module 2

Welcome to module 2

Welcome back to our Python journey!
In the previous videos, we learned all about the basics of Python.
We started at the very beginning by understanding how security analysts use Python.
We learned several building blocks of Python.
We went into detail learning about data types, variables, and basic statements.
Now, we'll add to this and learn more about how to write effective Python scripts.
We'll discover ways we can make our efforts more efficient.
The upcoming videos are going to start by introducing functions, which are very important in Python.
Functions allow us to put together a set of instructions that we can use again and again in our code.
Afterwards, we're going to learn about Python modules and libraries, which include collections of functions and data types that we can use with Python.
They help us gain access to functions without having to create them ourselves.
Lastly, we're going to talk about one of the most important rules of programming, and that is code readability.
We'll learn all about ways to make sure everyone can understand and work with your code.
I'm excited that you've decided to continue your Python journey with me, so let's start learning more!

Python functions in cybersecurity

Previously, you explored how to define and call your own functions. In this reading, you’ll revisit what you learned about functions and examine how functions can improve efficiency in a cybersecurity setting.

Functions in cybersecurity

A **function** is a section of code that can be reused in a program. Functions are important in Python because they allow you to automate repetitive parts of your code. In cybersecurity, you will likely adopt some processes that you will often repeat.

When working with security logs, you will often encounter tasks that need to be repeated. For example, if you were responsible for finding malicious login activity based on failed login attempts, you might have to repeat the process for multiple logs.

To work around that, you could define a function that takes a log as its input and returns all potentially malicious logins. It would be easy to apply this function to different logs.

Defining a function

In Python, you'll work with built-in functions and user-defined functions. **Built-in functions** are functions that exist within Python and can be called directly. The *print()* function is an example of a built-in function.

User-defined functions are functions that programmers design for their specific needs. To define a function, you need to include a function header and the body of your function.

Function header

The function header is what tells Python that you are starting to define a function. For example, if you want to define a function that displays an "*investigate activity*" message, you can include this function header:

def display_investigation_message():

The *def* keyword is placed before a function name to define a function. In this case, the name of that function is *display_investigation_message*.

The parentheses that follow the name of the function and the colon (:) at the end of the function header are also essential parts of the syntax.

Pro tip: When naming a function, give it a name that indicates what it does. This will make it easier to remember when calling it later.

Function body

The body of the function is an indented block of code after the function header that defines what the function does. The indentation is very important when writing a function because it separates the definition of a function from the rest of the code.

To add a body to your definition of the *display_investigation_message()* function, add an indented line with the *print()* function. Your function definition becomes the following:

def display_investigation_message():

    print("investigate activity")

Calling a function

After defining a function, you can use it as many times as needed in your code. Using a function after defining it is referred to as calling a function. To call a function, write its name followed by parentheses. So, for the function you previously defined, you can use the following code to call it:

display_investigation_message()

Although you'll use functions in more complex ways as you expand your understanding, the following code provides an introduction to how the *display_investigation_message()* function might be part of a larger section of code. You can run it and analyze its output:

```
def display_investigation_message():
    print("investigate activity")
application_status = "potential concern"
```

```
email_status = "okay"
if application_status == "potential concern":
    print("application_log:")
    display_investigation_message()
if email_status == "potential concern":
    print("email log:")
    display_investigation_message()
```

```
application_log:
investigate activity
```

The *display_investigation_message()* function is used twice within the code. It will print "*investigate activity*" messages about two different logs when the specified conditions evaluate to *True*. In this example, only the first conditional statement evaluates to *True*, so the message prints once.

This code calls the function from within conditionals, but you might call a function from a variety of locations within the code.

Note: Calling a function inside of the body of its function definition can create an infinite loop. This happens when it is not combined with logic that stops the function call when certain conditions are met. For example, in the following function definition, after you first call *func1()*, it will continue to call itself and create an infinite loop:

```
def func1():
    func1()
```

Key takeaways

Python's functions are important when writing code. To define your own functions, you need the two essential components of the function header and the function body. After defining a function, you can call it when needed.

Introduction to functions

As the complexity of our programs grow, it's also likely that we'll reuse the same lines of code.
Writing this code multiple times would be time-consuming, but luckily we have a way to manage this.
We can use functions.
A function is a section of code that can be reused in a program.
We already learned one function when we worked with print and used it to output specified data to the screen.
For example, we printed "Hello Python." There are many other functions.
Sometimes, we need to automate a task that might otherwise be repetitive if we did it manually.
Previously, we compared other key Python components to elements of a kitchen.
We compared data types to categories of food.
There are differences in how we handle vegetables and meat, and likewise, there are differences in how we handle different data types.
We then discussed how variables are like the containers you put food in after a meal; what they hold can change.
As far as functions, we can think about them like a dishwasher.
If you aren't using a dishwasher, you'll spend a lot of time washing each dish separately.
But a dishwasher automates this and lets you wash everything at once.
Similarly, functions improve efficiency.
They perform repetitive activities within a program and allow it to work effectively.
Functions are made to be reused in our programs.
They consist of small instructions and can be called upon any number of times and from anywhere in our programs.
Another benefit to functions is that if we ever had to make changes to them, we can make those changes directly in the function, and there'll be applied everywhere we use them.
This is much better than making the same changes in many different places within a program.
The print() function is an example of a built-in function.
Built-in functions are functions that exist within Python and can be called directly.
They are available to us by default.
We can also create our own functions.
User-defined functions are functions that programmers design for their specific needs.
Both types of functions are like mini-programs within a larger program.
They make working in Python much more effective and efficient.
Let's continue learning more about them.
it's also likely that we'll reuse the same lines of code..

Create a basic function

Let's start our exploration of user-defined functions by creating and then running a very simple function.
The first thing we need to do is define our function.
When we define a function, we basically tell Python that it exists.
The def keyword is needed for this.
def is placed before a function name to define a function.
Let's create a function that greets employees after they log in.
First, we'll comment on what we want to do with this code.
We want to define a function.
Now, we'll go to a new line and use the keyword def to name our function.
We'll call it greet_employee.
Let's look at this syntax a little more closely.
After our keyword def and the function name, we place parentheses.
Later, we'll explore adding information inside the parentheses, but for this simple function, we don't need to add anything.
Also, just like we did with conditional and iterative statements, we add a colon at the end of this header.
After the colon, we'll indicate what the function will do.
In our case, we want the function to output a message once the employee logs in.
So let's continue creating our function and tell Python to print this string.
This line is indented because it's part of this function.
So what happens if we run this code?
Does it print our message?
Let's try this.
It doesn't.
That's because you also have to call your function.
You may not realize it, but you already have experience calling functions.
Print is a built-in function that we've called many times.
So to call greet_employee, we'll do something similar.
Let's go with a new line.
We'll add another comment because now our purpose is to call our function.
And then, we'll call the greet_employee function.
We'll run it again.
This time it printed our welcome message.
Great work!
We've now defined and called a function.
This was a simple function.
We're going to learn something next that will add to the complexity of the functions you write.

