Python Scripting Basics

In this Module, we will cover the following Learning Units:

- Variables, Slicing, and Type Casting
- Lists and Dictionaries
- Loops, Logic, and User Input
- Files and Functions
- Modules and Web Requests
- Python Network Sockets
- Putting It All Together to Create a Web Spider

Each learner moves at their own pace, but this Module should take approximately 15.5 hours to complete.

Scripting[1] is an efficient way to perform or automate repetitive tasks. We can also use it to complete tasks on a large scale. For example, in an enterprise network, we might need to complete the same tasks on hundreds of hosts. Doing this manually would be a frustratingly tedious waste of time, but with scripting, we can accomplish this quickly and easily.

The basic blocks of scripts are *conditional statements*[2] and *loops*.[3] Generally speaking, we'll use these two items to create a script that processes input until something has been found or until there is no input left.

Scripts are text files processed by an interpreter. If there is an issue in a script, it can be easily fixed by modifying the file. *Python*[4] is a platform-independent language because the interpreter can be installed on both *nix-type and Windows operating systems.

Because we can directly edit script files, we don't need a development environment to compile the source code and create a machine code binary file. Practically speaking, this means that scripts are relatively easy to create and excellent for automating repetitive tasks such as bulk adding users to the system, backing up important files, creating remote backups, or tracking possible malicious activities within log files.

In this Module, we'll review the basics of scripting using the Python language.

---

[1] (Wikipedia, 2023), https://en.wikipedia.org/wiki/Scripting_language ↩

[2] (Wikipedia, 2023), https://en.wikipedia.org/wiki/Conditional_(computer_programming) ↩

[3] (TLDP, 2023), https://tldp.org/LDP/abs/html/loops1.html ↩

[4] (Python, 2023), https://www.python.org/doc/essays/blurb/ ↩

Variables, Slicing, and Type Casting

This Learning Unit covers the following Learning Objectives:

1. Find the Python version
2. Understand and set a shebang line
3. Write our first Python script
4. Understand basic variable types
5. Understand how to use different variable types
6. Slice strings
7. Understand and work with integer variables
8. Understand float variables
9. Understand Boolean variables
10. Understand type casting
11. Set variables to different data types using type casting

This Learning Unit will take approximately 180 minutes to complete.

To begin, we'll cover some basic items with scripting languages. This will start with how we can tell the system to execute our script and print output to the terminal. Let's dive right in and build our knowledge through each section.

Finding our Version of Python

Before working with our Python exercises, let's examine how to determine our installed version. To do this, let's execute **python -V** in the terminal.

```
kali@kali:~$ python -V
Python 3.9.10
```

Listing 1 - The currently installed version of Python is displayed

We can confirm that we currently have *Python 3.9.10* installed and working on our Kali machine. This is important to know since the syntax between versions can affect how our scripts run.

Now that we know our version number, let's write our first Python script.

Exercises

1. What is the option (with dash) to check the version of Python?

Answer

-V

2. Are the syntax requirements the same between Python 2 and Python 3? (yes/no)

Answer

no

Writing our First Python Script

Python is a popular and high-level programming language used by scientists and security professionals alike.

We'll begin our introduction to Python with a few simple examples and then progress to more complex and interesting scripts. We can't cover everything here, but the popularity of Python means there are many support options available if we run into issues.

Like Bash scripting, we can tell the system to interpret the script as a Python script with the *shebang*[1] and python path (#!/usr/bin/python). This way, we can save the file and set the executable flag with **chmod**, and execute it with **./fileName.py**. Additionally, we can run a Python script using the **python** command, which doesn't require the shebang line or the executable flag.

Let's review a simple "Hello World" script.

```
kali@kali:~$ cat pythonsample.py
#!/usr/bin/python
print("Scripting is fun!")
```

Listing 2 - Simple Python script

The script has two lines. The first line tells the OS that the script should be interpreted with Python. When the file is executed, a loader reads the file and the shebang tells the loader which interpreter to use by providing an absolute path to the interpreter.

The second line is a *print()* function that outputs the string "Hello World" to the terminal when we execute it.

Let's make **pythonsample.py** executable with **chmod**.

```
kali@kali:~$ chmod +x pythonsample.py
```

Listing 3 - The script is now executable

Now that the script is executable, let's execute it from the terminal. Because it is set as executable, we can run it without using the **python** command.

```
kali@kali:~$ ./pythonsample.py
Scripting is fun!
```

Listing 4 - The script is executed and the output is displayed to the terminal

The script was able to execute and displayed "Scripting is fun!" to the terminal.

Another variation is to run the script with the **python** command before the script.

```
kali@kali:~$ python pythonsample.py
Scripting is fun!
```

Listing 5 - The script executed with the python command

Running it this way, the shebang line is not needed. Although, it is a good practice to have it in the script file.

Let's practice what we learned in the following exercises:

---

[1] (Wikipedia, 2023), https://en.wikipedia.org/wiki/Shebang_(Unix) ↩

Resources

Some of the exercises require you to start the target machine(s) below.

Please note that the IP addresses assigned to your target machines may not match those referenced in the Module text and video.

PythonExercises

Exercises

1. What is the first line in the script called that specifies which interpreter to use?

Answer

shebang ✓

2. What is the script code to print "Python is fun!" to the terminal output? (Enter the entire line of code in Python 3 syntax).

Answer

print("Python is fun!") ✓

To get started with the following exercise, ssh into the exercise host with "offsec:offsec".

3. Write a Python script called **firstScript.py** that prints "Python is fun!" to the terminal output. Make sure the script is executable, has the appropriate shebang line, and wait a minute after the script completion for the flag to appear. The script file must be located in the **/home/offsec** directory, and the flag will appear in the **/home/offsec/flags** directory.

Answer

PYTHON{F1Rst_Scr1pt_Executed!} ✓

Setting Variables

Before we cover the types of variables we may encounter in Python, let's examine how we can set variables. We will use the *print()* function to display the value of the variables in the terminal.

To set a variable in our Python script, we will enter a variable name followed by an equal (=) sign and the value of the variable we want to set.

```
kali@kali:~$ cat variables.py
#!/usr/bin/python

companyName = "OffSec"

currentYear = 2023

print(companyName)
print(currentYear)
```

Listing 6 - Two variables are set in our script

We have two variables set in our script, followed by the *print()* functions to display them to the terminal. The first variable is called *companyName* and has the value of "OffSec". The next variable is called *currentYear* and has the value of "2022".

```
kali@kali:~$ ./variables.py
OffSec
2023
```

Listing 7 - The variable values are displayed in the terminal

As expected, the *print()* functions printed the values of the variables to the terminal.

Now that we covered setting variables, let's practice what we've learned with the following exercises.

Exercises

1. How could a variable of *color* be set to the value "red"?

Answer

color = "red"

2. How could a variable of *year* be set to the integer of "2022"?

Answer

year=2022

Data Types

Python is quite forgiving when it comes to data types, especially when compared to lower-level programming languages. Python variables can be converted from one data type to another in a process we call *type casting*,[1] which we cover later. Having mentioned this, it is still important to have a basic understanding of the different data types when scripting with Python.

We can set a variable to a value by using the equal sign (=). If we set a variable and use quotes around the value, this can affect how Python treats it. We will go over this in more detail shortly.

The *print()* function can be used to output information to the command-line similar to Bash's *echo* command. The syntax for Python 3 is as follows.

```
myString = "Hello World"
print(myString)

# or

print("Hello World")
```

Listing 8 - The Python print() function

During debugging, we may want to check a variable's type. We can do this with the built in *type()* function. We'll pass the variable into *type()* and output the type of data structure assigned to that variable.

```
kali@kali:~$ cat typeexample.py
#!/usr/bin/python

a = "banana"
print(a)
print(type(a))

b = 1337

print(b)
print(type(b))

kali@kali:~$ python typeexample.py
banana
<class 'str'>
1337
<class 'int'>
```

Listing 9 - Python data types

In this example, we created two variables, *a* and *b*. Our script prints the value of the variable and then its data type. Variable *a* is a *string* with the value of "banana" and variable *b* is an *integer* with the value of "1337".

Let's practice what we learned with the following exercises.

---

[1] (Wikipedia, 2023), https://en.wikipedia.org/wiki/Type_conversion ↩

Exercises

1. What data type would be assigned to the following?

```
a = 4
```

Answer

int ✓

2. What data type would be assigned to the following?

```
a = "56"
```

Answer

string ✓

Strings and Slicing

Many people new to Python may be familiar with *strings*[1] in other programming languages. These data types hold one or more letters, numbers, or symbols, and are typically set by using quotes.

Let's review two string examples:

```
myString = "Hello World"

anotherString = "ABC123!@#"
```

Listing 10 - String examples

In the code block above, we created two variables and assigned them different values.

A string can be converted to a data type called a *list* (we will cover this later). Lists and strings can be manipulated and *sliced*[2] using a few different methods. Slicing in Python is when we cut a string or list into sections. This is done to cut out just the parts of a string that we are interested in.

Let's say we are writing a Python script to scrape a website for any links to other pages. This is a very useful technique for a penetration tester hoping to gain more information about a target. Within the HTML code for the page we are scraping, each HTML anchor tag will appear something like this.

```
<a href="https://www.offsec.com/blog">Blog</a>
```

Listing 11 - An HTML anchor tag

To be able to work with this in our script, we'll want only the URL portion of the tag (https://www.offsec.com/blog), so we'll use string slicing to pull the URL out.

As a side note, because this string contains quotation marks ("), we will run into problems if we use the same syntax as we did earlier. Instead, we'll use single quotes (') around the string. Content between single quotes will not be interpreted.

Once we've created our variable, we can trim the ends of the string. To do this, we need to find the index of where the URL starts and the index of where it ends.

There are ways to find these automatically, but for this example, we'll just count. We need to count each character up to our URL with the first character being 0. The letter "h" in "https" is at index 9 so that's our starting point. If we keep counting to the end of the URL, we find that the letter "g" in "Blog" at the end of the URL is at index 47. Therefore, we want to slice the string from index 9 through index 48 (index 47 + 1), inclusively.

With the index of the start and end of our URL, we can slice it out of the full string and store it into a variable named *url* using the following syntax.

```
kali@kali:~$ cat tagslice.py
#!/usr/bin/python

tag = '<a href="https://www.offsec.com/blog">Blog</a>'
url = tag[9:48]

print(url)

kali@kali:~$ python tagslice.py
https://www.offsec.com/blog
```

Listing 12 - Slicing the HTML anchor tag

We can also slice out the URL from the full HTML anchor tag by using the *index()* function. First, we figure out what is always at the start of the string we want to slice out. In this case, it would be "https". Then, we need what will come after the string we want to slice out. In this case, it's the end double-quote of the URL and a greater-than symbol. Let's set these as individual string variables.

```
tag = '<a href="https://www.offsec.com/blog">Blog</a>'
start = "http"
end = "\">"
```

Listing 13 - Setting the start and end variables

Note that part of our second variable, *end*, includes a quotation mark. This could present a problem, but we're working around it by *escaping*[3] it using a backslash (\) character, which allows us to use the quotation marks that follow without invoking their special meaning.

We can use the *start* and *end* strings to get the index values of where those are located in a complete anchor tag. To do this, we will add *.index()* to the variable *tag*, and inside the index function, we will add the variables. Let's print those values.

```
kali@kali:~$ cat tagslice2.py
#!/usr/bin/python

tag = '<a href="https://www.offsec.com/blog">Blog</a>'
start = "http"
end = "\">"

print(tag.index(start))
print(tag.index(end))

kali@kali:~$ python tagslice2.py
9
48
```

Listing 14 - Running our slicing script

These numbers are similar to the values we got by counting before. Let's remove the *print()* functions and slice *tag* using the index of the *start* and *end* strings.

```
kali@kali:~$ cat tagslice3.py
#!/usr/bin/python

tag = '<a href="https://www.offsec.com/blog">Blog</a>'
start = "http"
end = "\">"
url = tag[tag.index(start):tag.index(end)]

print(url)

kali@kali:~$ python tagslice3.py
https://www.offsec.com/blog
```

Listing 15 - Improving our slicing script

Our script contains a new line of code, which may be a bit confusing at first glance. We've replaced "tag[9:48]" from Listing 12 with the values "tag.index(start)", which came out to 9, and "tag.index(end)", which was 48.

The advantage of a short script like this is that it will work on tags no matter how long or short they are.

---

[1] (Python, 2023), https://docs.python.org/3/tutorial/introduction.html#strings ↩

[2] (Wikipedia, 2023), https://en.wikipedia.org/wiki/Array_slicing ↩

[3] (Wikipedia, 2023), https://en.wikipedia.org/wiki/Escape_character ↩

Resources

Some of the exercises require you to start the target machine(s) below.

Please note that the IP addresses assigned to your target machines may not match those referenced in the Module text and video.

PythonExercises

Exercises

1. In the string "Hello", what is the index value of the letter "H"?

Answer

0

2. In the following code block, we have a variable named *Fact* that contains a string. What would the index slice be to extract "super" from the string?

```
Fact = "When a solution contains more of a solute than can be dissolved, it is known to be supersaturated."
```

Answer

Fact[83:88]

To get started with the following exercise, ssh into the exercise host with "offsec:offsec".

3. Modify **urlSlice.py** and add the values for the *start* and *end* variables, so only the full URL is printed to the terminal. After this is complete, wait up to a minute for the flag to appear in the **/home/offsec/flags/** directory.

Answer

PYTHON{Slicing_and_Dicing_Them_URLs}

Integers

Integer (or *int*)[1] variables are the basic ways to store whole numbers with a comparable value. Int variables are typically set by assigning a whole number without quotes to a variable name.

In the script below, we assign the value of "750" to a variable called *myInt*, then we print it to the terminal.

```
kali@kali:~$ cat intTest.py
#!/usr/bin/python
myInt = 750

print(myInt)

kali@kali:~$ python intTest.py
750
```

Listing 16 - Setting an Integer variable and printing it to the terminal

As expected, when we run the script, the output is "750".

If you use quotes to set a number to a variable, you are setting it as a string instead of an integer. This may lead to bugs or errors if comparisons are done. In the following example, the usage of quotes changes how Python interprets the value of the variable.

```
kali@kali:~$ cat intTest2.py
#!/usr/bin/python

myString = "750"
myInt = 750

print(myString)
print(myInt)
print(myInt + 1)
print(myString + 1)

kali@kali:~$ python intTest2.py
750
750
751
Traceback (most recent call last):
  File "/home/kali/intTest2.py", line 7, in <module>
    print(myString + 1)
TypeError: can only concatenate str (not "int") to str
```

Listing 17 - Working with integers and strings

It's interesting to note that the output of two of the four *print()* functions was the same, but Python was unable to add one to "750" when that value was a string.

It's an excellent idea to get familiar with reading output errors, researching them, and thinking about how we might fix them.

---

1 (Python, 2023), https://docs.python.org/3/library/stdtypes.html#typesnumeric ↩

Exercises

1. Will the following be considered a string or an integer? (string/integer)

```
number = "33"
```

Answer

string ✓

2. Will the following be considered a string or an integer? (string/integer)

```
number = 42
```

Answer

integer ✓

3. Using what we have learned so far, can integers be added to strings? (yes/no)

Answer

no ✓

Floats

If we want a variable to contain a number with a decimal, we can't use an integer. Instead, we will need to use a *Float*.[1] The nice thing is that Python will usually handle this for us, and we can typically treat floats the same as integer variables. For example, let's test what happens when we add a decimal value to an integer.

```
kali@kali:~$ cat floatTest.py
#!/usr/bin/python
a = 100

print(a)
print(type(a))

a = a + .5

print(a)
print(type(a))

kali@kali:~$ python floatTest.py
100
<class 'int'>
100.5
<class 'float'>
```

Listing 18 - Working with Float variables

As we found from the script execution, Python was able to change the integer to a float without us needing to do anything else.

Beyond strings and number variables, we must understand Boolean variables as well.

---

1 (Python, 2023), https://docs.python.org/3/tutorial/floatingpoint.html ↩

Exercises

1. In Python, do we need to do anything to convert an integer to a float when the number value becomes a decimal? (yes/no)

Answer

no

Booleans

Boolean[1] variables store an object value of "True" or "False". These types of variables are useful when using conditional statements but we'll get into that a little later. For now, it's important to understand that these are not string values of "True" or "False". Let's examine a code snippet.

```
# this may be set from a user database or after authentication
adminBool = False

if (adminBool)
    print("You are an admin!")
else
    print("You are NOT an admin!")
```

Listing 19 - Example working with Booleans

This snippet includes a conditional statement, which we'll cover later in this Module. For now, we'll just note that since the variable *adminBool* is *False*, this script would print "You are NOT an admin!".

So far, we covered strings, number variables, and Booleans. Now let's examine a way to change variable types from one to another with a process called *type casting*.

---

[1] (Wikipedia, 2023), https://en.wikipedia.org/wiki/Boolean_data_type#Python,_Ruby,_and_JavaScript ↩

Resources

Some of the exercises require you to start the target machine(s) below.

Please note that the IP addresses assigned to your target machines may not match those referenced in the Module text and video.

PythonExercises

Exercises

1. Is the following a Boolean variable? (yes/no)

```
isAdmin = True
```

Answer

yes ✓

2. Is the following a Boolean variable? (yes/no)

```
isAdmin = false
```

Answer

no ✓

3. Is the following a Boolean variable? (yes/no)

```
isAdmin = "False"
```

Answer

no ✓

To get started with the following exercise, ssh into the exercise host with "offsec:offsec".

4. In the **/home/offsec/booleanTest.py** script file, set the *skyIsBlue* Boolean so that the script prints "The sky is blue" to the terminal. When this is complete, wait up to a minute for the flag to appear in the **/home/offsec/flags/** directory.

Answer

PYTHON{It_is_due_to_Raleigh_scattering} ✓

Type Casting

Casting is a way to convert a variable type in Python. This can be done by using the appropriate casting function to modify the variable type to another. A reason to use this is when reading user input or data from an external source such as a text document or webpage.

For example, let's say we have two strings that contain numbers that we want to add together. This would occur in a scenario where these numbers were part of a longer string that we sliced.

If we try to add these variables together, we won't receive any errors, but the result is also unexpected.

```
kali@kali:~$ cat castTest.py
#!/usr/bin/python

numA = "86"
numB = "20"

print(type(numA))
print(type(numB))

print(numA + numB)

kali@kali:~$ python castTest.py
<class 'str'>
<class 'str'>
8620
```

Listing 20 - Setting up strings to test casting

The output shows that we concatenated the strings together instead of adding the numbers. We will need to cast the strings to integers before they can be added, using the *int()* function. For simplicity, we can do this right in the *print()* function.

Let's change one line in our script and then run it again.

```
kali@kali:~$ cat castTest.py
#!/usr/bin/python

numA = "86"
numB = "20"

print(type(numA))
print(type(numB))

print(int(numA) + int(numB))

kali@kali:~$ python castTest.py
<class 'str'>
<class 'str'>
106
```

Listing 21 - Example casting strings to integers

This can also be done with the *str()* function to convert an integer or float into a string data type. Let's modify the code slightly to demonstrate this.

```
kali@kali:~$ cat castTest.py
#!/usr/bin/python

numA = "86"
numB = "20"

print(type(numA))
print(type(numB))

newValue = int(numA) + int(numB)
print(newValue)
print(type(newValue))
print(type(str(newValue)))

kali@kali:~$ python castTest.py
<class 'str'>
<class 'str'>
106
<class 'int'>
<class 'str'>
```

Listing 22 - Example casting integer to string

As shown in the output, the integer variable was type cast to a string with the *str()* function.

Let's take this opportunity to apply what we've learned in the following exercises.

Resources

Some of the exercises require you to start the target machine(s) below.

Please note that the IP addresses assigned to your target machines may not match those referenced in the Module text and video.

PythonExercises

Exercises

For questions 1 - 3, use the following code snippet:

```
1  #!/usr/bin/python
2
3   firstName = "Dade"
4   LastName = "Murphy"
5   handle = "Zero Cool"
6   systemsCrashed = 1507
7   movieRunHours = 1.783
8   movieYear = "1995"
```

1. In the example Python code above, which line number declares a float variable?

Answer: 7

2. How many string variables are declared?

Answer: 4

3. How could we rewrite line 8 so that Python will interpret it as an integer?

Answer: movieYear = 1995

4. If the variable *movieYear* was kept as a string, how could we later type cast this as an integer?

Answer: int(movieYear)

5. Can strings and integers be printed in the same *print* function without type casting? (yes/no)

Answer: n

6. How could we type cast the *systemsCrashed* variable to a string?

Answer: str(systemsCrashed)

To get started with the following exercise, ssh into the exercise host with "offsec:offsec".

7. Without changing any of the variable declarations at the beginning of the script, adjust the print function within **/home/offsec/typeCasting.py**. After this is complete, wait a minute for the flag to appear in the **/home/offsec/flags/** directory.

Answer: PYTHON{Type_casting_with_Gr1t}

Lists and Dictionaries

This Learning Unit covers the following Learning Objectives:

1. Understand what lists and dictionaries are
2. Understand how lists and dictionaries can be used
3. Create lists
4. Create dictionaries

This Learning Unit will take approximately 90 minutes to complete.

So far, we have covered variables and how to work with them. Now let's examine some more complex variables that hold more than one value in them: lists and dictionaries.

Python Lists

A *list*[1] is a datatype that contains one or more variables in indexed order. The different types of variables can be contained within a list or a list can even contain other lists.

We can specify a list in Python by using square brackets.

```
kali@kali:~$ cat listTest.py
fruitList = ["apple", "banana", "orange"]

print(type(fruitList))

kali@kali:~$ python listTest.py
<class 'list'>
```

Listing 23 - Setting and verifying a list variable

As expected, when we check the data type of the *fruitList* variable, we show that it is a *list*.

Each item in the list has a corresponding index value that represents its location. In our previous example, "apple" has an index of 0, and "banana" would have an index of 1. If we know a value is contained in the list but don't know the index, we can find it by using the list *index()* method.

```
kali@kali:~$ cat listTest2.py
#!/usr/bin/python

fruitList = ["apple", "banana", "orange"]

print(fruitList.index("orange"))

kali@kali:~$ python listTest2.py
2
```

Listing 24 - Finding the index of an item in a list

Above, we searched for the index containing the value "orange". In this case, the "orange" value had an index of 2.

The *index()* method is also very helpful when slicing strings, which we touched on earlier.

If we would like to add an item to our list, we can use the *append()* method.

```
kali@kali:~$ cat listTest3.py
#!/usr/bin/python

fruitList = ["apple", "banana", "orange"]
fruitList.append("mango")

print(fruitList)

kali@kali:~$ python listTest3.py
["apple", "banana", "orange", "mango"]
```

Listing 25 - Adding an item to a list

In the list above, we added the value "mango" to the end of the list.

Inversely, we can remove items from a list, in the same way, using *remove()*.

```
kali@kali:~$ cat listTest4.py
#!/usr/bin/python

fruitList = ["apple", "banana", "orange", "mango"]
fruitList.remove("mango")

print(fruitList)

kali@kali:~$ python listTest4.py
["apple", "banana", "orange"]
```

Listing 26 - Removing an item form a list

Listing 26 shows the value "mango" being removed from the list.

If we would like to know the number of items in our list, we can use the *len()* function, which will return the number of items our list contains.

```
kali@kali:~$ cat listTest5.py
#!/usr/bin/python

fruitList = ["apple", "banana", "orange"]

print(len(fruitList))

kali@kali:~$ python listTest5.py
3
```

Listing 27 - Finding the length of a list

The length of the list in Listing 27 is 3, because there are 3 total items in the list.

---

[1] (Python, 2023), https://docs.python.org/3/tutorial/datastructures.html ↩

Resources

Some of the exercises require you to start the target machine(s) below.

Please note that the IP addresses assigned to your target machines may not match those referenced in the Module text and video.

PythonExercises

Exercises

Use the following line of code to answer Questions 1-4:

```
myList = ["a", "b", "c", "d"]
```

1. Write a line of Python to print the length of the above list.

Answer

print(len(myList))

2. How would we reference the string "c" in this list using its index value?

Answer

myList[2]

3. How would we append the value "e" to the list above?

Answer

myList.append("e")

4. How would we remove the value "b" from the list above?

Answer

myList.remove("b")

To get started with the following exercise, ssh into the exercise host with "offsec:offsec".

5. In the **/home/offsec/createList.py** script file, create a list under the variable name *characters* that contains the following names in order:

```
Guts
Griffith
Casca
```

When this is completed, wait a minute for the flag to appear in the **/home/offsec/flags/** directory.

Answer

PYTHON{What_4_L1sT}

Python Dictionaries

In Python, a *dictionary*[1] is a data structure that contains one or more *key-value*[2] pairs. We can use curly brackets to define a new dictionary and supply it with any initial key-value pairs.

```
theOne = {
    "firstName":"Thomas",
    "lastName":"Anderson",
    "occupation":"Programmer"
    }
```

Listing 28 - Example setting up a dictionary in Python

In Listing 28, we have three key-value pairs within the *theOne* dictionary.

To add an entry to our dictionary, we can simply reference the dictionary with an index of the key we want to add and define it as the value we want to set.

```
kali@kali:~$ cat dictTest.py
#!/usr/bin/python

theOne = {
    "firstName":"Thomas",
    "lastName":"Anderson",
    "occupation":"Programmer"
    }

theOne["company"] = "MetaCortex"

print(theOne)

kali@kali:~$ python dictTest.py
{'firstName': 'Thomas', 'lastName': 'Anderson',  'occupation': 'Programmer', 'company':
'MetaCortex'}
```

Listing 29 - Adding a key-value pair to a dictionary

In the code block above, we added the key-value pair of *company:MetaCortex* to the end of the *theOne* dictionary.

We reference a value in our dictionary by its key.

```
kali@kali:~$ cat dictTest.py
#!/usr/bin/python

theOne = {
    "firstName":"Thomas",
    "lastName":"Anderson",
    "occupation":"Programmer"
    }

theOne["company"] = "MetaCortex"

print(theOne["firstName"])

kali@kali:~$ python dictTest.py
Thomas
```

Listing 30 - Referencing an item in a dictionary by key

In Listing 30, we printed the value "Thomas" by referencing to its key of "firstName".

If we want to change the value of an existing key, we can specify the key name and the new value. The only difference between adding a new key-value pair and modifying one is the fact that the key-value pair already exists within the dictionary.

```
kali@kali:~$ cat dictTest.py
#!/usr/bin/python

theOne = {
        "firstName":"Thomas",
        "lastName":"Anderson",
        "occupation":"Programmer"
        }

theOne["company"] = "MetaCortex"

print(theOne)

theOne["occupation"] = "Superhero"

print(theOne)

kali@kali:~$ ./dictTest.py
{'firstName': 'Thomas', 'lastName': 'Anderson', 'occupation': 'Programmer', 'company':
'MetaCortex'}
{'firstName': 'Thomas', 'lastName': 'Anderson', 'occupation': 'Superhero', 'company':
'MetaCortex'}
```

Listing 31 - The occupation key-value pair was changed

The value of they key labelled "occupation" was changed from "Programmer" to "Superhero".

We can also retrieve a list of keys that are stored in a dictionary by using the *keys()* method.

```
kali@kali:~$ cat dictTest.py
#!/usr/bin/python

theOne = {
    "firstName":"Thomas",
    "lastName":"Anderson",
    "occupation":"Programmer"
    }

theOne["company"] = "MetaCortex"

print(theOne.keys())

kali@kali:~$ python dictTest.py
dict_keys(['firstName', 'lastName', 'occupation', 'company'])
```

Listing 32 - Printing out the keys of a dictionary

As shown in Listing 32, we added a new key-value pair. Then, we used the *keys()* method to output the names of the keys within the *theOne* dictionary.

---

[1] (Python, 2023), https://docs.python.org/3/tutorial/datastructures.html#dictionaries ↩

[2] (PCMag, 2023), https://www.pcmag.com/encyclopedia/term/key-value-pair ↩

Exercises

1. How would we declare a new dictionary named *myDict* with one key-value pair where the key is "one" with a value of the integer of 1? You can either use single or double quotes, but ensure there are no spaces in your answer.

Answer

myDict={"one":1}

2. How would we add a key-value pair to *myDict* where the key is "two" and the value is the integer 2? You can use either single or double quotes, but ensure there are no spaces in your answer.

Answer

myDict["two"]=2

Loops, Logic, and User Input

This Learning Unit covers the following Learning Objectives:

1. Create and iterate with a while loop
2. Create and iterate with a for loop
3. Create if/elif/else statements
4. Use user-generated input

This Learning Unit will take approximately 150 minutes to complete.

In this section, we will take what we learned about variables and use them in loops, conditional statements, and user input. This will enhance our capabilities to create programs that are more functional and user-friendly.

Loops

Looping in programming is one way to iterate on a conditional state or data structure. Python has two types of looping methods: *for*[1] and *while*.[2]

A *while* loop will repeat a code block as long as a conditional statement evaluates to True. In the following example, we set a variable *i* to 0. We then start our *while* loop with the condition that *i* is less than 10. Each run of the code block prints out the current value of *i* and then increments *i* by 1. This is evident with the statement *i += 1*, which is the same as *i = i + 1*. The symbol *+=* is an assignment operator. If we forget to increment our counter variable, this would run until the script is killed manually.

The syntax of loops is important in Python. Note that the loop statement has a colon (:) at the end of the line. This is then followed by indented instructions to complete in the loop under that loop statement. When the indentation isn't there, Python interprets that as being an instruction outside of the loop. Let's examine the following code:

```
i = 0
while i < 10:
    print(i)
    i += 1
```

Listing 33 - Example while loop in Python

Now let's use a while loop with Python lists. Let's consider the following script.

```
kali@kali:~$ cat whilelist.py
#!/usr/bin/python

namelist = ["Sleepy", "Sneezy", "Happy", "Grumpy", "Bashful", "Dopey", "Doc"]

print(namelist)

kali@kali:~$ ./whilelist.py
['Sleepy', 'Sneezy', 'Happy', 'Grumpy', 'Bashful', 'Dopey', 'Doc']
```

Listing 34 - There is a list of names that is output to the terminal

This may not seem like a worthwhile exercise, but linking a looping statement with lists can greatly impact what is achievable with our scripts. As it is, this output only prints out the values of the list, and we don't have control over any of those values.

Although we've learned about indexing in the Python lists section, let's take a different approach on how we could iterate through the list with indexes in a while loop. Let's get the number of values in the *namelist* list, and then iterate through the values to show each name on its own line.

The following code is added to the bottom of the previously shown script. Comments are added using the pound or hash-tag symbol (#) to provide some clarification on what each line is doing.

```
# Get the number of items in the list and store the value in a variable
nameListCount = len(namelist)
# Print a message with how many items are in the list
print("There are " + str(nameListCount) + " names in the name list.")

nameIndex = 0
while nameIndex < nameListCount:
    # Print the index number
    print(nameIndex)
    # Print the name at the current index
    print(namelist[nameIndex])
    # Add 1 to the index value before the loop starts over
    nameIndex = nameIndex + 1
```

Listing 35 - A while loop is created with variables to iterate through the names and show the index values in the list

Now that this is added to the bottom of the script, let's execute it to analyze what it does.

```
kali@kali:~$ ./whilelist.py
['Sleepy', 'Sneezy', 'Happy', 'Grumpy', 'Bashful', 'Dopey', 'Doc']
There are 7 names in the name list.
0
Sleepy
1
Sneezy
2
Happy
3
Grumpy
4
Bashful
5
Dopey
6
Doc
```

Listing 36 - The script shows the list, the number of names message, and iterates through the list to show the index number and value

In the output above, the list and a message with the number of names is displayed to the terminal. The list is then iterated through to show the respective index and name.

Of course, in deployment, we would want to remove the indexes, the list, and possibly even the message stating how many names are in the list. We did this to show how the while loop can be used to separate out values in the list and we'll take this concept further later in this Module. Now that we've covered a *while* loop, let's move on to a *for* loop.

A *for* loop will repeat a code block as many times as specified. Each iteration will store the current value of the sequence to a temporary variable and execute the code block accordingly. In the following example, we are using the *range*[3] command to create a list containing numbers 0 through 9. The first iteration of this loop will set the temporary variable *i* to 0 and then run the code block. Then it will set *i* to 1 and run the code block again. This will repeat until the range is depleted. Notice there is no need to increment the counter. This is a characteristic of a *for* loop. As shown above, the *while* loop will require some incremental process (nameIndex = nameIndex + 1) but with the *for* loop, this is done for us.

```
kali@kali:~$ cat forloop.py
#!/usr/bin/python

for i in range(10):
    print(i)

kali@kali:~$ ./forloop.py
0
1
2
3
4
5
6
7
8
9
```

Listing 37 - Example for loop in Python

Note that there are 10 iterations in the loop. Keep in mind that the index starts at 0, and the *range* function also starts at index 0 by default. We can modify the way this works by also modifying the range start, stop, and step values. The syntax for this is *range(start, stop, step)*. The *start* parameter is used to specify what position we want to start the loop count. The *stop* parameter is used to specify the ending position of the iterations. The *step* parameter is used to designate how many will be added in each iteration. The default for this is 1. Let's change the program to start at 10, and at 20, and use the step count of 2.

```
kali@kali:~$ cat forloop.py
#!/usr/bin/python

for i in range(10,20,2):
    print(i)

kali@kali:~$ ./forloop.py
10
12
14
16
18
```

Listing 38 - The range started at 10 and counted up by 2 to 18

As shown in Listing 38, the values were counted by 2's. This was the impact of the step parameter. The range started at 10 but didn't show the value of 20. This is the same concept as shown in the original script example, where the ending position is not reached.

