted.
3. Replace these URLs with different pages found on the **testsafebrowsing** page as shown below.



Updating API data

4. Click **Send**.
5. Review the response **Body** to see the alerts.

Reviewing results

Troubleshooting

If the response body is empty:

- Confirm the status is still **200 OK**. If you have a status that is in the **4xx** or **5xx** range, review your configuration. More details can be found at https://developers.google.com/safe-browsing/v4/status-codes[94]

[94] https://developers.google.com/safe-browsing/v4/status-codes

- Review the request body and look for typos. The most common are missing quotes and commas. If necessary, compare your request body to the original API example and/or restart from the example.
- Try different URLs from the testsafebrowsing website. Different pages will cause different types of alerts.
- Lastly, its possibly the testsafebrowsing site isn't working. Search online for other "test" pages you can submit.

How it works

API Body details

In the first PostMan example, we used the **Body** that was provided to us from the API example. This body is a JSON string that contains many details we can/should update.

```
{
  "client": {
    "clientId":      "yourcompanyname",
    "clientVersion": "1.5.2"
  },
  "threatInfo": {
    "threatTypes":      ["MALWARE", "SOCIAL_ENGINEERING"],
    "platformTypes":    ["WINDOWS"],
    "threatEntryTypes": ["URL"],
    "threatEntries": [
      {"url": "http://www.urltocheck1.org/"},
      {"url": "http://www.urltocheck2.org/"},
      {"url": "http://www.urltocheck3.com/"}
    ]
  }
}
```

API body

In the second PostMan example, we updated the **Body** to include different URLs to check that *should* return results. By using the testsafebrowsing web page, we can now see how results are returned from the API for both trustworthy and suspicious web sites. Using this information, we can see the type of information returned, and then make decisions based on that information.

There's more

More details about the available options can be found by using the Safe Browsing Lists API at https://developers.google.com/safe-browsing/v4/lists[95]. This API returns a list of all available threatTypes, platformTypes, and threatEntryTypes, and the available combinations.

client

The first section in the body is the client description. This information is used for logging and accounting and should be unique and descriptive. The clientId can be your company name, your application name, or even the company you are creating the application for.

The clientVersion should be a number that is incremented everytime you update your program. The clientId and clientVersion can be helpful for reporting later if you are supporting the application, and need to know which versions are still in use.

threatTypes

The threatTypes section specifies the specific types of threats to search for. It isn't always necessary to check for all types of threats. As of writing this, the following threatTypes are available:

- MALWARE
- SOCIAL_ENGINEERING
- POTENTIALLY_HARMFUL_APPLICATION
- UNWANTED_SOFTWARE

platformTypes

The platformTypes specifies the platform (normally the OS) to test for vulnerabilities. Because some vulnerabilities only exist on 1 operating system, we can limit our search to our current platform(s). As of writing this, the following platformTypes are available:

- ANY_PLATFORM
- WINDOWS
- LINUX
- OSX
- CHROME
- ANDROID
- IOS

[95] https://developers.google.com/safe-browsing/v4/lists

threatEntryTypes

The threatEntryTypes section identifies the type of entries to be evaluated. As of writing this, the following options are available:

- URL
- IP_RANGE

threatEntries

The threatEntries section includes the entry or entries to be evaluated.

See also

More details about the Safe Browsing API https://developers.google.com/safe-browsing/v4/reference/rest[96]

Scanning URLs with Python

Now that know how the Safe Browsing API works, what information is needed, how results are returned, and have confirmed our API key works, we can create our first script.

Getting ready

Edit the **/home/pi/secrets.ini** file and add the **GoogleSafeBrowsing** API key to the file.

In Visual Studio Code, browse to **start/CH09/CheckURL.py** and open it for editing.

How to do it

```
#!/usr/bin/env python3
# Script that checks URLs against Google's Safe Browsing API
# https://developers.google.com/safe-browsing/v4
# By Ed Goad
# date: 2/5/2021

# Import Python modules
import requests
import json
import configparser

```

[96] https://developers.google.com/safe-browsing/v4/reference/rest

```
def get_api_key(key_name):
    # Create the ConfigParser and load the file
    config = configparser.ConfigParser()
    config.read("/home/pi/secrets.ini")
    # Get the API key and return
    api_key = config["APIKeys"][key_name]
    return api_key

def check_safebrowsing_url(token, threat, platform, test_url):
    clientId = "test_python_client"
    clientVersion = "0.0.1"
    # Setup the base URL and Authorization header
    api_base = "https://safebrowsing.googleapis.com/v4"
    url = api_base + "/threatMatches:find?key=" + token
    # Configure API key and file hash
    params = { "client": {
                    "clientId": clientId,
                    "clientVersion": clientVersion
                },
                "threatInfo": {
                    "threatTypes": threat,
                    "platformTypes": platform,
                    "threatEntryTypes": "URL",
                    "threatEntries": {
                        "url": test_url
                    }
                }
            }
    # Perform the request
    response = requests.post(
        url,
        json=params
        )
    # Convert the JSON to Python objects
    items = response.json()
    return items


# Get API key
token = get_api_key("GoogleSafeBrowsing")

# Prompt user for platformTypes, Default to "LINUX"
platform = input("What platform are you scanning form [LINUX] ") \
```

```
55      or "LINUX"
56
57  # Default threatTypes to "MALWARE"
58  threat = input("What threat type [MALWARE] ") or "MALWARE"
59
60  # Prompt user for threatEntries
61  url = input("What URL to scan? ") or \
62      "http://testsafebrowsing.appspot.com/s/malware.html"
63
64  print(check_safebrowsing_url(token, threat, platform, url))
```

 Don't forget to commit the changes to Git and GitHub

How it works

The script starts on line **#51** where the API key is retrieved from the secrets.ini file.

Lines **#54** - **#60** prompt the user for the various information needed for the Safe Browsing API. Note that after the **input()** function there is an or statement and a default value is provided. This is a common method for providing default values in Python, so the user can simply press **Enter** to accept the default.

On line **#62**, the **check_safebrowsing_url()** function is called with the API key, platform, threat type, and URL to scan. The function, starting on lines **#21** - **#25**, begins by identifying base information such as the **clientID** and **clientVersion**.

On line **#27**, the script begins to create a Python dictionary object named **params**. This is a somewhat complex dictionary object because **params** dictionary object contains 2 additional dictionaries named **client** and **threatInfo**. The **threatInfo** dictionary object contains yet another dictionary named **threatEntries**.