Use parameters in functions

Previously, we defined and called our first function.
It didn't require any information from outside the function, but other functions might.
This means we need to talk about using parameters in functions.
In Python, a parameter is an object that is included in a function definition for use in that function.
Parameters are accepted into a function through the parentheses after a function name.
The function that we created in the last video isn't taking in any parameters.
Now, let's revisit another function called range() that does use parameters.
If you recall, the range() function generates a sequence of numbers from a start point to the value before the stop point.
Therefore,range() does include parameters for the start and stop indices that each accept an integer value.
For instance, it could accept integers 3 and 7.
This means the sequence it generates will run from 3 to 6.
In our previous example, we wrote a function that displayed a welcome message when someone logged in.
It would be even more welcoming if we included the employee's name with the message.
Let's define a function with a parameter so we can greet employees by name!
When we define our function, we'll include the name of the parameter that our function depends on.
We place this parameter, the name variable, inside the parentheses.
The rest of the syntax stays the same.
Now, let's go to the next line and indent so we can tell Python what we want this function to do.
We want it to print a message that welcomes the employee using the name that's passed into the function.
Bringing this variable into our print statement requires a few considerations.
Like before, we start with the welcome message we want to print.
In this case, though, we're not stopping our message after we tell them they're logged in.
We want to continue and add the employee's name to the message.
That's why we're placing a comma after "You're logged in" and then adding the name variable.
Since this is a variable and not a specific string, we don't place it in quotation marks.
Now that our function is set up, we're ready to call it with the specific argument that we want to pass in.
In Python, an argument is the data brought into a function when it is called.
For example, earlier, when we passed 3 and 7 into the range() function, these were arguments.
In our case, let's imagine we want to greet an employee named Charley Patel.
We'll call our greet_employee() function with this argument.
And when we run this, Charley Patel gets a personalized welcome message!
In this example, we only have one parameter in our function.
But we can have more.
Let's explore an example of this.

Maybe instead of a single name parameter, we have a parameter for first name and a second parameter for last name.
If so, we would need to adjust the code like this.
First, when we define the function, we include both parameters and separate them with a comma.
Then, when we call it, we also include two arguments.
This time we're greeting someone with the first name of Kiara and with the last name of Carter.
These are also separated by a comma.
Let's run this and welcome Kiara Carter!
As we just explored, using more than one parameter just requires a few adjustments.
Great work in this video!
We learned a lot about working with parameters in a function.
This understanding is something you'll need as you continue to write Python scripts.

Return statements

We previously learned how we can pass arguments into a function.
We can do more than pass information into a function.
We can also send information out of one!
Return statements allow us to do this.
A return statement is a Python statement that executes inside a function and sends information back to the function call.
This ability to send information back from a function is useful to a security analyst in various ways.
As one example, an analyst might have a function that checks whether someone is allowed to access a particular file and will return a Boolean value of "True" or "False" to the larger program.
We'll explore another example.
Let's create a function related to analyzing login attempts.
Based on the information it takes in, this function will compute the percentage of failed attempts and return this percentage.
The program could use this information in a variety of ways.
For example, it might be used to determine whether or not to lock an account.
So let's get started and learn how to return information from a function.
Just like before, we start by defining our function.
We'll name it calculate fails() and we'll set two parameters related to login attempts: one for total_attempts and one for failed_attempts.
Next, we'll tell Python what we want this function to do.
We want this function to store the percentage of failed attempts in a variable called fail_percentage.
We need to divide failed_attempts by total_attempts to get this percentage.
So far, this is similar to what we've learned previously.
But now, let's learn how to return the fail percentage.
To do this, we need to use the keyword return.
Return is used to return information from a function.
In our case, we'll return the percentage we just calculated.
So after the keyword return, we’ll type fail_percentage.
This is our variable that contains this information.
Now, we're ready to call this function.
We'll calculate the percentage for a user who has logged in 4 times with 2 failed attempts.
So, our arguments are 4 and 2.
When we run this, the function returns the percentage of failed attempts.
It’s5, or 50 percent, but in some Python environments, this might not be printed to the screen.
We cannot use the specific variable named fail_percentage outside of the function.
So, in order to use this information in another part of the program, we would need to return the value from the function and assign it to a new variable.
Let's check this out.
This time, when the function is called, the value that's returned is stored in a variable called

percentage.
Then, we can use this variable in additional code.
For example, we can write a conditional that checks if the percentage of failed attempts is greater than or equal to 50 percent.
When this condition is met, we can tell Python to print an "Account locked" message.
Let's run this code.
And this time, the percentage isn't returned to the screen.
Instead, we get the “Account locked” message.
Coming up, we'll discuss more functions, but this time, we'll go over a few that are ready for use and built in to Python!

Functions and variables

Previously, you focused on working with multiple parameters and arguments in functions and returning information from functions. In this reading, you'll review these concepts. You'll also be introduced to a new concept: global and local variables.

Working with variables in functions

Working with variables in functions requires an understanding of both parameters and arguments. The terms parameters and arguments have distinct uses when referring to variables in a function. Additionally, if you want the function to return output, you should be familiar with return statements.

Parameters

A **parameter** is an object that is included in a function definition for use in that function. When you define a function, you create variables in the function header. They can then be used in the body of the function. In this context, these variables are called parameters. For example, consider the following function:

```
def remaining_login_attempts(maximum_attempts, total_attempts):

    print(maximum_attempts - total_attempts)
```

This function takes in two variables, *maximum_attempts* and *total_attempts* and uses them to perform a calculation. In this example, *maximum_attempts* and *total_attempts* are parameters.

Arguments

In Python, an **argument** is the data brought into a function when it is called. When calling *remaining_login_attempts* in the following example, the integers *3* and *2* are considered arguments:

```
remaining_login_attempts(3, 2)
```

These integers pass into the function through the parameters that were identified when defining the function. In this case, those parameters would be *maximum_attempts* and *total_attempts*. *3* is in the first position, so it passes into *maximum_attempts*. Similarly, *2* is in the second position and passes into *total_attempts*.

Return statements

When defining functions in Python, you use return statements if you want the function to return output. The *return* keyword is used to return information from a function.

The *return* keyword appears in front of the information that you want to return. In the following example, it is before the calculation of how many login attempts remain:

```
def remaining_login_attempts(maximum_attempts, total_attempts):

    return maximum_attempts - total_attempts
```

Note: The *return* keyword is not a function, so you should not place parentheses after it.

Return statements are useful when you want to store what a function returns inside of a variable to use elsewhere in the code. For example, you might use this variable for calculations or within conditional statements. In the following example, the information returned from the call to *remaining_login_attempts* is stored in a variable called *remaining_attempts*. Then, this variable is used in a conditional that prints a "*Your account is locked*" message when *remaining_attempts* is less than or equal to *0*. You can run this code to explore its output:

```
def remaining_login_attempts(maximum_attempts, total_attempts):
    return maximum_attempts - total_attempts
remaining_attempts = remaining_login_attempts(3, 3)
if remaining_attempts <= 0:
    print("Your account is locked")
```

```
Your account is locked
```

In this example, the message prints because the calculation in the function results in *0*.