We can also do more with loops. Let's continue working with the *for* loop to demonstrate how we can reference dictionary items in a loop.

```
kali@kali:~$ cat forDictionary.py
#!/usr/bin/python

guts = {
        "Name":"Guts",
        "Personality":"gruff",
        "Weapon":"Dragon Slayer",
        "Armor":"Berserker Armor"
        }

print(guts)

kali@kali:~$ ./forDictionary.py
{'Name': 'Guts', 'Personality': 'gruff', 'Weapon': 'Dragon Slayer', 'Armor': 'Berserker Armor'}
```

Listing 39 - A dictionary is made and displayed

Earlier, we covered showing each of the keys with the *keys()* function. Let's iterate with a *for* loop to list each of the key-value pairs on separate lines (instead of displaying the entire dictionary as shown in the listing above). To do this, we'll add the following code at the bottom of the **forDictionary.py** script.

```
for key in guts.keys():
    print(key + ": " + guts[key])
```

Listing 40 - A loop to iterate through the keys and display the key-value pairs

With the above code added, let's execute the script.

```
kali@kali:~$ ./forDictionary.py
{'Name': 'Guts', 'Personality': 'gruff', 'Weapon': 'Dragon Slayer', 'Armor': 'Berserker Armor'}
Name: Guts
Personality: gruff
Weapon: Dragon Slayer
Armor: Berserker Armor
```

Listing 41 - Each key-value pair is displayed in the terminal

As expected, we iterated through each key-value pair and printed the key, followed by a colon (:) and space, and finally the associated value of the key. This occurred for each pair on a new line until the end of the dictionary.

This completes our coverage of *while* and *for* loops. Let's take a moment to practice what we've learned so far.

---

[1] (Python, 2023), https://docs.python.org/3/tutorial/controlflow.html#for-statements ↩

[2] (Python, 2023), https://docs.python.org/3/reference/compound_stmts.html#while ↩

[3] (W3Schools, 2023), https://www.w3schools.com/python/ref_func_range.asp ↩

Resources

Some of the exercises require you to start the target machine(s) below.

Please note that the IP addresses assigned to your target machines may not match those referenced in the Module text and video.

PythonExercises

Exercises

1. How many times will the loop below run?

```
x = 10

while x > 1:
    print(x)
    x -= 1
```

Answer

9

2. How many times will the below loop run?

```
dogs = ["poodles", "greyhounds", "pitbulls", "huskies"]

for d in dogs:
    print(d + "are great dogs")
```

Answer

4

3. What is the index number of "huskies"?

Answer

3

To get started with the following exercise, ssh into the exercise host with "offsec:offsec".

4. Print each dictionary key and key-value as shown in the demonstration above. The file is **/home/offsec/forDictionary.py**. After completion, wait one minute for the flag to appear in the **/home/offsec/flags/** directory. The format for each line should be "key: value". Note the colon (:) and space between key and value.

Answer

PYTHON{Guts_is_packing}

Conditional Statements

When scripting, there may be sections of code that we want to run in specific situations. To make this easier, we can use *conditional statements*[1] such as *if*, *elif*, and *else* logic.

In Python's *if* statements, the use of newlines and tabs changes how the logic is interpreted. If an *if* statement evaluates to True, then the code it will run is indented under the conditional statement. Just like looping statements, a colon and newline are required after the conditional statement.

```
if numApples > 100:
    print("That's a lot of apples!")
```

Listing 42 - Example If statement in Python

As long as the value of *numApples* is greater than 100, the program will execute the *print* function located inside the *if* statement.

When the *if* statement evaluates to False, the indented code block is skipped. If we have a related conditional statement, we can use the *elif* (short for "else if") statement. Many *elif* statements can be added as long as an initial *if* statement exists.

```
if numApples > 100:
    print("That's a lot of apples!")
elif numApples > 50:
    print("That's a very good amount of apples")
elif numApples > 30:
    print("That's a good amount of apples")
```

Listing 43 - Example if - elif statement in Python

In Listing 43, if the value of *numApples* is greater than 100, it will execute the *print* function inside the *if* statement. Then, it will skip the rest of the *elif* statements to continue with the program. Otherwise, it will continue to compare to the next *elif* statement.

If we would like to add a handler to run if all *if* and *elif* statements evaluate to false, we can use the *else* statement. If all previous *if* and *elif* statements resolve to false, the code under the *else* statement will be run. Notice in the example below that we don't have to specify in what case the code under *else* is run as it is a catch-all. The code under the *else* will only run if all other conditional statements in this conditional block evaluate to false.

```
if numApples > 100:
    print("That's a lot of apples!")
elif numApples > 50:
    print("That's a very good amount of apples")
elif numApples > 30:
    print("That's a moderate amount of apples")
else:
    print("Running low on apples!")
```

Listing 44 - Example if/elif/else statement in Python

Let's set the variable *numApples* to various numbers and review the effects when running the program. The variable must be set before the conditional statements.

```
numApples = 150
```

Listing 45 - We put that we have 150 apples as the variable value

Now let's execute the script.

```
kali@kali:~$ ./appleStock.py
That's a lot of apples!
```

Listing 46 - The output correlates with the condition that there are more than 100 apples

Let's change the value one more time. This time, we'll make it 15, which is less than the last checked condition of 30 apples.

```
numApples = 15
```

Listing 47 - The number of apples is now set to 15

Let's execute our script again to check if the output has changed.

```
kali@kali:~$ ./appleStock.py
Running low on apples!
```

Listing 48 - The output correlates with the else statement in the code

Now that we have experienced some conditional statements, let's practice what we've learned with some exercises.

---

[1] (Python, 2023), https://docs.python.org/3/tutorial/controlflow.html ↩

Resources

Some of the exercises require you to start the target machine(s) below.

Please note that the IP addresses assigned to your target machines may not match those referenced in the Module text and video.

PythonExercises

Exercises

Use the following lines of code to answer Questions 1-2.

```
1   if myApples < yourApples:
2       print("You have more apples")
3   elif yourApples > myApples:
4       print("I have more apples")
5   else:
6       print("We have the same amount of apples")
```

1. The above Python snippet doesn't work as expected. Which line is logically incorrect? (Enter the line number for the answer)

Answer

3

2. What should the line be changed to in order to make more logical sense? (Keep variable names in the same order)

Answer

elif yourApples < myApples:

User Input

While setting our variables inside our script worked for what we've covered so far, it doesn't make our program very interactive for the user. Prompting the user for input can greatly enhance the flexibility of the program and allow for different variations to be entered as the variable under test. Let's consider the following code.

```
kali@kali:~$ cat nameAge.py
#!/usr/bin/python

name = "Griffith"
age = 24

print("Hi " + name + "!")

if age >= 100:
    print("You are over 100 years old? What's your secret?")
elif age >= 70:
    print("You are over 70 years old? Are you retired or still working?")
elif age >= 60:
    print("You are over 60 years old? Will you be retiring soon?")
elif age >= 40:
    print("You are over 40 years old? What do you do for a living?")
elif age >= 20:
    print("You are over 20 years old? What do you want to do for your career?")
elif age >= 18:
    print("You are over 18 years old? That makes you a legal adult!")
else:
    print("It looks like you are under 18 years old.")
```

Listing 49 - There are two variables: name and age, that can be changed inside the script

In Listing 49, we create two variables, execute a *print()* function, and run some conditional statements based on the *age* variable.

Let's execute the script, as is, to examine the expected behavior.

```
kali@kali:~$ ./nameAge.py
Hi Griffith!
You are over 20 years old? What do you want to do for your career?
```

Listing 50 - The script is executed and the output for someone between 20 and 40 is displayed

The script is functional as written, but this doesn't allow anyone to execute the script and add their own name and age. Instead, for every change of the name and age, the script will have to be modified and saved before the next execution. This can be a cumbersome task, so let's make this script more user-friendly.

To keep things brief, we'll only focus on the variable lines. Let's use the *input()*[1] function to prompt the user to input the values to store as the variables. The syntax for the *input()* function is *input("prompt")*.

```
name = input("Please enter your name: ")
age = input("Please enter your age: ")
```

Listing 51 - Now the user should be able to enter their own values

Let's execute the script with the above changes to the variable lines.

```
kali@kali:~$ ./nameAge.py
Please enter your name: Griffith
Please enter your age: 24
Hi Griffith!
Traceback (most recent call last):
  File "/home/kali/./nameAge.py", line 9, in <module>
    if age >= 100:
TypeError: '>=' not supported between instances of 'str' and 'int'
```

Listing 52 - The modification to our script failed with a TypeError message

The script failed to execute. In the listing above, the error message indicates that something is wrong with a variable that is trying to mix a *'str'* and an *'int'* in the *age* comparison line. This is where we can use type casting, as covered earlier, to set the age variable input to be an integer.

```
name = input("Please enter your name: ")
age = int(input("Please enter your age: "))
```

Listing 53 - The type cast has been put around the input() function to force the variable into being an integer

Now that *age* is type cast as an integer, let's execute the script again.

```
kali@kali:~$ ./nameAge.py
Please enter your name: Griffith
Please enter your age: 24
Hi Griffith!
You are over 20 years old? What do you want to do for your career?
```

Listing 54 - The script works as expected

One note on user input, we need to be careful with what the user may do when using our program. Even though we set the *age* variable to be an integer through type casting, the user can still enter unexpected values. Let's try to simulate a user entering a string when prompted for their age.

```
kali@kali:~$ ./nameAge.py
Please enter your name: Griffith
Please enter your age: Femto
Traceback (most recent call last):
  File "/home/kali/./nameAge.py", line 4, in <module>
    age = int(input("Please enter your age: "))
ValueError: invalid literal for int() with base 10: 'Femto'
```

Listing 55 - The script fails with an error message for invalid input

Fixing this issue is out of scope for this Module, but we must keep this in mind while we write any code. Input validation is a very big reason a lot of security vulnerabilities exist. Keeping in mind that a user may enter special characters, characters that don't match the data type, or even input characters that may be thousands of characters long are all important to securely writing code.

---

[1] (GeeksforGeeks, 2023), https://www.geeksforgeeks.org/python-3-input-function/ ↩

Resources

Some of the exercises require you to start the target machine(s) below.

Please note that the IP addresses assigned to your target machines may not match those referenced in the Module text and video.

PythonExercises

Exercises

To get started with the following exercise, ssh into the exercise host with "offsec:offsec".

1. Modify **/home/offsec/nameAge.py** to accept user input with the *input()* function, similar to how we did in the lesson. Make the script executable when finished with the modifications to the code. When complete, wait a minute for the flag to appear in the **/home/offsec/flags/** directory.

Answer

PYTHON{Prompting_the_user}

Files and Functions

This Learning Unit covers the following Learning Objectives:

1. Open files
2. Read files
3. Write to files
4. Close files
5. Create functions
6. Understand function parameters
7. Return function values

This Learning Unit will take approximately 120 minutes to complete.

Working with files can be incredibly useful when writing Python scripts. This can be used to get data from a file and act upon that information in the code. It can also be used to change or even create files on a system. If we want to have a log or resulting output with our script, we need to understand how to work with files.

Functions can also help us manage our code and separate each goal within the program into smaller segments.

Working with Files

Reading and writing to files is an important function for ingesting or saving data for after the script ends. To open a file, we can use the *open*[1] command, set to a variable. We need to specify the file name and the mode. The mode can be read (*r*), write (*w*), append (*a*), or read+write (*r+*) for text. If we want to read or write binary data, we can append a *b* to the mode. Reading a binary file would require a read-binary (*rb*) mode, and writing binary to a file would require write-binary (*wb*). For our examples, we will work with text.

```
f = open("data.txt", "r")
```

Listing 56 - Example of opening a file in read mode

With the file opened and defined as variable *f*, we can now read the contents with the *read()*[2] method.

```
data = f.read()
```

Listing 57 - Example of reading a file to a variable

This will store the entire contents of the opened file as a string into a variable named *data*. This may not be the best option if we are working with large files such as log files. For larger files, we can limit how much we are storing by only reading one line of the file at a time using *readlines()* instead of *read()*. Using *readlines()* as a sequence in a *for* loop makes this very easy. Each iteration of the *for* loop will store the current line of the file we are reading as a temporary variable that we can work with.

```
f = open("data.txt", "r")

for line in f:
    print(line)
```

Listing 58 - Example of looping over lines of an opened file

If we would like to write some data to a file, we can open the file like before but in write (*w*) or append (*a*) mode depending on what we are trying to accomplish. Opening a file in write mode will overwrite the file if it already exists. Using append mode, we maintain the existing contents of the file and will write any new data to the end. Either way, we write data to the file using *write()*[3] with the data to write passed as an argument.

```
myData = "I'm sample data to be written to a file"

f = open("data.txt", "a")

f.write(myData)
```

Listing 59 - Example of opening a file in write mode and writing data to it

In Listing 59, we append by writing the value of *myData* to **data.txt**.

After reading or writing data to a file, we will need to close it. This can be done by using *close()*. Closing an opened file isn't necessary, but it is good practice. Depending on the situation, it can have many benefits, like allowing other programs the ability to access the file. There is much more theory and technical explanation behind this, but it is out of scope. For the purpose of this Module, remember that we *must* close an opened file because of best practices.

```
f.close()
```

Listing 60 - Closing a file

---

[1] (Python, 2023), https://docs.python.org/3/library/functions.html#open ↩

[2] (W3Schools, 2023), https://www.w3schools.com/python/ref_file_read.asp ↩

[3] (W3Schools, 2023), https://www.w3schools.com/python/python_file_write.asp ↩

Exercises

1. How would we open an existing file named **log.txt** referenced by the variable name *out* with the intention to add to the end of the file?

Answer

```
out = open("log.txt", "a")
```

2. How would we write the contents of a string variable named *logOutput* to the file opened in Question 1?

Answer

```
out.write(logOutput)
```

Python Functions

A *function*[1] is a code block that can be referenced later in either our script or another external script or program. Functions need to be defined in the script before they can be called. To define a function, we use *def* followed by the name of our function. The function definition line ends with parenthesis and a colon.

The big value of functions is that they can help organize code into small snippets. This makes the code much easier to manage, call, and even modify in the future. Instead of writing an entire program, each function goal can be completed and assembled to make the complete program. We will explore this more in the last section of this Module.

The idea behind this usage is to keep the lines of actual code per function under 30 lines. Not only is 30 lines easier to deal with than 500, but it can also help identify the area of code that may have an issue in the debugging process. Of course, 30 lines is an arbitrary goal to keep the functions short. If possible, it would be a better practice to lower this line count within the function as much as possible.

```
kali@kali:~$ cat function.py
#!/usr/bin/python

def hello():
    print("Hi there!")
```

Listing 61 - This function will print "Hi there!" to the terminal

We created a function called *hello()* that prints text to the terminal.

With the function written, let's execute the script.

```
kali@kali:~$ ./function.py
```

Listing 62 - Nothing is shown in the terminal

Nothing is shown in the terminal, despite the function having the *print()* function. The reason for this is we didn't call the function in the script. To call and execute the function, we need to specify the function name after the function. If we attempt to call the function before it is defined, the program will result in an error. This will happen because as far the program knows, the function does not (yet) exist.

Let's add the function call now.

```
kali@kali:~$ cat function.py
#!/usr/bin/python

def hello():
    print("Hi there!")

hello()
```

Listing 63 - The hello() function is called in the script

Now that we have a function call in the script, let's execute it and analyze the output.

```
kali@kali:~$ ./function.py
Hi there!
```

Listing 64 - The function was called and printed "Hi there!" to the terminal

Perfect! When we ran our script, the program knew a function called *hello()* existed because we defined it. Then, when we called it, the program executed the instructions inside the function. This resulted in the text printing to the terminal.

This was a very simple function that may not be useful for our needs. Despite this example, we'll review how these simple functions - without arguments - become incredibly useful later in this Module.

To expand on function use, we can supply arguments to be used in the function. Arguments are also known as parameters, operands, and variables. These are interchangeable in Python. They are passed to a function within the parentheses. A *return* statement is used to supply the function output back to our script in progress. Let's create a demonstration function to add two numbers and *return* the value.

```
def addNums(numA, numB):
    answer = numA + numB
    return answer
```

Listing 65 - Example Function in Python

In Listing 65, we created a function called *addNums*, which takes two arguments: *numA* and *numB*. Inside the function, both of these variables are added and the result is assigned to the *answer* variable. Finally, the function returns the *answer* variable.

We can now call *addNums()* later in our script using the function name and any passed arguments in parentheses.

```
kali@kali:~$ cat functTest.py
#!/usr/bin/python

def addNums(numA, numB):
    answer = numA + numB
    return answer

x = addNums(5, 7)

print(x)

kali@kali:~$ python functTest.py
12
```

Listing 66 - Calling a Function

In Listing 66, we took the function that we previously created and we added a *print()* function to print it to the terminal. It's important to note that the *return* statement does not print the result to the terminal. We can then take the return value and do something with it, like manipulate it further or print it to the terminal. As expected, 12 was printed because 5 + 7 = 12.

---

[1] (Python, 2023), https://docs.python.org/3/tutorial/controlflow.html#defining-functions ↩

Resources

Some of the exercises require you to start the target machine(s) below.

Please note that the IP addresses assigned to your target machines may not match those referenced in the Module text and video.

PythonExercises

Exercises

Use the following lines of code to answer Questions 1-2:

```
def myFunction( a b c )
    d = a + b
    answer = d * c
    return answer

print(myFunction(8, 5, 3))
```

1. Two symbol types are missing from the function declaration. Without any spaces in the answer, what are the two symbols required to fix the first line?

Answer

Answer | Verify

2. Once the function is fixed, what would be printed after running the above code?

Answer

Answer | Verify

Combining File Operations in a Function

We covered how to work with files and working with functions. Let's mix these two subjects together and accomplish the file operations to quickly store the file contents in a variable to work within our script. Let's consider the following script.

```
kali@kali:~$ cat fileManipulation.py
#!/usr/bin/python

def storeFile(file):
    f = open(file, 'r')
    contents = f.read()
    f.close()
    return contents

# Variable to store the filename
fileVar = "notes.txt"

contents = storeFile(fileVar)
print(contents)
```

Listing 67 - The function opens, reads, stores the contents into a variable, and closes the specified file

This script opens, reads, stores the contents into a variable, and closes the file within one function call. With this, we can call the function and pass the parameter with the *fileVar* variable, which was set above the function call. For this to work, the file must already exist. In this example, the file exists with the text shown in the script execution.

```
kali@kali:~$ ./fileManipulation.py
These are my amazing notes
```

Listing 68 - The file operations completed and the contents are displayed to the terminal

With this function, we can modify the value stored in the variable *f*, instead of modifying the file in any way. This may help prevent mistakes that may happen when working directly with the file. The convenience of the function is that the file is also closed as soon as it is no longer needed.

Let's practice what we learned with some exercises.

Exercises

1. If we wanted to open a file for writing, what would we put in place of the empty space in between the single quotes in the following line:

```
f = open(file, ' ')
```

Answer — View hints

Answer — Verify

2. If we wanted to create a file that doesn't exist and get an error if it already exists, what would we put in place of the empty space in between the single quotes in the following line:

```
f = open(file, ' ')
```

Answer — View hints

Answer — Verify

3. If we wanted to append to a file, what would we put in place of the empty space in between the single quotes in the following line:

```
f = open(file, ' ')
```

Answer — View hints

Answer — Verify

4. Consider the following code snippet:

```
def storeFile(file):
    f = open(file, 'r')
    f.close()
    return contents

# Variable to store the filename
FILE = "notes.txt"

f = storeFile(FILE)
print(f)
```

What will happen when the program is executed? (Enter the letter only, without the dot or text, corresponding with the answer)

```
a. The contents of the "notes.txt" file will be displayed in the terminal
b. An error occurs in the script execution
c. The filename changes to "storeFile.txt"
d. The letter "f" is displayed on the terminal
```

Answer

Answer — Verify

Modules and Web Requests

This Learning Unit covers the following Learning Objectives:

1. Define what a module is
2. Create a custom Python module
3. Import a module
4. Use an imported module within a script
5. Make a web request to pull a web page with Python

This Learning Unit will take approximately 120 minutes to complete.

We covered multiple subjects for creating our own Python programs, but the power behind Python's ease-of-use is its modules that are already built. To understand modules, we'll create our own and import them for use in a custom script. From there, we'll examine how to use pre-existing modules to make web requests to pull web page content.

Importing a Module

One of the best things about Python is the size of the community. There are lots of resources available to get help or find better ways to accomplish complex tasks. Sometimes, we may find that someone has already solved a complicated task for us and provided their code as a Python module. Examples of these are the *JSON*,[1] *Requests*,[2] and *NumPy*[3] modules.

We may also run into a situation where we are working with large and complex Python files. It may be better for us to split them up and import the functionality when needed by using modules. This can help us keep our code organized and clean. We can also re-use code more easily in other projects this way.

Let's start with importing our own code. First, let's create a new file named **myData.py** and initialize a couple of lists with some sample values and a function that prints out items in a list passed to it.

```
#!/usr/bin/python

fruit = ["apple", "banana", "orange", "mango"]

veg = ["carrot", "broccoli", "peas", "artichoke"]

def printItems(myList):
    for x in myList:
        print(x)
```

Listing 69 - Setting up a Python file to import

Now, we'll set up a new Python file in the same directory called **myMain.py**. We want to be able to run **myMain.py** and have it import the lists and function from **myData.py**. To do this, we use the *import*[4] statement. This is usually done at the top of the file below the shebang. There are a couple of different ways to import a module. We can import just the parts we want, or we can import the entire module. To import the entire module, we can use the *import* statement followed by the file we want to import (without the file extension).

```
#!/usr/bin/python

import myData
```

Listing 70 - Importing our local Python script

With our module imported, we can reference the lists and functions included in it by calling the module name and the variable name separated by a period. This import will first search for local modules of this name (in the same directory) and then search for modules of the same name in the *PYTHONPATH*, which is dependent on our OS and how Python was installed.

```
#!/usr/bin/python

import myData

print(myData.fruit)

print(myData.veg)

myData.printItems(myData.fruit)
```

Listing 71 - Working with imported data and functions from a basic import

This is very useful but typing "myData" every time we reference something from the module can be inefficient. Instead, we can just import what we want and remove the need to reference the module each time by using the *from* statement along with our import.

```
kali@kali:~$ cat myMain.py
#!/usr/bin/python

from myData import fruit, printItems

print(fruit)

printItems(fruit)

print(veg)

kali@kali:~$ python myMain.py
['apple', 'banana', 'orange', 'mango']
apple
banana
orange
mango
Traceback (most recent call last):
 File "/home/kali/myMain.py", line 9, in <module>
   print(veg)
NameError: name 'veg' is not defined
```

Listing 72 - Working with imported data and functions using From

Here, we are choosing which parts to import from our myData module. We imported the *fruit* list and *printItems()* function directly so we will be able to work with them in **myMain.py** without having to reference the module they came from.

We didn't import the *veg* list so Python produced an error when we tried to use it. Using this method, we can also import everything from myData by replacing the import statement with "from myData import *".

---

[1] (Python, 2023), https://docs.python.org/3/library/json.html ↩

[2] (Python-Requests, 2023), https://docs.python-requests.org/en/latest/ ↩

[3] (NumPy, 2023), https://numpy.org/ ↩

[4] (Python, 2023), https://docs.python.org/3/reference/import.html ↩

Resources

Some of the exercises require you to start the target machine(s) below.

Please note that the IP addresses assigned to your target machines may not match those referenced in the Module text and video.

PythonExercises

Exercises

1. When importing another program, do we keep the ".py" extension in the import statement? (yes/no)

Answer

Answer | Verify

2. Consider the following code snippets:

File: **myData.py**

```
#!/usr/bin/python

fruit = ["apple", "banana", "orange", "mango"]

veg = ["carrot", "broccoli", "peas", "artichoke"]

def printItems(myList):
    for x in myList:
        print(x)
```

File: **myMain.py**

```
#!/usr/bin/python

from myData import veg, printItems

print(fruit)

printItems(fruit)

print(veg)
```

Will the **myMain.py** script execute as written without errors? (yes/no)

Answer

Answer | Verify

3. Consider the following code snippet:

```
kali@kali:~$ cat Calculations.py
#!/usr/bin/python

def square(x):
    return x*x

def sqrt(x):
    sqrt = x / 2
    temp = 0
    while(sqrt != temp):
        temp = sqrt
        sqrt = (x/temp + temp) / 2
    return sqrt

def pow(x):
    power = x
    for i in range(1,y):
        power = power * x
    return power
```

If we want to only import the *pow()* function from **Calculations.py**, what would our import line be?

Answer | View hints

Answer | Verify

To get started with the following exercise, ssh into the exercise host with "offsec:offsec".

4. There are two script files in the **/home/offsec/import/** directory. Import the **importMe.py** script into **importTest.py**. When this is complete, figure out how to execute the function inside **importMe.py** to get the flag. Use **sudo** when executing the **importTest.py** script. (The function is already provided in **importTest.py**)

Answer | View hints

Answer | Verify

Web Requests

There are a lot of modules that we can leverage in Python. We will focus on the *requests*[1] module for making web requests.

```
kali@kali:~$ cat webRequest.py
#!/usr/bin/python

import requests
```

Listing 73 - The module will be imported in our webRequest script

The *requests* module contains multiple functions within it. Some common functions are: *get*, *status_code*, *headers*, *encoding*, *text*, and *json*. Let's work with *get*, *status_code*, and *text* to keep this section easier for the sake of learning.

Let's modify our script to request the webpage at "https://www.offsec.com/offsec/game-hacking-intro/", store that in a variable, and show the status of that webpage.

```
kali@kali:~$ cat webRequest.py
#!/usr/bin/python

import requests

page = requests.get('https://www.offsec.com/offsec/game-hacking-intro/')
print(page.status_code)
```

Listing 74 - The web page is stored in the page variable and the status will be printed to the terminal on execution

The web page contents will be stored in the *page* variable. Using that variable, we can check the status of the web response with the *status_code* function. Let's execute the script and identify the status of the webpage.

```
kali@kali:~$ ./webRequest.py
200
```

Listing 75 - The HTTP response is 200

The *HTTP response*[2] code is *200*, which means the page was successfully reached. Knowing the HTTP response code can be useful when making requests to web resources. If the resource is blocked or unreachable, that request could have an error message, be ignored, or even halt the program execution.

Even though the HTTP response is useful, this isn't what we truly wanted from the request. Let's add another function under the *status_code* call, *text*.

```
kali@kali:~$ cat webRequest.py
#!/usr/bin/python

import requests

r = requests.get('https://www.offsec.com/offsec/game-hacking-intro/')
print(r.status_code)
print(r.text)
```

Listing 76 - The text function call in the module is added to the script

Now that we have the *text* function call in our script, let's execute it.

```
kali@kali:~$ ./webRequest.py
200

<!doctype html>

  <html class="no-js"  lang="en-US">

        <head>
                <meta charset="utf-8">

                <!-- Force IE to use the latest rendering engine available -->
                <meta http-equiv="X-UA-Compatible" content="IE=edge">

                <!-- Mobile Meta -->
                <meta name="viewport" content="width=device-width, initial-scale=1.0">
                <meta class="foundation-mq">
...
```

Listing 77 - The source code of the web page is displayed on the terminal

The source code of the webpage is displayed on the terminal. The output in the listing above is trimmed to save space, but the contents of the webpage could be manipulated in our script.

Let's practice what we learned with the following exercises:

---

[1] (Python-Requests, 2023), https://docs.python-requests.org/en/latest/ ↩

[2] (Mozilla, 2023), https://developer.mozilla.org/en-US/docs/Web/HTTP/Status ↩

Resources

Some of the exercises require you to start the target machine(s) below.

Please note that the IP addresses assigned to your target machines may not match those referenced in the Module text and video.

PythonExercises

Exercises

1. We stored a webpage in the variable *page*. What is the correct syntax to refer to the web HTTP response code value of this variable?

Answer

Answer — Verify

2. What is the function in the requests module to display the webpage contents? (supply only the function)

Answer

Answer — Verify

To get started with the following exercise, ssh into the exercise host with "offsec:offsec".

3. Write a script inside the **/home/offsec/webDownloader.py** file that pulls the root webpage from the localhost. Print the contents of the webpage to the terminal in the script. Once this is completed, wait a minute for the flag to appear in the **/home/offsec/flags/** directory.

Answer — View hints

Answer — Verify

Python Network Sockets

This Learning Unit covers the following Learning Objectives:

1. Write a Python network client
2. Connect to a server and read the content received
3. Send data to a server with the Python network client

This Learning Unit will take approximately 90 minutes to complete.

In this section, we'll cover how to write a Python socket script. *Network sockets*[1] are endpoints for sending and receiving data across the network. Simply put, they are the backbone of server/client relationships. We'll only be covering client socket programming with Python to connect to a remote server.

---

[1] (Wikipedia, 2023), https://en.wikipedia.org/wiki/Network_socket ↩

Creating the Python Socket Client

To get started with our Python network client script, we'll first need to import the *socket* module.

```
kali@kali:~$ cat networkClient.py
#!/usr/bin/python

import socket
```

Listing 78 - The socket module is imported in our script

From here, we need to set a socket variable. In our case, we'll name ours *s*.

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

Listing 79 - The socket variable is set to 's'

The variable may seem complicated, so let's break down what each part of the socket declaration is. There is a *socket()* function that has two parameters: *AF_INET* and *SOCK_STREAM*.[1] The *AF_INET* parameter specifies that the IP address will be an IPv4 address. The *SOCK_STREAM* parameter specifies that the socket will use a TCP connection. Now that the socket variable is set, we can connect to a remote server. Let's add the connection code now. For this demonstration, we'll use the IP address 192.168.50.101 and port 9999.

```
s.connect(("192.168.50.101", 9999))
```

Listing 80 - The socket is connected

It is important to note that the IP and port are provided as a single parameter in the *connect()* function. Now that we added the connection code, let's check if the server sends anything to the client. We can do this with the *recv()* function.

```
print(s.recv(1024))
```

Listing 81 - The data sent from the server will be displayed on the terminal

The receive value of 1024 in the above listing is the buffer size for the data receipt. This value sets the number of bytes that can be received from the server. It can be changed to be lower or higher, up to near 64,000 bytes. Raising our buffer to that size would be impractical, and 1024 bytes is a fair amount to specify for typical usage.

After we receive the data from the server, we will close our socket connection with the *close()* function. Let's add this and review our script.

```
kali@kali:~$ cat networkClient.py
#!/usr/bin/python

import socket

s = socket.socket(socket.AF_INET,socket.SOCK_STREAM)

s.connect(("192.168.50.101", 9999))

print(s.recv(1024))
s.close()
```

Listing 82 - The Python network socket client is complete

Let's execute our script against the remote server to analyze the result.

```
kali@kali:~$ ./networkClient.py
b'You are connected.\nGoodbye'
```

Listing 83 - The client connected to the server, read the incoming data, and closed the connection

Interestingly enough, there is a *b* before the string of data. The string also has a *newline character*[2] and didn't interpret that as a new line. The *b* is signifying that the data is in a binary format. On the server, the data being sent is encoded. We can decode this data in our client with the *decode()* function. Let's modify the *print()* function to decode the data being received.

```
print(s.recv(1024).decode())
```

Listing 84 - The data received will now be decoded

Now that we have *decode()* added to our *print()* function, let's execute the script again.

```
kali@kali:~$ ./networkClient.py
You are connected.
Goodbye
```

Listing 85 - The output from the server looks better

The server only sent some data. Of course, most service applications are much more complex and can take data as input from the client.

> *The service on port 9999 will be changed to account for the ability for the client to send data. These are examples that should be read along with, as opposed to doing the activity. The service that we are connecting to on port 9999 will be rewritten to show another type of interaction we can get from services of this nature.*

Since the service was changed, let's just send our script to the newly modified service to analyze what may be on it.

```
kali@kali:~$ ./networkClient.py
Please send a number to be squared
```

Listing 86 - The server is now requesting a number be sent

Now, the server is requesting a number to be sent to the socket so it will be squared and returned. Let's send a number to the server with the *send()* function.

```
kali@kali:~$ cat networkClient.py
#!/usr/bin/python

import socket

s = socket.socket(socket.AF_INET,socket.SOCK_STREAM)

s.connect(("192.168.50.101", 9999))

print(s.recv(1024).decode())
s.send("5".encode())
s.close()
```

Listing 87 - The number is added with the send function and encoded

The number *5* is sent as a string and encoded for the server to understand the value. Let's execute the script again.

```
kali@kali:~$ ./networkClient.py
Please send a number to be squared
```

Listing 88 - The message is the same as before

The message returned from the server is the same as before. This is due to us not reading any new data that may have been sent as a result of our number. Let's add another *recv()* line to our script before the connection is closed.

```
kali@kali:~$ cat networkClient.py
#!/usr/bin/python

import socket

s = socket.socket(socket.AF_INET,socket.SOCK_STREAM)

s.connect(("192.168.50.101", 9999))

print(s.recv(1024).decode())
s.send("5".encode())
print(s.recv(1024).decode())
s.close()
```

Listing 89 - The additional data should be printed

Let's execute the script again to test if the server sends anything after we sent the number 5.

```
kali@kali:~$ ./networkClient.py
Please send a number to be squared
25
```

Listing 90 - The number we sent was squared and returned

In our last execution, the server did accept our number 5 and return its square, 25.

In this section, we covered how to create a Python network socket client to send and receive data from a server.

---

[1] (GeeksforGeeks, 2023), https://www.geeksforgeeks.org/socket-programming-python/ ↵

[2] (Wikipedia, 2023), https://en.wikipedia.org/wiki/Newline ↵

Resources

Some of the exercises require you to start the target machine(s) below.

Please note that the IP addresses assigned to your target machines may not match those referenced in the Module text and video.

PythonExercises

Exercises

1. Define a variable named *conn*, and set that to a socket that uses an IPv4 address and a TCP connection.

Answer — View hints

Answer — Verify

2. What is the function (in the socket module) to connect to a remote server? (supply only the function name)

Answer

Answer — Verify

3. How many parameters does the *connect()* function take?

Answer — View hints

Answer — Verify

4. What is the function to get data that is sent from a server? (supply only the function name)

Answer

Answer — Verify

5. What is the function to send data to a server? (supply only the function name)

Answer

Answer — Verify

6. Is the data sent in binary or text mode?

Answer

Answer — Verify

For the following exercises, create Python network socket client programs to meet the conditions of each question.