 Use the debugger to walk through the script for more details on how it works.

This construction of the **params** dictionary may be the most complex and difficult to understand portion of the script. Using the debugger to view the contents of the variable can be helpful in understanding how we are constructing the **Request body**.

There's more

This sample script is only a beginning for full-fledged tool. We can further expand the script to include several checks such as validating the user input, and confirming the API returns a success status code.

Once our script is fully functioning, we can customize it to pass lists of URLs to scan multiple pages at once. One possible use for this is to scan your corporate website to ensure no malicious data is hosted on it.

See also

Python dictionary object

https://www.w3schools.com/python/python_dictionaries.asp[97]

https://docs.python.org/3/tutorial/datastructures.html#dictionaries[98]

https://realpython.com/python-dicts/[99]

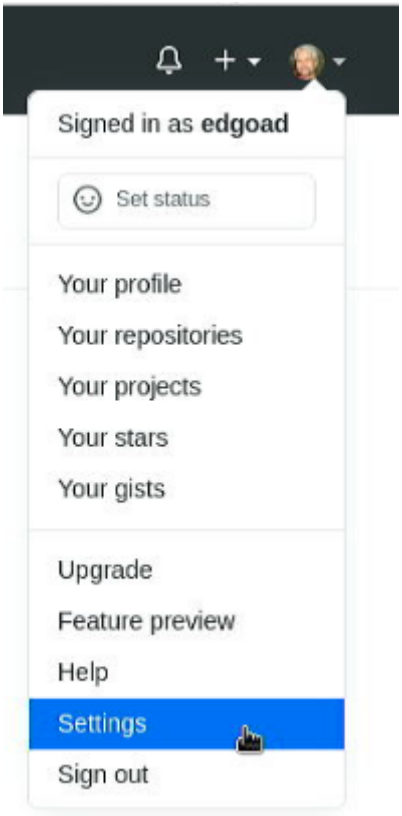
https://www.programiz.com/python-programming/dictionary[100]

[97] https://www.w3schools.com/python/python_dictionaries.asp
[98] https://docs.python.org/3/tutorial/datastructures.html#dictionaries
[99] https://realpython.com/python-dicts/
[100] https://www.programiz.com/python-programming/dictionary

Appendix - Using Git

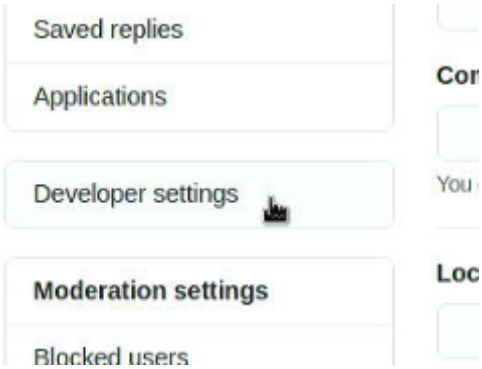
Creating a Personal Access Token

1. If not already, log into **GitHub**.
2. Click on your name / Avatar in the upper right corner and select **Settings**.