Note: When Python encounters a *return* statement, it executes this statement and then exits the function. If there are lines of code that follow the *return* statement within the function, they will not be run. The previous example didn't contain any lines of code after the *return* statement, but this might apply in other functions, such as one containing a conditional statement.

Global and local variables

To better understand how functions interact with variables, you should know the difference between global and local variables.

When defining and calling functions, you're working with local variables, which are different from the variables you define outside the scope of a function.

Global variables

A **global variable** is a variable that is available through the entire program. Global variables are assigned outside of a function definition. Whenever that variable is called, whether inside or outside a function, it will return the value it is assigned.

For example, you might assign the following variable at the beginning of your code:

```
device_id = "7ad2130bd"
```

Throughout the rest of your code, you will be able to access and modify the *device_id* variable in conditionals, loops, functions, and other syntax.

Local variables

A **local variable** is a variable assigned within a function. These variables cannot be called or accessed outside of the body of a function. Local variables include parameters as well as other variables assigned within a function definition.

In the following function definition, *total_string* and *name* are local variables:

```
def greet_employee(name):

    total_string = "Welcome" + name

    return total_string
```

The variable *total_string* is a local variable because it's assigned inside of the function. The parameter *name* is a local variable because it is also created when the function is defined.

Whenever you call a function, Python creates these variables temporarily while the function is running and deletes them from memory after the function stops running.

This means that if you call the *greet_employee()* function with an argument and then use the *total_string* variable outside of this function, you'll get an error.

Best practices for global and local variables

When working with variables and functions, it is very important to make sure that you only use a certain variable name once, even if one is defined globally and the other is defined locally.

When using global variables inside functions, functions can access the values of a global variable. You can run the following example to explore this:

```
username = "elarson"
def identify_user():
    print(username)
identify_user()
```

```
elarson
```

The code block returns "*elarson*" even though that name isn't defined locally. The function accesses the global variable. If you wanted the *identify_user()* function to accommodate other usernames, you would have to reassign the global username variable outside of the function. This isn't good practice. A better way to pass different values into a function is to use a parameter instead of a global variable.

There's something else to consider too. If you reuse the name of a global variable within a function, it will create a new local variable with that name. In other words, there will be both a global variable with that name and a local variable with that name, and they'll have different values. You can consider the following code block:

```
username = "elarson"
print("1:" + username)
def greet():
    username = "bmoreno"
    print("2:" + username)
greet()
print("3:" + username)
```

```
1:elarson
2:bmoreno
3:elarson
```

The first print statement occurs before the function, and Python returns the value of the global *username* variable, "*elarson*". The second print statement is within the function, and it returns the value of the local *username* variable, which is "*bmoreno*". But this doesn't change the value of the global variable, and when *username* is printed a third time after the function call, it's still "*elarson*".

Due to this complexity, it's best to avoid combining global and local variables within functions.

Key takeaways

Working with variables in functions requires understanding various concepts. A parameter is an object that is included in a function definition for use in that function, an argument is the data brought into a function when it is called, and the *return* keyword is used to return information from a function. Additionally, global variables are variables accessible throughout the program, and local variables are parameters and variables assigned within a function that aren't usable outside of a function. It's important to make sure your variables all have distinct names, even if one is a local variable and the other is a global variable.

Explore built-in functions

Now that we know how to create our own functions, let's also explore a few of Python's built-in functions.
As we discussed previously, built-in functions are functions that exist within Python and can be called directly.
Our only job is to call them by their name!
And we've already described a few throughout the course; for example, Python's print() and type() functions.
Let's quickly review those two built-in functions before learning about new ones!
First, print() outputs a specified object to the screen.
And then, the type() function returns the data type of its input.
Previously, we've been using functions independently from one another.
For example, we asked Python to print something, or we asked Python to return the data type of something.
As we begin to explore built-in functions, we'll often need to use multiple functions together.
We can do this by passing one function into another as an argument.
For example, in this line of code, Python first returns the data type of "Hello" as a string.
Then, this returned value is passed into the print() function.
This means the data type of string will be printed to the screen.
print() and type() are not the only functions you'll see used together in this way.
In all cases, the general syntax is the same.
The inner function is processed first and then its returned value is passed to the outer function.
Let's consider another aspect of working with built-in functions.
When working with functions, you have to understand what their expected inputs and outputs are.
Some functions only expect specific data types and will return a type error if you use the wrong one.
Other functions need a specific amount of parameters or return a different data type.
The print() function, for example can take in any data type as its input.
It can also take in any number of parameters, even ones with different data types.
Let's explore the input and output of the print() function.
We'll enter three arguments.
The first contains string data.
Then, a comma is used to separate this from the second argument.
This second argument is an integer.
Finally, after another comma, our third argument is another string.
Now, let's run this code.
Perfect!
This printed out just as expected!
The type() function also takes in all data types, but it only accepts one parameter.
Let's explore this input and output too.
Our first line of code will first determine the data type of the word "security" and then pass what it

returns into a print() function.
And the second line of code will do the same thing with the value of 73.2.
Now, let's run this and see what happens.
Python first returns output that tells us that the word "security" is string data.
Next, it returns another line of output that tells us that 73.2 is float data.
Now, we know what to consider before using a built-in function.
We have to know exactly how many parameters it requires and what data types they can be.
We also need to know what kind of output it produces.
Let's learn a couple of new built-in functions and think about this.
We'll start with max().
The max() function returns the largest numeric input passed into it.
It doesn't have a defined number of parameters that it accepts.
Let's explore the max() function.
We'll pass three arguments into max() in the form of variables.
So let's first define those variables.
We'll set the value of a to 3, b to 9, and c to 6.
Then, we'll pass these variables into the max() function and print them.
Let's run this.
It tells us the highest value among those is 9.
Now, let's study one more built-in function: the sorted() function.
The sorted() function sorts the components of a list.
This function can be very useful in a security setting.
When working with lists, we often have to sort them.
With lists of numbers, we sort them from smallest to largest or the other way around.
With lists of string data, we might need to sort them alphabetically.
Imagine you have a list that contains usernames in your organization and you wanted to sort them alphabetically.
Let's use Python's sorted() function for this.
We'll specify our list through a variable named usernames.
In this list, we'll include all of the usernames we want to sort.
Now, we'll use the sorted() function to sort these names by passing the usernames variable into it.
And then we'll pass its output into the print statement so it can be displayed on the screen.
When we run it, everything is now in order!
These are just a few of the built-in functions available for your use.
As you work more in Python, you'll become familiar with others that can help you in your programs.

Work with built-in functions

Previously, you explored built-in functions in Python, including *print()*, *type()*, *max()*, and *sorted()*. **Built-in functions** are functions that exist within Python and can be called directly. In this reading, you’ll explore these further and also learn about the *min()* function. In addition, you'll review how to pass the output of one function into another function.

print()

The *print()* function outputs a specified object to the screen. The *print()* function is one of the most commonly used functions in Python because it allows you to output any detail from your code.