7. Connect to the exercise host on port 6666. The flag will be provided by the server.

Answer

Answer — Verify

8. Connect to the exercise host on port 7777 and fulfill the expected server demands.

Answer

Answer — Verify

9. Connect to the exercise host on port 7777 and send unexpected data.

Answer

Answer — Verify

10. Connect to the exercise host on port 8888 and fulfill the expected server demands.

Answer — View hints

Answer — Verify

Putting It All Together

This Learning Unit covers the following Learning Objectives:

1. Write a program using pseudocode
2. Create a program flowchart
3. Combine all of the previous sections to make a spider

This Learning Unit will take approximately 180 minutes to complete.

In this section, we'll be creating a program from scratch.

We've covered a lot of material regarding Python scripting. Let's walk through the development process of writing a complete and functional script that will leverage many of the concepts we covered.

The script we will be writing is a *web spider*.[1] Before we get into working with actual Python scripting, let's define what we want to create, how we plan to go about it, and write a program in pseudocode.

---

[1] (Wikipedia, 2023), https://en.wikipedia.org/wiki/Web_crawler ↩

Writing Programs in Pseudocode

Before writing any program, we need to have an idea of what and how we want to accomplish the task the program will complete. With this in mind, many programmers don't take the time to organize their programs. Instead, they jump directly into typing the code with the mindset that doing anything else would be time wasted. We are not going to adopt that mentality. We will take the time to organize our plan and write it in plain language (instead of writing it in the actual programming language). This is called *pseudocode*.[1] The time spent writing our pseudocode will save us time in the long run as we develop the program.

Let's begin with defining what the goal of our script is.

```
The script will download a web page, find all of the links on that page, and
recursively collect links on the website after following all of the links and display
them when complete.
```

Listing 91 - The goal of our script

This may sound like a cluttered definition. Let's break down what this program will do. The idea is that the script will be supplied with a web page to make a request. From there, the script will search for other links within the web page that was originally requested. It will store these links in a list and follow each link programmatically and repeat the process until no unique links are found. If this still doesn't make sense, that's ok. We'll cover this further as we build and organize our pseudocode.

Let's get started with our pseudocode and how we might handle each part of the scripting process. We defined the beginning step already.

```
1. The script will make a request to a web page - supplied by the user.
```

Listing 92 - The first step of the program

Next, the script needs to collect any links inside the requested web page. This can be handled in different ways, but we'll simplify this to parsing the web page for the "http" string to identify any URLs on the page. These need to be stored in a list after they are parsed. Next, let's translate this into pseudocode.

```
1. The script will make a request to a web page - supplied by the user.
2. The web page requested will be parsed to search for any URLs starting with 'http'.
3. Each found URL will be stored in a list variable.
```

Listing 93 - The pseduocode is expanded

Now, the script will need to follow each found URL and repeat the process. There's an issue with our idea though. What if the same URL is found on multiple of the found URLs in the list? We can create a dictionary of the URLs with a value of whether they have been searched or not. In the beginning, no URL, except the URL that was provided at the beginning of the script execution, should have been followed. After we have a way to keep track of which URLs have been requested or not, we need to repeat the request/parse process for each subsequent page. Let's modify our pseudocode to add these new steps:

```
1. The script will make a request to a web page - supplied by the user.
2. The web page will be added to a dictionary (isFollowed) with a value of 'yes'.
3. The web page requested will be parsed to search for any URLs starting with 'http'.
4. Each found URL will be stored in a list variable.
5. Each URL in the list variable will be checked against the dictionary (isFollowed) to
check if there is a 'yes' entry for that URL link.
6a. If it has already been followed, the URL will not be requested again.
6b. If it has not been followed, the URL will be requested and repeat the above
process.
```

Listing 94 - There's a decision in the pseudocode

Control over the web spider is extremely important. We need to also accommodate for a scope in our search term. We can add another search term in our parsing and handle it in a couple of ways. To make sure we don't create any unnecessary traffic to websites, we'll be writing this spider for our web server on the Exercise Host. We can filter URLs that do not end with the last octet of our exercise host. This will prevent the spider from leaving our website and continuing to spider websites that may have been referenced within ours.

Let's add this to our pseudocode with the ending action.

```
1. The script will make a request to a web page - supplied by the user.
2. The web page will be added to a dictionary (isFollowed) with a value of 'yes'.
3. The web page requested will be parsed to search for any URLs starting with 'http'
and the last octet of our exercise host IP.
4. Each found URL will be stored in a list variable.
5. Each URL in the list variable will be checked against the dictionary (isFollowed) to
check if there is a 'yes' entry for that URL link.
6a. If it has already been followed, the URL will not be requested again.
6b. If it has not been followed, the URL will be requested and repeat the above
process.
7. When the process is finished, print the list of URLs to the terminal each on their
own line.
```

Listing 95 - The pseudocode is complete, and we can move on to the next step.

Now that we have our pseudocode finished, let's take a moment to review what we learned with the following exercises:

---

[1] (GeeksforGeeks, 2023), https://www.geeksforgeeks.org/how-to-write-a-pseudo-code/ ↩

Exercises

1. What is it called when we write out a program outline in plain language?

Answer — View hints

Answer — Verify

2. Overall, does writing pseudocode before writing any programming language save time? (yes/no)

Answer

Answer — Verify

3. Pseudocode is written in the programming language we plan to use. (True/False)

Answer

Answer — Verify

Creating a Program Flowchart

To expand our organization into a format we can visualize, we'll begin creating a *flowchart*[1] of how we expect our script to work. This will be translating our already written pseudocode into a graphical medium.

Unlike writing pseudocode, creating a flowchart for the program may not be as valuable of an investment in time. Despite this, it can help us further realize how our program should be structured and translated into the programming language we are going to use. In our case, we will be writing a Python script. We can visualize the flow and the shapes to operations in the programming language. This will become more apparent as we continue making the flowchart.

For this Module, we'll be using the diagramming application *Lucidchart*.[2] There is a free-to-use tier that has limitations, but it should be fine with our need to create the flowchart for our web spider. We won't go into the usage of Lucidchart or other diagramming applications. Instead, we'll simply focus on translating our pseudocode to a visual format.

As a recap, let's review our pseudocode again.

```
1. The script will make a request to a web page - supplied by the user.
2. The web page will be added to a dictionary (isFollowed) with a value of 'yes'.
3. The web page requested will be parsed to search for any URLs starting with 'http'
and the last octet of our exercise host IP.
4. Each found URL will be stored in a list variable.
5. Each URL in the list variable will be checked against the dictionary (isFollowed) to
check if there is a 'yes' entry for that URL link.
6a. If it has already been followed, the URL will not be requested again.
6b. If it has not been followed, the URL will be requested and repeat the above
process.
7. When the process is finished, print the list of URLs to the terminal each on their
own line.
```

Listing 96 - The pseudocode we made earlier

Translating this into a flowchart may resemble the following image:

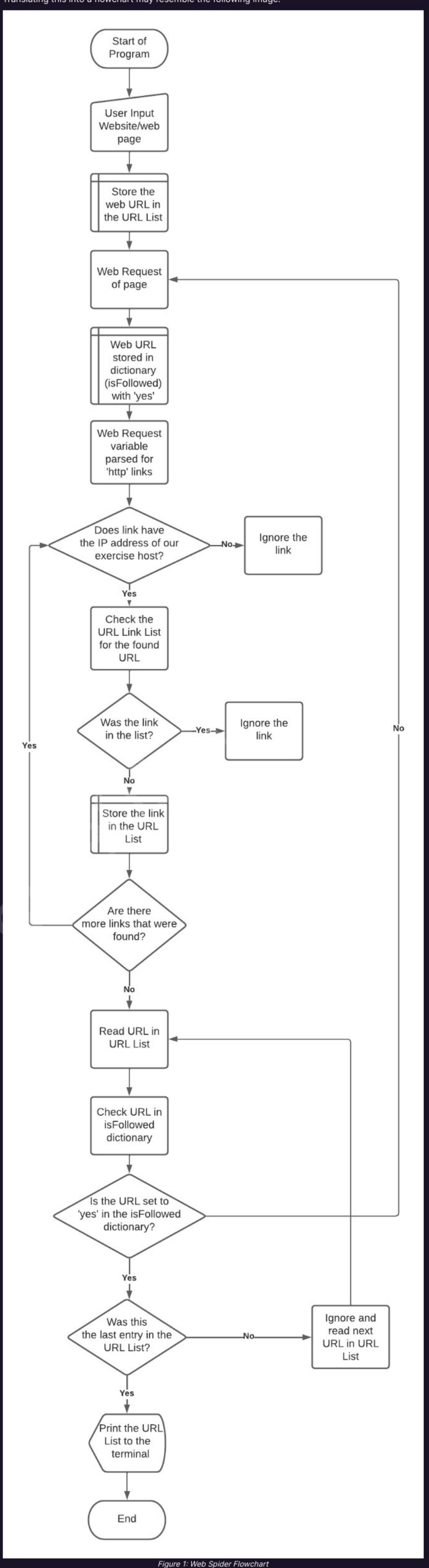

Figure 1: Web Spider Flowchart

At the beginning of the flowchart, we added a clear starting point in the diagram. The first thing that we need for the program to succeed is a target URL, which will be supplied by the end-user. The user may supply this as a terminal input or as a variable within the script. Despite having a flowchart, we can decide on the mechanisms handling these tasks when we program our script later.

After the starting web URL is provided, the program will store this URL in a URL list. From here, the program will request the webpage. After the webpage is requested and stored, the URL used will be stored in a dictionary, called *isFollowed*, followed by a value of "yes". After this is stored, the requested webpage will be parsed for any links that contain "http" in them.

With the webpage filtered for the "http" string in a link, the program will determine if that link contains the IP address of our host. This is to keep our spider from wandering outside of our targeted site. If it does have our target IP address, the link found will be checked in the URL list we started before. If it is already there, that link will also be ignored. If it wasn't found, the link will get stored in the URL list. The page will continue to get parsed for links with "http" in them and follow this same behavior until all of the links are found and the appropriate action is taken on them.

When all the links are parsed out of the webpage, the program will refer to the URL list again. Starting from the beginning of the list, the program will check if that URL is in the *isFollowed* dictionary variable. If the URL is either not in the dictionary or has not had the value of "yes" set to it, the URL will be used to request that webpage. It will then follow all of the steps covered above.

In the event the URL is set to "yes" in the *isFollowed* dictionary, the next URL list value will be read. This next URL will be checked if it has been followed as well. These loops will continue until all of the URL list values have been checked.

When the last URL list value check is complete, the URL list variable will be printed to the terminal. Once all the discovered links are printed to the terminal, the application will close.

This concludes our flowchart of the spidering program. Let's move on to writing the script.

---

[1] (Wikipedia, 2023), https://en.wikipedia.org/wiki/Flowchart ↩

[2] (Lucidchart, 2023), https://www.lucidchart.com/pages/ ↩

Creating the Spider

Now that we have our spider program idea in pseudocode and a flowchart, we'll translate this plan to actual code. Instead of providing the entire code base and explaining it, let's work through each step of programming this script as laid out in the flowchart. This way, we can conduct our scripting tests while writing the program, instead of having a final solution up front.

As we know, we need to have the start of our program. Let's name our script **webSpider.py** and add the shebang line.

```
kali@kali:~$ cat webSpider.py
#!/usr/bin/python3
```

Listing 97 - The program file is started with the shebang

The shebang in this file is a bit different in that the interpreter is calling on *python3* instead of just python. This will ensure that Python version 3 is used for our script.

According to our flowchart diagram, our first action is to take user input of a webpage. The simplest way for us to do this is to add a variable for this value. Let's add a variable called *URL* and assign it to our exercise host address. For this demonstration, the address will be 192.168.50.101.

```
kali@kali:~$ cat webSpider.py
#!/usr/bin/python3

URL = "http://192.168.50.101/"
```

Listing 98 - The initial URL has been set

With the initial URL set, we now need to store this in a URL list variable. We'll call ours *urlList*. Instead of creating the *urlList* variable with the initial value in it, let's start with setting the variable with no values and add the *URL* variable in it.

```
kali@kali:~$ cat webSpider.py
#!/usr/bin/python3

URL = "http://192.168.50.101/"
urlList = []

urlList.append(URL)
```

Listing 99 - The urlList list variable is defined and the URL is added to it

Instead of continuing forward, let's test our current code as written. The question is, "Are we properly adding the value expected to the list variable?" The easy way to test this is to *print* the urlList variable before and after the *append()* statement. Let's add this now.

```
kali@kali:~$ cat webSpider.py
#!/usr/bin/python3

URL = "http://192.168.50.101/"
urlList = []

print(urlList)
urlList.append(URL)
print(urlList)
```

Listing 100 - The debugging print functions are added

With the debugging *print* functions added, let's execute the script to make sure the *urlList* variable is being handled appropriately.

```
kali@kali:~$ chmod +x webSpider.py

kali@kali:~$ ./webSpider.py
[]
['http://192.168.50.101/']
```

Listing 101 - The urlList has been appropriately populated

With our test, we now know that the *URL* value has been added appropriately to the *urlList* variable. From here, we need to make a web request of that URL page and store it in a variable. As we covered before, to do this, we must *import* a module called *requests*. From there, we can use that module to make the request. We'll store the web request in a variable called *page*. For the sake of cleaning up our code, let's remove those debugging *print()* functions as well.

```
kali@kali:~$ cat webSpider.py
#!/usr/bin/python3

import requests

URL = "http://192.168.50.101/"
urlList = []

urlList.append(URL)

page = requests.get(URL)
```

Listing 102 - The requests module is imported and the URL is requested

With the requests module imported in the *page* variable population, we can take another moment to debug our program. Let's add a print statement for the content of the web page after the request to make sure we are pulling in what we expect.

```
kali@kali:~$ cat webSpider.py
#!/usr/bin/python3

import requests

URL = "http://192.168.50.101/"
urlList = []

urlList.append(URL)

page = requests.get(URL)
print(page.text)
```

Listing 103 - The debugging print statement is added to verify if the webpage is getting pulled in appropriately

With the debugging *print* statement in place, let's execute the script again.

```
kali@kali:~$ ./webSpider.py
<!doctype html>
<html> <!-- used for the background of entire website -->

<head>
    <!-- This cheesy site was made by RedHatAugust -->
    <title>The Secret World of MPG</title>
```

Listing 104 - The page is being shown as expected

Great! This is what we expected. Now, let's remove the debugging *print* statement and move on to the next step.

The next step in our flowchart is to store the URL in a dictionary variable called *isFollowed* and assign it the value "yes". We'll approach this the same way as the *urlList* variable and define the dictionary at the beginning of our script. From here, we can add the dictionary entry and its value after the page request. We'll also add a debugging *print* statement after the entry to verify the dictionary is populated correctly.

```
kali@kali:~$ cat webSpider.py
#!/usr/bin/python3

import requests

URL = "http://192.168.50.101/"
urlList = []
isFollowed = {}

urlList.append(URL)

page = requests.get(URL)
isFollowed[URL] = "yes"
print(isFollowed)
```

Listing 105 - The dictionary variable and entry are added

Let's execute the script to ensure it is working as expected.

```
kali@kali:~$ ./webSpider.py
{'http://192.168.50.101/': 'yes'}
```

Listing 106 - The dictionary has the expected values

Perfect! Now we will remove the *print* statement again and move on to the next step in our flowchart.

Here, we need to parse the page for any links that have "http" in them. We will accomplish this by using a *for* loop with a *split()*[1] method. The *split()* method will separate the variable based on a newline character. Instead of going back and forth, let's do this with the debugging built-in to test if we can iteratively read through each line in the *page* variable. To do this, we'll declare a *counter* variable before the *for* loop and set the value to "0". Inside the loop, we will print the *counter*, the line in the *page* variable, then increment the *counter* variable. For now, we have not considered the fact we need to search for the "http" string.

```
kali@kali:~$ cat webSpider.py
#!/usr/bin/python3

import requests

URL = "http://192.168.50.101/"
urlList = []
isFollowed = {}

urlList.append(URL)

page = requests.get(URL)
isFollowed[URL] = "yes"

counter = 0
for line in page.text.split("\n"):
    print(counter)
    print(line)
    counter = counter + 1
```

Listing 107 - The for loop is created to show each line of the page variable

With this, let's execute our script.

```
kali@kali:~$ ./webSpider.py
0
<!doctype html>
1
<html> <!-- used for the background of entire website -->
2

3
<head>
4
    <!-- This cheesy site was made by RedHatAugust -->
```

Listing 108 - The loop is able to iterate through each line of the page variable

The goal of this step is to identify any link that has "http" in it. Let's remove the counter and print of each line and modify the script to check if "http" is in the line it iterates. We will print only these lines for this next step.

```
kali@kali:~$ cat webSpider.py
#!/usr/bin/python3

import requests

URL = "http://192.168.50.101/"
urlList = []
isFollowed = {}

urlList.append(URL)

page = requests.get(URL)
isFollowed[URL] = "yes"

for line in page.text.split("\n"):
    if "http" in line:
        print(line)
```

Listing 109 - Each line will be searched for the "http" string and print it if it is found

Let's execute the script.

```
kali@kali:~$ ./webSpider.py
    <link href="http://192.168.50.101/css/main.css" rel="stylesheet" type="text/css" />
            <li><a href="http://192.168.50.101/index.html" class="current">Home</a>
</li>
            <li><a href="http://192.168.50.101/gettingStarted.html">Getting Started</a>
</li>
            <li><a href="http://192.168.50.101/techniques.html">Techniques</a></li>
            <li><a href="http://192.168.50.101/tutorials.html">Painting Tutorials</a>
</li>
            <li><a href="http://192.168.50.101/shops.html">Miniature Manufacturers</a>
</li>
    <img class="imgRight" src="http://192.168.50.101/images/Crazy_Gaming_Table.jpg"
alt="Crazy Tabletop Game Setup!" caption="This is way more than I have ever seen, but
this is a wargamer's dream!" width="200px" height="150px">
    <!-- Image taken from:
https://c2.staticflickr.com/4/3044/2775801876_f168dfcb66_b.jpg -->
            <li><a href="http://192.168.50.101/gettingStarted.html">Getting Started</a>
</li>
            <li><a href="http://192.168.50.101/techniques.html">Techniques</a></li>
            <li><a href="http://192.168.50.101/tutorials.html">Painting Tutorials</a>
</li>
            <li><a href="http://192.168.50.101/shops.html">Miniature Manufacturers</a>
</li>
    <img class="imgRight" src="http://192.168.50.101/images/Painter-Tux-Beret-Art-
Artist-Brush-Blotch-Colour-161318.png" width="40px" height="37px" alt="Shhh... It's a
secret.">
    <!-- Image taken from:  http://maxpixel.freegreatpicture.com/Painter-Tux-Beret-Art-
Artist-Brush-Blotch-Colour-161318 -->
```

Listing 110 - Every line in the page with "http" is printed to the terminal

In the output, there are links that do not contain the IP address of our host. Let's filter this further with a nested if statement.

```
for line in page.text.split("\n"):
    if "http" in line:
        if "192.168.50.101" in line:
            print(line)
```

Listing 111 - An additional string check for our IP Address is added under the check for the "http" string

Let's execute the script again to check if the output changes.

```
kali@kali:~$ ./webSpider.py
    <link href="http://192.168.50.101/css/main.css" rel="stylesheet" type="text/css" />
            <li><a href="http://192.168.50.101/index.html" class="current">Home</a>
</li>
            <li><a href="http://192.168.50.101/gettingStarted.html">Getting Started</a>
</li>
            <li><a href="http://192.168.50.101/techniques.html">Techniques</a></li>
            <li><a href="http://192.168.50.101/tutorials.html">Painting Tutorials</a>
</li>
            <li><a href="http://192.168.50.101/shops.html">Miniature Manufacturers</a>
</li>
    <img class="imgRight" src="http://192.168.50.101/images/Crazy_Gaming_Table.jpg"
alt="Crazy Tabletop Game Setup!" caption="This is way more than I have ever seen, but
this is a wargamer's dream!" width="200px" height="150px">
            <li><a href="http://192.168.50.101/gettingStarted.html">Getting Started</a>
</li>
            <li><a href="http://192.168.50.101/techniques.html">Techniques</a></li>
            <li><a href="http://192.168.50.101/tutorials.html">Painting Tutorials</a>
</li>
            <li><a href="http://192.168.50.101/shops.html">Miniature Manufacturers</a>
</li>
    <img class="imgRight" src="http://192.168.50.101/images/Painter-Tux-Beret-Art-
Artist-Brush-Blotch-Colour-161318.png" width="40px" height="37px" alt="Shhh... It's a
secret.">
```

Listing 112 - The links are now limited to those that contain the IP address of our target host

Instead of printing the line to the terminal, we need to filter out the links. We will do this with a *start* and *end* index to get the URL of each line and print that to the terminal. Let's do this the same way as covered in the *Strings and Slicing* section of this Module.

```
start = "http"
end = "\">"
for line in page.text.split("\n"):
    if "http" in line:
        if "192.168.50.101" in line:
            print(line[line.index(start):line.index(end)])
```

Listing 113 - The start and end index parameters are set and the line will be parsed

Let's execute the script and find out if we were able to parse the links correctly.

```
kali@kali:~$ ./webSpider.py
Traceback (most recent call last):
  File "/home/kali/./webSpider.py", line 20, in <module>
    print(line[line.index(start):line.index(end)])
ValueError: substring not found
```

Listing 114 - The substring failed

The script failed to execute. The error message indicates that there is an issue with the *substring*, so our parsing did not work as intended. Let's debug this issue to review what is going on. To start, let's *print* the index values for the *start* and the *end* conditions.

```
start = "http"
end = "\">"
for line in page.text.split("\n"):
    if "http" in line:
        if "192.168.50.101" in line:
            print(line)
            print(line.index(start))
            print(line.index(end))
            #print(line[line.index(start):line.index(end)])
```

Listing 115 - The print functions should indicate where the indexes are being assigned

Let's execute the script again.

```
kali@kali:~$ ./webSpider.py
    <link href="http://192.168.50.101/css/main.css" rel="stylesheet" type="text/css" />
16
Traceback (most recent call last):
  File "/home/kali/./webSpider.py", line 22, in <module>
    print(line.index(end))
ValueError: substring not found
```

Listing 116 - The issue appears to be with the end index

The script failed again. The *line* and the *start* index were printed to the terminal, which would indicate that the *end* index is the issue. Let's review the link lines again. To save time, only two lines will be shown in the following listing.

```
<link href="http://192.168.50.101/css/main.css" rel="stylesheet" type="text/css" />
            <li><a href="http://192.168.50.101/index.html" class="current">Home</a>
</li>
```

Listing 117 - The ending condition is different between the lines displayed

In the example listing above, the URL links have a different ending condition. With this, we can't create an index condition of "">"for all lines. Instead, the first line shown above ends the URL link with a "" " condition. Let's write up a short *if/else* statement to check for the potential ending condition and set the *end* variable to the found condition.

```
start = "http"
for line in page.text.split("\n"):
    if "http" in line:
        if "192.168.50.101" in line:
            if "\">" in line:
                end = "\">"
            else:
                end = "\" "
            print(line)
            print(line.index(start))
            print(line.index(end))
            #print(line[line.index(start):line.index(end)])
```

Listing 118 - The if/else condition is added to check for the potential ending index value

Let's execute the script again to analyze the difference in output.

```
kali@kali:~$ ./webSpider.py
    <link href="http://192.168.50.101/css/main.css" rel="stylesheet" type="text/css" />
16
50
            <li><a href="http://192.168.50.101/index.html" class="current">Home</a>
</li>
25
73
...
```

Listing 119 - The script executed without failure

There's a lot of terminal output with the method we used to debug. Let's cut out all the debugging statements and print the new URL values to find out if the slicing is working as expected.

```
start = "http"
for line in page.text.split("\n"):
    if "http" in line:
        if "192.168.50.101" in line:
            if "\">" in line:
                end = "\">"
            else:
                end = "\" "
            print(line[line.index(start):line.index(end)])
```

Listing 120 - The debugging lines were removed and we should get the URLs on execution now

Let's execute the script to find the URL results.

```
kali@kali:~$ ./webSpider.py
http://192.168.50.101/css/main.css
http://192.168.50.101/index.html" class="current
http://192.168.50.101/gettingStarted.html
http://192.168.50.101/techniques.html
http://192.168.50.101/tutorials.html
http://192.168.50.101/shops.html
http://192.168.50.101/images/Crazy_Gaming_Table.jpg" alt="Crazy Tabletop Game Setup!"
caption="This is way more than I have ever seen, but this is a wargamer's dream!"
width="200px" height="150px
http://192.168.50.101/gettingStarted.html
http://192.168.50.101/techniques.html
http://192.168.50.101/tutorials.html
http://192.168.50.101/shops.html
http://192.168.50.101/images/Painter-Tux-Beret-Art-Artist-Brush-Blotch-Colour-
161318.png" width="40px" height="37px" alt="Shhh... It's a secret.
```

Listing 121 - There are lines that have more than the URL in the output

So close! There are many URL lines that are correctly sliced, but there are a few lines that have extra HTML data in them. Let's set this to a variable and run an additional test on the variable to determine if there are any quotes (") in the line. If there are, we can slice the line again with the *end* variable set to """.

```
start = "http"
for line in page.text.split("\n"):
    if "http" in line:
        if "192.168.50.101" in line:
            if "\">" in line:
                end = "\">"
            else:
                end = "\" "
            sliced = line[line.index(start):line.index(end)]
            if "\"" in sliced:
                end = "\""
                print(sliced[sliced.index(start):sliced.index(end)])
            else:
                print(sliced)
```

Listing 122 - There is an additional check for quotes (") in the sliced line

If the *sliced* variable has quotes (") in it, it will be sliced an additional time with a new *end* condition for the index. Otherwise, the program will assume the line was fine and print the *sliced* variable without modification.

```
kali@kali:~$ ./webSpider.py
http://192.168.50.101/css/main.css
http://192.168.50.101/index.html
http://192.168.50.101/gettingStarted.html
http://192.168.50.101/techniques.html
http://192.168.50.101/tutorials.html
http://192.168.50.101/shops.html
http://192.168.50.101/images/Crazy_Gaming_Table.jpg
http://192.168.50.101/gettingStarted.html
http://192.168.50.101/techniques.html
http://192.168.50.101/tutorials.html
http://192.168.50.101/shops.html
http://192.168.50.101/images/Painter-Tux-Beret-Art-Artist-Brush-Blotch-Colour-
161318.png
```

Listing 123 - The URL lines appear as expected

The output of the parsing algorithm we set appears to be working now. Now that we have the expected output, we need to refer to the *urlList* and add the link if it isn't in the list already. To do this, let's take a moment away from this section of the code and add a custom function at the top of our script called *checkUrlList*. This function will take a parameter and loop through the *urlList* list variable to check if it exists or not. It will then return *True* or *False* based on the search result.

```
def checkUrlList(URL):
    if URL in urlList:
        return True
    else:
        return False
```

Listing 124 - The function takes an argument and checks each value in the urlList variable for that argument

As we've been doing, let's add debug print functions to test the function. Let's do this after our first URL is added to the *urlList* list. We'll add one print statement that should result in a "True" return and another that should result in a "False" return.

```
urlList.append(URL)
print(checkUrlList(URL))
print(checkUrlList("http://www.offsec.com/"))
```

Listing 125 - The debug print functions are added

Let's execute the script and find out what each debugging statement returns.

```
kali@kali:~$ ./webSpider.py
True
False
...
```

Listing 126 - The returns displayed to the terminal as expected

Now that we validated the *checkUrlList* function, let's incorporate it in the http parsing section. If the link is found in the *urlList*, it will be ignored. If it is not on the list, it will be added.

We can remove all the debugging *print()* functions from the code as well. Instead of the print functions, let's set the *sliced* value to a different variable, called *parsedURL*. For the sake of brevity, we'll also add the debugging *print* statement outside of the loop to analyze if the sliced URLs were added to the *urlList* variable.

```
start = "http"
for line in page.text.split("\n"):
    if "http" in line:
        if "192.168.50.101" in line:
            if "\">" in line:
                end = "\">"
            else:
                end = "\" "
            sliced = line[line.index(start):line.index(end)]
            if "\"" in sliced:
                end = "\""
                parsedURL = sliced[sliced.index(start):sliced.index(end)]
            else:
                parsedURL = sliced

            if checkUrlList(parsedURL) == False:
                urlList.append(parsedURL)

print(urlList)
```

Listing 127 - The parsed links should be stored in the urlList list variable

Let's test the script and analyze the output.

```
kali@kali:~$ ./webSpider.py
['http://192.168.50.101/', 'http://192.168.50.101/css/main.css',
'http://192.168.50.101/index.html', 'http://192.168.50.101/gettingStarted.html',
'http://192.168.50.101/techniques.html', 'http://192.168.50.101/tutorials.html',
'http://192.168.50.101/shops.html',
'http://192.168.50.101/images/Crazy_Gaming_Table.jpg',
'http://192.168.50.101/gettingStarted.html', 'http://192.168.50.101/techniques.html',
'http://192.168.50.101/tutorials.html', 'http://192.168.50.101/shops.html',
'http://192.168.50.101/images/Painter-Tux-Beret-Art-Artist-Brush-Blotch-Colour-
161318.png']
```

Listing 128 - The URL links were added to the list

Now that the links were stored in the *urlList* variable, we can read the *urlList* and compare the values with the *isFollowed* dictionary. If the entry doesn't exist, the whole process needs to start over. If the entry does exist and has a value of "yes", the URL entry will be skipped.

Let's build another function that will handle checking if a URL is in the dictionary *isFollowed* and if there is a value set to "yes" for that entry. If either one of these conditions is not met, the function will return *False*. Let's name the function *isFollowedCheck* and put it after our previously written function.

```
def isFollowedCheck(URL):
    for entry in isFollowed.keys():
        if URL != entry:
            return False
        else:
            if isFollowed[URL] == "yes":
                return True
            else:
                return False
```

Listing 129 - The function to check a URL and if the URL was requested

Let's test the function after we previously set the initial *URL* variable dictionary value to "yes". Again, we'll test for a *True* and a *False* return.

```
isFollowed[URL] = "yes"
print(isFollowedCheck(URL))
print(isFollowedCheck("http://www.offsec.com/"))
```

Listing 130 - The debugging print functions are added after the initial URL was set to "yes" in isFollowed

Let's execute the script to test our function.

```
kali@kali:~$ ./webSpider.py
True
False
```

Listing 131 - The function works as expected

With the test of our function complete, we can remove the *print()* functions. This is where our flowchart may come in handy. Our entire script flow is about to change with that loop from determining if the URL was followed or not. If we analyze the flowchart, the flow of the program should execute back to the beginning of our request operation.

We can interpret this choice to return to the beginning of our request operation to be a loop that starts with that. Once the *urlList* list variable is exhausted and all links recorded in that list are followed, the program will continue past the loop. The last thing we must do is print the *urlList* list to the terminal. For the sake of presentation, let's put that in a *for* loop and print each link on its own line.

Indentation matters in Python. The entire script will need to be shifted to the appropriate spacing under the *for* loop and maintain the structure of the code already written. Here is our completed spider script:

```
kali@kali:~$ cat webSpider.py
#!/usr/bin/python3
import requests

URL = "http://192.168.50.101/"
urlList = []
isFollowed = {}

def checkUrlList(URL):
   if URL in urlList:
       return True
   else:
       return False

def isFollowedCheck(URL):
   for entry in isFollowed.keys():
       if URL != entry:
           return False
       else:
           if isFollowed[URL] == "yes":
               return True
           else:
               return False

urlList.append(URL)

for URL in urlList:
   if isFollowedCheck(URL) != True:
       page = requests.get(URL)
       isFollowed[URL] = "yes"

       start = "http"
       for line in page.text.split("\n"):
           if "http" in line:
               if "192.168.50.101" in line:
                   if "\">" in line:
                       end = "\">"
                   else:
                       end = "\" "
                   sliced = line[line.index(start):line.index(end)]
                   if "\"" in sliced:
                       end = "\""
                       parsedURL = sliced[sliced.index(start):sliced.index(end)]
                   else:
                       parsedURL = sliced
                   if checkUrlList(parsedURL) == False:
                       urlList.append(parsedURL)
                       isFollowed[parsedURL] = "no"

for URL in urlList:
   print(URL)
```

Listing 132 - The script is completed by closing the logic loop through the urlList list variable

To finalize this, let's run the script to determine if our loop was successful.

```
kali@kali:~$ ./webSpider.py
http://192.168.50.101/
http://192.168.50.101/css/main.css
http://192.168.50.101/index.html
http://192.168.50.101/gettingStarted.html
http://192.168.50.101/techniques.html
http://192.168.50.101/tutorials.html
http://192.168.50.101/shops.html
http://192.168.50.101/images/Crazy_Gaming_Table.jpg
http://192.168.50.101/images/Painter-Tux-Beret-Art-Artist-Brush-Blotch-Colour-
161318.png
```

Listing 133 - The spider printed out all the pages found on the site

The spider took a moment to complete the iterations and page requests. When completed, the links are displayed to the terminal output.

Let's take a moment to practice what was covered with the following exercise.

---

[1] (GeeksforGeeks), https://www.w3schools.com/python/ref_string_split.asp ↩

Resources

Some of the exercises require you to start the target machine(s) below.

Please note that the IP addresses assigned to your target machines may not match those referenced in the Module text and video.

PythonExercises

Exercises

1. Determine the IP address of the exercise host and replicate the spider script as covered in this section. The flag will be a hidden link somewhere on the site and will be reported by the spider if it works as designed.