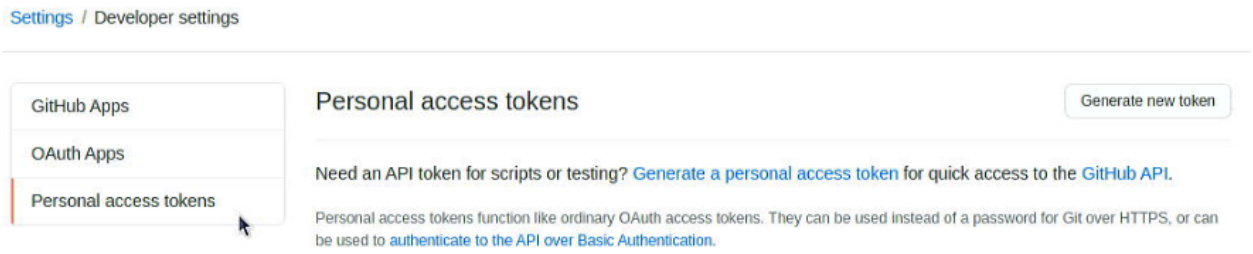
GitHub settings

3. On the left, click **Developer settings**.



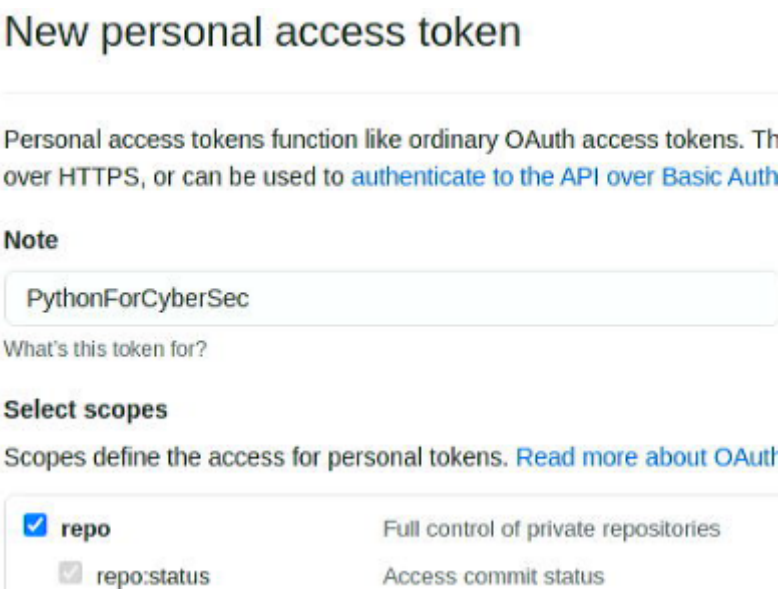
Developer settings

4. Select **Personal access tokens** and click **Generate new token**.



Generating token

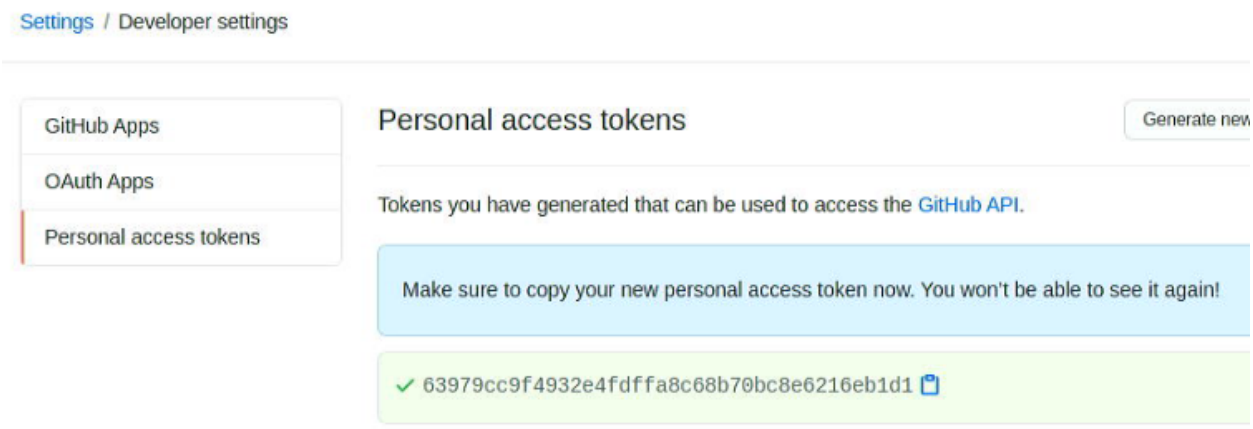
5. Give the token a description/name and select the scope of the token.



Setting token scope

- I selected **repo** only to facilitate pull, push, clone, and commit actions.
- Click the link **Read more about OAuth scopes** for details about the permission sets.

6. Click **Generate token**.
7. Copy the token - this is your new password!



Copying token

> **NOTE:** The token is only available at this point. If you lose the token, you will need to regenerate it.

Cloning GitHub locally

Once we have a token, we need to configure the local Git client with a username and email address. On the Raspberry Pi, open a terminal window. Type in the following commands to configure username, email, and credential helper, replacing the values in the brackets (<>) with your username and email.

```
git config --global user.name "<your name here>"
git config --global user.email "<your email here>"
git config --global credential.helper cache
git config -l
```

> **NOTE:** The option `credential.helper cache` command will assist us in keeping a copy of the Personal Access Token on the computer. This means it won't prompt us for passwords everytime we use it.

Now that Git is configured, we will clone from our GitHub repository with the following command:

```
git clone <github URL here>
```

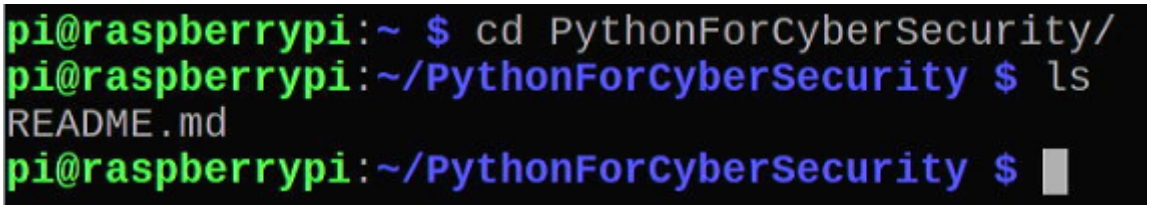
Cloning repository

When prompted for **Username** and **Password**, enter your **GitHub username** and the **Personal Access Token** you created. When finished, it will say several objects were identified.

Successful clone

> **NOTE:** If you lost the token or its not working, delete and recreate the token on the website

When the repository is cloned, it creates a folder with the same name as the repository. You can use **cd** to change into is and **ls** to view the contents.



Viewing local repository

Pushing changes to GitHub

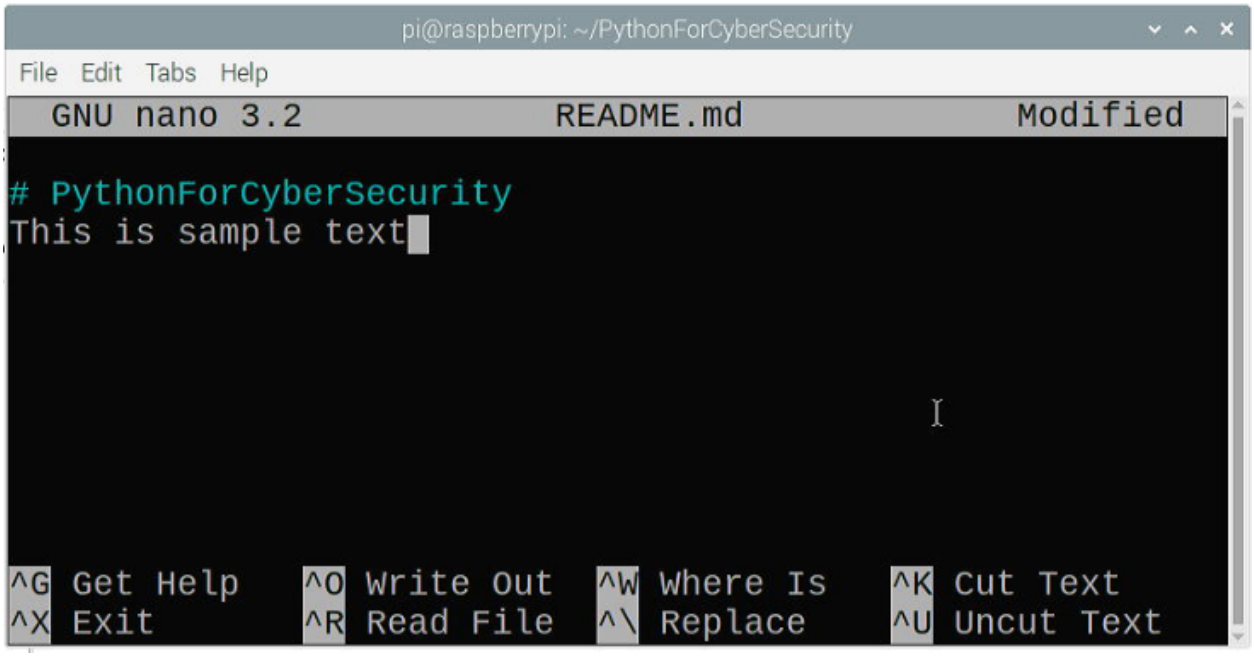
To push changes back to GitHub, it is a simple process of making changes, **add**ing the changes, **commit**ting them, **push**ing them to GitHub, and reviewing them online.