To use the *print()* function, you pass the object you want to print as an argument to the function. The *print()* function takes in any number of arguments, separated by a comma, and prints all of them. For example, you can run the following code that prints a string, a variable, another string, and an integer together:

```
month = "September"
print("Investigate failed login attempts during", month, "if more than", 100)
```

```
Investigate failed login attempts during September if more than 100
```

type()

The *type()* function returns the data type of its argument. The *type()* function helps you keep track of the data types of variables to avoid errors throughout your code.

To use it, you pass the object as an argument, and it returns its data type. It only accepts one argument. For example, you could specify *type("security")* or *type(7)*.

Passing one function into another

When working with functions, you often need to pass them through *print()* if you want to output the data type to the screen. This is the case when using a function like *type()*. Consider the following code:

```
print(type("This is a string"))
```

```
<class 'str'>
```

It displays *str*, which means that the argument passed to the *type()* function is a string. This happens because the *type()* function is processed first and its output is passed as an argument to the *print()* function.

max() and min()

The *max()* function returns the largest numeric input passed into it. The *min()* function returns the smallest numeric input passed into it.

The *max()* and *min()* functions accept arguments of either multiple numeric values or of an iterable like a list, and they return the largest or smallest value respectively.

In a cybersecurity context, you could use these functions to identify the longest or shortest session that a user logged in for. If a specific user logged in seven times during a week, and you stored their access times in minutes in a list, you can use the *max()* and *min()* functions to find and print their longest and shortest sessions:

```
time_list = [12, 2, 32, 19, 57, 22, 14]
print(min(time_list))
print(max(time_list))
```

```
2
57
```

sorted()

The *sorted()* function sorts the components of a list. The *sorted()* function also works on any iterable, like a string, and returns the sorted elements in a list. By default, it sorts them in ascending order. When given an iterable that contains numbers, it sorts them from smallest to largest; this includes iterables that contain numeric data as well as iterables that contain string data beginning with numbers. An iterable that contains strings that begin with alphabetic characters will be sorted alphabetically.

The *sorted()* function takes an iterable, like a list or a string, as an input. So, for example, you can use the following code to sort the list of login sessions from shortest to longest:

```
time_list = [12, 2, 32, 19, 57, 22, 14]
print(sorted(time_list))
```

```
[2, 12, 14, 19, 22, 32, 57]
```

This displays the sorted list.

The *sorted()* function does not change the iterable that it sorts. The following code illustrates this:

```
time_list = [12, 2, 32, 19, 57, 22, 14]
print(sorted(time_list))
print(time_list)
```

```
[2, 12, 14, 19, 22, 32, 57]
[12, 2, 32, 19, 57, 22, 14]
```

The first *print()* function displays the sorted list. However, the second *print()* function, which does not include the *sorted()* function, displays the list as assigned to *time_list* in the first line of code.

One more important detail about the *sorted()* function is that it cannot take lists or strings that have elements of more than one data type. For example, you can't use the list *[1, 2, "hello"]*.

Key takeaways

Built-in functions are powerful tools in Python that allow you to perform tasks with one simple command. The *print()* function prints its arguments to the screen, the *type()* function returns the data type of its argument, the *min()* and *max()* functions return the smallest and largest values of an iterable respectively, and *sorted()* organizes its argument.

Resources for more information

These were just a few of Python's built-in functions. You can continue learning about others on your own:

- The Python Standard Library documentation

: A list of Python's built-in functions and information on how to use them

Activity: Define and call a function

Introduction

As a security analyst, when you're writing out Python code to automate a certain task, you'll often find yourself needing to reuse the same block of code more than once. This is why functions are important. You can call that function whenever you need the computer to execute those steps. Python not only has built-in functions that have already been defined, but also provides the tools for users to define their own functions. Security analysts often define and call functions in Python to automate series of tasks.

In this lab, you'll practice defining and calling functions in Python.

Tips for completing this lab

As you navigate this lab, keep the following tips in mind: - `### YOUR CODE HERE ###` indicates where you should write code. Be sure to replace this with your own code before running the code cell. - Feel free to open the hints for additional guidance as you work on each task. - To enter your answer to a question, double-click the markdown cell to edit. Be sure to replace the "[Double-click to enter your responses here.]" with your own answer. - You can save your work manually by clicking File and then Save in the menu bar at the top of the notebook. - You can download your work locally by clicking File and then Download and then specifying your preferred file format in the menu bar at the top of the notebook.

Scenario

Writing functions in Python is a useful skill in your work as a security analyst. In this lab, you'll define and a call a function that displays an alert about a potential security issue. Also, you'll work with a list of employee usernames, creating a function that converts the list into one string.

Task 1

The following code cell contains a user-defined function named `alert()`.

For this task, analyze the function definition, and make note of your observations.

You won't need to run the cell in order to answer the question that follows. But if you do run the cell, note that it will not produce an output because the function is just being defined here.

```
# Define a function named `alert()`

def alert():
    print("Potential security issue. Investigate further.")
```

Hint 1

When analyzing the function definition, make sure to observe the function body, which is the indented block of code after the function header. The function body tells you what the function does.

Question 1

Summarize what the user-defined function above does in your own words. Think about what the output would be if this function were called.

it prints Potential security issue. Investigate further. IF the def function is called

Task 2

For this task, call the `alert()` function that was defined earlier and analyze the output.

Be sure to replace the `### YOUR CODE HERE ###` with your own code before running the following cell.

```
# Define a function named `alert()`

def alert():
    print("Potential security issue. Investigate further.")

# Call the `alert()` function

alert()
```

```
Potential security issue. Investigate further.
```

Hint 1

To call the function, write `alert()` after the function definition. Note that the function can be called only after it's defined.

Question 2

What are the advantages of placing this code in a function rather than running it directly?

its re-usable

Task 3

Functions can include other components that you've already worked with. The following code cell contains a variation of the `alert()` function that now uses a `for` loop to display the alert message multiple times.

For this task, call the new `alert()` function and observe the output.

Be sure to replace the `### YOUR CODE HERE ###` with your own code before running the following cell.

```
# Define a function named `alert()`

def alert():
    for i in range(3):
        print("Potential security issue. Investigate further.")

# Call the `alert()` function

alert()
```

```
Potential security issue. Investigate further.
Potential security issue. Investigate further.
Potential security issue. Investigate further.
```

Hint 1

To call the function, write `alert()` after the function definition. Note that the function can be called only after it's defined.

Question 3

How does the output above compare to the output from calling the previous version of the `alert()` function? How are the two definitions of the function different?

its able to be called 3 times in a row instead of once in the 2nd example

Task 4

In the next part of your work, you're going to work with a list of approved usernames, representing users who can enter a system. You'll be developing a function that helps you convert the list of approved usernames into one big string. Structuring this data differently enables you to work with it in different ways. For example, structuring the usernames as a list allows you to easily add or remove a username from it. In contrast, structuring it as a string allows you to easily place its contents into a text file.

For this task, start defining a function named `list_to_string()`. Write the function header.