Answer

Answer | Verify

Network Scripting

In this Module, we will cover the following Learning Units:

- Write a client program with Python - I
- Write a client program with Python - II
- Write a server program with Python
- Write a port scanner with Python
- Website interaction with Python - I
- Website interaction with Python - II
- Capture and send packets with Scapy

Note that this Module relies on knowledge relevant to Linux administration, scripting, and networking. Should you feel the need to, please brush up on those subject areas before completing this Module.

Creating network scripts can improve our flexibility, consistency, and efficiency when it comes to interacting with targets over a network. The ability to write our own network scripts can help us dramatically reduce the number of repetitive tasks we need to make. Perhaps more importantly, it can allow us to understand how programs and applications communicate with each other on a deeper level.

In some cases, we can even write our own network scripts to troubleshoot tools when they are not working as expected. In addition, writing our own network scripts can come in handy when we do not have access to tools, for example, on a compromised target. It can be time-consuming to transfer our normal programs and tools onto a target system, and they can get flagged by antivirus or other security defenses that are in place.

Write a Client with Python - I

This Learning Unit covers the following Learning Objectives:

1. Understand the concept of network sockets and their use cases
2. Build a basic networking client in Python
3. Use a loop to initiate multiple connections to a listening server

Building a Basic Client

Although there are many programming languages that we can use to complete our tasks, Python is a very popular language that penetration testers use to create their network scripts. This is due to its ease of use and the large number of libraries available for it.

For programs and systems to communicate with each other on a network, they use *sockets*[1] and the *socket API*[2] to send messages back and fourth. A socket[3] is essentially an endpoint that allows network communication to flow between two programs running over a network. We can implement network sockets on several different channel types.

As we begin to write our scripts, we recommend that you follow along and type the syntax in your own Python files. Try not to copy/paste the code, because writing it yourself can help reinforce understanding and memory.

Let's use the following Python code to start creating a script that uses the *socket*[4] module. We'll import the socket library for our Python script, and then call the *socket.socket* method.

```
#!/usr/bin/python3
#client.py

import socket

s = socket.socket(<socket_family>, <socket_type>, <protocol>)
```

Listing 1 - Initial Python script

Above, we set the *socket.socket* method to the variable *s*. The *socket.socket* method itself contains three (currently unset) variables: *socket_family*, *socket_type*, and *protocol*. Let's examine these variable placeholders.

The *socket_family* variable allows us to specify a protocol domain that will act as the transport mechanism. The most common, AF_INET, is used for IPv4 Internet addressing and AF_INET6 is used for IPv6 Internet addressing. AF_UNIX is the address family for *Unix Domain Sockets* (UDS).[5] This socket family allows the operating system to pass data directly from process to process, without going through the network stack.

The *socket_type* variable allows the communication between two endpoints. The socket type is usually either SOCK_DGRAM for the *User Datagram Protocol* (UDP)[6] or SOCK_STREAM for the *Transmission Control Protocol* (TCP).[7]

The *protocol* variable can be used to specify the protocol number. It is usually set to 0, which is the default value that will be set if it is not specified.

To supply these variables with values, we will create a socket that will communicate with an IPv4 address to transmit our communication over TCP.

```
#!/usr/bin/python3
#client.py

import socket

client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

Listing 2 - Creating an IPv4 TCP socket

Notice that we have not specified a protocol, so it will default to a value of 0 as mentioned above.

We will continue to expand on this script as we create our client. Before we proceed, we need to understand some of the methods that are built into the socket module. We'll slowly build up our client by applying each relevant method. At the end of this process, we will have a fully functional networking client that can handle errors and receive data of arbitrary length.

---

[1] (Steve's Internet Guide, 2021), http://www.steves-internet-guide.com/tcpip-ports-sockets/ ↩

[2] (Python Software Foundation, 2021), https://docs.python.org/3/library/socket.html ↩

[3] (Geeks for Geeks, 2021), https://www.geeksforgeeks.org/socket-programming-python/ ↩

[4] (Python Software Foundation, 2021), https://python.readthedocs.io/en/latest/library/socket.html ↩

[5] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Unix_domain_socket ↩

[6] (Wikipedia, 2021), https://en.wikipedia.org/wiki/User_Datagram_Protocol ↩

[7] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Transmission_Control_Protocol ↩

Socket Methods

The most common type of socket applications are client-server applications, such as the one we are currently building. This involves a client making a *request* to the server. The client then receives a *response* from the server.

The socket module comes with various methods to facilitate the various actions a client (or a server) will make during such a communication. There are three sets of socket methods that we need to be aware of: client socket methods, server socket methods, and general socket methods. Usually - though not always - a client will invoke client socket methods, a server will invoke server socket methods, and both programs can make use of the general methods.

The *socket.gethostname()* method[1] returns the name of the current system. We'll be using it in our scripts to test execution on our local machine. This means that if we want to run our scripts against an external server, we'll need to specify the IP address of the remote target.

```
#!/usr/bin/python3
#client.py

import socket

client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
host = socket.gethostname()
port = 8080
```

Listing 3 - Setting the target and port

In the above listing, we specify our own localhost with the *socket.gethostname()* method, and then we specify the port we want to connect on as an integer.

The *socket.connect(address)* method[2] is used to initiate a connection with the server. The method requires that we specify a single host and port to connect on, which we defined in the *host* and *port* variables.

```
#!/usr/bin/python3
#client.py

import socket

client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
host = socket.gethostname()
port = 8080

client.connect((host, port)) # Connect to our client
```

Listing 4 - Initiating a connection

Notice the double parentheses within the *s.connect((host, port))* syntax. The reason for this is because the socket module treats *(host, port)* as a single argument. If we only included one pair of parentheses, Python would interpret our syntax as attempting to provide two arguments to a method that only accepts one.

Now that we understand how to use the *socket.connect* method, there are some general socket methods that we need to become familiar with to use with our client so that it can receive data and terminate.

The *socket.recv(bufsize)* method[3] allows the client to receive a TCP message from the socket. The *bufsize* (buffer size) argument defines the maximum amount of data that the method can receive at any one time.

```
#!/usr/bin/python3
#client.py

import socket

client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
host = socket.gethostname()
port = 8080

client.connect((host, port)) # Connect to our client
msg = client.recv(1024)

print (msg.decode('ascii'))
```

Listing 5 - Receiving and displaying data

In the above listing, a client would connect to a server, and then print out any data it receives from the server via the *socket.recv()* method.

We now have enough code to connect to a server.

> *If you are following along, you will not yet have the server code to test out your client, so for now simply make sure that the client code you have written is identical to the code above.*

```
kali@kali:~$ python3 client.py
Connection Established
```

Listing 6 - Connecting to a server

In the above listing, we execute our client against a server running on our own localhost, and receive a message from the server that the connection has been established.

The *socket.close()* method[4] is pretty straightforward as it will just close the socket. This can be invoked from either end and will terminate the connection between the client and the server.

```
#!/usr/bin/python3
#client.py

import socket

client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
host = socket.gethostname()
port = 8080

client.connect((host, port)) # Connect to our client
msg = client.recv(1024)
client.close()

print (msg.decode('ascii'))
```

Listing 7 - Closing the connection

We've now built up a fully functioning client program in Python. Let's review what we've learned so far. Our script is designed to connect to a local server that is running on port 8080. The *socket.connect()* method will establish the connection. If the connection is successful, the client will receive a message from the server with *socket.recv()*. The *socket.close()* method will close the client, and then the *print* function[5] will decode and display the message from the server.

Once we have finished writing the program, we need to save the file to our system. We'll call it **client.py**.

---

[1] (Python Software Foundation, 2021), https://docs.python.org/3/library/socket.html#socket.gethostname ↩

[2] (Python Software Foundation, 2021), https://docs.python.org/3/library/socket.html#socket.socket.connect ↩

[3] (Python Software Foundation, 2021), https://docs.python.org/3/library/socket.html#socket.socket.recv ↩

[4] (Python Software Foundation, 2021), https://docs.python.org/3/library/socket.html#socket.socket.close ↩

[5] (Python Software Foundation, 2021), https://docs.python.org/3/library/functions.html#print ↩

Resources

Some of the exercises require you to start the target machine(s) below.

Please note that the IP addresses assigned to your target machines may not match those referenced in the Module text and video.

Network-Scripting

Exercises

1. What socket family will allow us to connect to a system that has an IPv4 Address?

Answer

Answer | Verify

2. Take a look at the following syntax: s.connect(("127.0.0.1",9090)) . What kind of socket method is the method being invoked here?

```
A. Client
B. Server
C. Socket
D. General
```

Answer

Answer | Verify

3. This is a scripting exercise. Use Python to connect to the server on port 2000 of the provided VM. When a client connects to the server, it will receive a certain response. With your Python script, send this exact response back to the server, and you will receive a second response. Your task is to send and receive 10 connections to and from the server within 15 seconds to obtain the flag.

Answer | View hints

Answer | Verify

Write a Client Program in Python - II

This Learning Unit covers the following Learning Objectives:

1. Add error handling mechanisms to a Python networking client
2. Handle data of unknown size with a Python networking client
3. Create interactive sockets to dynamically communicate with a listening server

Write a Client with Python - I
Socket Methods

Write a Client Program in Python - II
Error Handling: Try and Except Clauses

Resource Center

Error Handling: Try and Except Clauses

Sometimes, our client code may not work as desired, because the server responds in a way that we don't expect, or because it isn't working properly. To make our program more robust, we can introduce *error handling*[1] that will tell the client what to do if it encounters an error.

Python makes use of *try* and *except* statements to handle errors. Here is an example of what a *try* and *except* pair might look like in psedo-code.

```
try:
    do something
    break
except <exception type>:
    print an error statement
```

Listing 8 - Some try/except psuedo-code

A *try* statement attempts to execute any code within the *try* block. If the code within the *try* block executes successfully, the program will skip over the *except* block and continue its execution flow.

If, however, it does encounter an error (also known as an *exception*), execution flow will immediately jump to the corresponding *except* block with the corresponding *exception type*. Then, the code within the *except* block is executed to *handle* the exception. Once the *except* block is finished executing, the program will continue its execution flow.

Finally, if there is no corresponding exception type for the error encountered within the *try* block, the program will stop its execution because it won't know how to continue. This is called an *unhandled exception*.

Let's go ahead and add *try* and *except* statements to our Python client.

```
#!/usr/bin/python3
#client.py

import socket

client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

host = socket.gethostname()
port = 8080

try:
    client.connect((host, port))
    msg = client.recv(1024)
    client.close()
    print (msg.decode('ascii'))

except ConnectionRefusedError:
    print ("The server is not accepting our connection request!")
    exit(1)

print ("This sentence will only print if the except block was not executed.")
```

Listing 9 - Adding a try/except to our client

The above code checks for the specific exception called *ConnectionRefusedError*, which indicates if a server is unwilling, or unable, to accept the client's connection. We can run our client code against a non-existent server to validate the *try* and *except* blocks.

```
kali@kali:~$  python3 client.py
The server is not accepting our connection request!
```

Listing 10 - Attempting to connect to a non-existent server

Notice how the last line of our program is not executed. This is because the *except* block prints a statement and then exits the program before the last line can be run.

Here, we are only using the *except* block to print a statement and exit. However, we could create far more complex instructions, allowing our program to perform diagnostics, connect to another server, or do some unrelated thing. In the following exercise, you will use *try* and *except* statements to allow the program to reconnect to a server an arbitrary number of times.

---

[1] (Python Software Foundation, 2021), https://docs.python.org/3/tutorial/errors.html ↩

Resources

Some of the exercises require you to start the target machine(s) below.

Please note that the IP addresses assigned to your target machines may not match those referenced in the Module text and video.

Network-Scripting

Exercises

1. The server running on port 2001 of the provided VM is buggy; it only responds sometimes. Make sure that your client program has a means of handling errors and reconnecting. As in the previous exercise, connect 10 successful times in 15 seconds to obtain the flag.

Answer — View hints

Answer — Verify

Handling Unknown Data Size

In the above demonstration, we've assumed that the server is only going to send our client 1024 bytes of data. Sometimes, we may not be able to anticipate the exact size of the server's response. Let's execute our client against a server that will send us more than 1024 bytes and examine what happens.

We've adjusted our server to send the "Connection Established" string followed by 2000 "A" characters. We'll then execute our client code and pass the results to **sed**, **tr**, and **wc** to count the exact number of A's we receive from the server.

```
kali@kali:~$ python3 client.py | sed 's/[^A]//g' | tr -d '\n' | wc -c
    1002
```

Listing 11 - Counting the A's from the server

Since the "Connection Established " string is 22 characters long, only 1002 out of 2000 A's are sent to the client. This is because our implementation of the *socket.recv* method only allows 1024 bytes of data to be received by our client.

The *socket* module documentation suggests that the value used for each particular *socket.recv* call should be a small power of 2, such as 1024, 2048, or 4096. When we don't know how many bytes we'll be receiving, we can process the response in a loop by using smaller chunks until all data has been received. In the following exercise, you will add a loop to the client program so that it can receive arbitrary chuncks of data.

Resources

Some of the exercises require you to start the target machine(s) below.

Please note that the IP addresses assigned to your target machines may not match those referenced in the Module text and video.

Network-Scripting

Exercises

1. Connect to the server on port 2002 of the provided VM. The response you receive will be of unknown length, so build in some provisions in your client script to handle the responses using loops. You will need to connect to the server several times to receive the flag.

Answer

View hints

Answer

Verify

Interactive Sockets

Sometimes we may want to establish a connection with a server, and then send it data based on the information it provides us. To do this, we can create an *Interactive Socket* with the Telnet library.[1] Let's begin with our original client code, and import *telnetlib*.

```
#!/usr/bin/python3
#interactive-client.py

import socket
import telnetlib

client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
host = socket.gethostname()
port = 8080

client.connect((host, port)) # Connect to our client
msg = client.recv(1024)
client.close()

print (msg.decode('ascii'))
```

Listing 12 - Importing telnetlib

The *telnetlib* allows for an implementation of the Telnet[2] protocol. We will be modifying our script to make use of the *telnetlib.interact()* method, which will allow us to interact with the server dynamically. To implement this method, we'll create a function that we will call after our client has connected to the server.

```
#!/usr/bin/python3
#interactive-client.py

import socket
import telnetlib

def interact(socket):
    t = telnetlib.Telnet()
    t.sock = s
    t.interact()

client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
host = socket.gethostname()
port = 8080

client.connect((host, port)) # Connect to our client
msg = client.recv(1024)
print (msg.decode('ascii'))

client.close()
```

Listing 13 - Defining the interact function

Notice that the function that we've created is named "interact". But the name of the telnet function is also called "interact"! This is not essential: we could call our function whatever we wish. However, it is a good reminder that we should always understand the *scope*[3] of different code blocks so that we do not become confused.

The mechanism that allows this client to maintain interactivity with the server is all done via Telnet's own *interact* function. This function sets up a *while true* loop that keeps reading and writing data. Since *while true* is always true, it continues to read and write data from and to the server until the connection is severed.

Next, we'll call our new function after we connect to the server.

```
#!/usr/bin/python3
#interactive-client.py

import socket
import telnetlib

def interact(socket):
    t = telnetlib.Telnet()
    t.sock = socket
    t.interact()

client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
host = socket.gethostname()
port = 8080

client.connect((host, port)) # Connect to our client
msg = client.recv(1024)
print (msg.decode('ascii'))
interact(client)
client.close()
```

Listing 14 - Interacting with the socket

In the above listing, our client code is now capable of interacting dynamically with the server it connects to. Note that this functionality depends on two things:

1. The server must allow the connection to stay open. If it closes the connection, our client will not be able to send any further data.
2. The server must be configured to receive the data we send to it. Since we haven't yet implemented our own server, you can try the above client code against the remote VM in the following exercise.

---

[1] (Python Software Foundation, 2021), https://docs.python.org/3/library/telnetlib.html ↩

[2] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Telnet ↩

[3] (Wikipedia, 2022), https://en.wikipedia.org/wiki/Scope_(computer_science) ↩

Resources

Some of the exercises require you to start the target machine(s) below.

Please note that the IP addresses assigned to your target machines may not match those referenced in the Module text and video.

Network-Scripting

Exercises

1. Use your Python skills to connect to the server on port 2003 of the provided VM. The server will send any clients that connect to it some questions. Answer all the questions correctly to obtain the flag.

Answer

Answer | Verify

Write a Server with Python

This Learning Unit covers the following Learning Objectives:

1. Understand the server socket methods, like *socket.bind* and *socket.listen*
2. Build a basic networking server in Python
3. Implement interactive sockets in a client and server

Write a Client Program in Python - II
Interactive Sockets

Write a Server with Python
Building a Basic Server

Building a Basic Server

We'll start building our sever by importing the socket module, initializing a socket, and defining a host and port:

```
#!/usr/bin/python3
#server.py

import socket

server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
host = socket.gethostname()
port = 8080
```

Listing 15 - Initial server code

Note that this code is almost identical to the beginning of our client-side code. We'll now introduce a few more socket methods that our server will need to make use of.

The *socket.bind(address)* method[1] *binds*, or assigns, a specific port to our program. In this case, we want to bind our server to the port we defined in the *port* variable.

```
#!/usr/bin/python3
#server.py

import socket

server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
host = socket.gethostname()
port = 8080

server.bind((host, port))
```

Listing 16 - Binding to a port

Like the *socket.connect* method, *socket.bind* expects a complete address as input in the format of (host, port). This is why there are double parentheses in the method call.

The *socket.listen(int)* method[2] tells the server to listen for incoming connections and expects an integer.

```
#!/usr/bin/python3
#server.py

import socket

server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server = socket.gethostname()
port = 8080

server.bind((host, port))
server.listen(2) # Wait for a client connection. Only 2 clients can connect to the
server
print('Server is listening for incoming connections')
```

Listing 17 - Listening for a client connection

The integer specified in *socket.listen()* represents the number of clients the server will allow to connect to itself simultaneously. Once the server is listening, it reports its status via the *print* function.

The *socket.accept()* method[3] returns a pair of values (*conn, address*) where *conn* represents a new socket that will send and receive messages, and *address* represents the client address bound to the socket.

```
#!/usr/bin/python3
#server.py

import socket

server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server = socket.gethostname()
port = 8080

server.bind((host, port))
server.listen(2) # Wait for a client connection. Only 2 clients can connect to the
server
print('Server is listening for incoming connections')

while True:
    conn, address = server.accept() # Establish the connection with the client
    print("Connection Received from %s" % str(addr))
```

Listing 18 - Allowing incoming connections

In the above listing, we start a *while True*[4] loop that allows incoming connections via the newly created *conn*, *address* pair. Once a connection has been established, our server will report that the connection has occured.

The *socket.send(bytes)* general socket method[5] allows a client or server to send data to the socket. This data can then be received via the *socket.recv* method. The *bytes* argument will provide several bytes that will be sent to the socket. Specifying these bytes can change how the method interacts with the receiving machine.

```
#!/usr/bin/python3
#server.py

import socket

server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server = socket.gethostname()
port = 8080

server.bind((host, port))
server.listen(2) # Wait for a client connection. Only 2 clients can connect to the
server
print('Server is listening for incoming connections')

while True:
    conn, address = server.accept() # Establish the connection with the client
    print("Connection Received from %s" % str(addr))
    msg = 'Connection Established'+ "\r\n"
    conn.send(msg.encode('ascii'))
```

Listing 19 - Sending data to the socket

In the above listing, we use the new *conn* socket to send the text "Connection Established" to the client once it connects to the server.

We have now almost completed our server. The last functionality we will add is the ability to close the socket from the server-side with *socket.close()*.

```
#!/usr/bin/python3
#server.py

import socket

host = socket.gethostname()
port = 8080

server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind((host,port))
server.listen(2)
print('Server is listening for incoming connections')

while True:
    conn,addr = server.accept()
    print("Connection Received from %s" % str(addr))
    msg = 'Connection Established'+ "\r\n"
    conn.send(msg.encode('ascii'))
    conn.close()
```

Listing 20 - Closing the socket

Notice how we close the *conn* socket and not the original *server* socket, to allow our server to keep running and to accept further connections.

Let's summarize what we have learned. After creating a new socket, we use the *socket.bind()* method, which binds the program to a specified IP address and port. This allows the server to listen for incoming requests. To make sure the server is listening for these requests, we use the *socket.listen()* method. Once the client has requested to connect, we use the *socket.accept()* method to accept the connection, and the *socket.send()* method to send a message back to the client. Finally, we invoke *socket.close()* to terminate the connection.

Once the script has been written, we will save the Python program as **server.py**.

---

[1] (Python Software Foundation, 2021), https://docs.python.org/3/library/socket.html#socket.socket.bind ↩

[2] (Python Software Foundation, 2021), https://docs.python.org/3/library/socket.html#socket.socket.listen ↩

[3] (Python Software Foundation, 2021), https://docs.python.org/3/library/socket.html#socket.socket.accept ↩

[4] (Python Software Foundation, 2021), https://docs.python.org/3/reference/compound_stmts.html#while ↩

[5] (Python Software Foundation, 2021), https://docs.python.org/3/library/socket.html#socket.socket.send ↩

Testing our Client and Server

To complete this section of the Learning Unit, a working version of the Python client built in the 'Write a Client Program in Python' Learning Units is required.

Now that we have built our client and server programs, let's find out if we can get them to work together. First, we'll open up two terminal windows: one to run the client and the other to run the server. In Terminal One, we start our server and should receive the following output on the console:

```
kali@kali:~$ python3 server.py
Server is listening for incoming connections
```

Listing 21 - Starting our server

We then run the client program in the Terminal Two window. We should receive the following output:

```
kali@kali:~$ python3 client.py
Connection Established

kali@kali:~$
```

Listing 22 - Running our client

To confirm that the connection between our client and our server was successful, we can check the Terminal One window running the server.

```
kali@kali:~$ python3 server.py
Server is listening for incoming connections
Connection Recieved from ('127.0.0.1', 48904)
```

Listing 23 - Output from the server

Above, notice how port 48904 is allocated to the client after initiating a socket connection to the server on port 8080. Reading through our client and server Python code, we never specified the number 48904 anywhere. When you execute your own version of the program, you'll find that a different number is used. This is because the operating system itself automatically allocates a temporary *ephemeral port*[1] based on a pre-defined range, to ensure that a port is always available to assign to a connecting client. This allows multiple clients to connect simultaneously.

---

[1] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Ephemeral_port ↩

Resources

Some of the exercises require you to start the target machine(s) below.

Please note that the IP addresses assigned to your target machines may not match those referenced in the Module text and video.

Network-Scripting

Exercises

1. What three methods do we need to implement in a server to allow remote clients to connect to it? Name the three methods in alphabetical order. Prepend each method with "socket.".

Answer — View hints

Answer — Verify

2. How many values does the socket.accept() method return?

Answer

Answer — Verify

3. This is a scripting challenge. First, make sure that your server can accept at least four connections at once. Then, use SSH to login to the container running on port 2004 of the target VM with the credentials root:root. Run the *binary* located at /root to connect back to your server and receive the flag.

Answer

Answer — Verify

Write a Port Scanner with Python

This Learning Unit covers the following Learning Objectives:

1. Understand how a port scanner retrieves information from targets
2. Use the time module in Python to track how long a program runs for
3. Build a basic port scanner in Python with the time and socket modules
4. Understand and apply the concept of port knocking

Write a Server with Python
Testing our Client and Server

Write a Port Scanner with Python
Using the Socket Module to Create a Port S...

Resource Center

Using the Socket Module to Create a Port Scanner

In this Learning Unit, we are going to build a simple port scanner using the *socket* and *time*[1] libraries. Port scanning allows us to locate open ports that are available on a particular host. As penetration testers, we can configure our port scanner to retrieve information about the ports, assess what services are running on each port, and even guess which OS may be running on the host.

We'll begin our script by importing the relevant modules and by invoking the *time.time()* method.[2]

```
#!/usr/bin/python3
#scanner.py

import socket
import time

startTime = time.time()
```

Listing 24 - Our initial Python code

The *time.time()* method returns the time at which the Python interpreter runs the line of code it is located on. We use this method to store the initial time of the program's execution in the *startTime* variable.

At the end of the script, we'll use *time.time()* once again to store the future time in another variable (*endTime*). By subtracting the value of *endTime* from *startTime*, we can calculate how long it takes for the program to complete its total execution.

Next, we'll allow the user to specify which target they want to scan via the *input* method. We then use *socket.gethostbyname()*[3] to convert the provided hostname into an IP address.

```
#!/usr/bin/python3
#scanner.py

import socket
import time

startTime = time.time()

target = input('Please specify the host that you want to scan: ')
target_IP = socket.gethostbyname(target)
print ('Initiating Scan for host: ', target_IP)
```

Listing 25 - Allowing the user to specify a target

Alternatively, we could omit the line beginning with "target_IP", and simply allow the user to provide an IP address as input rather than a hostname.

> *When you execute your scan against the exercise machine, you may want to adjust this portion of the script.*

Next, we'll employ a *for* loop to determine which ports we want to scan on our target. Instead of the familiar *socket.connect()* method, we'll use *socket.connect_ex()*[4] to initiate the connection. *socket.connect_ex()* does the same thing as *socket.connect()*, but it returns an error indicator upon success or failure. In particular, it will return 0 when it executes successfully. This means that when 0 is returned from this method, we know that the specific port we were scanning at the time was open.

```
#!/usr/bin/python3
#scanner.py

import socket
import time

startTime = time.time()

target = input('Please specify the host that you want to scan: ')
target_IP = socket.gethostbyname(target)
print ('Initiating Scan for host: ', target_IP)

for i in range(1, 1000):
    scanner = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    conn = scanner.connect_ex((target_IP, i))
    if(conn == 0):
        print ('Port %d: OPEN' %(i))
    scanner.close()
```

Listing 26 - Scanning ports in a loop

In the *for* loop, we are iterating over ports 1 through 1000. We can easily change these values, or better yet allow the user to specify which ports they want to scan via command line arguments. We'll leave further improvements to the script as an exercise to the reader.

Finally, we simply need to run *time.time()* again and calculate how long the scan takes, as described above.

```
#!/usr/bin/python3
#scanner.py

import socket
import time

startTime = time.time()

target = input('Please specify the host that you want to scan: ')
target_IP = socket.gethostbyname(target)
print ('Initiating Scan for host: ', target_IP)

for i in range(1, 1000):
    scanner = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    conn = scanner.connect_ex((target_IP, i))
    if(conn == 0):
        print ('Port %d: OPEN' %(i))
    scanner.close()

endTime = time.time()
totalTime = endTime - startTime
print('Total Time: %s' %(totalTime))
```

Listing 27 - Calculating the scan duration

Executing the script will prompt us to input a hostname and it will conduct the port scan. The script should generate output similar to the following.

```
kali@kali:~$ python3 scanner.py
Please specify the host that you want to scan: localhost
Initiating Scan for host:  127.0.0.1
Port 22: OPEN
Port 80: OPEN
Port 8080: OPEN
Total Time: 3.3422038555145264
```

Listing 28 - Running our port scanner

In the above example, we've specified *localhost* as the target to scan. Since the script uses the *socket.gethostbyname()* method, it allows us to specify the IPv4 hostname of a machine and retrieve its IP address. As a refresher, localhost[5] usually resolves to the loopback IP address, 127.0.0.1.

We find that we have three ports listening on our local VM, and that the script took approximately 3.34 seconds to execute from start to finish. We can expand this script by providing it a means of looping through a list of hosts by creating nested *for* loops, or by having it send different kinds of packets to the hosts it connects to.

---

[1] (Python Software Foundation, 2021), https://docs.python.org/3/library/time.html ↩

[2] (Python Software Foundation, 2021), https://docs.python.org/3/library/time.html#time.time ↩

[3] (Python Software Foundation, 2021), https://docs.python.org/3/library/socket.html#socket.gethostbyname ↩

[4] (Python Software Foundation, 2021), https://docs.python.org/3/library/socket.html#socket.socket.connect_ex ↩

[5] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Localhost ↩

Resources

Some of the exercises require you to start the target machine(s) below.

Please note that the IP addresses assigned to your target machines may not match those referenced in the Module text and video.

Network-Scripting

Exercises

1. Recreate the port scanner in this section. Then, target the ports 3000 to 3999 of the target VM. In numerical order, what ports are open? Enter your answer in the following format: WWWW, XXXX, YYYY, ZZZZ

Answer

Answer | Verify

Port Knocking

Port Knocking[1] is a means by which external users can open a gated port on a machine by first connecting to a predetermined list of other ports in a specific order. Think of it like entering a PIN on a mobile device: if you input the correct numbers in the correct order, the phone will unlock. Similarly, assuming the machine's firewall has been configured in such a way, "knocking" on the correct ports in the correct order will open the gated port.

A port knocking implementation can add a small but non-negligible layer of security to a system, because it prevents port scans such as the one we have executed above. Since the port scan will iterate through a loop, it is very unlikely that the firewall will be configured to open the gated port based on the exact rules that our scan follows. An external user will need to have intimate knowledge of the system first, before being able to connect to the gated service.

Port knocking rules can be made arbitrarily complex, increasing the knowledge of the system required by the external user. Note, however, that port knocking alone is not a sufficient security system, because it relies on security by obscurity. It can be compared to single-factor authentication, representing something the user knows as the only means of authenticating. In this case, the thing the user know is the order of ports to knock.

---

[1] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Port_knocking ↩

Resources

Some of the exercises require you to start the target machine(s) below.

Please note that the IP addresses assigned to your target machines may not match those referenced in the Module text and video.

Network-Scripting

Exercises

1. Modify your port scanner so that it knocks precisely and in numerical order on the eight ports that have Pronic numbers in the range 4000 to 4999. You may need to look up the definition of "Pronic number" to determine which ports to scan. Once you have performed the port knocking sequence, use the credentials Aristotle:Lyceum to SSH to the newly opened port 2222. What is the flag on the user's desktop?

Answer — View hints

Answer — Verify

Website Interaction with Python - I

This Learning Unit has the following Learning Objectives:

1. Understand the benefits of connecting to a website programmatically
2. Communicate over HTTP with Python
3. Parse HTML responses with Python

Write a Port Scanner with Python
Port Knocking

Website Interaction with Python - I
The Transport Layer: Using the Python Sock...

The Transport Layer: Using the Python Sockets Module with HTTP

Imagine that we have identified a web server as our target and we need to learn more about the service. In this situation, we can create a script with Python that will send HTTP requests to our web server to analyze how it responds to us. This technique is very useful because we can learn more about the web server and how it communicates to our client, and to assess if it is vulnerable to any exploits that we are aware of.

In this section, we'll use the *socket* module again to create a raw TCP connection to a web server running on port 80. Since we are using a raw socket connection, we need to provide the specific HTTP request to be sent to the server as data.

The socket module operates on the equivalent of the Transport layer of the OSI or TCP/IP network reference models. We therefore must encapsulate the Application layer protocol (i.e. HTTP) data that we want to send through the Transport layer.

We'll begin our script by importing the socket module and defining a remote host and port.

```
#!/usr/bin/python3
#http-sockets.py

import socket

remote_host = "www.offensive-security.com"
remote_port = 80
```

Listing 29 - Our initial Python code

We call our script **http-sockets.py**. Note that we cannot call the script **http.py** because Python has a built-in module by the same name.

Next, let's store the HTTP request we want to make to the server inside a variable, which we'll aptly call *request*. To learn more details about the format of an HTTP request, please refer to the Web Applications Basics Module.

```
#!/usr/bin/python3
#http-sockets.py

import socket

remote_host = "www.offensive-security.com"
remote_port = 80

request = "GET / HTTP/1.1\r\nHost: www.offensive-security.com\r\n\r\n"
```

Listing 30 - Creating the request

We need to specify the exact request we want our server to send to the server, so we must include the precise syntax and newlines ("\r\n") that HTTP expects.

The next portion of the script is similar to creating a Python client. We'll initialize a socket, and then use it to connect to the server. Once we're connected, we'll use *socket.send()* to make our request, using our *request* variable.

```
#!/usr/bin/python3
#http-sockets.py

import socket

remote_host = "www.offensive-security.com"
remote_port = 80

request = "GET / HTTP/1.1\r\nHost: www.offensive-security.com\r\n\r\n"

client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect((remote_host,remote_port))
client.send(request.encode())
```

Listing 31 - Connecting to the server

Finally, we want to catch the response that the server sends us, so we'll use the *socket.recv(bytes)* method and then decode and print it.

```
#!/usr/bin/python3
#http-sockets.py

import socket

remote_host = "www.offensive-security.com"
remote_port = 80

request = "GET / HTTP/1.1\r\nHost: www.offensive-security.com\r\n\r\n"

client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect((remote_host,remote_port))
client.send(request.encode())

response = client.recv(4096)
print(response.decode())
```

Listing 32 - Receiving and displaying the response

It is important to note that the *send()* method requires a byte-like object argument, not a string. We can use the *encode()* method on a variable to convert its content to bytes, and use the *decode()* method to convert bytes to a string.

As mentioned above, the *recv()* method is used to receive the response from the server. This method requires an argument, which is the maximum number of data in bytes to be received.

Exercises

1. Recreate the script and modify it to reach www.megacorpone.com . What popular operating system distribution is www.megacorpone.com running on?

Answer

Answer | Verify

2. What branch of technology does MegaCorp One focus on? Use the HTML code from the server's response to answer this question.

Answer

Answer | Verify

The Application Layer: GET Requests with Python

Python allows us to communicate directly over HTTP instead of opening up raw network sockets. In this section we will use the *requests*[1] library to create an HTTP GET[2] request and display the response.

Let's start our script by importing the *requests* module and defining a target URL that we want to make a request to.

```
#!/usr/bin/python3
#web-client.py

import requests

url = "http://www.offensive-security.com"
```

Listing 33 - Our initial Python code

Next, we'll employ our first requests method, *requests.get()*.[3] This method will make an HTTP request to the URL provided to it as argument, and returns a *Response* object. The *Response* object can then be parsed and formatted in various ways.