We will begin by editing the **README.md** file using the **nano** editor. Nano is a simple text editor that is installed on the Raspberry Pi OS, and is fairly easy to use. To edit the README file:

1. In your console window, ensure you are in the folder of the cloned GitHub repository.
2. Type `nano README.md`

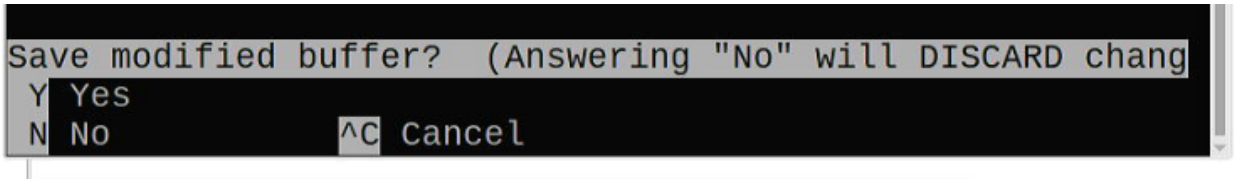
- **Note:** file names and commands are case-sensitive on Linux.

3. On the keyboard, press the "down arrow" to move the cursor, and then type in some text.



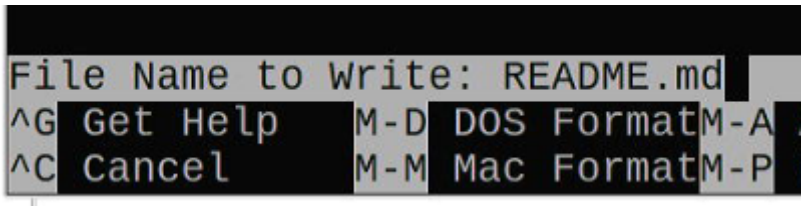
Editing README.md

4. On the keyboard, press **CTRL + X**.
   - **NOTE:** You can see at the bottom of the window the section "**^X Exit**" that suggests this will exit the editor.
5. When prompted **Save modified buffer?** Press **Y** on the keyboard.



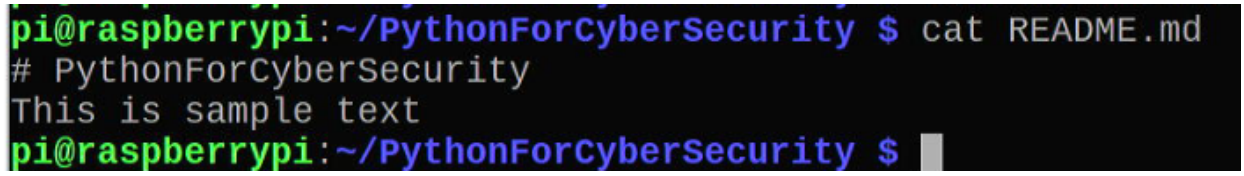
Saving file

6. When prompted for **File Name to Write**, hit **Enter** on the keyboard to accept the default.



Confirm file name

7. The **nano** editor closes automatically. You can confirm the changes were saved by typing `cat README.md`

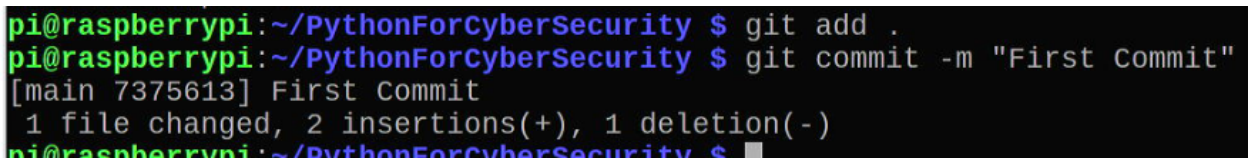


Viewing file contents

Now that changes have been made to the local file, we first **add** the changes, then **commit** them and lastly **push** them to GitHub.

1. To add the changes to the local repository, type `git add .` (note the trailing period ".")
2. To commit the changes, type `git commit -m "first commit"`

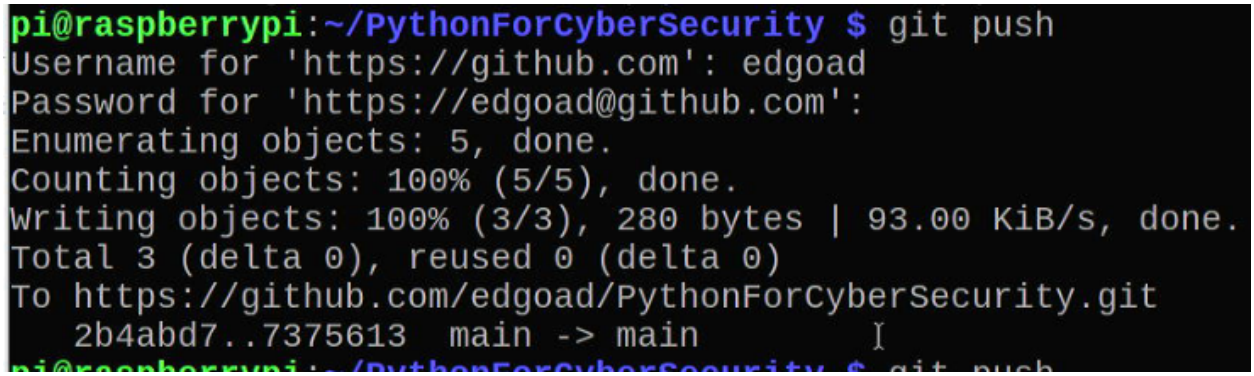
```
1 git add .
2 git commit -m "first commit"
```