Be sure to replace the `### YOUR CODE HERE ###` with your own code. Note that running this cell will produce an error since this cell will just contain the function header; you'll write the function body and complete the function definition in a later task.

```
# Define a function named `list_to_string()`

def list_to_string():
    string = "string"
    print(string)
list_to_string
```

```
<function __main__.list_to_string()>
```

Hint 1

To write the function header, start with the `def` keyword, followed by the name of the function, parentheses, and a colon.

Task 5

Now you'll begin to develop the body of the `list_to_string()` function.

In the following code cell, you're provided a list of approved usernames, stored in a variable named `username_list`. Your task is to complete the body of the `list_to_string()` function. Recall that the body of a function must be indented. To complete the function body, write a loop that iterates through the elements of the `username_list` and displays each element. Then, call the function and run the cell to observe what happens.

Be sure to replace each `### YOUR CODE HERE ###` with your own code before running the following cell.

```
# Define a function named `list_to_string()`

def list_to_string():

  # Store the list of approved usernames in a variable named `username_list`

    username_list = ["elarson", "bmoreno", "tshah", "sgilmore", "eraab", "gesparza", "alevitsk", "wjaffrey"]
    for username in username_list:
        print(username)
  # Write a for loop that iterates through the elements of `username_list` and displays each element



# Call the `list_to_string()` function

list_to_string()
```

```
elarson
bmoreno
tshah
sgilmore
eraab
gesparza
alevitsk
wjaffrey
```

Hint 1

The `for` loop in the body of the `list_to_string()` function must iterate through the elements of `username_list`. So, use the `username_list` variable to complete the `for` loop condition.

Hint 2

In each iteration of the `for` loop, an element of `username_list` should be displayed. The loop variable `i` represents each element of `username_list`. To complete the `print()` statement inside the `for` loop, pass`i` to the `print()` function call.

Hint 3

To call the function, write `list_to_string()` after the function definition. Recall that the function can be called only after it's defined.

Question 4

What do you observe from the output above?

IT prints all the usernames in the list

Task 6

String concatenation is a powerful concept in coding. It allows you to combine multiple strings together to form one large string, using the addition operator (`+`). Sometimes analysts need to merge individual pieces of data into a single string value. In this task, you'll use string concatenation to modify how the `list_to_string()` function is defined.

In the following code cell, you're provided a variable named `sum_variable` that initially contains an empty string. Your task is to use string concatenation to combine the usernames from the `username_list` and store the result in `sum_variable`.

In each iteration of the `for` loop, add the current element of `username_list` to `sum_variable`. At the end of the function definition, write a `print()` statement to display the value of `sum_variable` at that stage of the process. Then, run the cell to call the `list_to_string()` function and examine its output.

Be sure to replace each `### YOUR CODE HERE ###` with your own code before running the following cell.

```
# Define a function named `list_to_string()`

def list_to_string():

  # Store the list of approved usernames in a variable named `username_list`

    username_list = ["elarson", "bmoreno", "tshah", "sgilmore", "eraab", "gesparza", "alevitsk", "wjaffrey"]

  # Assign `sum_variable` to an empty string

    sum_variable = ""

  # Write a for loop that iterates through the elements of `username_list` and displays each element

    for username in username_list:
        sum_variable += username

    # Add a comma and space after each username, except the last one
        if username != username_list[-1]:
            sum_variable += ", "
    # Display the value of `sum_variable`
        print(sum_variable)

# Call the `list_to_string()` function

list_to_string()
```

```
elarson,
elarson, bmoreno,
elarson, bmoreno, tshah,
elarson, bmoreno, tshah, sgilmore,
elarson, bmoreno, tshah, sgilmore, eraab,
elarson, bmoreno, tshah, sgilmore, eraab, gesparza,
elarson, bmoreno, tshah, sgilmore, eraab, gesparza, alevitsk,
elarson, bmoreno, tshah, sgilmore, eraab, gesparza, alevitsk, wjaffrey
```

Hint 1

Inside the `for` loop, complete the line that updates the `sum_variable` in each iteration. The loop variable `i` represents each element of `username_list`. Since you need to add the current element to the current value of `sum_variable`, place `i` after the addition operator `(+)`.

Hint 2

Use the `print()` function to display the value of `sum_variable`. Make sure to pass in `sum_variable` to the call to `print()`.

Question 5

What do you observe from the output above?

Username_list[-1] represents the end of the list

Task 7

In this final task, you'll modify the code you wrote previously to improve the readability of the output.

This time, in the definition of the `list_to_string()` function, add a comma and a space (`", "`) after each username. This will prevent all the usernames from running into each other in the output. Adding a comma helps clearly separate one username from the next in the output. Adding a space following the comma as an additional separator between one username and the next makes it easier to read the output. Then, call the function and run the cell to observe the output.

Be sure to replace each `### YOUR CODE HERE ###` with your own code before running the following cell.

```
# Define a function named `list_to_string()`

def list_to_string():

  # Store the list of approved usernames in a variable named `username_list`

    username_list = ["elarson", "bmoreno", "tshah", "sgilmore", "eraab", "gesparza", "alevitsk", "wjaffrey"]

  # Assign `sum_variable` to an empty string
```

```
    sum_variable = ""

  # Write a for loop that iterates through the elements of `username_list` and displays each element

    for username in username_list:
        sum_variable += username

    # Add a comma and space after each username, except the last one
        if username != username_list[-1]:
            sum_variable += ", "
    # Display the value of `sum_variable`
        print(sum_variable)

# Call the `list_to_string()` function
list_to_string()
```

```
elarson,
elarson, bmoreno,
elarson, bmoreno, tshah,
elarson, bmoreno, tshah, sgilmore,
elarson, bmoreno, tshah, sgilmore, eraab,
elarson, bmoreno, tshah, sgilmore, eraab, gesparza,
elarson, bmoreno, tshah, sgilmore, eraab, gesparza, alevitsk,
elarson, bmoreno, tshah, sgilmore, eraab, gesparza, alevitsk, wjaffrey
```

Hint 1

Inside the `for` loop, complete the line that updates the `sum_variable` in each iteration. The loop variable `i` represents each element of `username_list`. After the current element is added to the current value of `sum_variable`, add a string that contains a comma followed by a space. To complete this step, place `", "` after the last addition operator (`+`).

Hint 2

To call the function, write `list_to_string()` after the function definition. Note that the function can be called only after it's defined.