```
#!/usr/bin/python3
#web-client.py

import requests

url = "http://www.offensive-security.com"

response = requests.get(url)
print(response.content.decode())
```

Listing 34 - Making a GET request

In the above listing, we use the *requests.content()*[4] method to read the content of the *Response* object in bytes. We pass the output to the *decode()* method and print it, so that we can view it in plain text.

This basic script can be modified in a varity of ways that can allow us to extract the precise information we're looking for from a web server.

During a reconaisance phase of a penetration test (for example), we might only be interested in determining the status code of the various pages. The *requests.status_code()*[5] method extracts the response status code from the *Response* object

```
#!/usr/bin/python3
#web-client.py

import requests

url = "http://www.offensive-security.com/doesnotexist.html"

response = requests.get(url)
print(response.status_code)
```

Listing 35 - Extracting the status code

As a refresher, status codes are grouped into the following classes:

- Code 100-199: Informational Responses
- Code 200-299: Successful Responses
- Code 300-399: Redirects
- Code 400-499: Client Errors
- Code 500-599: Server Errors

When the script is executed, we will obtain the status code of the URL we have specified. The URL we are requesting is **www.offensive-security.com/doesnotexist.html**. Since this page does not exist, the server will respond with a client error status code (404).

Next, let's modify the script to return only the response headers from the server. We can do this with the predictably named *requests.headers()*[6] method. We can analyze headers to get a better understanding of the web server and how it is interacting with clients.

```
#!/usr/bin/python3
#web-client.py

import requests

url = "http://www.offensive-security.com"

response = requests.get(url)
print(response.headers)
```

Listing 36 - Retrieving the response headers

In this iteration of the script, we send a request to **http://www.offensive-security.com** and print the response headers we receive from the server.

Finally, the *requests.text()*[7] method allows us to print the full response in unicode.

```
#!/usr/bin/python3
#web-client.py

import requests

url = "http://www.offensive-security.com"

response = requests.get(url)
print(response.text)
```

Listing 37 - Displaying the full response

This script makes a simple request to **http://www.offensive-security.com** and prints out the response. Remember that *requests.content()* should be used when the server is serving data in binary format, and *requests.text()* when it is serving textual data.

> *If you are uncertain about a specific website, try both!*

---

[1] (Reitz, 2021), https://docs.python-requests.org/en/master/ ↩

[2] (Mozilla, 2021), https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/GET ↩

[3] (Reitz, 2021), https://docs.python-requests.org/en/latest/api/#requests.get ↩

[4] (Reitz, 2021), https://docs.python-requests.org/en/latest/api/#requests.Response.content ↩

[5] (Reitz, 2021), https://docs.python-requests.org/en/latest/api/#requests.Response.status_code ↩

[6] (Reitz, 2021), https://docs.python-requests.org/en/latest/api/#requests.Response.headers ↩

[7] (Reitz, 2021), https://docs.python-requests.org/en/latest/api/#requests.Response.text ↩

The exercises below require you to interact with the target webserver using Python. You will not be able to reach the required pages with your regular browser!

Resources

Some of the exercises require you to start the target machine(s) below.

Please note that the IP addresses assigned to your target machines may not match those referenced in the Module text and video.

Network-Scripting

Exercises

1. Why can't we call our Python HTTP client "http.py"?

```
A. Because Python3 doesn't work with http.
B. Because Python3 has another module called http.py.
C. Because Python3 cannot execute multiple web-clients at the same time.
```

Answer

Answer — Verify

2. Write as Python script to do a HTTP GET request on port 8080 of the target VM and get the webpage content. What is the flag on the index.html page?

Answer

Answer — Verify

3. Write as Python script to do a HTTP GET request on port 8080 at /1.html. This site will give you the first character of the flag. The directories /2.html to /50.html will give you the remaining characters. What is the complete flag?

Answer

Answer — Verify

Parsing HTML

If you've used the *requests.content()* or *requests.text()* methods above, you've likely found that the response can contain a significant amount of HTML data. Sometimes, we may want to get specific information from a site without all the clutter involved with a full response. *Web Scraping* is a process of sweeping information that is contained on a webpage and extracting the information we're interested in. As penetration testers, it is sometimes easier to retrieve the data we are looking for by writing a script than trying to look for the data manually from the website.

Lets start by using the *urllib3*[1] module to send an HTTP request that will obtain the data from the webpage. The *urllib3* library is used by the *requests* library; in this case, we will import it explicitly just so that we can become familiar with different syntax.

```
#!/usr/bin/python3
#parse.py

import urllib3

http = urllib3.PoolManager()

url = 'http://www.megacorpone.com'

response = http.request('GET', url)
print(response.data.decode('utf-8'))
```

Listing 38 - Our initial urllib3 code

Here, we create a variable called *http* that calls the *urllib3* module, and we use the *PoolManager*[2] method to sort the unordered results. The *url* variable is then used to call the website we are sending an HTTP request to. We can change the *url* variable to specify other domains or IP addresses.

The above script will print the output from the target website onto the terminal, but it will display as raw HTML code. Let's go ahead and execute the script, and pass the output to the **head** command to snip it.

```
kali@kali:~$ python3 parse.py | head -n 20
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <meta name="description" content="">
    <meta name="author" content="">
    <link rel="shortcut icon" href="assets/ico/favicon.ico">

    <title>MegaCorp One - Nanotechnology Is the Future</title>

    <!-- Bootstrap core CSS -->
    <link href="assets/css/bootstrap.css" rel="stylesheet">

    <!-- Custom styles for this template -->
    <link href="assets/css/style.css" rel="stylesheet">
    <link href="assets/css/font-awesome.min.css" rel="stylesheet">
```

Listing 39 - The first 20 lines of output

To make the data more easily readable, we can use another module called *BeautifulSoup*.[3] This module takes the raw HTML and XML files from urlopen and pulls the data to help parse the information we have retrieved from the webpage.

We can include our new module by modifying our script as follows.

```
#!/usr/bin/python3
#parse.py

import urllib3
from urllib.request import urlopen
from bs4 import BeautifulSoup

url = urlopen("http://www.megacorpone.com")

page = url.read()
soup = BeautifulSoup(page, features="html.parser")

print(soup)
```

Listing 40 - Using the BeautifulSoup module

In the above listing, we import two libraries from *urllib3* and *bs4*. The *urlopen* library allows us to retrieve the raw data returned by the server. The *beautifulsoup* library is responsible for the actual parsing of the output.

```
<!DOCTYPE html>

<html lang="en">
<head>
<meta charset="utf-8"/>
<meta content="IE=edge" http-equiv="X-UA-Compatible"/>
<meta content="width=device-width, initial-scale=1" name="viewport"/>
<meta content="" name="description"/>
<meta content="" name="author"/>
<link href="assets/ico/favicon.ico" rel="shortcut icon"/>
<title>MegaCorp One - Nanotechnology Is the Future</title>
<!-- Bootstrap core CSS -->
<link href="assets/css/bootstrap.css" rel="stylesheet"/>
<!-- Custom styles for this template -->
<link href="assets/css/style.css" rel="stylesheet"/>
<link href="assets/css/font-awesome.min.css" rel="stylesheet"/>
<!-- Just for debugging purposes. Don't actually copy this line! -->
<!--[if lt IE 9]><script src="../../assets/js/ie8-responsive-file-warning.js"></script>
<![endif]-->
<!-- HTML5 shim and Respond.js IE8 support of HTML5 elements and media queries -->
<!--[if lt IE 9]>
```

Listing 41 - Our script output using BeautifulSoup

While the output of the above listing isn't any more readable than the output provided by the original script, BeautifulSoup has many methods we can use to narrow, sort, and display our desired text. For example, the *text* method will print only the textual content of the web page. The following exercises allow you to flex your ability to gather specific data from a web server.

> *These exercises may take a little bit of extra research, so make sure to read up on the relevant library documentation if you get stuck.*

---

[1] (Petrov, 2021), https://urllib3.readthedocs.io/en/stable/ ↩

[2] (Petrov, 2021), https://urllib3.readthedocs.io/en/stable/reference/urllib3.poolmanager.html ↩

[3] (Richardson, 2020), https://www.crummy.com/software/BeautifulSoup/bs4/doc/ ↩

Resources

Some of the exercises require you to start the target machine(s) below.

Please note that the IP addresses assigned to your target machines may not match those referenced in the Module text and video.

Network-Scripting

Exercises

1. The website on port 8080 of the target VM has multiple pages under the directory /crawling. Use your Python skills to GET the content on all the pages and find the flag.

Answer

Answer | Verify

2. Visit the website on port 8080 of the target VM under the /table directory. The table found on the page contains the flag, but each row contains a different letter. Use Python to make a request to this page and parse the response.

Answer

Answer | Verify

Website Interaction with Python - II

This Learning Unit has the following Learning Objectives:

1. Make HTTP POST requests using Python
2. Analyze HTTP request headers with Python
3. Become comfortable programming repetitive web-based actions

Website Interaction with Python - I
Parsing HTML

Website Interaction with Python - II
POST Requests and Parameters with Python

POST Requests and Parameters with Python

The requests module also allows us to send data to a server via a *POST*[1] request. In a POST request, the data that is sent to the server is stored in the request body of an HTTP request. This is in contrast to a GET request, where data is sent directly via a URL. A common use-case for POST requests employed by many websites are web forms, such as those used when subscribing to a site.

Let's examine a script that will make a POST request and then return the response from the server.

```
#!/usr/bin/python3
#web-client2.py

import requests

url = 'http://www.offensive-security.com'

info = {'check-key': 'check-value'}
post = requests.post(url, data = info)
print(post.text)
```

Listing 42 - Our POST request Python script

The script in the above listing POSTs a request to **www.offensive-security.com** and sends the data contained within the *info* variable. Once the data has been submitted, the response will be printed out in text.

There are other HTTP methods that we can use in our Python scripts such as *PUT*,[2] *DELETE*,[3] *HEAD*,[4] and *OPTIONS*[5] to interact with the web server. We encourage you to experiment with writing scripts for each of these.

---

[1] (Mozilla, 2021), https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/POST ↩

[2] (Mozilla, 2021), https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/PUT ↩

[3] (Mozilla, 2021), https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/DELETE ↩

[4] (Mozilla, 2021), https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/HEAD ↩

[5] (Mozilla, 2021), https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/OPTIONS ↩

Resources

Some of the exercises require you to start the target machine(s) below.

Please note that the IP addresses assigned to your target machines may not match those referenced in the Module text and video.

Network-Scripting

Exercises

1. The page at port 8080 of the target server called /basic-post only accepts POST requests. Make any POST request to the page to receive the flag.

Answer

Answer | Verify

2. You can authenticate to page at port 8080 of the target server called /login-1 with the username 'thobbes' and the password 'leviathan'. Make a POST request to the page with the above credentials to get the flag.

Answer

Answer | Verify

3. You can authenticate to the page at port 8080 of the target server called /login-2 with the username 'rdescartes' and the password 'discourse'... however, the password is followed by the five characters ! @ # % & in some unknown order. For example, the password might be discourse#!@&%, or it might be discourse%&@!#. Use Python to iterate through all possible POST requests to determine the password, and login to get the flag.

Answer

Answer | Verify

4. The page on port 8080 of the target server called /bijection accepts an integer value that corresponds to the letter position of the flag. For example:

- /bijection?index=0 will return the character 'O'
- /bijection?index=1 will return the character 'S'
- /bijection?index=2 will return the character '{'

Use your Python skills to create a script that will get the entire flag. Note, the page will only accept POST requests!

Answer

Answer | Verify

Request Headers and Non-Text-based Content

An HTTP header allows the client and server to pass additional information in the request or the response. The header consists of a case-insensitive name followed by a semi-colon (:), and then a value. A *request header* contains detailed information about the resource that is being queried. A *response header* holds additional information about the response. For example, a response header might include the location of the server.

One of the most important headers for us to learn about is the *Content-Type*[1] header. This header is used to indicate the original media type of the resource.

In the response, the Content-Type header tells the client what type of content type will be displayed.

For us to identify the content type from the server, we need to send a specific request to the server and have the response print the output of the HTTP Headers. The following script makes a GET request to **www.offensive-security.com** and prints the response's HTTP headers.

```
#!/usr/bin/python3
#headers.py

import requests

url = "http://www.offensive-security.com"
response = requests.get(url)
print(response.headers)
```

Listing 43 - Displaying the response headers

The server will respond with the following output.

```
{'Server': 'Sucuri/Cloudproxy', 'Date': 'Wed, 09 Jun 2021 02:32:42 GMT', 'Content-
Type': 'text/html; charset=UTF-8', 'Content-Length': '14838', 'Connection': 'keep-
alive', 'X-Sucuri-ID': '17005', 'X-XSS-Protection': '1; mode=block', 'X-Frame-Options':
'SAMEORIGIN', 'X-Content-Type-Options': 'nosniff', 'Strict-Transport-Security': 'max-
age=31536000; includeSubdomains; preload', 'Content-Security-Policy': 'upgrade-
insecure-requests;', 'Link': '<https://www.offensive-security.com/>; rel=shortlink',
'Vary': 'Accept-Encoding,User-Agent', 'Content-Encoding': 'gzip', 'X-Sucuri-Cache':
'HIT'}
```

Listing 44 - Response header output

In the above listing, the *'Content-Type': 'text/html; charset=UTF-8'* section indicates that the page will display the information in text/html format. Keep in mind that the content-type will always change depending on the request sent to the server.

---

[1] (Mozilla, 2021), https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Type ↩

Resources

Some of the exercises require you to start the target machine(s) below.

Please note that the IP addresses assigned to your target machines may not match those referenced in the Module text and video.

Network-Scripting

Exercises

1. The directory at port 8080 of the target server called /headers has ten subpages called /headers/1 through /headers/10. Each page has a custom header called "Flag" that contains a portion of the flag. Use Python to piece together all the components of the flag.

Answer

Answer | Verify

2. The page at port 8080 of the target server called /object returns a binary that when run, prints out the flag. Use python to save the binary and then run it to get the flag.

Answer

Answer | Verify

3. The page at port 8080 of the target server called /about.html contains a list of 30 employees, their email addresses, and their favorite colors. Only one of these users can login to the page at /login-3. Use Python to determine which user has a valid account by analyzing the responses to your requests. What is the first name of the valid user?

Answer

Answer | Verify

4. The valid account's password is their favorite colleague's first name and their boss's favorite color twice in a row. For example, if their best friend is Jacob and their boss is Carly, then the password is JacobOrangeJacobOrange. Use this knowledge to authenticate to the website to get the flag.

Answer

Answer | Verify

Capturing and Sending Packets with Scapy

This Learning Unit covers the following Learning Objectives:

1. Understand why text-based packet manipulation can be a powerful tool
2. Capture network traffic using Scapy
3. Send and receive packets using Scapy

Website Interaction with Python - II
Request Headers and Non-Text-based Cont...

Capturing and Sending Packets with Scapy
Introduction to Scapy

Introduction to Scapy

Scapy[1] is a flexible Python-based packet manipulation program. The purpose of using Scapy is mainly for two things: to sends packets and receive answers.

As penetration testers, we can use Scapy to craft custom packets that can be sent through a variety of protocols. Scripting with Scapy will give us the ability to define a set of packets, how and when to send them, and how to analyze the responses.

Many penetration testers use Scapy to conduct certain tasks such as network discovery, network scanning, packet capture, and much more. The features in Scapy can simulate some of the common tools that we use in our engagments. Common tools that we could use to replace with Scapy are *hping*,[2] *arpspoof*,[3] *arp-sk*,[4] *p0f*,[5] *tcpdump*,[6] *tshark*,[7] and even some parts of *nmap*.[8]

---

[1] (Biondi, 2021), https://scapy.net/ ↩

[2] (Sanfilippo, 2006), http://www.hping.org/ ↩

[3] (die.net, 2021), https://linux.die.net/man/8/arpspoof ↩

[4] (manned.org, 2021), https://manned.org/arp-sk/99e329e1 ↩

[5] (Zalewski, 2014), https://lcamtuf.coredump.cx/p0f3/ ↩

[6] (The Tcpdump Group, 2021), https://www.tcpdump.org/ ↩

[7] (Wireshark Foundation, 2021), https://www.wireshark.org/docs/man-pages/tshark.html ↩

[8] (Nmap.org, 2021), https://nmap.org/ ↩

Scapy Commands

Scapy provides a variety of commands to help us analyze the packets we are capturing. There are two ways we can use Scapy depending on the situation we are in. We can use a terminal window or we can import the library into a Python script. For this Learning Unit, we'll invoke Scapy from the command line with the **scapy** command.

```
kali@kali:~$ sudo scapy
Password:
INFO: Can't import PyX. Won't be able to use psdump() or pdfdump().

                     aSPY//YASa
             apyyyyCY//////////YCa       |
            sY//////YSpcs  scpCY//Pp     | Welcome to Scapy
 ayp ayyyyyyySCP//Pp           syY//C    | Version 2.4.4
 AYAsAYYYYYYYY///Ps              cY//S   |
         pCCCCY//p          cSSps y//Y   | https://github.com/secdev/scapy
         SPPPP///a          pP///AC//Y   |
              A//A            cyP////C   | Have fun!
              p///Ac            sC///a   |
              P////YCpc           A//A   | Craft me if you can.
       scccccp///pSP///p          p//Y   |                 -- IPv6 layer
      sY/////////y  caa           S//P   |
       cayCyayP//Ya              pY/Ya
        sY/PsY////YCc           aC//Yp
         sc  sccaCY//PCypaapyCP//YSs
                  spCPY//////YPSps
                       ccaacs
                                       using IPython 7.20.0
>>>
```

Listing 45 - Launching Scapy from the command line

Let's examine some of the most common Scapy commnads.

- **ls()**: Displays all of the protocols supported by Scapy
- **explore()**: Displays all protocols in a clear GUI
- **lsc()**: Displays a list of commands and functions that are supported.
- **conf**: Displays our configuration options in Scapy
- **help()**: Gets help on a specific Scapy command. For example, we can run **help** on the **sniff** command.

```
>>> help(sniff)

Help on function sniff in module scapy.sendrecv:

sniff(*args, **kwargs)
    Sniff packets and return a list of packets.

    Args:
        count: number of packets to capture. 0 means infinity.
        store: whether to store sniffed packets or discard them
        prn: function to apply to each packet. If something is returned, it
             is displayed.
```

Listing 46 - Getting help for the sniff command

Capturing Packets with Scapy

Recall that packet capture usually requires elevated permissions. Therefore we need to make sure we provide Scapy with *sudo* privileges before running it, to allow it to sniff for packets.

To begin capturing packets with Scapy, we are going to use the *sniff()* function to help us capture all network traffic from our system. The *sniff()* function has a few parameters that we can use to filter out the network traffic.

The *iface* parameter allows us to specify network interface we want to capture packets.

The *count* parameter captures the supplied number of packets. If we remove this option or set it to 0, Scapy will continue to sniff for packets until we stop the program.

The *prn* parameter prints the results we are sniffing. We need to use a call-back function name to print the results. For example, the *x:x.summary()* syntax names our capture "x", and then uses the *summary()* method to print out the high level results of the capture on the terminal screen.

The *filter* parameter will allows us to capture only packets that are interesting to us.

Let's apply these options, capture 10 TCP packets on the eth0 interface, and print them to the terminal window.

```
>>> sniff(iface="eth0", count=10, prn = lambda x: x.summary(), filter="tcp")
Ether / IP / TCP 192.168.60.200:5901 > 192.168.47.4:53910 PA / Raw
Ether / IP / TCP 192.168.60.200:5901 > 192.168.47.4:53910 PA / Raw
Ether / IP / TCP 192.168.47.4:53910 > 192.168.60.200:5901 A
Ether / IP / TCP 192.168.60.200:5901 > 192.168.47.4:53910 A / Raw
Ether / IP / TCP 192.168.60.200:5901 > 192.168.47.4:53910 PA / Raw
Ether / IP / TCP 192.168.60.200:5901 > 192.168.47.4:53910 A / Raw
Ether / IP / TCP 192.168.60.200:5901 > 192.168.47.4:53910 PA / Raw
Ether / IP / TCP 192.168.47.4:53910 > 192.168.60.200:5901 A
Ether / IP / TCP 192.168.47.4:53910 > 192.168.60.200:5901 A
Ether / IP / TCP 192.168.47.4:53910 > 192.168.60.200:5901 A
<Sniffed: TCP:10 UDP:0 ICMP:0 Other:0>
```

Listing 47 - Sniffing traffic with scapy

> *In the above listing, we have set our network interface to eth0. If you are using the VPN to connect to the labs, make sure that you use the correct interface name provided by the VPN connection.*

We can also set a particular sniff command to a variable. Below, we set the *pkts* variable to the summary of the desired capture, and then print it out with Python's standard *print* function.

```
>>> pkts = sniff(iface="eth0", count=10, prn = lambda x: x.summary(), filter="tcp")
Ether / IP / TCP 192.168.60.200:5901 > 192.168.47.4:53910 PA / Raw
Ether / IP / TCP 192.168.60.200:5901 > 192.168.47.4:53910 PA / Raw
Ether / IP / TCP 192.168.47.4:53910 > 192.168.60.200:5901 A
Ether / IP / TCP 192.168.60.200:5901 > 192.168.47.4:53910 A / Raw
Ether / IP / TCP 192.168.60.200:5901 > 192.168.47.4:53910 PA / Raw
Ether / IP / TCP 192.168.60.200:5901 > 192.168.47.4:53910 A / Raw
Ether / IP / TCP 192.168.47.4:53910 > 192.168.60.200:5901 A
Ether / IP / TCP 192.168.47.4:53910 > 192.168.60.200:5901 A
Ether / IP / TCP 192.168.60.200:5901 > 192.168.47.4:53910 PA / Raw

>>> print(pkts)
<Sniffed: TCP:10 UDP:0 ICMP:0 Other:0>
```

Listing 48 - Sniffing with a variable

To show the details of a single packet, we can use *x.show()* instead of *x.summary()* as the value to the *prn* parameter.

```
>>> sniff(iface="eth0", count=1, prn = lambda x: x.show(), filter="tcp")
###[ Ethernet ]###
  dst= 00:50:56:bf:0a:ef
  src= 00:50:56:bf:5b:69
  type= IPv4
###[ IP ]###
     version= 4
     ihl= 5
     tos= 0x0
     len= 7292
     id= 63715
     flags= DF
     frag= 0
     ttl= 64
     proto= tcp
     chksum= 0x387b
     src= 192.168.60.200
     dst= 192.168.47.4
     \options\
###[ TCP ]###
        sport= 5901
        dport= 53910
        seq= 1791883037
        ack= 3124877518
        dataofs= 8
        reserved= 0
        flags= PA
        window= 501
        chksum= 0x98c
        urgptr= 0
        options= [('NOP', None), ('NOP', None), ('Timestamp', (4108292854,
3224841510))]
###[ Raw ]###
           load= '\x17\x03\x03@\x18\x00\x00\x00\x00\x00\x00\x1c\xf5v0\'\x10
R\xc1\xe2\xe4\xdf~\x95M\x96K
...
```

Listing 49 - Displaying the details of a packet

Saving Packets with Scapy

To save the packets we collected, we can use the *wrpcap* command to save the information into a **.pcap** file.

```
>>> pkts = sniff(iface="eth0", count=10, prn = lambda x: x.summary(), filter="tcp")
Ether / IP / TCP 192.168.60.200:5901 > 192.168.47.4:53910 PA / Raw
Ether / IP / TCP 192.168.60.200:5901 > 192.168.47.4:53910 PA / Raw
Ether / IP / TCP 192.168.47.4:53910 > 192.168.60.200:5901 A
Ether / IP / TCP 192.168.60.200:5901 > 192.168.47.4:53910 A / Raw
Ether / IP / TCP 192.168.60.200:5901 > 192.168.47.4:53910 PA / Raw
Ether / IP / TCP 192.168.60.200:5901 > 192.168.47.4:53910 A / Raw
Ether / IP / TCP 192.168.60.200:5901 > 192.168.47.4:53910 PA / Raw
Ether / IP / TCP 192.168.47.4:53910 > 192.168.60.200:5901 A
Ether / IP / TCP 192.168.47.4:53910 > 192.168.60.200:5901 A
Ether / IP / TCP 192.168.60.200:5901 > 192.168.47.4:53910 PA / Raw

>>> wrpcap('sniffed.pcap', pkts)
```

Listing 50 - Saving the packets

This will allow us to use tools such as tcpdump or Wireshark to further analyze the network traffic that was saved into the **.pcap** file from Scapy.

```
kali@kali:~$ sudo tcpdump -r sniffed.pcap
reading from file sniffed.pcap, link-type EN10MB (Ethernet), snapshot length 65535
12:15:35.367793 IP 192.168.60.200.5901 > 192.168.47.4.53910: Flags [P.], seq
1806351782:1806359022, ack 3124922719, win 501, options [nop,nop,TS val 4109078037 ecr
3225626688], length 7240
12:15:35.367815 IP 192.168.60.200.5901 > 192.168.47.4.53910: Flags [P.], seq
7240:14480, ack 1, win 501, options [nop,nop,TS val 4109078037 ecr 3225626688], length
7240
12:15:35.368652 IP 192.168.47.4.53910 > 192.168.60.200.5901: Flags [.], ack 14480, win
4121, options [nop,nop,TS val 3225626737 ecr 4109078037], length 0
12:15:35.368667 IP 192.168.60.200.5901 > 192.168.47.4.53910: Flags [.], seq
14480:15928, ack 1, win 501, options [nop,nop,TS val 4109078037 ecr 3225626737], length
1448
12:15:35.370214 IP 192.168.60.200.5901 > 192.168.47.4.53910: Flags [P.], seq
15928:30408, ack 1, win 501, options [nop,nop,TS val 4109078039 ecr 3225626737], length
14480
12:15:35.370227 IP 192.168.60.200.5901 > 192.168.47.4.53910: Flags [.], seq
30408:31856, ack 1, win 501, options [nop,nop,TS val 4109078039 ecr 3225626737], length
1448
12:15:35.370384 IP 192.168.60.200.5901 > 192.168.47.4.53910: Flags [P.], seq
31856:40544, ack 1, win 501, options [nop,nop,TS val 4109078039 ecr 3225626737], length
8688
12:15:35.370989 IP 192.168.47.4.53910 > 192.168.60.200.5901: Flags [.], ack 30408, win
4121, options [nop,nop,TS val 3225626740 ecr 4109078037], length 0
12:15:35.371060 IP 192.168.47.4.53910 > 192.168.60.200.5901: Flags [.], ack 40544, win
4068, options [nop,nop,TS val 3225626740 ecr 4109078039], length 0
12:15:35.378232 IP 192.168.60.200.5901 > 192.168.47.4.53910: Flags [P.], seq
40544:56472, ack 1, win 501, options [nop,nop,TS val 4109078047 ecr 3225626740], length
15928
```

Listing 51 - Analyzing the traffic

Exercises

1. What function can be used in Scapy to sniff traffic?

Answer

Answer | Verify

2. What parameter to sniff() would you need to provide to capture a total of 3000 packets from the wire?

Answer

Answer | Verify

Methods for Sending and Receiving Packets with Scapy

Scapy can also be used to send and receive packets. Depending on the packet or group of packets we want to send to a target, we should expect a response back. Scapy has a few different types of send and receive methods.

- **send**
  - *sendp()*: Sends Layer 2 packets.
  - *send()*: Sends Layer 3 Packets.
- **sr**
  - *sr()*: This method sends and receives packets at layer 3. If this method is set then it will return answered and unanswered packets.
  - *sr1()*: This method sends packets at layer 3 but it will only return the first answer or sent packets.
  - *srp()*: This method sends and receives packets at layer 2. If this method is set then it will return answered and unanswered packets.
  - *srp1()*: This method sends packets at layer 2 but it will only return the first answer or sent packets.

Sending a Packet with Scapy

To send a packet to our target we first need to create the packet. To create the packet we will use the following syntax:

```
>>> packet = IP (dst = "offensive-security.com")
```

Listing 52 - Creating a packet

With the above command, we have created an IP packet that will be sent to **www.offensive-security.com**. Let's make our packet slightly more sophisticated.

```
>>> packet = IP (dst = "offensive-security.com")/ICMP()/"Ping Offsec"
```

Listing 53 - Improving our packet

This additional syntax creates a packet that will be sent to the IP Layer and it will specifically be going through the ICMP protocol. A raw payload is included to make sure the destination receives the message we included. To verify the information and options we set, we run the following.

```
>>> packet.show
<bound method Packet.show of <IP  frag=0 proto=icmp dst=Net('offensive-security.com') |
<ICMP  |<Raw  load='Ping Offsec' |>>>>
```

Listing 54 - Reviewing the packet contents

This command will display the information contained in the packet. Once we have verified our options for the packet, we can now send it to our target.

```
>>> send(packet)
.
Sent 1 packets.
```

Listing 55 - Sending the packet

In the above listing, we receive a response from Scapy that the packet has been sent. However, if we want to view the output of the response, we need to use another terminal window to catch the packet.

Let's continue to use Scapy to capture the packet response by using the following syntax in a new terminal window.

```
>>> capture=sniff(filter="icmp", iface="eth0", count=2, prn=lambda x:x.summary())
```

Listing 56 - Capturing the packet

Then, we can resend the packet with **send(packet)** in the original terminal. Our listening terminal will produce the following output.

```
>>> capture=sniff(filter="icmp", iface="eth0", count=2, prn=lambda x:x.summary())
Ether / IP / ICMP 192.168.60.200 > 192.124.249.5 echo-request 0 / Raw
```

Listing 57 - Receiving the packet

A spoofed packet disguises its source IP address, so that the receiver believes that the packet originates from a different source. Packet spoofing can be used by attackers to misrepresent where they are, or to impersonate other users.

Sending and Receiving a Response from Scapy

Earlier, we introduced the *sr()* send and response method. Using this method is almost the same as using *send()*. The only difference between using *send()* and *sr()* is that we will be able to receive a response immediately with *sr()*, instead of opening up a new capture.

We'll modify our previous syntax to utilize the *sr()* method.

```
>>> packet = IP (dst="offensive-security.com")/ICMP()/"Hello Offsec"

>>> sr(packet)
Begin emission:
Finished sending 1 packets.
.....................................................^c
Received 186 packets, got 0 answers, remaining 1 packets
(<Results: TCP:0 UDP:0 ICMP:0 Other:0>,
 <Unanswered: TCP:0 UDP:0 ICMP:1 Other:0>)
```

Listing 58 - Using sr() to send and receive a packet

Scapy was able to finish sending the packet and it was able to obtain 186 packets until it exited when we manually sent an interrupt (ctrl-c).

The following exercises will test your Scapy skills. There is a server running on the target VM on port 9876. The server is listening for packets coming from Scapy. When it is provided with the correct inputs, it will write a flag onto the target's file system.

To access the file system, use the *sftp* command with 'sftp offensive@TARGET-IP'. The password to logon is 'security'. Once you are connected, use *cd* to move into the *file-transfers* directory.

In another terminal, run Scapy with elevated permissions and then complete each exercise below. For each exercise, after you have sent the correct input via scapy, run *ls* on the target server to retrieve the corresponding flag and enter it as your answer.

Resources

Some of the exercises require you to start the target machine(s) below.

Please note that the IP addresses assigned to your target machines may not match those referenced in the Module text and video.

Network-Scripting

Exercises

Complete the following exercises with the in-browser Kali client.

1. Send an IP packet to the server listening on port 9876. The IP packet must arrive at the target server with a TTL of 99, and does not need to contain any data.

Answer — View hints

Answer — Verify

2. Send an ICMP packet to the server listening on port 9876. The ICMP packet should contain the data "Hello, Offsec!".

Answer

Answer — Verify

3. Send a UDP packet to the server listening on port 9876.

Answer

Answer — Verify

4. Send a TCP ACK packet to the server listening on port 9876, with the source port of 22.

Answer

Answer — Verify

Data Manipulation in Python

In this Topic, we will cover the following Learning Units:

- Refresh our Basic Knowledge of Python Data
- Understand Sets, Lists, and Dictionaries
- Understand Different Base Representations
- Learn to Convert and Display Data Objects
- Manipulate Binary Large Objects
- Understand User Defined Data Structures
- Understand Data Structures as Records

Each learner moves at their own pace, but this Topic should take approximately 7.5 hours to complete.

Python Data Basics

In any computer system, the basic forms of data, bits and bytes, are low level constructs. Python provides us with the ability to work in higher level constructs such as *strings*, *integers*, *boolean*, and *floating point*.

In this Learning Unit, we'll refresh our knowledge and skills of basic data types. We'll cover the following Learning Objectives:

- Refresh our knowledge of strings
- Refresh our knowledge of integers
- Refresh our knowledge of floating points
- Explore complex numbers
- Refresh our knowledge of booleans
- Understand Python's bytes

This Learning Unit should take about 90 minutes to complete.

For this Topic, we'll mostly use the interactive console, which can be started as follows.

```
kali@kali:~$ python3
Python 3.9.9 (main, Jan 12 2022, 16:10:51)
[GCC 11.2.0] on linux
Type "help", "copyright", "credits", or "license" for more information.
>>>
```

Listing 1 - Starting Python

Working with Strings

One of the most commonly used data types in Python is the string object, which is zero or more characters concatenated together. A string may be any length and may include spaces and special characters, such as a new line. We can store it as a variable or use it directly as a quoted string. Because much of the basic manipulation of strings is explained in the introductory Python scripting Topic, we'll briefly cover the functions we use to manipulate strings as a refresher.

```
>>> astring = "The world is not "

>>> bstring = "flat."

>>> astring = astring+bstring

>>> print(astring.upper())
THE WORLD IS NOT FLAT.