First commit

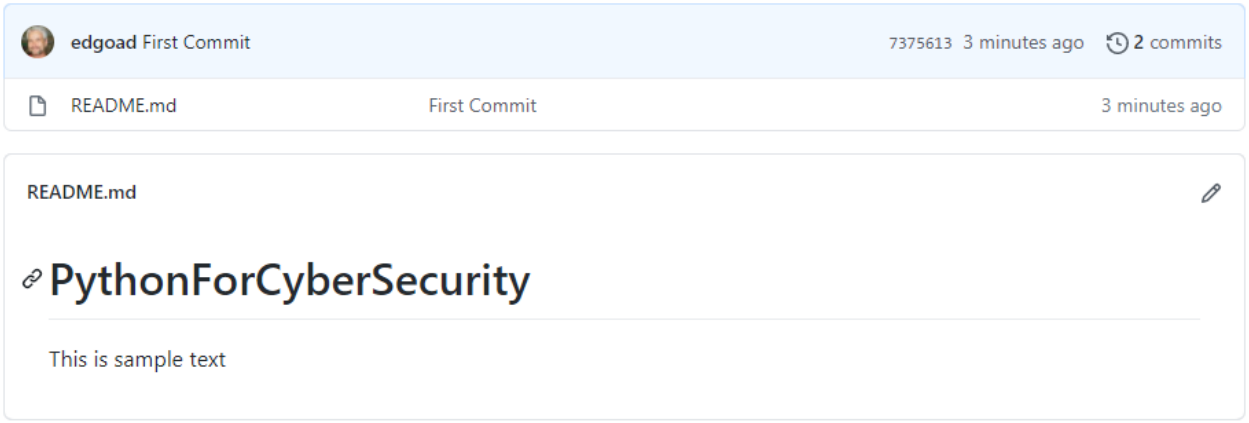
> **NOTE**: The text in quotes is your "commit comment". Every commit must have a comment and can be used later to assist in finding version changes

3. To push the changes to GitHub, `type git push`



Pushing to GitHub

Once the changes have been pushed, log into GitHub via the web browser and browse to your repository. You will now see the **README.md** file has been updated, and the contents are shown on the page.

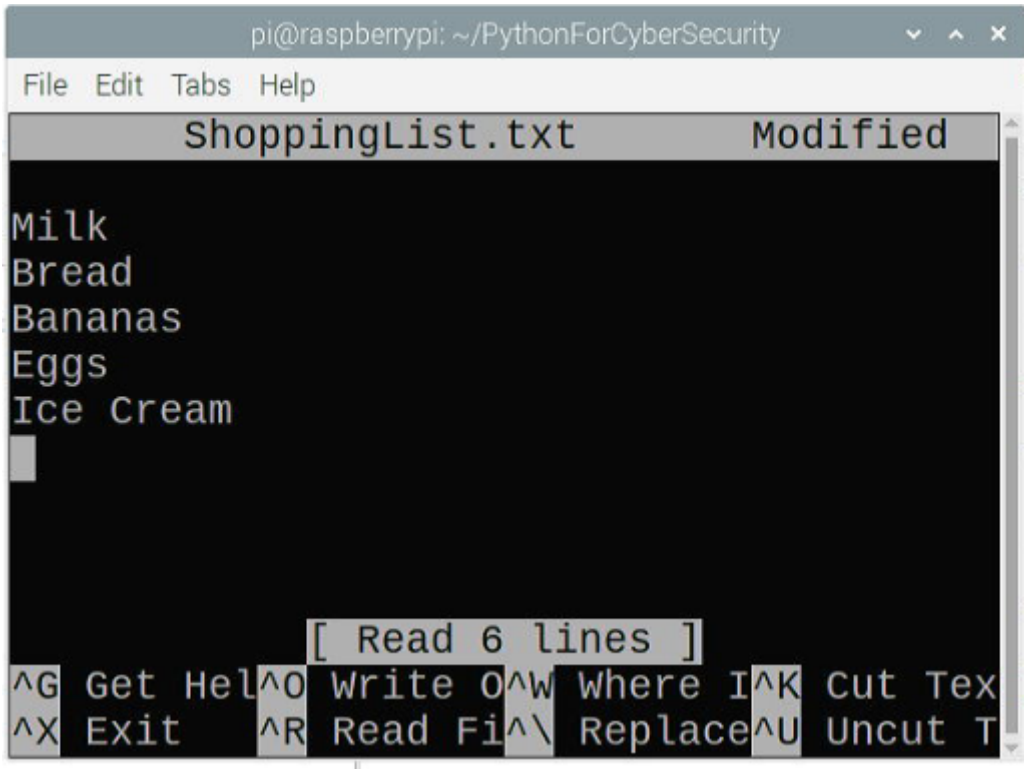


Viewing changes online

Tracking changes over time

One of the great benefits of Git and GitHub is the ability to track changes over time. To highlight this, we will use the idea of creating and updating a shopping list. First, we will create a Shopping List, and make several changes to it, ensuring we commit the changes to Git and GitHub each time.

1. On the command line, type `nano ShoppingList.txt`
2. Inside the editor, enter the following lines.

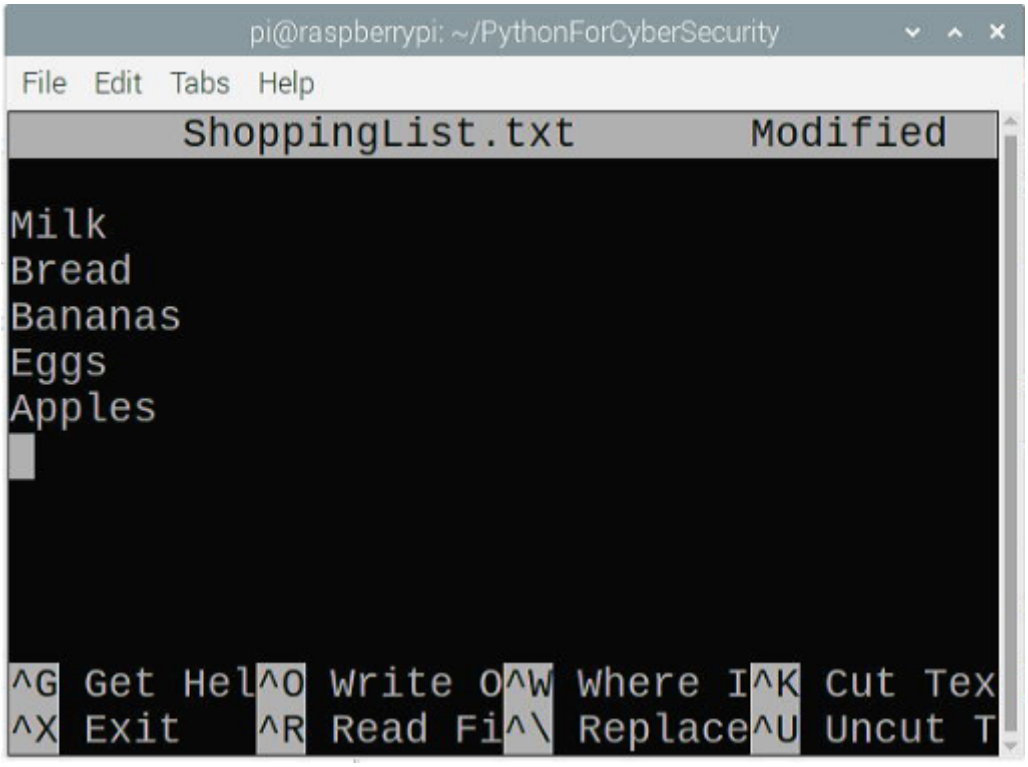


Editing ShoppingList

3. Once finished, press **CTRL + X**.
4. When prompted to save, press **Y**.
5. When prompted for the filename, press **Enter** to accept the default name.
6. Back at the terminal, run the following commands to commit the changes.

```
git add .
git commit -m "created shopping list"
git push
```