Question 6

What do you notice about the output from the function call this time?

its the same but the original difference was that it didnt contain the comma

Conclusion

What are your key takeaways from this lab?

def functions are cool, but also so is the if username != username_list[-1]: syntax for if you wanna do something when list ends

Modules and libraries

Hello again!
Previously, we learned about built-in functions in Python.
Built-in functions come standard with every version of Python and consist of functions such as print(), type(), max(), and many more.
To access additional pre-built functions, you can import a library.
A library is a collection of modules that provide code users can access in their programs.
All libraries are generally made up of several modules.
A module is a Python file that contains additional functions, variables, classes, and any kind of runnable code.
Think of them as saved Python files that contain useful functionality.
Modules may be made up of small and simple lines of code or be complex and lengthy in size.
Either way, they help save programmers time and make code more readable.
Now, let's focus specifically on the Python Standard Library.
The Python Standard Library is an extensive collection of usable Python code that often comes packaged with Python.
One example of a module from the Python Standard Library is the re module.
This is a useful module for a security analyst when they're tasked with searching for patterns in log files.
Another module is the csv module.
It allows you to work efficiently with CSV files.
The Python Standard Library also contains glob and os modules for interacting with the command line as well as time and datetime for working with timestamps.
These are just a few of the modules in the Python Standard Library.
In addition to what's always available through the Python Standard Library, you can also download external libraries.
A couple of examples are Beautiful Soup for parsing HTML website files and NumPy for arrays and mathematical computations.
These libraries will assist you as a security analyst in network traffic analysis, log file parsing, and complex math.
Overall, Python libraries and modules are useful because they provide pre-programmed functions and variables.
This saves time for the user.
I encourage you to explore some of the libraries and modules we discussed here and the ways they might be helpful to you as you work in Python.

Import modules and libraries in Python

Previously, you explored libraries and modules. You learned that a **module** is a Python file that contains additional functions, variables, classes, and any kind of runnable code. You also learned that a **library** is a collection of modules that provide code users can access in their programs. You were introduced to a few modules in the Python Standard Library and a couple of external libraries. In this reading, you'll learn how to import a module that exists in the Python Standard Library and use its functions. You'll also expand your understanding of external libraries.

The Python Standard Library

The **Python Standard Library** is an extensive collection of Python code that often comes packaged with Python. It includes a variety of modules, each with pre-built code centered around a particular type of task.

For example, you were previously introduced to the the following modules in the Python Standard Library:

- The *re* module, which provides functions used for searching for patterns in log files
- The *csv* module, which provides functions used when working with *.csv* files
- The *glob* and *os* modules, which provide functions used when interacting with the command line
- The *time* and *datetime* modules, which provide functions used when working with timestamps

Another Python Standard Library module is *statistics*. The *statistics* module includes functions used when calculating statistics related to numeric data. For example, *mean()* is a function in the *statistics* module that takes numeric data as input and calculates its mean (or average). Additionally, *median()* is a function in the *statistics* module that takes numeric data as input and calculates its median (or middle value).

How to import modules from the Python Standard Library

To access modules from the Python Standard Library, you need to import them. You can choose to either import a full module or to only import specific functions from a module.

Importing an entire module

To import an entire Python Standard Library module, you use the *import* keyword. The *import* keyword searches for a module or library in a system and adds it to the local Python environment. After *import*, specify the name of the module to import. For example, you can specify *import statistics* to import the *statistics* module. This will import all the functions inside of the *statistics* module for use later in your code.

As an example, you might want to use the *mean()* function from the *statistics* module to calculate the average number of failed login attempts per month for a particular user. In the following code block, the total number of failed login attempts for each of the twelve months is stored in a list called *monthly_failed_attempts*. Run this code and analyze how *mean()* can be used to calculate the average of these monthly failed login totals and store it in *mean_failed_attempts*\

```
import statistics
monthly_failed_attempts = [20, 17, 178, 33, 15, 21, 19, 29, 32, 15, 25, 19]
mean_failed_attempts = statistics.mean(monthly_failed_attempts)
print("mean:", mean_failed_attempts)
```

```
mean: 35.25
```

The output returns a mean of *35.25*. You might notice the outlying value of *178* and want to find the middle value as well. To do this through the *median()* function, you can use the following code:

```
import statistics
monthly_failed_attempts = [20, 17, 178, 33, 15, 21, 19, 29, 32, 15, 25, 19]
median_failed_attempts = statistics.median(monthly_failed_attempts)
print("median:", median_failed_attempts)
```

```
median: 20.5
```

This gives you the value of *20.5*, which might also be useful for analyzing the user's failed login attempt statistics.

Note: When importing an entire Python Standard Library module, you need to identify the name of the module with the function when you call it. You can do this by placing the module name followed by a period (.) before the function name. For example, the previous code blocks use *statistics.mean()* and *statistics.median()* to call those functions.

Importing specific functions from a module

To import a specific function from the Python Standard Library, you can use the *from* keyword. For example, if you want to import just the *median()* function from the *statistics* module, you can write *from statistics import median*.

To import multiple functions from a module, you can separate the functions you want to import with a comma. For instance, *from statistics import mean, median* imports both the *mean()* and the *median()* functions from the *statistics* module.

An important detail to note is that if you import specific functions from a module, you no longer have to specify the name of the module before those functions. You can examine this in the following code, which specifically imports only the *median()* and the *mean()* functions from the *statistics* module and performs the same calculations as the previous examples:

```
from statistics import mean, median
monthly_failed_attempts = [20, 17, 178, 33, 15, 21, 19, 29, 32, 15, 25, 19]
mean_failed_attempts = mean(monthly_failed_attempts)
print("mean:", mean_failed_attempts)
median_failed_attempts = median(monthly_failed_attempts)
print("median:", median_failed_attempts)
```

```
mean: 35.25
median: 20.5
```

It is no longer necessary to specify *statistics.mean()* or *statistics.median()* and instead the code incorporates these functions as *mean()* and *median()*.

External libraries

In addition to the Python Standard Library, you can also download external libraries and incorporate them into your Python code. For example, previously you were introduced to Beautiful Soup (*bs4*) for parsing HTML files and NumPy (*numpy*) for arrays and mathematical computations. Before using them in a Jupyter Notebook or a Google Colab environment, you need to install them first.

To install a library, such as *numpy*, in either environment, you can run the following line prior to importing the library:

%pip install numpy

This installs the library so you can use it in your notebook.

After a library is installed, you can import it directly into Python using the *import* keyword in a similar way to how you used it to import modules from the Python Standard Library. For example, after the *numpy* install, you can use this code to import it:

import numpy

Key takeaways

The Python Standard Library contains many modules that you can import, including *re*, *csv*, *os*, *glob*, *time*, *datetime*, and *statistics*. To import these modules, you must use the *import* keyword. Syntax varies depending on whether or not you want to import the entire module or just specific functions from it. External libraries can also be imported into Python, but they need to be installed first.