>>> print(astring.lower())
the world is not flat.
```

Listing 2 - Manipulating String Case

This code starts by setting *astring* and *bstring* to parts of a sentence and then concatenating them with the plus (+) operator. The *upper()* and *lower()* functions are used to change the case of the sentence.

When dealing with multiple items we can hold them in a single data structure and access them using indexes. We call this a list or an array of items. A character string can be manipulated as an array of characters. We can find its length with the *len* function and index into it with "0" as the first character. Let's pick out the first and last character, and then the word "flat".

```
>>> print(len(astring))
22

>>> print(astring[0])
T

>>> print(string[21])
.

>>> print(astring[-1])
.

>>> print(astring[17:21])
flat
```

Listing 3 - Strings as Character Arrays

We've used the character index starting from the beginning, but we've also used the character index as a negative value to start from the end, with "-1" being the last character in the string.

We can refer to the part of the string that we are manipulating as a *substring* or a slice. A substring has an inclusive start point and an exclusive end point. In other words, the string we are taking out is up to, but not including, the end point.

We can't directly assign a string element or substring, but we can replace part of our string, as shown below.

```
>>> string[13:21]="spherical"
Traceback (most recent call last)
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment

>>> astring=astring.replace("not flat","spherical")

>>> print(astring)
The world is spherical.
```

Listing 4 - Substring Replacement

We can also use the asterisk (*) as a multiplier for character assignments.

```
>>> a = "A" * 6

>>> print(a)
AAAAAA
```

Listing 5 - Repeating Characters

Single or double quotes can both be used to denote strings. Having multiple string delimiters is useful when we want to embed a string within a string. Alternatively, we can achieve the same by prefixing with the backslash (\), also known as an *escape prefix*.

```
>>> astring = 'The world is "almost" spherical.'

>>> print(astring)
The world is "almost" spherical.

>>> astring="The world is \"almost\" spherical."
The world is "almost" spherical.
```

Listing 6 - Embedding Delimeters

There are other useful escape sequences, such as *newline* (\n) and *tab* (\t), which can be used as shown below.

```
>>> pstring = "Mercury\tVenus\tEarth\nMars\tJupiter\tSaturn\n"

>>> print(astring)
Mercury Venus   Earth
Mars    Jupiter Saturn
```

Listing 7 - Special Characters

Now that we've had a quick refresher on strings, let's move on to integers.

Working with Integers

We'll set an integer variable in Python (which supports unlimited length integers) using an assignment statement. An integer can be positive or negative and we can use the normal arithmetic operators. Examples of this are shown below.

```
>>> inum1 = 164
>>> inum2 = 37
>>> inum3 = inum1 * inum2
>>> print(inum3)
6068

>>> inum3 = inum3 + 1
>>> print(inum3)
6069

>>> type(inum3)
<class 'int'>

>>> inum4 = inum3 / 23
>>> print(inum4)
263

>>> type(inum4)
<class 'int'>

>>> inum5 = inum3 % 23
>>> print(inum5)
20

>>> inum4 += 1
>>> print(inum4)
264
```

Listing 8 - Integers in Python

The example above shows the way in which the integer division operator (/) results in an integer answer, with any remainder being discarded. The *modulo* (%) operator will provide the remainder if needed. We can also use the *type* function to identify the type of variable we are dealing with. The listing shows the use of the "+=" operator as shorthand for adding a fixed value to a variable.

There are many mathematical functions we can use with integers. Two we'll be using later in the Topic are *int()* to convert a string to an integer, and *str()* to convert an integer to a string. This is particularly useful when we're reading a number from the console as we can input a string and then convert it to a number using the *int()* function, or we can directly convert the input. Both ways are shown below.

```
>>> astring = input("Enter a number: ")
Enter a number: 12

>>> type(astring)
<class 'str'>

>>> inum = int(astring)
>>> print(inum)
12

>>> inum = int(input("Enter a number: "))
Enter a number: 12

>>> print(inum)
12

>>> type(inum)
<class 'int'>

>>> astring = str(inum+1)+" green bottles"
>>> print(astring)
13 green bottles
```

Listing 9 - Converting Between Integer and String

Let's now move on to floating point numbers.

Working with Floating Points

Floating point variables are numbers which have numbers both in front of and behind the decimal point, and is a decimal (i.e., using the values 0-9) form of representing fractions. For example, if we divide three by 2 we get 1.5, denoting one and a half. We can set a floating point variable in Python by including a decimal point in the assignment with either zero or nothing after it. As long as the decimal point is used, it's considered to be a floating point number.

```
>>> fnum1 = 12.75
>>> type(fnum1)
<class 'float'>

>>> fnum2=67.
>>> type(fnum2)
<class 'float'>
```

Listing 10 - Setting a Floating Point Value

The standard mathematical operators work as we'd expect and return floating point results. The integer division and modulo operators work on floating point numbers, returning floating point values.

```
>>> fnum3 = fnum2 / fnum1
>>> print(fnum3)
5.254901960784314

>>> type(fnum3)
<class 'float'>

>>> fnum4 = fnum2 / fnum1
>>> print(fnum4)
5.0

>>> type(fnum4)
<class 'float'>

>>> fnum5 = fnum2 % fnum1
>>> print(fnum5)
3.25
```

Listing 11 - Once a Float, Always a Float

The issue we run into with floating point numbers is due to the way they are internally stored by Python. Let's do a simple addition of two floating point numbers and print the result.

```
>>> fnum1 = 0.1
>>> fnum2 = 0.2
>>> fnum3 = fnum1 + fnum2
>>> print(fnum3)
0.30000000000000004

>>> fnum3==0.3
False
```

Listing 12 - A Floating Anomaly

We may not want to have a floating point number shown in its completely accurate form, and in some cases we just can't do that. Consider when we divide 5 by 3. The value is 1.66666.. but the fractional digit 6 continues to infinity. We will often decide we want, say a 4 digit *level of precision*,[1] and then represent the number in that form. We could just cut it off, and call 5 divided by 3 to be 1.666, or we might *round*[2] the last digit by adjusting it based on the digit following it. If it's 5 or greater, we can increase our last digit by 1. This would make 5 divided by 3 rounded to 4 digits of precision as 1.667.

When we're working with computer-based numbers, we will often define the level of precision in bits rather than decimal display values. Single precision in this context means what we can fit in one 32 bit word, and double precision means what we can fit in two 32 bit words.

```
>>> round(fnum3,1)
0.3
```

Listing 13 - Rounding Floats

We can convert between integer and floating point easily enough and sometimes this is done automatically.

```
>>> inum1 = 164
>>> inum2 = 37
>>> fnum1 = inum1 / inum2
>>> print(fnum1)
4.4324324324324325

>>> type(fnum1)
<class 'float'>

>>> inum3 = int(fnum1)
>>> print(inum3)
4

>>> type(inum3)
<class 'int'>

>>> fnum2 = float(inum3)
>>> print(fnum2)
4.0
```

Listing 14 - Converting Between Floats and Integers

If we display a very small floating point value, we'll get a slightly different result with larger numbers.

```
>>> fnum1 = 0.002
>>> print(type(fnum1),fnum1)
<class 'float'> 0.002

>>> fnum2 = 0.000002
>>> print(type(fnum2),fnum2)
<class 'float'> 2e-06

>>> astring = str(fnum2)
>>> print(astring)
2e-06
```

Listing 15 - Very Small Floating Points

Python will display the smaller number in scientific notation, which also occurs if we convert the value to a string. Sometimes a calculation resulting in scientific notation can have a lot of decimal places we may want to round. We can't use a round function because we aren't actually dealing with decimal places. What we can do, however, is use a special prefix to specify the level of accuracy we want, as shown below.

```
>>> fnum3 = 1.4966517743e17

>>> print("%.3g"%fnum3)
1.49e17
```

Listing 15 - Rounded Floats

Let's move on to complex numbers in Python.

---

[1] (GeekforGeeks, 2022), https://www.geeksforgeeks.org/floating-point-representation-basics/ ↩

[2] (Math is Fun), https://mathsisfun.com/rounding-numbers.html ↩

Exploring Complex Numbers

Python offers the ability to manipulate complex numbers. These numbers, which consist of real and imaginary parts, occur when carrying out square root operations on negative numbers. They are used in engineering to analyze concepts like structural vibrations and the behavior of fluids. A complex number is represented in the form *x+yj* where *x* is the real part and *y* the imaginary part.

```
>>> cnum1 = 17+3j
>>> print(cnum1)
(17+3j)

>>> type(cnum1)
<class 'complex'>

>>> cnum2 = complex(17,3)
>>> print(cnum2)
(17+3j)
```

Listing 16 - Complex Numbers in Python

We can also use the *complex* function to yield a complex value and pick out parts of the complex numbers using suffixes.

```
>>> print(cnum1.real)
17.0

>>> print(cnum1.imag)
3.0

>>> print(cnum1.conjugate())
(17-3j)
```

Listing 17 - Complex Number Parts

We can manipulate complex numbers with mathematical operators and use them in functions. We can't, however, directly convert complex numbers to int or float, but we can use the *int()* function on the *.real* and *.imag* suffix parts.

We'll leave complex numbers for now and review another form of data: booleans.

Working with Booleans

Boolean variables take the *true* or *false* value. These are either created by direct assignment of the pre-defined names, True and False, or as a result of a comparison operation. Let's demonstrate some of these.

```
>>> bval1 = True
>>> print(bval1)
True

>>> if bval1:
...    print("It's true!")
...
It's true!

>>> if inum1 > inum2:
...     print(inum1,"is greater than",inum2)
...
164 is greater than 37
```

Listing 18 - Boolean Values

The Boolean operators are *and*, *or*, and *not*. We can use the *bitwise* operators "&" and "|" as shorthand for *and* and *or*, respectively.

```
>>> bval1 = True
>>> bval2 = False
>>> print(bval1 and bval2)
False

>>> print(bval1 & bval2)
False

>>> print(bval1 or bval2)
True

>>> print(bval1 | bval2)
True

>>> print(not bval2)
True
```

Listing 19 - Boolean Operators

That completes the main data type refresher, so let's finish by reviewing the larger structures we can build with these data types.

Understanding Python Bytes

There's a class of built-in data, known as *bytes*,[1] which is used quite extensively in Python. A byte is an eight bit value and is declared by using the *b* prefix, or the *encode* function.

```
>>> b1 = b'A'
>>> type(b1)
<class 'bytes'>

>>> b2 = "A".encode()
>>> type(b2)
<class 'bytes'>

>>> print(b2)
b'A'
```

Listing 20 - Declaring a Byte using an ASCII Character

While byte literals can be declared using ASCII[2] characters, there are some values that don't appear in the ASCII character set. For these, we need to use an escape sequence to assign a bytes value. We'll typically use a hexadecimal constant, which we'll discuss shortly. Here is an example.

```
>>> b1 = b'\x7F'
>>> type(b1)
<class 'bytes'>

>>> print(b1)
b'7f'
```

Listing 21 - Declaring a Byte using a Hexadecimal Constant

When we print a bytes value that is non-ASCII, it is displayed in hexadecimal form. Bytes are of particular significance as some Python system functions require input in the form of bytes.

Python provides a *bytes()* function, but it has some interesting characteristics and we need to be careful when using it. Let's find out why by using it to create a byte character and then trying to create a byte number.

```
>>> b1 = bytes('A')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: string argument without an encoding

>>> b1 = bytes('A'.encode())
>>> print(b1)
b'A'

>>> b2 = bytes(123)
>>> print(b2)
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
0\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x
00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'

>>> b1 = bytes([127])
>>> print(b1)
b'\x7f'
```

Listing 21 - Byte Mistakes

In this example, we used the *bytes()* function together with the *encode()* function to deliver a byte object. However, when we use the bytes function with a number, it doesn't return the bytes representation of that number. It returns an array of bytes of that many zeroes. To get the value of 127, we need to pass it the value in square brackets - in other words, as a single value list.

We'll cover byte arrays in more detail when we discuss lists.

Next, let's do a few exercises before we learn about the way different based numbers are manipulated in Python.

---

[1] (Programiz, 2022), https://www.programiz.com/python-programming/methods/built-in/bytes ↩

[2] (ASCIITable.com, 2022), https://www.asciitable.com/ ↩

Exercises

1. What data type results from doing an integer divide of the floating point value 124.00?

Answer

Answer | Verify

2. How do we extract the remainder of dividing 1325 by 91?

Answer

Answer | Verify

3. We have a string variable *str* that contains "It is time to break the chains of life, if you follow you will find what's beyond reality". What is the substring expression to extract the phrase "chains of life"?

Answer

Answer | Verify

4. We have an imaginary number, *cnum1*, which we want to print in the form "The real part is xx and the imaginary part is yy". Write the print statement for that.

Answer View hints

Answer | Verify

5. The formula for gravitational force is: n = (G x m1 x m2)/(r x r) The result, *n*, is the gravitational force in Newtons. The value of the gravitational constant *G* is 6.674e-11. The mass of the first object is *m1*, let's use earth at 5.972e24 kg. The mass of the second object is *m2*, let's use the sun at 1.989e30 kg. The distance *r* between the sun and earth is 1.496e11 km.

Use Python to calculate the gravitational force between the earth and the sun. Enter the answer in scientific notation with 3 digits accuracy.

Answer

Answer | Verify

6. I want to create a type bytes value of 65. How do I do this?

Answer View hints

Answer | Verify

Sets, Lists, and Dictionaries

In this Learning Unit, we'll cover sets, lists, and dictionaries - composing structures that are built from the basic data types. We'll cover the following Learning Objectives:

- Manipulate sets
- Learn about lists
- Explore tuples
- Work with dictionaries

This Learning Unit should take approximately 45 minutes to complete.

Python Data Basics
Understanding Python Bytes

Sets, Lists, and Dictionaries
Manipulating Sets

Resource Center

My Kali

VPN

Manipulating Sets

A *set* is a one dimensional array of items, which may be of different data types. A set isn't ordered or indexed - it's just a collection of items. We can initialize a set, and we can add and remove from it, but we can't change the value of an element. We also can't access the elements directly. We can loop through the set, check for the existence of an entry, and carry out set operations such as *unions*.

One use of sets is to store multiple values of a common item. For example, we might create a set called "squalls", which stores security qualifications. We can initialize it as an empty set.

```
>>> squalls = set()
```

Listing 22 - Create an Empty Set

We have to use the *set()* function to initialize a set. We can keep it empty as shown above, or we can initialize it with an set of values as shown below.

```
>>> squalls = set(['CISSP','CertIV','BadCert'])
```

Listing 23 - Create a Populated Set

Let's add the OSCP qualification to the set.

```
>>> squalls.add('OSCP')

>>> print(squalls)
set(['CertIV', 'CISSP', 'BadCert', 'OSCP'])
```

Listing 24 - Adding to a Set

We can also use curly braces to create a set with values. We'll review how creating an empty set would be ambiguous later.

```
>>> vowels = {"a","e","i","o","u"}

>>> type(vowels)
<class 'set'>
```

Listing 25 - Creating a Populated Set

We can remove an element from a set, either using the *remove()* or *discard()* functions. The remove function will fail if the element to be removed doesn't exist, but the discard function won't. We'll use the discard function to remove the "BadCert".

```
>>> squalls.discard('BadCert')

>>> print(squalls)
set(['CertIV', 'CISSP', 'OSCP'])
```

Listing 26 - Taking from a Set

We can check whether a value exists in a set using the *in* operator.

```
>>> 'OSCP' in squalls
True
```

Listing 27 - Checking for a Set Value

We can loop through a set, for example to print each element.

```
>>> for qual in squalls:
...   print(qual)
...
CertIV
CISSP
OSCP
```

Listing 28 - Looping Through a Set

Let's define another set and check the results of carrying out a union and an intersection operation.

```
>>> squalls = set(['CISSP','CertIV','OSCP'])
>>> squally = set(['CISSP','CEH','CISA'])

>>> squat = squalls.union(squally)
>>> squat
set(['CertIV', 'CISSP', 'CISA', 'CEH', 'OSCP'])

>>> squalls | squally
set(['CertIV', 'CISSP', 'CISA', 'CEH', 'OSCP'])

>>> squat = squalls.intersection(squally)
set(['CISSP'])

>>> squalls & squally
set(['CISSP'])
```

Listing 29 - Set Operations

The *union()* function will create a new set with all unique elements of the two sets, and *intersection()* will return only those elements that exist in both sets.

We can use the line or pipe (|) as a shorthand for union and ampersand (&) as a shorthand for intersection. We've also used the interactive implicit *print* function - just stating the variable name.

Working with Lists

A list is set of items, which inherently, aren't in any specific order. It allows duplicates and we can change individual values as required.

We've already used a list to manipulate a string as a sequence of characters. We can do this by using square brackets to hold a single subscript to point to one character or by using a pair of subscripts to define a substring.

We can build a list by initializing it as empty and then appending values or by initializing the list with values.

```
>>> list1 = []
>>> print(list1)
[]

>>> list2 = [16, 7 ,"phoenix", 3.14159]
>>> print(list2)
[16,7,'phoenix',3.14159]

>>> list1.append(88)
>>> list1[0]=16
>>> list1.extend([7,"phoenix",3.14159])

>>> print(list1)
[16, 7, 'phoenix', 3.14159]

>>> print(len(list1))
4
```

Listing 30 - Creating Lists

We need to be careful when working with lists that we don't accidentally delete our list. Using append, for example, works directly on the list and returns a result of null. So if we by mistake issue the command as an assignment, the results is an empty none-type value.

```
>>>list1=list1.append(4)
>>>print(list1)
>>>
```

Listing 31 - Making a List mistake

Lists are very useful ways of arranging and manipulating data. In the example above, we initialized a null list and a list containing mixed values (two integers, a string, and float). We then constructed the same list by appending a single value, replacing an element's value, and extending with multiple values.

We can even include a list as an element within a list. We can then use subscripts to access elements, and where we have an embedded list, use double subscripts to access an element in the embedded list.

```
>>> list1.append(list2)

>>> print(list1)
[16, 7, 'phoenix', 3.12159, [16, 7, 'phoenix', 3.12159]]

>>> print(list1[2])
'phoenix'

>>> print(list1[4])
[16, 7, 'phoenix', 3.12159]

>>> print(list1[4][3]])
3.12159
```

Listing 32 - Embedding Lists

We can also use *del* to remove a list element and replace multiple elements by using a subscript pair consisting of a start and ending index.

```
>>> del list2[2]

>>> print(list2)
[16, 7, 3.12159]

>>> list2[1,3]=[32,48]

>>> print(list2)
[16, 32, 48]
```

Listing 33 - List Element Assignment

Lists retain the values in the order we set them in. However, if we want to keep the order of the added values, we can use the *sort* function to sort our list or create a new sorted list with the *sorted* function. Let's sort our first list.

```
>>> list1.sort()

>>> print(list1)
[3.12159, 7, 16, 'phoenix']
```

Listing 34 - Sorting Lists

Of course, sorting lists with consistent item types is more sensible, but Python is very forgiving.

We can also insert a new item at a specific point in the list. If we have a sorted list, we can loop through it to insert the item at its correct place.

```
>>> list3 = [1, 3, 5, 7, 13, 17, 19, 23]

>>> list3.insert(4,11)

>>> print(list3)
[1, 3, 5, 7, 11, 13, 17, 19, 23]
```

Listing 35 - Insert into a List

We'll continue reviewing different types of data representation shortly, but let's demonstrate how lists can provide a mapping for hexadecimal numbers. To do this, we can declare a list of hex values. For example, if we then want to obtain the hexadecimal digit for 12, we can do that by indexing into the list.

```
>>> hexlist = ['0','1','2','3','4','5','6','7','8','9','A','B','C','D','E','F']

>>> print(hexlist[12])
'C'
```

Listing 36 - Indexing Lists

We can concatenate lists by simply adding them together, either as list expressions or list variables.

```
>>> list3 = list3 + [29, 31, 37]

>>> print(list3)
[1, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37]
```

Listing 37 - List Concatenation

Let's move on to another special form of lists called *tuples*.

Exploring Tuples

A tuple is very similar to a list, but it is *immutable*. This means that once the values in a tuple have been set, they cannot be changed. We declare a tuple with parentheses.

```
>>> tup1 = (1, 2, 3)

>>> type(tup1)
<class 'tuple'>

>>> print(tup1)
(1, 2, 3)
```

Listing 38 - Declaring a Tuple

As with lists, a tuple can contain embedded data, including lists or other tuples, and is indexed the same way.

```
>>> tup2 = (1,2, ['alfa', 'bravo'], b'ABCDEF')

>>> type(tup2[0])
<class 'int'>

>>> type(tup2[2])
<class 'list'>

>>> type(tup2[3])
<class 'bytes'>

>>> type(tup2[2][1])
<class 'str'>
```

Listing 39 - Complex Tuples

Tuples provide a useful data structure for related items of unchanging data.

Let's move on to the final list-like data structure in Python, *paired entry lists*, also known as *dictionaries* or *associative arrays*.[1]

---

[1] (Brilliant, 2022), https://brilliant.org/wiki/associative-arrays ↩

Using Dictionaries

A dictionary is a list in which one element of a paired entry is a key and the other element is its corresponding value.

Let's set up a small dictionary containing the first four letters of the English alphabet and associate them with the first four letters of the NATO phonetic alphabet. We'll then use the dictionary to find a value and add more entries.

```
>>>  dict1 = {"A":"Alfa", "B":"Bravo", "C":"Charlie", "D":"Delta"}

>>>  print(dict1["C"])
'Charlie'

>>> dict1.update({"E":"Echo"})

>>> print(dict1)
{'A':'Alfa', 'B':'Bravo', 'C':'Charlie', 'D':'Delta', 'E':'Echo'}

>>> type(dict1)
<class 'dict'>
```

Listing 40 - Setting up a Dictionary

The key in a dictionary has to be a unique string. However, as with lists, we can embed different data types, including lists and dictionaries, as values in the dictionary.

```
>>> dict2 = {"101561":["Smith","Daniel","OpsGroup","09/02/2019"],\
       "105361":["Jones","Brian","OpsGroup","11/07/2021"]}

>>> for empid in iter(dict2):
...         print(empid,dict2[empid][1],dict2[empid][0])
...
101561 Daniel Smith
105361 Brian Jones
```

Listing 41 - Setting up a Dictionary

This provides us with a powerful way of structuring data.

Exercises

1. We have two skill sets, skills1 and skills2. What expression would result in the set of common elements in the sets?

Answer

Answer Verify

2. What is another name for an associative array?

Answer

Answer Verify

3. What condition would we use to determine whether the set skills1 contained the skill 'OSCP'?

Answer

Answer Verify

4. We have a list called list1 that contains the values: [1, 4, 5, 6, 8, 11, 24, 31, 32] We want to add the value 17, in order, into this list. What command do we use?

Answer

Answer Verify

5. There are some traps we can fall into when manipulating data with Python. Create list2, append to it, and print it using the code snippet below:

```
list2 = [1, 2, 3]
list2 = list2.append(4)
print(list2)
```

What is the result that is printed?

Answer

Answer Verify

6. Continuing from the previous question, check the data type of list2. What is the output when running the command as shown in the following code snippet?

```
type(list2)
```

Answer

Answer Verify

7. We declare c56 = (1,2,3). We can determine its type by entering type(c56). What type of object is c56?

Answer

Answer Verify

Different Base Representations

In this Learning Unit, we'll cover the following Learning Objectives:

- Represent and manipulate binary values
- Represent and manipulate octal numbers
- Represent and manipulate hexadecimal numbers

This Learning Unit should take approximately 90 minutes to complete.

We'll typically use decimal integers in Python, but there are other ways to represent integers as well. The three most useful alternative representations we'll find are binary, octal, and hexadecimal. We'll learn how they are represented in Python and how we transform a number into another base for calculations and printing.

Manipulating Binary Values

Binary numbers[1] take the values 0 and 1. While 0 and 1 are still equal to their face values, the binary equivalent of 2 is "10", 3 is "11", and 4 is "100". Instead of each position in the number representing a decimal digit 0-9, it represents a binary digit 0-1. Binary numbers work on *base 2*.

We can do math with binary numbers as we would with our normal decimal system, but need to remember to carry over at 2 instead of 10. So adding 1001 (or 9 in decimal) to 1010 (10 in decimal) gives 10011 (which is the binary representation of 19).

Binary numbers can be declared in Python using the *0b* prefix. Let's declare a binary variable and check its type.

```
>>> b1=0b01000001

>>> print(chr(b1))
'A'

>>> type(b1)
<class 'int'>
```

Listing 42 - Declaring a binary value

Now we know binary numbers in Python are manipulated as integers and we can declare them as such using all the standard integer operations. We still might want to declare a value in binary form if, for example, we're using it as a mask to pick out a certain bit in a set of flags. This often happens when dealing with system and network activity, such as manipulating the header fields in a TCP packet with a set of flags as shown below.

Figure 1: TCP Header

In order to know whether this transmission was an acknowledgement, we have to check whether the *ACK* flag is set. To isolate and check that bit, we'll need to carry out some bitwise operations.

Bitwise operators are used to manipulate binary numbers. There are four bitwise logic operators: *AND*, *OR*, *XOR* and *NOT*. AND operates on two inputs and returns a 1 where both bits are 1, OR operates on two inputs and returns a 1 where one (or both) of the bits are 1, XOR operates on two inputs and returns a 1 where only one of the two bits is 1, and NOT operates on one input and returns the opposite of the bit. There are two additional operators, "<<" and ">>", which shift the bits to the left or right by a specified number of bit positions. The operations are demonstrated below.

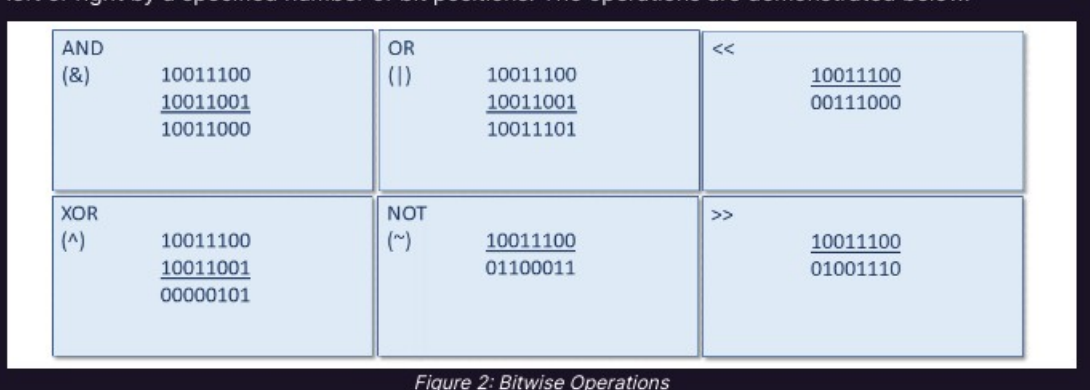

Figure 2: Bitwise Operations

We can use the bitwise operations on bit (integer) variables.

```
>>> bit1 = 0b01001011

>>> bit2 = 0b00011101

>>> bit3 = bit1 & bit2

>>> print(b3)
9
```

Listing 43 - ANDing a Binary Value

Computers work at the binary level, but display their work in the 8-bit form of a byte or in multiple bytes. A set of 2 bytes (16 bits) is called a *word* and 4 bytes (32 bits) is called a *double word* (DWORD).

Let's do some bitwise manipulation using bytes. We'll use a flag byte to extract the ACK bit from the TCP Header by adding **AND** to the flag byte against the bit sequence 00010000. If the value is not zero, the ACK bit is set and Python recognizes not zero as true. The bit sequence to extract a bit is often called a *mask* while the process of adding AND is known as masking. We can use the word AND or we can use *&* as a shorthand version of the operator.

```
>>> flag_byte = 0b00011000

>>> ack=flag_byte & 0b00010000

>>> if ack:
...  print("Ack set")
...
Ack set
```

Listing 44 - Masking a Binary Value

We also might want to set or unset a bit. Let's set the ACK bit using the OR function. The ACK bit is in position 4 with the lowest order byte bit being position 0. We can use the left shift function in conjunction with the OR (|) function to set it.

```
>>> flag_bits = 0b00000000

>>> flag_bits = flag_bits | 1<<4

>>> print(bin(flag_bits))
0b1000
```

Listing 45 - Setting a Binary Value

Of course, we can clear a bit using the same technique and the AND operator.

XOR is a commonly used operator with a very useful characteristic: when it's applied twice, the original result is uncovered. This makes XOR ideal for cryptographic functions. Let's create a cipher and then recover it.

```
>>> plaintext = 0b01101010
>>> key = 0b10101010
>>> ciphertext = plaintext ^ key

>>> print(bin(ciphertext))
'0b11000000'

>>> decode = ciphertext ^ key

>>> print(bin(decode))
'0b1101010'
```

Listing 46 - Using Binary XOR to Encipher a Byte

In the example above, we've used the *bin()* function to display an integer in its binary form.

Because of its use in cryptography, XOR is useful for manipulating plaintext in the form of Python strings. Let's code up a variation on the *sxor()* function[2] to perform string XOR.

```
>>> def sxor(s1,s2):
...    tkey = s2 * int(len(s1)/len(s2)+1)
...    return ''.join(chr(ord(a) ^ ord(b)) for (a,b) in zip(s1,tkey))
...
```

Listing 47 - The String XOR Function

We provide the *sxor()* function with two parameters: a string to encrypt and a key to encrypt it with. The first thing we'll do is repeat the key so it's at least as long as the string we're encrypting. We'll use the ratio of the string lengths to determine how many times to repeat it and use the multiply operator to concatenate the key.

The return line starts by clearing the return value and then looping for the length of the message, joining the encrypted message character-by-character to the result. The core of the statement is the exclusive OR ("ord(a) ^ ord(b)") in which the integer value of the message character is XOR'd with the integer value of the key at the corresponding point. The resulting integer is converted back to a character (*chr()*).

The *zip* function takes two iterables as arguments and returns an iterator that generates a sequence of character pairs. It generates pairs until the end of one of its inputs, which is why we made the key longer than the string.

Now that we understand the function, let's use it to encrypt and recover a message.

```
>>> msg = 'Always think of what is useful and not beautiful. Beauty will come of its
own accord.'
>>> key = 'BlueVariedWine'
>>> cipher = sxor(msg,key)

>>> print(cipher.encode('hex'))
030002042f12521d0d0d39024e0a244c020d371552001644221a0b033700550b3915520b0005221d0703370
05b451404131c111d771e07092e4c160a3b04520603443e1d1d452d1b1b4537021106170079

>>> recovery = sxor(cipher,key)

>>> print(recovery)
Always think of what is useful and not beautiful. Beauty will come of its own accord.
```

Listing 48 - Using Binary XOR to Encipher Strings

This is a pretty useful function to add to our toolbox. We can code up a very similar function for byte strings.

```
>>> def bxor(b1,b2):
...    tkey = b2*int(len(b1)/len(b2)+1)
...    return bytes([a^b for (a,b) in zip(b1,tkey)])
...

>>> b1 = b'I no longer know whether I wish to drown myself in love, vodka, or the
sea.'
>>> b2 = b'kafkaesque'
>>> cipher = bxor(b1,b2)

>>> print(cipher)
b'"A\x08\x04A\t\x1c\x1f\x12\x00\x19A\r\x05\x0e\x12S\x06\x1d\x00\x1f\t\x03\x19A,S\x06\x1
c\x16\x03A\x12\x04A\x01\x01\x1e\x02\x0bK\x0c\x1f\x18\x04\t\x15Q\x1c\x0bK\r\t\x1d\x04IS\
x07\x1a\x01\x00\x00JK\x0e\x17S\x05\x1d\x00K\x12\x03\nO'

>>> recovery = bxor(cipher,b2)

>>> print(recovery)
b'I no longer know whether I wish to drown myself in love, vodka, or the sea.'
```

Listing 48 - Using Binary XOR to Encipher Byte Arrays

We've covered the basic representation and manipulation of bits, so let's move on to *octal* numbers.

---

[1] (Rod Castor, 2020), https://towardsdatascience.com/binary-hex-and-octal-in-python-20222488cee1 ↩

[2] (Mark Byers, 2010), https://stackoverflow.com/questions/2612720/how-to-do-bitwise-exclusive-or-of-two-strings-in-python ↩

Octal Numbers

Octal[1] representation is less common, but it's still important to understand how to manipulate octal data. Octal numbers are declared with the *0o* prefix, have a base of 8, and are represented by the digits 0-7. Let's declare and check an octal number.

```
>>> o1 = 0o342127
>>> type(o1)
<class 'int'>

>>> o2 = o1 + 0o412661
>>> print(oct(o2))
0o755010

>>> print(o2)
252424
```

Listing 49 - Declaring and Using Octal Values

Octal numbers, as with binary numbers, are an integer, so they can be manipulated with the same operations. We can use the octal representation solely with the *oct()* function and the same variable can be displayed or manipulated as an integer.

Continuing with the example above, if we want to obtain the octal value without the *0o* prefix, we can remove it as shown below.

```
>>> po1 = 0o755010
>>> print(oct(po1)[2:])
755010

>>> print(oct(po1).replace('0o',''',1))
755010

>>> no1 = -0o4415
>>> print(oct(no1).replace("0o",'',1))
-4415
```

Listing 50 - Removing the Octal Prefix

Removing parts of a string is called taking a *slice* from the string. We'll often bump into this terminology when doing stringwise operations.

While we can just remove the first two characters of the octal string if we know the value is positive, using the *replace* function provides a better solution as it works for both negative and positive octal values.

We can convert a string of decimal digits to an integer with the *int()* function, which assumes a base of 10. In order to convert a string of octal digits into an integer, we can use the same method but include the optional second parameter to *int()* to specify the base. For octal, this is 8. Including the *0o* prefix in the string is optional.