7. Once the file is commit to Git, open the file again by typing `nano ShoppingList.txt`
8. Inside the editor remove **Ice Cream** and add **Apples**

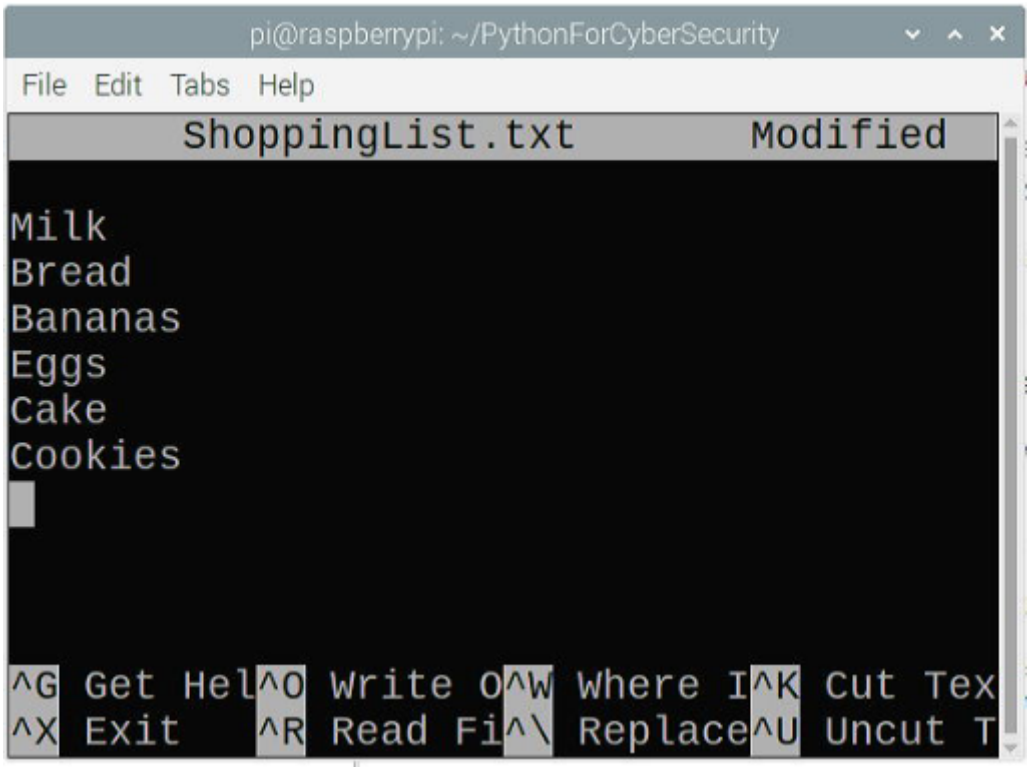


Editing ShoppingList

9. Save the file by pressing **CTRL + X** , press **Y** , and **Enter** to accept the default name.
10. Back at the terminal, run the following commands to commit the changes.

```
git add .
git commit -m "Removed Ice Cream"
git push
```

11. Once more, open the file by typing `nano ShoppingList.txt`
12. Inside the editor, add **Cake** and **Cookies** on separate lines.



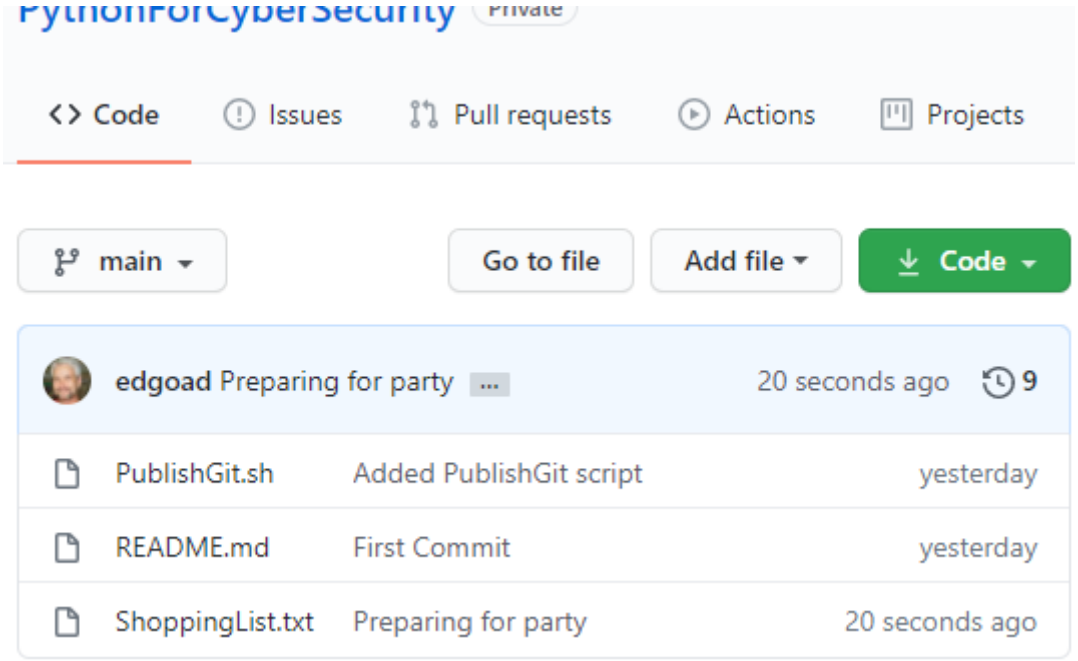
Editing ShoppingList

13. Save the file by pressing **CTRL + X** , press **Y** , and **Enter** to accept the default name.
14. Back at the terminal, run the following commands to commit the changes.

```
git add .
git commit -m "Preparing for party"
git push
```