Code readability

Welcome back!
One of the advantages to programming in Python is that it's a very readable language.
It also helps that the Python community shares a set of guidelines that promote clean and neat code.
These are called style guides.
A style guide is a manual that informs the writing, formatting, and design of documents.
As it relates to programming, style guides are intended to help programmers follow similar conventions.
The PEP 8 style guide is a resource that provides stylistic guidelines for programmers working in Python.
PEP is short for Python Enhancement Proposals.
PEP 8 provides programmers with suggestions related to syntax.
They're not mandatory, but they help create consistency among programmers to make sure that others can easily understand our code.
It's essentially based on the principle that code is read much more often than it's written.
This is a great resource for anyone who wants to learn how to style and format their Python code in a manner consistent with other programmers.
For example, PEP 8 discusses comments.
A comment is a note programmers make about the intention behind their code.
They are inserted in computer programs to indicate what the code is doing and why.
PEP 8 gives specific recommendations, like making your comments clear and keeping them up-to-date when the code changes.
Here's an example of code without a comment.
The person who wrote it might know what's going on, but what about others who need to read it?
They might not understand the context behind the failed-attempts variable and why it prints "Account locked" if it's greater than 5.
And the original writer might need to revisit this code in the future, for example, to debug the larger program.
Without the comment, they would also be less efficient.
But in this example we've added a comment.
All the readers can quickly understand what our program and its variables are doing.
Comments should be short and right to the point.
Next, let's talk about another important aspect of code readability: indentation.
Indentation is a space added at the beginning of a line of code.
This both improves readability and ensures that code is executed properly.
There are instances when you must indent lines of code to establish connections with other lines of code.
This groups the indented lines of code together and establishes a connection with a previous line of code that isn't indented.
The body of a conditional statement is one example of this.

We need to make sure this printed statement executes only when the condition is met.
Indenting here provides this instruction to Python.
If the printed statement were not indented, Python would execute this printed statement outside of the conditional and it would always print.
This would be problematic because you would get a message that updates are needed, even if they're not.
To indent, you must add at least one space before a line of code.
Typically, programmers are two to four spaces for visual clarity.
The PEP 8 style guide recommends four spaces.
At my first engineering job, I wrote a script to help validate and launch firewall rules.
Initially, my script worked well, but it became hard to read a year later when we were trying to expand its functionality.
My programming knowledge and coding style had evolved over that year, as had the coding practices of my teammates.
Our organization did not use a coding style guide at that time, so our codes were very different, hard to read, and did not scale well.
This caused a lot of challenges and required additional work to fix.
Ensuring that code is readable and can be modified over time is why it's important for security professionals to adhere to coding style guides and why style guides are so important for organizations to utilize.
The ability to write readable code is key when working in Python.
As we head into the next part of our course, we'll continue to develop effective code practices for better readability.

Ensure proper syntax and readability in Python

Previously, you were introduced to the PEP 8 style guide and its stylistic guidelines for programmers working in Python. You also learned about how adding comments and using correct indentation makes your code more readable. Additionally, correct indentation ensures your code is executed properly. This reading explores these ideas further and also focuses on common items to check in the syntax of your code to ensure it runs.

Comments

A **comment** is a note programmers make about the intentions behind their code. Comments make it easier for you and other programmers to read and understand your code.

It’s important to start your code with a comment that explains what the program does. Then, throughout the code, you should add additional comments about your intentions behind specific sections.

When adding comments, you can add both single-line comments and multi-line comments.

Single-line comments

Single-line comments in Python begin with the (#) symbol. According to the PEP 8 style guide, it’s best practice to keep all lines in Python under 79 characters to maintain readability, and this includes comments.

Single-line comments are often used throughout your program to explain the intention behind specific sections of code. For example, this might be when you're explaining simpler components of your program, such as the following *for* loop:

```
# Print elements of 'computer_assets' list

computer_assets = ["laptop1", "desktop20", "smartphone03"]

for asset in computer_assets:

    print(asset)
```

Note: Comments are important when writing more complex code, like functions, or multiple loops or conditional statements. However, they're optional when writing less complex code like reassigning a variable.

Multi-line comments

Multi-line comments are used when you need more than 79 characters in a single comment. For example, this might occur when defining a function if the comment describes its inputs and their data types as well as its output.

There are two commonly used ways of writing multi-line comments in Python. The first is by using the hashtag (#) symbol over multiple lines:

```
# remaining_login_attempts() function takes two integer parameters,

# the maximum login attempts allowed and the total attempts made,

# and it returns an integer representing remaining login attempts

def remaining_login_attempts(maximum_attempts, total_attempts):

    return maximum_attempts - total_attempts
```

Another way of writing multi-line comments is by using documentation strings and not assigning them to a variable. Documentation strings, also called docstrings, are strings that are written over multiple lines and are used to document code. To create a documentation string, use triple quotation marks (""" """).

You could add the comment to the function in the previous example in this way too:

```
"""

remaining_login_attempts() function takes two integer parameters,

the maximum login attempts allowed and the total attempts made,

and it returns an integer representing remaining login attempts

"""
```

Correct indentation

Indentation is space added at the beginning of a line of code. In Python, you should indent the body of conditional statements, iterative statements, and function definitions. Indentation is not only necessary for Python to interpret this syntax properly, but it can also make it easier for you and other programmers to read your code.

The PEP 8 style guide recommends that indentations should be four spaces long. For example, if you had a conditional statement inside of a *while* loop, the body of the loop would be indented four spaces and the body of the conditional would be indented four spaces beyond that. This means the conditional would be indented eight spaces in total.

```
count = 0

login_status = True

while login_status == True:

    print("Try again.")

    count = count + 1

    if count == 4:

        login_status = False
```

Maintaining correct syntax

Syntax errors involve invalid usage of the Python language. They are incredibly common with Python, so focusing on correct syntax is essential in ensuring that your code runs. Awareness of common errors will help you more easily fix them.

Syntax errors often occur because of mistakes with data types or in the headers of conditional or iterative statements or of function definitions.

Data types

Correct syntax varies depending on data type:

- Place string data in quotation marks.
  - Example: *username* = "*bmoreno*"
- Do not add quotation marks around integer, float, or Boolean data types.
  - Examples: *login_attempts = 5*, *percentage_successful = .8*, *login_status = True*
- Place lists in brackets and separate the elements of a list with commas.
  - Example: *username_list = ["bmoreno", "tshah"]*

Colons in headers

The header of a conditional or iterative statement or of a function definition must end with a colon. For example, a colon appears at the end of the header in the following function definition:

```
def remaining_login_attempts(maximum_attempts, total_attempts):

    return maximum_attempts - total_attempts
```

Key takeaways

The PEP 8 style guide provides recommendations for writing code that can be easily understood and read by other Python programmers. In order to make your intentions clear, you should incorporate comments into your code. Depending on the length of the comment, you can follow conventions for single-line or multi-line comments. It's also important to use correct indentation; this ensures your code will run as intended and also makes it easier to read. Finally, you should also be aware of common syntax issues so that you can more easily fix them.

Resources for more information

Learning to write readable code can be challenging, so make sure to review the PEP 8 style guide and learn about additional aspects of code readability.

- PEP 8 - Style Guide for Python Code
- : The PEP 8 style guide contains all standards of Python code. When reading this guide, it's helpful to use the table of contents to navigate through the concepts you haven't learned yet.