```
>>> so1 = "0o755010"
>>> int1 = int(so1,8)

>>> print(oct(int1))
0o755010

>>> so2 = "755010"
>>> int2 = int(so2,8)

>>> print(oct(int2))
0o755010

>>> so3 = "-0o4415"
>>> int3 = int(so3,8)

>>> print(oct(int3))
-0o4415
```

Listing 51 - Converting Octal Strings to Integer

---

[1] (Rod Castor, 2020), https://towardsdatascience.com/binary-hex-and-octal-in-python-20222488cee1 ↩

Hexadecimal Numbers

Hexadecimal[1] numbers are integers with a base of 16, with the digits 10-15 being represented by the characters A-F (lowercase is also acceptable). Hexadecimal numbers use the prefix *0x* and integers can be displayed using the *hex()* function.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

```
  0x13A7                0xF00D
+ 0x26FF              - 0x8AAD
  ------              --------
  0x3BA6                0x3560
```

Figure 3! Hexadecimal (Base 16) Numbers

The figure above provides an example of hexadecimal addition and subtraction. Again, these operations are standard integer operations in Python and we can display the results in either decimal or hexadecimal, or using any other base, as we wish. We can display values in hexadecimal as shown below.

```
>>> hex1 = 0x13A7
>>> hex2 = 0x26FF

>>> print(hex(hex1+hex2))
0x3ba6

>>> hex3 = 0xF00D
>>> hex4 = 0x8AAD

>>> print(hex(hex3-hex4))
0x3560
```

Listing 52 - Declaring and Using Hexadecimal Numbers

We'll often want to manipulate the two hexadecimal digits of an 8-bit byte individually. These are called *nibbles*,[2] which are categorized into high-order nibble (bits 7-4) and low-order nibble (bits 3-0). We can isolate the low-order nibble by adding AND to the byte with the hexadecimal value *0x0F*. We can obtain the high-order nibble by shifting the byte four bits to the right.

```
>>> hexbyte = 0xBE
>>> low_nibble = hexbyte & 0x0F

>>> print(hex(low_nibble))
0xe

>>> high_nibble = hexbyte >> 4

>>> print(hex(high_nibble))
0xb
```

Listing 53 - Nibbling Hexadecimal Numbers

When we're dealing with text files, we'll often navigate by using line numbers. However, when reviewing binary files, we'll typically use hexadecimal addresses as offsets from the beginning of the file. An example of this is shown below.

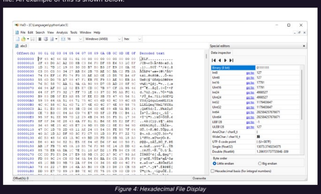

Figure 4: Hexadecimal File Display

Each line in the middle panel is 16 bytes, so the address sensibly progresses by increments of 0x10 bytes per line. Let's try this in our own Python script.

```
1. fname = input("filename: ")
2. f = open(fname,"rb")
3. offset = 0x0
```

Listing 54 - Hexit Part I: Opening files

The first thing we need to do is request a filename, which we'll open for reading in binary mode at line 2. We won't code in any error checking for this demonstration, so let's assume the file exists. At line 3, we'll set the initial offset address to zero as we're at the start of the file.

Let's continue setting up an outer per-line loop.

```
4. infile = True
5. while infile:
6.     addr=str(hex(offset)[2:].zfill(8))
7.     hexline = ""
8.     asciline = ""
```

Listing 55 - Hexit Part II: The Outer Line Loop

In this code fragment, we're setting a flag called *infile*, which indicates we are still processing data. We'll use this to signal from our inner loop (yet to come) to the outer loop at the end of file. The outer loop continues while this is true (line 5). At line 6, we get the hexadecimal value of our *offset*, remove the leading "0x", include leading zeroes to a fixed length of 8 characters, and store this as *addr*. Next, we clear the hexadecimal and the ASCII[3] (printable) portions of the line.

Let's now script up the inner loop for the 16 bytes we'll display on a line.

```
9.     for x in range(0x10):
10.        byte = f.read(1)
11.        if len(byte) == 0:
12.            hexline = hexline.ljust(48)
13.            infile = False
14.            break
15.        if ord(byte) > 29:
16.            aschr = chr(byte[0])
17.        else:
18.            aschr = "."
19.        asciline = asciline + aschr
20.        hexline = hexline + hex(byte[0])[2:].zfill(2) + " "
```

Listing 56 - Hexit Part III: The Inner Byte Loop

We'll start by doing an inner *for* loop sixteen times. For clarity, we're using the hexadecimal value "0x10" to indicate this will be for an address increment of 0x10. At line 10, we read one byte from our file, which the *read* function delivers to us in type bytes. If the length is zero, which we can check at line 11, then we've reached the end of file. In this case, we'll continue the hexadecimal part of the line with trailing spaces to a length of 48 characters, set the flag for our outer loop, and leave the inner loop.

At line 15, we'll check whether the decimal value of the byte is greater than 29, in which case we'll obtain its character representation by using the expression *byte[0]*, which will return the byte as an integer. We could have also used *ord(byte)* to obtain the integer value. If the byte is less than 30, it's an unprintable character that we'll replace with a period. We'll add this to the ASCII part of line 19. Then, at line 20, we'll obtain its hexadecimal form, remove the "0x" prefix, and add it to the hexadecimal portion of the line.

All that's left is to continue looping through the file, printing out our file dump.

```
21.    print(addr+ "  "+hexline+ "  "+asciline)
22.    offset = offset + 0x10
23. f.close()
```

Listing 57 - Hexit Part IV: The Ending

Once the inner loop terminates, or we exit, we'll print the line address at line 21 (hexadecimal portion and ASCII portion). At line 22, we'll add the address increment 0x10 and loop for the next line. Once we have printed the last line, we can close the file at line 23 and terminate.

The full code listing is below:

```
fname = input("Filename: ")
f = open(fname,"rb")
offset = 0x0
infile = True
while infile:
    addr= str(hex(offset)[2:].zfill(8))
    hexline = ""
    asciline = ""
    for x in range(0x10):
        byte = f.read(1)
        if len(byte) == 0:
            hexline = hexline.ljust(48)
            infile = False
            break
        if ord(byte) > 29:
            aschr = chr(byte[0])
        else:
            aschr = "."
        asciline = asciline + aschr
        hexline = hexline + hex(byte[0])[2:].zfill(2) + " "
    print(addr+ "  "+hexline+ "  "+asciline)
    offset = offset + 0x10
f.close()
```

Listing 58 - Hexit Code Listing

We've already found that we have to manipulate data between representations, and the key transformation we need to be familiar with is bytes to integers and strings. We can convert a byte into an integer by either using the *ord* function or subscripting it.

```
>>> b1 = b'A'

>>> type(b1)
<class 'bytes'>

>>> type(b1[0])
<class 'int'>

>>> type(ord(b1))
<class 'int'>
```

Listing 59 - Manipulating a Hex Byte

We'll often have to deal with data that consists of multiple bytes (reading a section of data from a file, for instance). We'll move data between string and bytes form, and while we can use subscripts to convert it byte-by-byte, it is easier to use the *encode* and *decode* functions.

```
>>> str1 = "Hey hey, my my, rock'n'roll will never die"

>>> type(str1)
<class 'str'>

>>> type(str1.encode())
<class 'bytes'>

>>> b1 = b"Hey hey, my my, rock'n'roll will never die"

>>> type(b1)
<class 'bytes'>

>>> type(b1.decode())
<class 'str'>
```

Listing 60 - Encoding and Decoding Bytes

We can also (though less often) convert float values to hexadecimal using the *float.hex()* function.

```
>>> f1 = 3.14159

>>> print(f1oat.hex(f1))
0x1.921f9f01b866ep+1
```

Listing 61 - Hexadecimal for Floating Points

---

[1] (Rod Castor, 2020), https://towardsdatascience.com/binary-hex-and-octal-in-python-20222488cee1 ↩

[2] (Wikipedia, 2022), https://en.wikipedia.org/wiki/Nibble ↩

[3] (ASCIITable.com, 2022), https://www.asciitable.com/ ↩

Exercises

1. Use the XOR bitwise operator to achieve the ultimate alchemy - transform 'iron' into 'gold'. What is the key?

Answer — View hints

Answer — Verify

2. We want to check the URG bit setting in the TCP header flags byte (flag_bits). Using the left shift bit approach as we covered for the ACK bit, what is the Python expression which will return true if the URG bit is set?

Answer — View hints

Answer — Verify

3. We have an octal string called "status". What is the function call we would use to return the integer value of that string?

Answer

Answer — Verify

4. In Kali, code up the script to dump out files in hexadecimal as presented above. Then create a new file and enter the following text exactly as shown.

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Using your hexadecimal dump script, what is the hexadecimal value at offset 1C?

Answer

Answer — Verify

Converting and Displaying Data Types

In this Learning Unit we'll cover the following Learning Objectives:

- Convert integer data objects
- Convert byte data objects
- Convert character data objects
- Convert between strings and hexadecimal strings

This Learning Unit should take approximately 60 minutes to complete.

Now that we have our foundation of understanding data types, let's learn to convert them from one form to another.

Introducing Conversions

Our basic data types are integers and strings and the bytes object is used when interacting with the underlying system. Characters are a special atomic string.

We'll use data objects in their native forms and represent them as native strings, but we may also want to represent them in another form. We'll learn how to work with various data types in hexadecimal, and we can take a similar approach for binary, octal, or any other exotic base.

Characters in their "western" form are represented as 8 bits - a single byte. However, because computers support more complex languages such as Arabic and Japanese, the representation of characters often uses two bytes. This is known as the *Unicode*[1] form.

Let's review how we move from one data object form to another and how we move between displayable strings. The functions we'll use are summarized in the figure below.

Data Object				To String		From String	
	Int	Byte	Character	Native	Hexadecimal	Native	Hexadecimal
Int	-	.to_bytes() [[]]	chr()	str()	hex()	int()	int(,16)
Byte	int.from_bytes()	-	.decode()	str()	.hex()	.encode()	bytes.fromhex()
Character	ord()	.encode()	-	-	.encode().hex()	-	bytearray.fromhex().decode()

Figure 5: Atomic Conversion

[1] (Python.org, 2022), https://docs.python.org/2/tutorial/introduction.html#unicode-strings ↩

Converting Integers

We can convert an integer to a byte using the *.to_bytes()* function or we can use the bytes conversion on the integer as a list entry.

```
>>> i1 = 99

>>> b1 = i1.to_bytes(1,'big')

>>> print("to_bytes:",type(b1),b1)
to bytes: <class 'bytes'> b'c'

>>> b2 = bytes([i1])

>>> print("In list: ",type(b2),b2)
iterable: <class 'bytes'> b'c'
```

Listing 62 - Integer to Bytes

Because integers can exceed a single byte, the *to_int* function has to cope with multiple bytes. It takes two arguments: the number of bytes we want and whether the most significant is first or last. Dealing with a single byte makes no difference, but we need to be consistent if we're using larger integers. We've used *'big'*, which means the first byte is the most significant. If we use the list entry method, the integer must be less than 256 characters hence, a single byte.

Converting an integer to a character is simple enough with the *chr()* function. If the value is greater than 255, *chr()* will return a unicode special language character. The *chr()* function has no meaning for integers larger than two bytes.

```
>>> i1 = 99

>>> print(chr(i1))
c
```

Listing 63 - Integer to Character

The example in the figure below shows the character representation of the two-byte value 468: a caron (that's an upside down caret!), which is used in a number of non-English languages.

```
>>>
>>>
>>> i2 = 468
>>> print(chr(i2))
ǔ
>>>
```

Figure 6: Unicode Integers

To convert an integer to its decimal string representation is quite simple: we'll use the *str()* function. To convert it back, we'll use *int()*.

```
>>> print(str(i1))
99

>>> i2 = int("42")
```

Listing 64 - Display Integer as Decimal String

Using the *hex()* function to display an integer in hexadecimal form is straightforward. If the integer is greater than 255, this results in a "big" order of hexadecimal bytes. Recovering an integer from hexadecimal form requires that we use the *int()* function with the optional base argument set to 16 (or 8 if we want to recover from an octal string, 2 for binary, etc.).

```
>>> i1 = 99

>>> print(hex(i1))
0x63

>>> i2 = 333

>>> print(hex(i1))
0x14d

>>> h1 = "0x01ff"

>>> i2 = int(h1,16)

>>> print(i2)
511
```

Listing 65 - Display Integer as Hexadecimal String

Now let's move on to bytes.

Converting Bytes

Computers use a lot of byte data so it's important we understand how to work in bytes. First, let's learn to convert a single byte to an integer.

```
>>> b1 = b'c'

>>> i2 = int.from_bytes(b1,'big')

>>> print(i2)
99
```

Listing 66 - Byte to Integer

We can use the *decode()* function to convert a byte to a character.

```
>>> b1 = b'c'

>>> print(b1.decode())
c
```

Listing 67 - Byte to Character

We can use the *str()* function to provide the native string representation of a bytes object with the *b'* prefix. The *encode()* function converts the string back and does not expect the prefix.

```
>>> b1 = b'c'

>>> print(str(b1))
b'c'

>>> b2 = 'c'.encode()

>>> print(type(b2),b2)
<class 'bytes'> b'c'
```

Listing 68 - Byte as a Native String

When representing bytes objects as hexadecimal, we can't provide a byte value as an argument to the hex function, so we'll need to use the *hex()* method instead. Note that this does not include the hexadecimal prefix *0x*.

```
>>> b1 = b'c'

>>> b2 = b1.hex()

>>> print(type(b2),b2)
<class 'str'> 63
```

Listing 69 - Byte as a Hexadecimal String

The resulting string is the hexadecimal digits 63, presented without the prefix.

To convert from a hexadecimal string back to binary, we'll use the *bytes.fromhex()* function, again without a prefix.

```
>>> h1 = '01FF'

>>> b2 = bytes.fromhex(h1)

>>> print(type(b2),b2)
<class 'bytes'> b'\x01\xff'
```

Listing 70 - Hexadecimal String to Bytes

In this case, the resulting byte value has a prefix of b'.

Converting Characters

A character is the special case of a single element of a string, and is of type string. We can convert a character (but not a string of length greater than 1) into its integer value by using the *ord()* function.

```
>>> ch1 = 'c'

>>> print(type(ch1))
<class 'str'>

>>> print(ord(ch1))
99
```

Listing 71 - Character to Integer

A character can be converted to a byte using the *encode* method, which applies to any length string.

```
>>> ch1 = 'c'

>>> b1 = ch1.encode()

>>> print(type(b1),b1)
<class 'bytes'> b'c'
```

Listing 72 - Character to Bytes

Because a character's native form is a string, we don't need to convert it. However, we can convert a character to its hexadecimal form by using its encoded representation. This converts any length string, including length 1 characters, but does not include the *0x* prefix.

```
>>> ch1 = 'c'

>>> hex1 = ch1.encode().hex()

>>> print(hex1)
63
```

Listing 73 - Character as a Hexadecimal String

Unfortunately, the Python2 *decode* ("hex") function doesn't exist in Python3, so we need to use the *bytearray* function to recover a character from a hexadecimal string.

```
>>>  hex1 = '63'

>>>  print(bytearray.fromhex(hex1).decode())
c
```

Listing 74 - Hexadecimal String to Character

String to Hexadecimal String

We can convert between strings of length greater than 1 and hexadecimal in the same way we converted single character strings.

```
>>> str1 = "Gothic"

>>> print(str1.encode().hex())
476f74686963

>>> hex1 = "47656E7265"

>>> print(bytearray.fromhex(hex1).decode())
Genre
```

Listing 75 - String to and from Hexadecimal String

Exercises

1. We convert an integer to a hexadecimal string using the *hex()* function and a byte to a hexadecimal string using the *.hex()* method. Which one - function or method - returns a "0x" prefix?

Answer

Answer | Verify

2. What function do we use to convert a hexadecimal string back to bytes?

Answer

Answer | Verify

3. Convert the integer value 482737218405 into hexadecimal and then convert the hexadecimal string to a character string. What is the result?

Answer View hints

Answer | Verify

4. What function would we use to convert the character "p" to a byte value?

Answer

Answer | Verify

Manipulating Binary Large Objects in Python

In this Learning Unit, we'll cover the following Learning Objectives:

- Learn about arrays of bytes and byte arrays
- Manage BLOBs as bytes

This Learning Unit should take approximately 30 minutes to complete.

Converting and Displaying Data Types
String to Hexadecimal String

Manipulating Binary Large Objects in Python
Arrays of Bytes and Byte Arrays

Arrays of Bytes and Byte Arrays

A byte array is a list of type bytes and is declared using the prefix *b* or the *bytes()* or *bytearray()* functions. Let's start with the *b* prefix.

```
>>> ba1 = b'ABCDEFG'

>>> type(ba1)
<class 'bytes'>

>>> print(ba1)
b'ABCDEFG'

>>> print(ba1[0])
65

>>> type(ba1[0])
<class 'int'>
```

Listing 76 - Declaring Byte Arrays with "b"

We can also create a byte array using the *bytes()* function.

```
>>> ba2 = bytes([1,3,55,7,11,13,17,19,23,31,37])

>>> type(ba2)
<class 'bytes'>

>>> print(ba2)
b'\x01\x03\x05\x07\x0b\r\x11\x13\x17\x1f%'
```

Listing 77 - Declaring Byte Arrays with Bytes()

The display now includes hexadecimal digits because Python doesn't show bytes with a value less than 32 as displayable characters. It also includes special escape characters such as (\r) for carriage return. We'll cover how we declare and manipulate hexadecimal values shortly.

We can also create an array of bytes using the *encode* function that we used to prepare a single bytes value. We can also use *decode* to turn the bytes object back to a string.

```
>>> ba3 = 'ABCDEFG'.encode()

>>> type(ba3)
<class 'bytes'>

>>> bs3 = ba3.decode()

>>> type(bs3)
<class 'str'>

>>> print(bs3)
ABCDEFG
```

Listing 78 - Encoding and Decoding Bytes

The fourth way of creating an array of bytes is to use the *bytearray()* function.

```
>>> ba4 = bytearray([1,3,55,7,11,13,17,19,23,31,37])

>>> type(ba4)
<class 'bytearray'>

>>> print(ba4)
bytearray(b'\x01\x03\x05\x07\x0b\r\x11\x13\x17\x1f%')
```

Listing 79 - Declaring Byte Arrays with bytearray()

This now presents a slightly different type and print result. The *ba4* variable is a byte array list, rather than a bytes list.

The final way to create a byte array is to convert it from a list using the *bytearray()* function. Previously, we set up a list of prime numbers called *list3*. Let's set it up again and convert it to a byte array.

```
>>> list3=[1,3,55,7,11,13,17,19,23,31,37]

>>> ba5 = bytearray(list3)

>>> type(ba5)
<class 'bytearray'>

>>> print(ba5)
bytearray(b'\x01\x03\x05\x07\x0b\r\x11\x13\x17\x1d\x1f%')
```

Listing 80 - Byte Arrays

Arrays of bytes and byte arrays are not the same. Let's check out the difference when we try to modify a value.

```
>>> ba5[0] = 6

>>> print(ba5)
bytearray(b'\x06\x03\x05\x07\x0b\r\x11\x13\x17\x1f%')

>>> ba3[0] = 6
Traceback (most recent call last)
  File "<stdin>", line 1,in <module>
TypeError: 'bytes' object does not support item assignment
```

Listing 81 - The Difference Between Bytes and Bytearrays

We call arrays of bytes immutable because the values in the array can't be changed. When we try to change an element in *ba3*, which is an array of bytes, we get an error. Byte arrays are mutable, and so changing an element of *ba5* is not a problem.

We can manipulate arrays of bytes to an extent. For example, we can create new bytes objects by adding bytes objects together.

```
>>> ba6 = bytes([1,2,3,4,5,6,7])

>>> ba7 = bytes([8,9,10,11,12])

>>> ba6 = ba6 + ba7

>>> print(ba6)
b'\x01\x02\x03\x04\x05\x06\x07\x08\t\n\x0b\x0c'
```

Listing 82 - Concatenating Byte Objects

This is possible because, while *ba6* is immutable, Python has actually created a new bytes variable but retained the same name and discarded the old *ba6*. Note that in the result above, we have some more special byte representations: the integer value 9 is *tab* (\t) and the value 10 is *newline* (\n).

We have more flexibility with byte array objects because not only can we change elements, but also append and remove them. Let's do some more work with our *ba5* byte array.

```
>>> ba5.append(65)

>>> print(ba5)
bytearray(b'\x06\x03\x05\x07\x0b\r\x11\x13\x17\x1f%A')
```

Listing 83 - Working with Bytearrays

We've added the integer value 65, which is the ASCII character "A", to the byte array. We can add byte arrays as we can with arrays of bytes. We can also delete one or a range of elements from a byte array, using the **del** command.

```
>>> del ba5[0:2]

>>> print(ba5)
bytearray(b'\x05\x07\x0b\r\x11\x13\x17\x1f%A')
```

Listing 84 - Deleting Bytearray Elements

In this example, we removed the elements at indexes 0 and 1 of *ba5*.

There are some functions we can use with arrays of bytes and with byte arrays. One of the more useful is *find()*, which finds the first occurrence of a byte or string of bytes in the object. Let's check this on the *ba6* bytes object we previously defined.

```
>>> ba6.find(b'\x05')
4
```

Listing 85 - Finding Bytes

The fifth array element, with index 4, is the first occurrence of the hexadecimal bytes 0×05.

Managing BLOBs as Bytes

The term *BLOB*[1] refers to large objects in binary form, such as an image or audio file. We can read a blob as we would any other binary file.

Python is supported by libraries that provide specific functions related to various types of blobs. The *PIL*[2] library is an example of this and is installed in Kali, but it has problems showing images. For the purposes of this Topic, we'll demonstrate the use of the Python *wand*[3] library.

> *Wand isn't installed on your in-browser Kali system and you don't need this for the exercises, but if you're running your own Kali platform and want to try it out, you can install wand using pip (**pip install wand**).*

Let's script some Python code into a file called **blob1.py** to read an image file and provide us with information about it.

```
1.  from wand.image import Image
2.  fin = input("Enter image file: ")
3.  with open(fin,"rb"):
4.    image_blob - f.read()
5.    print("BLOB stored as: ",type(image_blob))
6.    print("Length of BLOB: ",len(image_blob))
```

Listing 86 - Reading a BLOB

At line 1, we import the *Image library* from the wand module. We then ask for a file name, and at line 3, open the file as a binary file for reading onto an array of bytes. We read the complete file into an object called *image_blob* at line 4. At line 5, we display its class and at line 6, its length.

Let's now investigate the contents of *image_blob* using the image library functions.

```
7.    with Image(blob=image_blob) as img:
8.      print("Image size:     ",img.height,"x",img.width)
9.  f.close()
```

Listing 87 - Investigating an Image

At line 7, we define *img* as the wand object we're working on then at line 8, we can extract the height and width of the image using the attributes extracted by wand. Let's run the code and review what we get. We'll use one of the background images in Kali for this.

```
kali@kali:~$ python3 blob1.py
Enter image file:  /usr/share/backgrounds/xfce/xfce-blue.jpg
BLOB stored as:  <class 'bytes'>
Length of BLOB:  197011
Image size:      1800 x 2880
```

Listing 88 - Running the Blob Script

We can write out the file in the same way we read it, using the *f.write()* command. Alternatively, we can use wand's *save()* function. Let's use wand's file format conversion capability to convert a **.jpg** to a **.png** and then save it. We'll code this up as **blob2.py**.

```
1.  from wand.image import Image
2.  fin = input("Enter file name: ")
3.  fout = fin.replace(".jpg",".png")
4.  with open(fin,"rb") as f:
5.    image_blob = f.read()
6.    with Image(blob=image_blob) as img:
7.       img.format = 'png'
8.       img.save(filename=fout)
9.  f.close()
```

Listing 89 - JPG to PNG BLOB Converter

As before, we take a filename and read the file into a bytes object. At line 3, we replace the ".jpg" extension on the input file with a ".png" extension to form our output filename. We'll still pass our bytes object to the image library to manipulate as *img*, and at line 7, instruct wand to convert the format to PNG. At line 8, we save the file. That's it! We have just written a blob manipulation script to convert jpg to png!

Let's run this and then use our **blob1** script to check the png file. We've moved the **xfce-blue.jpg** into our home folder for this run.

```
kali@kali:~$ python3 blob2.py
Enter image name:  xfce-blue.jpg
kali@kali:~$ python3 blob1.py
Filename:  xfce-blue.png
Blob stored as:  <class 'bytes'>
Length of blob:  1052066
Image size:     1800 x 2880
```

Listing 90 - Running the JPG to PNG BLOB Converter

The **xfce-blue.png** file has been created, and we can determine it's much larger than the jpg file we started with.

---

[1] (Ionos, 2022), https://www.ionos.com/digitalguide/websites/web-development/binary-large-object ↩

[2] (Wikipedia, 2022), https://en.wikipedia.org/wiki/Python_Imaging_Library ↩

[3] (Wand, 2022), https://docs.wand-py.org/en/0.6.7/ ↩

Exercises

1. When we read a binary file, what type of data is provided by the *read()* function?

Answer

Answer | Verify

2. If we have a 2-byte bytes list with integer values of 7 and 10, what do we get when we print it?

Answer | View hints

Answer | Verify

3. Is a byte array immutable?

Answer

Answer | Verify

User-Defined Data Structures

In this Learning Unit, we'll cover the following Learning Objectives:

- Build stacks of data
- Double up on our lists
- Create graph structures
- Grow trees in Python
- Work with FIFO queues

This Learning Unit should take approximately 90 minutes to complete.

While Python offers some of the basic data types and structures, we can also create our own data structures. The potential types of user data structures are unlimited, so it's useful to review some examples.

Building Stacks of Data

Stacks are a data structure widely used in computing, especially for calling subroutines with parameters. The stack is a dynamic list that ensures the last item on the list is the first item out. This is sometimes called a *Last In First Out* (LIFO)[1] data structure.

Python provides a simple way of creating stacks using the *pop()* function on lists.

```
>>> stack = []

>>> stack.append(0x001F30)

>>> stack.append(0x002C5C)

>>> stack.append(0x001F38)

>>> print(hex(stack.pop()))
0x1f38

>>> print(hex(stack.pop()))
0x2c5c
```

Listing 91 - Using POP() for Stacks

Therefore, lists are inherently designed so they can be used as stacks.

---

[1] (Steve Campbell, 2022), https://www.guru99.com/python-queue-example.html ↩

Doubling Up on our Lists

We may want to have a data structure that keeps complex data objects in a particular order. While we can use the *sort* function on a simple list, it becomes more difficult when we have complex list entries. A common data structure for achieving this is a *linked list*,[1] which uses a list type object to store entries that contain a pointer to the next item in the list. We need to have a pointer to the first item in our linked list (which may not be the first item in the base list) and then navigate around the list by following the pointers, which will typically be list indices. The final entry in the list has a null (zero) pointer.

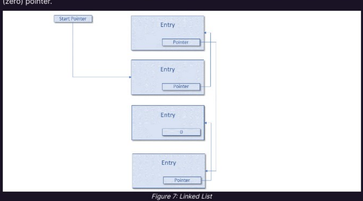

Figure 7: Linked List

To add an entry, we need to navigate through the list and insert it in the correct position, adjusting the pointers of both its preceding and seceding entries.

A special form of linked list is the *circular linked list*, in which the last entry points back to the first entry. In fact, this structure may have no concept of first or last entries.

Let's build a linked list script.

```
1.  linky = [["FirstPointer",0,"",""]]
2.  while True:
3.    action = input("(A)dd, (R)emove, (L)ist, (D)ump, (Q)uit: ").upper()
4.    if action == "Q":
5.      break
6.    if (action == "A" or action == "R"):
7.      name = input("Enter name: ")
```

Listing 92 - Linked List I - Starting Up

We start our script by creating a list with the first pointer as its first entry and the next entry index set to zero. Because this is a list of one entry, which is a list in itself, and not a list of four entries, we'll use double square brackets. We'll then start looping and continue until a "Q" is detected at line 3, at which point we'll break out of the loop and exit the script.

At line 6, we'll find whether we want to have an entry name and request it. We'll keep the code simple and not check that we're adding a unique name.

Let's now code up the dump and list actions.

```
8.    if action == 'D':
9.      print(linky)
10.   if action == 'L':
11.     index = linky[0][1]
12.     while index>0:
13.       print(linky[index][0])
14.       index = linky[index][1]
```

Listing 93 - Dumping and Listing the Linked List

Dumping the linked list is easy. We just print it in raw form. For listing in the correct order at line 11, we set the index to the first entry then loop until we get to the end where the next point is zero. At line 13, we print the name and then at line 14, set the index to the pointer for the next entry.

Now let's add an entry. We already have a name, so we need additional information to add into the list at the correct point. We need to take care of the boundary cases of the new entry coming first or last.

```
15.   if action == 'A':
16.       bdate = input("Birth date as yymmdd: ")
17.       sdate = input("Start date as yymmdd: ")
18.       newx = len(linky)
19.       linky.append([name,0,bdate,sdate])
20.       index = 0
21.       while True:
22.         next = linky[index][1]
23.         if next==0:
24.           linky[index][1] = newx
25.           break
26.         if linky[next][0] > name:
27.           linky[index][1] = newx
28.           linky[newx][1] = next
29.           break
30.         index = next
```

Listing 94 - Adding and Linked List Node

We start the process by requesting the information for the entry. At line 18, we record index for the new entry as *newx*, the next entry in the list, and then append it. At this point, it has no link to the next entry.

At line 20, we'll start with index set to the first entry in the list, and navigate through the linked list. At line 22, we record the pointer to the next entry in *next* and check whether this is zero, meaning there are no more entries. If so, we set the next entry pointer in the current node to our new entry and we're finished.

At line 26, we'll check whether our new entry should come before the next entry in the list. If so, at line 27, we'll set the current entry to point to our new entry and our new entry to point to the next one, and we're finished.

Failing that, we'll move on to the next entry and then loop around to do the checks again.

We'll finish our scripting with the removal function. Note that in order to keep the script simple, we won't actually delete the list entry, but leave it as unlinked "garbage".

If interested, you can check out how to do garbage collection[2] on this linked list!

```
31.   if action == 'R':
32.     index = 0
33.     while True:
34.       next = linky[index][1]
35.       if next == 0:
36.         print("Entry not found")
37.         break
38.       if name == linky[next][0]:
39.         linky[index][1] = linky[next][1]
40.         break
41.       index = next
```

Listing 95 - Removing a Linked List Node

Again, we'll start at the beginning of the list and navigate the links. At line 34, we'll set the *next* to the next entry in the list. If this is zero, we've reached the end of the list. This will only happen if the entry we're trying to remove doesn't exist. We'll print an error message and finish the action.

At line 38, we'll check whether the next entry is the one to be removed. If so, we claim the pointer for the entry after that and set our next entry to that one, which removes it.

If the next entry isn't the one, we move along the chain at line 41 and loop again.

The full code listing is here.

```
linky = [["FirstPointer",0,"",""]]
while True:
  action = input("(A)dd, (R)emove, (L)ist, (D)ump, (Q)uit: ").upper()
  if action == "Q":
    break
  if (action == "A" or action == "R"):
    name = input("Enter name: ")

  if action == 'D':
    print(linky)

  if action == 'L':
    index = linky[0][1]
    while index>0:
      print(linky[index][0],"born",linky[index][2])
      index = linky[index][1]

  if action == 'A':
    bdate = input("Birth date as yymmdd: ")
    sdate = input("Start date as yymmdd: ")
    newx = len(linky)
    linky.append([name,0,bdate,sdate])
    index = 0
    while True:
      next = linky[index][1]
      if next == 0:
        linky[index][1] = newx
        break
      if linky[next][0] > name:
        linky[index][1] = newx
        linky[newx][1] = next
        break
      index = next

  if action == 'R':
    index = 0
    while True:
      next = linky[index][1]
      if next == 0:
        print("Entry not found")
        break
      if name == linky[next][0]:
        linky[index][1] = linky[next][1]
        break
      index = next
```

Listing 96 - The Full Linky

Let's run this and check it out.

```
kali@kali:~$ python3 linky.py
(A)dd, (R)emove, (L)ist, (D)ump, (Q)uit: a
Enter name: Grant.G
Birth date as yymmdd: 710812
Start date as yymmdd: 181105
(A)dd, (R)emove, (L)ist, (D)ump, (Q)uit: a
Enter name: Hendrik.A
Birth date as yymmdd: 760816
Start date as yymmdd: 040812
(A)dd, (R)emove, (L)ist, (D)ump, (Q)uit: a
Enter name: Doe.J
Birth date as yymmdd: 880616
Start date as yymmdd: 220108
(A)dd, (R)emove, (L)ist, (D)ump, (Q)uit: l
Doe.J born 880616
Grant.G born 710812
Hendrik.A born 760816
(A)dd, (R)emove, (L)ist, (D)ump, (Q)uit: r
Enter name: Grant.G
(A)dd, (R)emove, (L)ist, (D)ump, (Q)uit: l
Doe.J born 880616
Hendrik.A born 760816
(A)dd, (R)emove, (L)ist, (D)ump, (Q)uit: q
```

Listing 97 - Running our Linked List Script

We'll find that entries are added and removed, and that we've maintained a linked list that we can display in name order.

Another form of linked list has a forward and a backward pointer so it can be navigated in either direction. We could implement this by using an additional field so that our initial entry resembles this:

```
linky = [["FirstPointer",0,0,"",""]]
```

Listing 98 - First Entry for a Doubly Linked List Script

We would then add in the backward link when we manipulate entries in the list. In this example, [1] is the backward pointer and [2] is the forward pointer.