Now that our file has several edits, with each edit being committed to Git and GitHub, we can now view those edits. The easiest way to see these is to login to GitHub

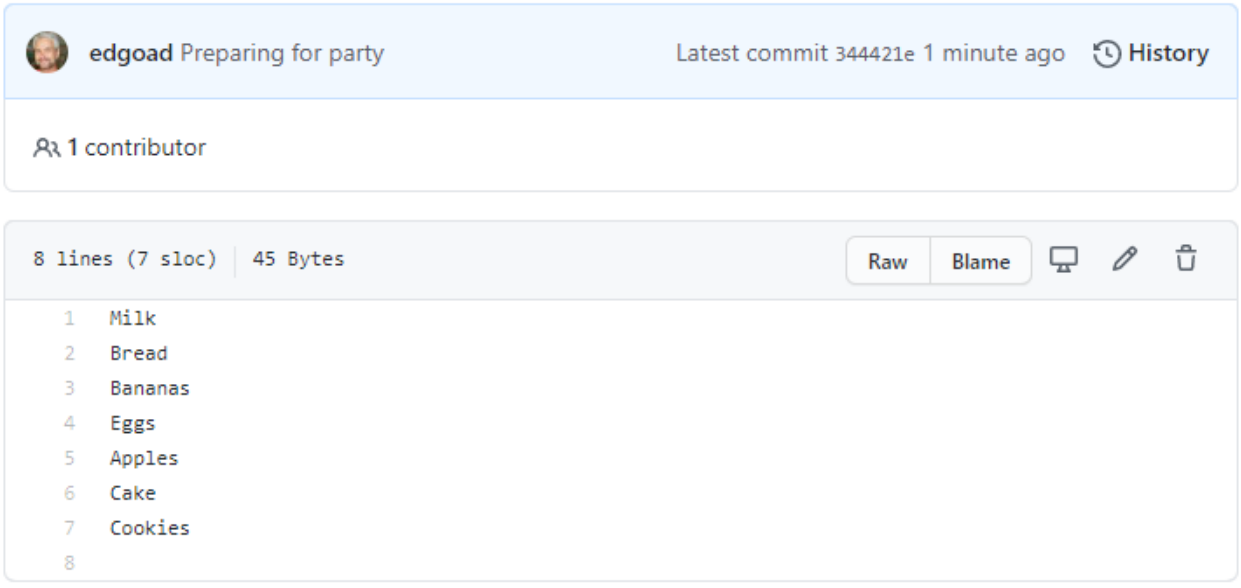
1. Open a web browser and login to **GitHub.com**.
2. If necessary, select the repository and you will see the **ShoppingList.txt** file has been added.



Viewing files online

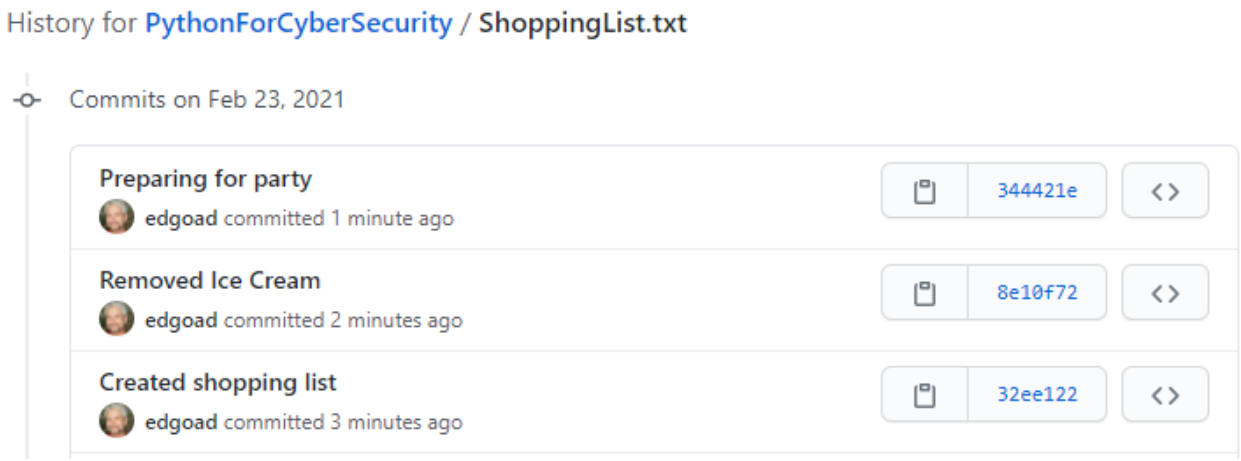
Note the comment next to the file and how recently the file was updated.

3. Click on the **ShoppingList.txt** and you will see the most recent contents of the file.



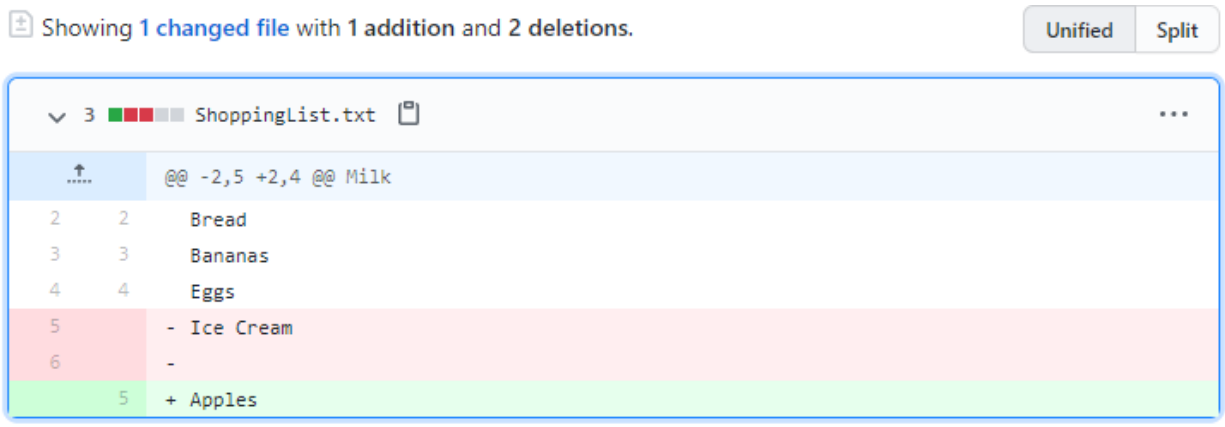
Viewing ShoppingList

4. In the box above the file contents, you will see a link named **History**. When you click this link it will show you a history of changes to the file.