Dorsa: Use Python efficiently on a cybersecurity team

Hi, my name is Dorsa and I'm a security engineer.
What I love the most about my job is that I get to look at different infrastructures and system designs on a daily basis.
One piece of advice for individuals who are starting out in their cybersecurity profession, it's very important to work collaboratively in Python and one of the key aspects of that is to listen to the feedback that your team members provide.
Python allows for many different ways of accessing different information.
When you share Python code snippets amongst your team members, it allows the code to be more uniform and the coding process to be more efficient.
It makes the code base a lot more readable and it allows for other engineers to work on your code after you, I've seen many examples of when collaboratively written Python code has been helpful in the industry.
One of the examples is when at Google we wrote a collaboratively written code base which allowed for an onboarding process to be reduced from six or seven hours to a couple of minutes.
Collaboration was a key part of this process because otherwise it would have taken many many years for one single individual to write it.
One individual is not able to understand every fine detail of each system and if we don't have multiple engineers working on it, they would have made this process a lot more difficult.
Communication is very important when you're working in a team and especially if you're developing code in Python, you need to express whether you need help throughout the process because your team members are there to ensure that you are successful.
At the end of the day, your success means that your team is also successful.
As you advance in your career as someone who writes code in Python, you'll realize that there are a lot of functions and methods that are just sticking around on the internet and you will be able to find them with a quick search and those methods will come in handy and you will be able to reuse them for your pieces of code.
A really good resource for you to learn new skills and expand your Python coding skills is to talk to your colleagues, attend meetups, talk to different security professionals who don't work at your company because everyone has an insight on how to make your coding skills, especially in cybersecurity better.
Hi, my name is Dorsa and I'm a security engineer.
What I love the most about my job is that I get to look at different infrastructures and system designs on a daily basis.
One piece of advice for individuals who are starting out in their cybersecurity profession, it's very important to work collaboratively in Python and one of the key aspects of that is to listen to the feedback that your team members provide.
Python allows for many different ways of accessing different information.

When you share Python code snippets amongst your team members, it allows the code to be more uniform and the coding process to be more efficient.
It makes the code base a lot more readable and it allows for other engineers to work on your code after you, I've seen many examples of when collaboratively written Python code has been helpful in the industry.
One of the examples is when at Google we wrote a collaboratively written code base which allowed for an onboarding process to be reduced from six or seven hours to a couple of minutes.
Collaboration was a key part of this process because otherwise it would have taken many many years for one single individual to write it.
One individual is not able to understand every fine detail of each system and if we don't have multiple engineers working on it, they would have made this process a lot more difficult.
Communication is very important when you're working in a team and especially if you're developing code in Python, you need to express whether you need help throughout the process because your team members are there to ensure that you are successful.
At the end of the day, your success means that your team is also successful.
As you advance in your career as someone who writes code in Python, you'll realize that there are a lot of functions and methods that are just sticking around on the internet and you will be able to find them with a quick search and those methods will come in handy and you will be able to reuse them for your pieces of code.
A really good resource for you to learn new skills and expand your Python coding skills is to talk to your colleagues, attend meetups, talk to different security professionals who don't work at your company because everyone has an insight on how to make your coding skills, especially in cybersecurity better.

Wrap-up; Reference guide: Python concepts from module 2

Great work on making it this far in the Python course!
You've put in a lot of work and effort towards learning more about how you can use Python effectively and efficiently.
Let's quickly recap the concepts you learned throughout the videos.
First, we started by understanding the role of functions in Python.
They can save you a lot of time!
You learned how to incorporate built-in functions and how to develop your own function to meet your needs.
We then shifted our focus towards modules and libraries, which gave us access to a lot more functions than those built into Python.
Lastly, we moved to learning about code readability and best practices to write clean, understandable code.
With these understandings, you're ready to learn how powerful Python can really be for task automation and how it can help you going forward as a security analyst.
Thank you for taking the time to go through this course with me.
I'll meet you in the next videos!

Reference guide: Python concepts from module 2

Google Cybersecurity Certificate

User-defined functions

The following keywords are used when creating user-defined functions.

`def`

Placed before a function name to define a function

```
def greet_employee():
```

Defines the `greet_employee()` function

```
def calculate_fails(total_attempts, failed_attempts):
```

Defines the `calculate_fails()` function, which includes the two parameters of `total_attempts` and `failed_attempts`

`return`

Used to return information from a function; when Python encounters this keyword, it exits the function after returning the information

```
def calculate_fails(total_attempts, failed_attempts):
    fail_percentage = failed_attempts / total_attempts
    return fail_percentage
```

Returns the value of the `fail_percentage` variable from the `calculate_fails()` function

Built-in functions

The following built-in functions are commonly used in Python.

`max()`

Returns the largest numeric input passed into it

```
print(max(10, 15, 5))
```

Returns 15 and outputs this value to the screen

`min()`

Returns the smallest numeric input passed into it

```
print(min(10, 15, 5))
```

Returns 5 and outputs this value to the screen

`sorted()`

Sorts the components of a list (or other iterable)

```
print(sorted([10, 15, 5]))
```

Sorts the elements of the list from smallest to largest and outputs the sorted list of `[5, 10, 15]` to the screen

```
print(sorted(["bmoreno", "tshah", "elarson"]))
```

Sorts the elements in the list in alphabetical order and outputs the sorted list of `["bmoreno", "elarson", "tshah"]` to the screen

Importing modules and libraries

The following keyword is used to import a module from the Python Standard Library or to import an external library that has already been installed.

`import`

Searches for a module or library in a system and adds it to the local Python environment

```
import statistics
```

Imports the `statistics` module and all of its functions from the Python Standard Library

```
from statistics import mean
```

Imports the `mean()` function of the `statistics` module from the Python Standard Library

```
from statistics import mean, median
```

Imports the `mean()` and `median()` functions of the `statistics` module from the Python Standard Library

Comments

The following syntax is used to create a comment. (A comment is a note programmers make about the intention behind their code.)

`#`

Starts a line that contains a Python comment

```
# Print approved usernames
```

Contains a comment that indicates the purpose of the code that follows it is to print approved usernames

`"""` (documentation strings)

Starts and ends a multi-line string that is often used as a Python comment; multi-line comments are used when you need more than 79 characters in a single comment

```
"""
The estimate_attempts() function takes in a monthly
     login attempt total and a number of months and
     returns their product.
"""
```

Contains a multi-line comment that indicates the purpose of the `estimate_attempts()` function

Glossary terms from module 2

Terms and definitions from Course 7, Module 2

Argument (Python): The data brought into a function when it is called

Built-in function: A function that exists within Python and can be called directly

Comment: A note programmers make about the intention behind their code

Function: A section of code that can be reused in a program

Global variable: A variable that is available through the entire program

Indentation: Space added at the beginning of a line of code

Library: A collection of modules that provide code users can access in their programs

Local variable: A variable assigned within a function

Module: A Python file that contains additional functions, variables, classes, and any kind of runnable code

Parameter (Python): An object that is included in a function definition for use in that function

PEP 8 style guide: A resource that provides stylistic guidelines for programmers working in Python

Python Standard Library: An extensive collection of Python code that often comes packaged with Python

Return statement: A Python statement that executes inside a function and sends information back to the function call

Style guide: A manual that informs the writing, formatting, and design of documents

User-defined function: A function that programmers design for their specific needs