```
      if linky[next][0]>name:
        linky[index][2] = newx
        linky[newx][1] = index
        linky[newx][2] = next
        linky[next][1] = newx
        break
```

Listing 97 - Adding a Doubly Linked Entry

Here, we are first setting the forward point of the current node to our new node, setting the backward pointer of our new node to the current node, and the forward node to the next node. The backward pointer of the next node (which would have originally been the current node) then points back to our new node.

[1] (Doogal Simpson, 2020), https://levelup.gitconnected.com/things-every-software-engineer-should-know-linked-lists-4841f75614ba ↩

[2] (Stanford University, 2022), https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/lectures/18/Slides18.pdf ↩

Creating Graph Structures

A *graph*[1] is a data structure consisting of nodes that are connected by *lines* (also called *edges*). These are used widely in mathematics to represent real world problems. The classic example is the *Traveling Salesman*[2] problem, which asks for the shortest path a salesman could take to visit all of the towns that are connected by a specified number of roads. Our standard car navigator uses this kind of challenge when finding the shortest path to our destination.

In Python, a graph is created by having a node data structure, which contains an identifier, or name, and list of other nodes to which it is connected.

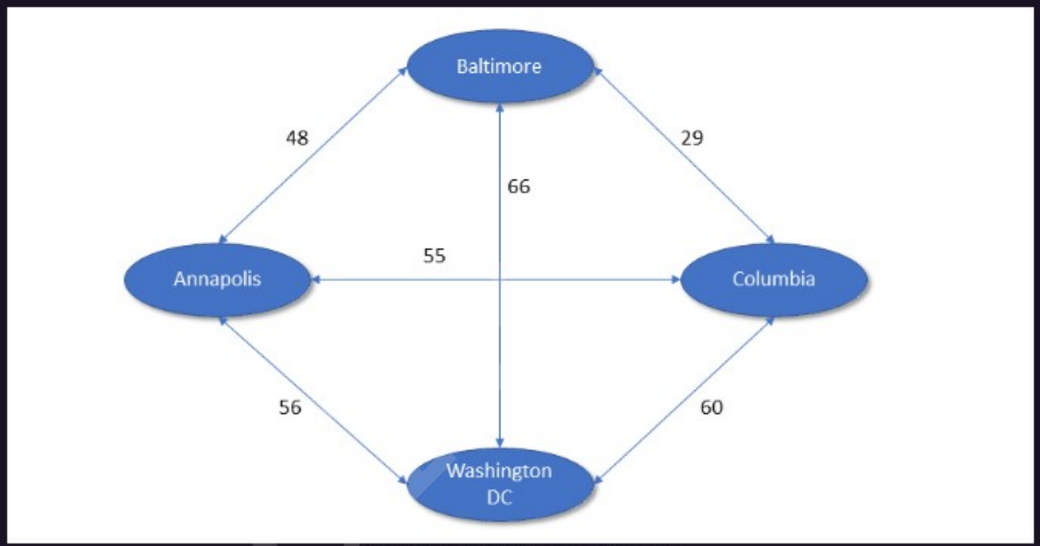

Figure 8: Travelling Salesman Graph

Our figure above shows a small traveling salesman problem for the *Beltway Bandits*[3] consisting of four cities as nodes, and showing the distance between them in kilometers. For the data structure to store this, we can use a list in which each entry is itself a list, consisting of two list elements: a name as the first element and a list of three tuples as the second element. Each of the tuples contain the name of the other cities and the distance from this city to the other. Let's create the list and append the entry for Baltimore.

```
kali@kali:~$ nodes = []

kali@kali:~$ nodes.append(["Baltimore",[("WashingtonDC",66), \
("Annapolis",48),("Columbia",29)]])
```

Listing 99 - Setting up a Graph Node

We'll leave the travelling salesman problem here and move on to a more familiar data structure: a *tree*.

---

[1] (Wolfram Mathworld, 2022), https://mathworld.wolfram.com/Graph.html ↩

[2] (Sandipan Dey, 2020), https://sandipanweb.wordpress.com/2020/12/08/travelling-salesman-problem-tsp-with-python/ ↩

[3] (GhostsofDC, 2018), https://ghostsofdc.org/2018/01/19/origin-term-beltway-bandit/ ↩

Growing Trees in Python

Trees are a special kind of graph that consist of nodes and links connecting them. First, trees start at a root and extend through layers of parent nodes and their children nodes. Second, each node will have just one parent node but zero or more children nodes. A consequence of this is that a tree cannot contain what is known as a *cycle* where a set of nodes are connected in a ring.

Trees are widely used in coding to represent everything from *family trees*[1] to multi-level structures of *Active Directory trees*.[2]

Let's review a problem in mathematics known as the *Steiner Problem in Graphs*.[3] This asks how we would connect a set of nodes with a specified distance from one another in a graph with minimum distance possible.

We'll focus on the Steiner Problem in Graphs as an example of coding a tree structure in Python. For real world context, an amino acid is coded with three RNA *nucleotides*,[4] each of which can take the values A, G, C, or U. The use of a tree structure to determine the shortest path between amino acids of different animals has been used in charting possible evolutionary paths.[5]

We'll not try to solve the problem in Python, but let's create its tree structure and add the ability to move nodes within the graph to manually test it. At each node, we'll store a name and a string of six characters from the nucleotide set *AGCU*, and make the distance between two nodes equal to the number of different values at the same index positions in the character sequences. We'll implement a doubly-linked list so that each node connects back to its one parent, and forward to zero or more children.

We'll use a small model for our coding. Our start point is as shown in the figure below.

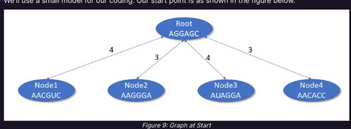

Figure 9: Graph at Start

The names of the nodes and the six-character sequences are stored in a *fixed-field* text file consisting of a sixteen-character name, space, and the six-character sequence. The actual data and the distances between nodes is as shown in the figure above, but we will calculate these so there is no need to store them.

Let's review the code for this.

```
1.  graph=[]
2.  index = 0
3.  f = open("graph.txt","r")
4.  for line in f:
5.    graph.append([line[0:16].strip(),line[17:23],0,[]])
6.    if index > 0:
7.      graph[0][3] += ([index])
8.    index += 1
```

Listing 100 - Graph Script I - The Start

At lines 1 and 2, we initialize the graph object to an empty list and we set the current index at 0. We then open **graph.txt** containing the data for the graph. For each line in the file, we loop through lines 5 to 8.

At line 5, we'll append a new entry to the list. The entry is itself a four element list. The first element is the node name, which we take from the first sixteen characters of the input line. We strip any leading or trailing blanks. The second element is the six characters of input starting one space after the node name. These are the values we'll be using to calculate distance. The third value represents the parent of this node. We'll leave the root node and every other node set to zero to reflect the fact that they all have the root node as their parent. The last element of the four element list entry is a list of all children, which will be empty for all the nodes we'll add.

At line 6, we'll check whether we're adding the root node. If not, we'll add the current entry to the list at line 7, which is contained as the fourth element of the root node. This indicates that this entry is another child of the root.

At line 8, we increment the index and loop around for the next line to be read. This loop will terminate when we have read all lines in the file.

Let's progress to the next stage of our script where we'll calculate the total distance between each node, also called the weight of the graph.

```
9.  while True:
10.   weight = 0
11.   for node in graph:
12.     distance = 0
13.     code1 = node[1]
14.     code2 = graph[node[2]][1]
15.     for i in range(6):
16.       if code1[i] != code2[i]:
17.         distance += 1
18      weight += distance
19.   print("Graph weight: ",weight)
```

Listing 101 - Graph Script II - The Calculation

At line 9, we'll start an infinite loop, or at least a loop until we break out of it. At line 10, we'll set the weight to zero and at line 11, we'll start a loop through every node. At line 12, we'll set the distance between this node and its parent to zero and extract the six character code sequence from this node at line 13. At line 14, we'll use the parent value from this node (the third element in the list, which makes up the node) to index into the graph to obtain the six-character code sequence from this node's parent. At the beginning of course, the parent will be root for every node, but that will change as we continue.

At line 15, we'll loop through each character for this node and its parent to check if they are the same. If not, we'll increment the distance at line 17. Once we've checked all six codes, we add the distance to the accumulating weight of the graph at line 18.

Once we've worked through all nodes in the graph, we print out the total weight of the graph at line 19. It's important to note that we start at the beginning of the list of nodes in the graph, which means that we check root against itself as its own parent. However, as there's zero distance, we don't need to code up an exclusion for this.

Let's now work through the last segment of code in which we can move a node in the graph. This will then loop around to again calculate the new weight of the graph.

```
20.   n1 = input("Node:       ")
21.   if len(n1) == 0:
22.     break
23.   n2 = input("New parent: ")
24.   i1 = 0; i2 = 0
25.   for i in range(len(graph)):
26.     if graph[i][0] == n1:
27.       i1 = i
28.     if graph[i][0] == n2:
29.       i2 = i
30.   if (i1==0 | i1==i2):
31.     print("Invalid node!")
```

Listing 102 - Graph Script III - Preparing to Move a Node

At line 20, we'll request the name of the node to be moved. If nothing is entered and just Return is pressed, we'll break out of the infinite loop and finish. At line 23, we'll request the name of the desired new parent node. We've now got two names, but we need to find their indices in the graph. The simplest way to do this is to loop through the graph checking the first entry in the list for each node and setting *i1* at line 27 to the index of the node to be moved and *i2* at line 29 to the index of the new parent node. We'll do a check at line 30 to make sure we have entered a valid name (we're not trying to move a node to itself), and if necessary, we'll report an error.

We are now ready to effect the node move.

```
32.   else:
33.     i3 = graph[i1][2]
34.     graph[i3][3].remove(i1)
35.     graph[i1][2] = i2
36.     graph[i2][3] += [i1]
37.  f.close()
```

Listing 103 - Graph Script IV - Changing the Graph

At line 33, we'll extract the index of the old parent node into *i3*. At line 34, we'll navigate to the old parent node and remove the child entry from its list of children. At line 35, we'll move the new parent node index into the parent node of the node being moved and follow this by adding the node we're moving as a child in its new parent node. We've now moved the node and can loop around to the recalculation of the graph weight.

The full listing follows.

```
graph = []
index = 0
f = open("graph.txt","r")
for line in f:
  graph.append([line[0:16].strip(),line[17:23],0,[]])
  if index > 0:
    graph[0][3] += [index]
  index += 1
while True:
  weight = 0
  for node in graph:
    distance = 0
    code1 = node[1]
    code2 = graph[node[2]][1]
    for i in range(6):
      if code1[i] != code2[i]:
        distance += 1
    weight += distance
  print("Graph weight: ",weight)
  n1 = input("Node:       ")
  if len(n1) == 0:
    break
  n2 = input("New parent: ")
  i1 = 0; i2 = 0
  for i in range(len(graph)):
    if graph[i][0] == n1:
      i1 = i
    if graph[i][0] == n2:
      i2 = i
  if (i1==0 | i1==i2):
    print("Invalid node!")
  else:
    i3 = graph[i1][2]
    graph[i3][3].remove(i1)
    graph[i1][2] = i2
    graph[i2][3] += [i1]
f.close()
```

Listing 104 - The Full Graph Script

Let's run the program and move the nodes around.

```
kali@kali:~$ python3 graph.py
Graph weight:  14
Node:       Node1
New parent: Node2
Graph weight:  13
Node:       Node3
New parent: Node2
Graph weight:  11
Node:       Node4
New parent: Node1
Graph weight:  10
Node:
```

Listing 105 - Graph Script V - Running the Graph Script

We've moved around the nodes and achieved an overall weight reduction. The resulting graph is shown below.

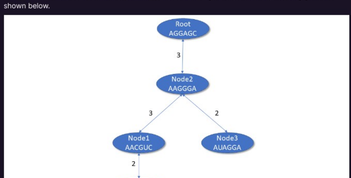

Figure 10: Graph at End

This isn't the ultimate graph script. In fact, it doesn't enforce proper connectivity (we can separate the graph into independent subgraphs and create cycles), but it serves to demonstrate the principles.

We'll leave graphs here and move on to our final user data structure: *FIFO queues.*

---

[1] (Ahsen Parwez, 2020), https://medium.com/@ahsenparwez/building-a-family-tree-with-python-and-graphviz-e4afb8367316 ↵

[2] (OmniSecu, 2022), https://www.omnisecu.com/windows-2003/active-directory/what-is-active-directory-tree.php ↵

[3] (Nelson Maculan, 1987), https://www.sciencedirect.com/science/article/abs/pii/S0304020808732365 ↵

[4] (NIH, 2022), https://www.genome.gov/genetics-glossary/Nucleotide ↵

[5] (Shore, 1979), https://mro.massey.ac.nz/handle/10179/13628 ↵

Working with FIFO Queues

A *First In First Out* (FIFO)[1] queue can be implemented in Python as a list where entries are added to the end and taken from the beginning. Let's inspect a simple script to implement a queue.

```
1.  import random
2.  queue = []
3.  while True:
4.    adding = input("New customer: ")
5.    if adding == "quit":
6.      break
7.    if len(adding) != 0:
8.      queue.append(adding)
9.    if random.randrange(10) > 6:
10.     if len(queue) > 0:
11.       print("Now consulting with: ",queue[0])
12.       queue.remove(queue[0])
13.   print("Queue length: ",len(queue))
```

Listing 106 - FIFO Script

We're going to have random service times in our queue, so at line 1, we'll import the *random* library. At line 2, we'll initialize the queue to an empty list. Then at line 3, we'll loop until line 5, where we'll quit by entering the word **quit**. We can add a new customer to the list by entering their name at line 4, or we can just press Return to not add a new customer. At line 7, if another customer has come in, we append them to the queue.

At lines 9 and 10, well check whether to take someone off the queue (randomly, if our random number is greater than 6 and people are still in the queue). If so, we reference the first in queue at index 0 in line 11 and then remove them from the queue at line 12. At each iteration of the loop, we display the queue length.

```
kali@kali:~$ python3 fifo.py
New customer: David Brown
Queue length:  1
New customer: Jake Backington
Queue length:  2
New customer: Sarah McNally
Now consulting with:  David Brown
Queue length:  2
New customer: Jim Mattson
Queue length:  3
New customer: Sally Prentice
Queue length:  4
New customer: Julia McKenzie
Now consulting with:  Jake Backington
Queue length:  4
New customer:
Queue length:  4
New customer:
Now consulting with:  Sarah McNally
Queue length:  3
New customer: quit
```

Listing 107 - Running the FIFO Script

The queue will grow and shrink as we enter and remove customers. Those taken off the queue are removed in a FIFO manner.

Using a list to implement a queue is an easy solution, but it does not scale well. However, it is sufficient for the purposes of this Topic.

---

[1] (Steve Campbell, 2022), https://www.guru99.com/python-queue-example.html ↩

Exercises

1. Using the diagram for our Beltway Bandit example, how would you define the entry in the nodes structure for the Annapolis node? Enter the full nodes with the cities in the order WashingtonDC, Columbia, Baltimore and enter your answer on a single line with no spaces.

Answer

Answer | Verify

2. How many parent nodes can we have for a child node in a tree data structure?

Answer

Answer | Verify

3. We are operating a FIFO queue. What list index do we use to remove the next queue item?

Answer

Answer | Verify

4. On your Kali workstation, create the **graph.txt** file and add another entry called *Steiner1* with nucleotide coding AAGGGC. Copy the **graph.py** code and run it. Make the parent of both Node1 and Node2 the new Steiner1 node. Make the parent of Node4, as previously, Node1. Make the parent of Node3, as previously, Node2. What is your final graph weight?

Answer

Answer | Verify

5. What form of Python data type should we use to implement a stack?

Answer

Answer | Verify

6. In a doubly-linked list, what process do we use to remove un-linked records?

Answer

Answer | Verify

Data Structures as Records

In this Learning Unit, we'll cover three Learning Objectives:

- Work with data records
- Work with databases
- Work with XML and JSON

This Learning Unit should take about 45 minutes to complete.

User-Defined Data Structures
Working with FIFO Queues

Data Structures as Records
Working with Data Records

Working with Data Records

We've covered a lot of material in this Topic, so we'll finish with just a brief review of data records. In business systems, we'll often talk about data records, which is a structured set of data stored on a file. While we'll typically use *SQL* as the file system, more *NoSQL* systems, such as *MongoDB*[1] and AWS's *DynamoDB*,[2] are being adopted. Business applications read a record from a file, process it, and store it back. Let's inspect how we'd do that in Python.

We can use basic data types to hold the elements of the record, but a better approach is to use the Python *dataclasses*[3] module. With this, we can define a data record that closely resembles the database record, and we can then manipulate data in a more structured way than with individual data items. Dataclasses are more elegant and they're easy to use. As an example, let's investigate how we might manage a database of our favorite original Bardcore composers.

We'll store a data record that holds the composer's name, nationality, dates of birth and death, an exemplar composition, and a list of the composer's compositions.

Let's start by defining the class and creating a data record.

```
>>> from dataclasses import dataclass

>>> @dataclass
... class Composer:
...     Name: str
...     Nationality: str
...     YearBorn: int
...     YearDied: int
...     Exemplar: str
...     Compositions: list
...
>>> comp1 = Composer("Hildegard von Bingham","German",1098,1179,"O Euchari",[])

>>> type(comp1)
<class '__main__.Composer'>

>>> print(comp1.Name,str(comp1.YearBorn)+'-'+str(comp1.YearDied))
Hildegard von Bingham 1098-1179
```

Listing 108 - Establishing the Composer Class

This is now showing as a user-defined class type, but we can use its elements as though they were basic data types. We'll use the data record name as a prefix. We've only created one data record but we can create as many as we like, giving them different names (or adding them to lists).

We can store data values into the elements of the class, as we would for a base variable. We've been advised that we should store *Ordo Virtutum* as the best exemplar of von Bingham's work, so let's update that. We'll also start to populate her compositions.

```
>>> comp1.Exemplar = "Ordo Virtutum"
>>> comp1.Compositions.append("O Euchari")
>>> comp.Compositions.append("Ordo Virtutum")
>>> comp.Compositions.append("O Virtus Sapientiae")
```

Listing 109 - Manipulating the Composer Record

Working with a data record is no different than working with individual data items. We just have the items properly organised within a record.

---

[1] (MongoDB, 2022), https://www.mongodb.com/ ↩

[2] (NetApp,2020), https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html ↩

[3] (Python, 2022), https://docs.python.org/3/library/dataclasses.html ↩

Working with Databases

We'll use a *SQLite* database to store our composer details. SQLite is built into Python, so we can use it without having to install any external modules. The first thing we'll do is to create the same data record for our composers in SQLite. To do that, we'll need to normalize our data record into a composer *masterfile* and a separate file that lists compositions by composer.

Let's build a script to manage our data.

```
1.  from dataclasses import dataclass
2.  import sqlite3
3.  from os.path import exists
4.  if not exists("music.db"):
5.    print("Creating database")
6.  conn = sqlite3.connect("music.db")
```

Listing 110 - Music I - Creating the Database

We'll be using a number of standard modules in this script. We've already used the dataclass module and we will be using the SQLite3 module also. We've also imported the *os.path* module so that we can check whether our database exists at line 4 and if not, provide a status message.

At line 6, we'll connect to the database, which will be created if it doesn't exist. Let's now create the tables we need in the database. To do this, we execute an SQL command using the connection we've just made. We'll use the **IF NOT EXISTS** clause so that this won't affect the existing tables.

```
7.  sq = 'CREATE TABLE IF NOT EXISTS Composer(Name TEXT NOT NULL, Nationality TEXT,
BornYear INT, DiedYear INT, Exemplar TEXT);'
8.  status = conn.execute(sq)
9.  sq = 'CREATE TABLE IF NOT EXISTS Compositions(Composer TEXT NOT NULL,Composition
TEXT NOT NULL);'
10. status = conn.execute(sq)
11. curse = conn.cursor()
```

Listing 111 - Music II - Creating the Tables

We execute both SQL statements and then set the cursor ready for use later.

Let's now create the data record class. To do this, let's start with what is known as the data class *decorator(@dataclass)*, and then define the class data structure. Python takes care of all the rest of the class requirements.

```
12. @dataclass
13. class Composer:
14.   Name: str
15.   Nationality: str
16.   BornYear: int
17.   DiedYear: int
18.   Exemplar: str
19.   Compositions: list
20. comp = Composer("","",0,0,"",[])
21.
```

Listing 112 - Music III - Creating the Record

We've now made sure we have a database with the tables we need, we've made the connection, and we've defined our data record. We could have created the database and tables manually at the command line, but having it in the script makes it simpler if we want to share or relocate the script. At line 20, we'll create a blank data record ready to use.

Now let's script the main program loop and the *Add Composer* function.

```
22.  while True:
23.    action = input("{A}dd Composer, {I}nsert Composition, {L}ist Music, {Q}uit:
").upper()
24.    if action == 'Q':
25.      break
26.
27.    if action == 'A':
28.      comp.Name = input("Name:         ")
29.      comp.Nationality = input("Nationality: ")
30.      comp.BornYear = input("Year Born:   ")
31.      comp.DiedYear = input("Year Died:   ")
32.      while True:
33.        opus = input("Composition: ")
34.        if len(opus) == 0:
35.          break
36.        comp.Compositions.append(opus)
37.        ex = input("Is this the exemplar Y/N? ").upper()
38.        if ex = "Y":
39.          comp.Exemplar=comp.Compositions[len(comp.Compositions)-1]
```

Listing 113 - Music IV - Getting Composer Data

We'll start the main loop and ask for an action. For this short demonstration script, we'll just have three functions: add a composer, insert another composition for a composer, and list the database. At line 24, we'll check whether we should quit, and if so, break out of the loop. Then at line 27, we'll check whether we are adding a new record. If so, let's request the main record data and store it in our data record.

We'll then request the initial compositions we want to store for the composer. At line 33, we'll ask for the composition and check whether it's a blank line. If so, we'll stop requesting compositions. Otherwise, we'll append it to our list of compositions.

The final task is to check if we want this to be the exemplar for the composer at line 37, and if so, store the composition in the exemplar field at line 39.

Now that we've got the data, we need to store it in the database.

```
40.    sq = 'INSERT INTO Composer(Name, Nationality, BornYear, DiedYear, Exemplar)
VALUES('
41.      sq = sq + "'"+comp.Name+"', "
42.      sq = sq + "'"+comp.Nationality+"', "
43.      sq = sq + comp.BornYear+", "
44.      sq = sq + comp.DiedYear+", "
45.      sq = sq + "'"+comp.Exemplar+"');"
46.      curse.execute(sq)
47.      for i in range(len(comp.Compositions)):
48.        sq = 'INSERT INTO Composition(Composer,Composition) VALUES('
49.        sq = sq + "'"+comp.Name+"', '"+comp.Compositions[i]+"');"
50.        curse.execute(sq)
51.      conn.commit()
```

Listing 114 - Music V - Adding a Record to the Database

Lines 40 to 45 build the query, which is executed in line 46 to insert a new Composer record. The Composition list at this stage is left blank. Following this, each of the entered compositions is put into an SQL query and at line 50, are written to the SQL Compositions table. We've now completed the addition of a Composer together with their Compositions. At line 51, we'll commit the changes to the SQLite database.

```
52.  if action == 'I':
53.    comp.Name = input("Name: ")
54.    sq = 'SELECT Name FROM Composer WHERE Name='+"'"+comp.Name+"';"
55.    curse.execute(sq)
56.    rows = curse.fetchall
57.    if len(rows) == 0:
58.      print("No such Composer")
59.    else:
60.      comp.Composition[0] = input("Composition: ")
61.      sq = 'INSERT INTO Composition(Composer,Composition) VALUES('
62.      sq = sq + "'"+comp.Name+"', "+comp.Composition+"');"
63.      curse.execute(sq)
64.      conn.commit()
```

Listing 115 - Music VI - Adding a Composition

We've also included the "I" option to enable more compositions to be added to the composer. We'll first ask for the composer's name and put that into an SQL statement so that we can check that the Composer exists at line 55. If they do, then at line 60, we'll request and include the composition in a query that we'll execute at line 63. Let's commit our change at line 64.

```
65.  if action == 'L':
66.    curse.execute('SELECT * FROM Composer;')
67.    library = curse.fetchall()
68.    for composer in library:
69.      print("Composer: "+composer[0])
70.      sq = 'SELECT * FROM Composition WHERE Composer="'+composer[0]+'";'
71.      curse.execute(sq)
72.      music = curse.fetchall()
73.      for opus in music:
74.        print("  "+opus[1])
```

Listing 116 - Music VII - Listing the Database

The last action we can take is to list the database. At line 66, we select all the composer records and at line 67, we'll read them into the library object. At line 68, we'll iterate through the library. At line 69, we iterate through printing the composer name and then create a query string to select the associated compositions from the compositions table at line 70. Let's execute this and then iterate through printing each composition under the composer.

All that's left is to close our cursor and the connection and we're done.

```
75.  curse.close()
76.  conn.close()
```

Listing 117 - Music VIII - Closure

Here is the full listing:

```
from dataclasses import dataclass
import sqlite3
from os.path import exists

if not exists("music.sqlite"):
  print("Creating database")
conn = sqlite3.connect("music.db")
sq = 'CREATE TABLE IF NOT EXISTS Composer(Name TEXT NOT NULL, Nationality TEXT,
BornYear INT, DiedYear INT, Exemplar TEXT);'
status = conn.execute(sq)
sq = 'CREATE TABLE IF NOT EXISTS Compositions(Composer TEXT NOT NULL,Composition TEXT
NOT NULL);'
status = conn.execute(sq)
curse = conn.cursor()

@dataclass
class Composer:
  Name: str
  Nationality: str
  BornYear: int
  DiedYear: int
  Exemplar: str
  Compositions: list
comp = Composer("","",0,0,"",[])

while True:
  action = input("{A}dd Composer, {I}nsert Composition, {L}ist Music, {Q}uit:
").upper()
  if action == 'Q':
    break

  if action == 'A':
    comp.Name = input("Name:         ")
    comp.Nationality = input("Nationality: ")
    comp.BornYear = input("Year Born:   ")
    comp.DiedYear = input("Year Died:   ")
    while True:
      opus = input("Composition: ")
      if len(opus) == 0:
        break
      comp.Compositions.append(opus)
      ex = input("Is this the exemplar Y/N? ").upper()
      if ex == "Y":
        comp.Exemplar=comp.Compositions[len(comp.Compositions)-1]
    sq = 'INSERT INTO Composer(Name, Nationality, BornYear, DiedYear, Exemplar)
VALUES('
    sq = sq + "'"+comp.Name+"', "
    sq = sq + "'"+comp.Nationality+"', "
    sq = sq + comp.BornYear+", "
    sq = sq + comp.DiedYear+", "
    sq = sq + "'"+comp.Exemplar+"');"
    curse.execute(sq)
    for i in range(len(comp.Compositions)):
      sq = 'INSERT INTO Composition(Composer,Composition) VALUES('
      sq = sq + "'"+comp.Name+"', "+comp.Compositions[i]+"');"
      curse.execute(sq)
    conn.commit()

  if action == 'I':
    comp.Name = input("Name: ")
    sq = 'SELECT Name FROM Composer WHERE Name='+"'"+comp.Name+"';"
    curse.execute(sq)
    rows = curse.fetchall
    if len(rows) == 0:
      print("No such Composer")
    else:
      comp.Composition[0] = input("Composition: ")
      sq = 'INSERT INTO Composition(Composer,Composition) VALUES('
      sq = sq + "'"+comp.Name+"', "+comp.Composition+"');"
      curse.execute(sq)
      conn.commit()

  if action == 'L':
    curse.execute('SELECT * FROM Composer;')
    library = curse.fetchall()
    for composer in library:
      print("Composer: "+composer[0])
      sq = 'SELECT * FROM Composition WHERE Composer="'+composer[0]+'";'
      curse.execute(sq)
      music = curse.fetchall()
      for opus in music:
        print("  "+opus[1])

curse.close()
conn.close()
```

Listing 118 - The Music Library Program in Full

Let's run this.

```
kali@kali:~$ python3 music.py
Creating database
{A}dd Composer, {I}nsert Composition, {L}ist Music, {Q}uit: a
Name:          Hildegard von Bingen
Nationality:  German
Year Born:    1098
Year Died:    1179
Composition: O Euchari
Is this the exemplar Y/N? y
Composition:  Ordo Virtutum
Is this the exemplar Y/N?
Composition: O Virtus Sapientiae
Is this the exemplar Y/N?
Composition:
{A}dd Composer, {I}nsert Composition, {L}ist Music, {Q}uit: l
Composer: Hildegard von Bingen
  O Euchari
  Ordo Virtutum
  O Virtus Sapientiae
{A}dd Composer, {I}nsert Composition, {L}ist Music, {Q}uit:
```

Listing 119 - Running the Music Library Program

We have a running program that uses data classes to manage our music library data in memory and uses SQL on disk.

Working with JSON and XML

We'll often bump into textual data structured in *XML*[1] or *JSON*[2] form, and it's useful to understand how to best manage it in our Python code.

Let's start with XML. The *Document Object Model* (DOM) [3] is an application programming interface based on XML and uses tree structures to manage XML data. Web page data consists of opening and closing tags.

Figure 11: Web Page Source

If we select *show view page source* from a website (right-click to get the context menu), we'll be presented with the raw HTML that makes up the page. It starts with an "HTML" tag and finishes with its closing counterpart "/HTML". Within these, there are "HEAD, /HEAD", "TITLE, /TITLE", and many other tags that contain their own particular parts of the web page.

We can use XML to describe any form of data records by using a field name and including the field contents in between the tags. Let's encode our composers in the form.

```
<Library>
  <Composer>
    <Name>Hildegard von Bingen</Name>
    <Nationality>German</Nationality>
    <BornYear>1098</BornYear>
    <DiedYear>1179</DiedYear>
    <Exemplar>O Euchari</Exemplar>
    <CompositionList>
      <Composition>O Euchari</Composition>
      <Composition>Ordo Virtutum</Composition>
      <Composition>O Virtus Sapientiae</Composition>
    </CompositionList>
  </Composer>
  <Composer>
    <Name>Jaufre Rudel</Name>
    <Nationality>Occitan</Nationality>
    <BornYear>1120</BornYear>
    <DiedYear>1147<DiedYear>
    <Exemplar></Exemplar>
    <CompositionList>
      <Composition>Quan Lorossinhol el Follos</Composition>
      <Composition>Lanquan lo temps renovelha</Composition>
    </CompositionList>
  </Composer>
</Library>
```

Listing 120 - An XML Composer Library

We'll call this **library.xml** and use it in our example script below.

Python provides an XML module called *ElementTree*[4] that we can use (there are others, but ElementTree will work nicely for this demonstration). Let's check out how we manipulate XML data with *etree*.

```
>>> import xml.etree.ElementTree as et

>>> tree = et.parse('library.xml')

>>> root = tree.getroot()

>>> print(len(root))
2
```

Listing 121 - Loading XML with ElementTree

We've imported the XML module and can access the Element Tree library as *et*. Let's first use the *parse* function to load our XML document into ElementTree's internal tree structure. We can then get the root node and determine how many entries we have in the library by checking the length of the root (i.e., its direct children).

Now let's access the data in the tree.

```
>>> print(root.tag)
Library

>>> print(root[0].tag)
Composer

>>> print(root[0][0].text)
Hildegard von Bingen
```

Listing 122 - Reading Tree Tags and Text

We can iterate across or over the tree. Let's list our composers.

```
>>> for i in range(len(root)):
...   print(root[i][0].text)
...
Hildegard von Bingen
Jaufre Rudel
```

Listing 123 - Iterating Across the Tree

We'll leave XML at this point and move on to JSON. JSON, or *Javascript Object Notation*, is used extensively in modern systems and applications, especially in cloud services. Python has a JSON[5] *encoder* and *decoder* module, which makes it easy to load and save JSON data as a dictionary.

```
{"Composer": [
   {"Name":"Hildegard von Bingen",
    "Nationality":"German",
    "BornYear":1098,
    "DiedYear":1179,
    "Exemplar":"O Euchari",
    "CompositionList": [
       {"Composition":"O Euchari"},
       {"Composition":"Ordo Virtutum"},
       {"Composition":"O Virtus Sapientiae"}
     ]
   },

   {"Name":"Jaufre Rudel",
    "Nationality":"Occitan",
    "BornYear":1120,
    "DiedYear":1147,
    "Exemplar":"Quan Lorossinhol el Follos",
    "CompositionList": [
      {"Composition":"Quan Lorossinhol el Follos" },
      {"Composition":"Lanquan lo Temps Renovelha"}
     ]
   }
 ]
}
```

Listing 124 - A JSON Composer Library

We have a copy of our library in JSON form called **library.json**. We can load this directly into a dictionary using the Python JSON module functions.

```
>>> import json

>>> f = open("library.json","r")

>>> json.load(f)

>>> type(library)
<class 'dict'>
```

Listing 125 - Reading JSON into a Dictionary

Now that we have a dictionary set up, we can use our normal dictionary manipulation to deal with it.

```
>>> for i in library(["Composers"]):
...   print(i["Name"])
...
Hildegard von Bingen
Jaufre Rudel
```

Listing 125 - Manipulating our Composer Dictionary

This is only an introduction, but there's so much more we can do with XML and JSON file data manipulation in Python. We'll save that for another lesson.

---

[1] (Mozilla, 2022), https://developer.mozilla.org/en-US/docs/Web/XML/XML_introduction ↩

[2] (JSON.org, 2022), https://www.json.org/json-en.html ↩

[3] (Wikipedia, 2022), https://en.wikipedia.org/wiki/Document_Object_Model ↩

[4] (Steph Howson, 2018), https://www.datacamp.com/community/tutorials/python-xml-elementtree ↩

[5] (Python.org, 2022), https://docs.python.org/3/library/json.html ↩

Exercises

1. What library do we need to import to work with data records in Python?

Answer

Answer — Verify

2. What decorator do we use to create a data record specification?

Answer

Answer — Verify

3. What connection function do we need to use to ensure any changes we make in the database are written to disk?

Answer

Answer — Verify

4. We define a dataclass called "vendor". What type of data object will this be?

Answer

Answer — Verify

5. What data structure is ideally suited to loading JSON data?

Answer

Answer — Verify