Viewing file history

5. If you click on the link that says **Removed Ice Cream** , you will be taken to a page that shows you the changes (additions and deletions) to the file.



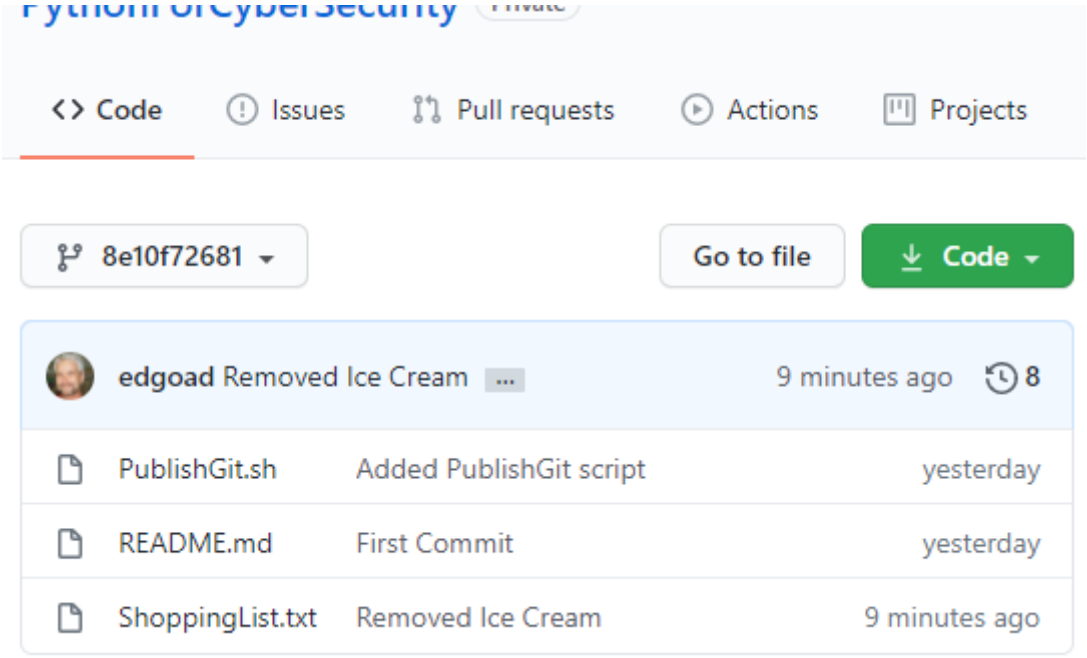
Viewing changes

- The minus signs (-) and red background identify lines that were removed (deletions).
- The plus signs (+) and green background identify lines that were added (additions).
- **NOTE**: My original file had a blank/empty line after Ice Cream, which I deleted when I added Apples.

6. If you return to the prior page (that shows the commit history), to the right of each commit you will see a button with a less-than and greater-than sign on it.

Viewing history

7. If you click this button to the right of **Removed Ice Cream**, it will take you back to the beginning of the repository.



Historical view of repository

8. However, if you look at the comments, you will see that ShoppingList.txt now shows **Removed Ice Cream**.
    - This essentially moved you "back in time" to when this commit occurred.
    - If other files had changed during/after this commit, they would be returned to the same "point in time".
    - This allows you to move an entire project to a specific point without worrying about version differences.

Simple sync script

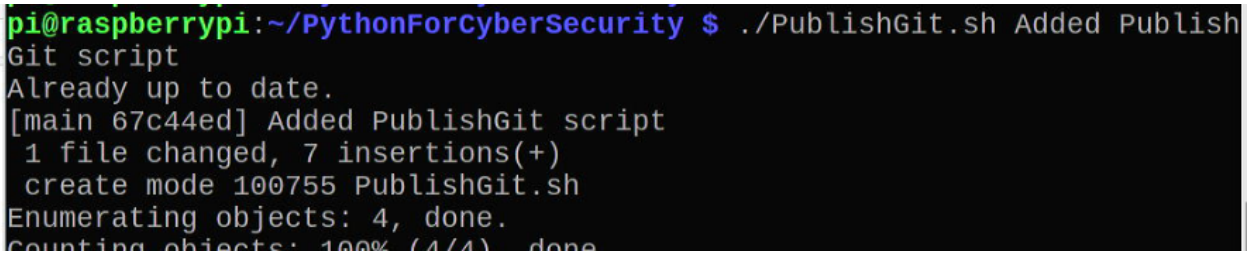
When working with the command line, it can become tiresome to frequently need to add, commit, and push changes. To make this easier, you can create a Linux BASH script

1. On the command line, type `nano PublishGit.sh`
2. Inside the editor, enter the following 5 lines:

```
#!/bin/bash
git pull
git add .
git commit -m "$*"
git push
```

3. Once finished, press **CTRL + X**.
4. When prompted to save, press **Y**.
5. When prompted for the filename, press **Enter** to accept the default.
6. To make the script executable, run the command `chmod +x PublishGit.sh`

To use the script from the command line, call it by typing `./PublishGit.sh` with a comment added to the end.



Output of PublishGit

See Also

https://guides.github.com/activities/hello-world/[101]

https://guides.github.com/introduction/git-handbook/[102]

https://product.hubspot.com/blog/git-and-github-tutorial-for-beginners[103]

[101] https://guides.github.com/activities/hello-world/
[102] https://guides.github.com/introduction/git-handbook/
[103] https://product.hubspot.com/blog/git-and-github-tutorial-for-beginners

Thank You

What Did You Think of Python for Cybersecurity?

First of all, thank you for purchasing this book. I know you could have picked any number of books to read, but you picked this book and for that I am extremely grateful.

I hope that it added at value and quality to your everyday life, has taught you new things, and increased your knowledge. If so, it would be really nice if you could share this book with your friends and family by posting to Facebook, Twitter, and other social media.

If you enjoyed this book and found some benefit in reading this, I'd like to hear from you and hope that you could take some time to post a review on Amazon. Your feedback and support will help this author to greatly improve his writing craft for future projects and make this book even better.

You can follow this link to http://www.amazon.com/dp/B098DDQN1M[104] now.

I want you, the reader, to know that your review is very important and so, if you'd like to leave a review, all you have to do is click here and away you go. I wish you all the best in your future success!

Thank you
Ed Goad

[104] http://www.amazon.com/dp/B098DDQN1M