module 4 (36% of the course lmfao)

- Welcome to module 4
- Automate cybersecurity tasks with Python
- Essential Python components for automation
- Clancy: Continual learning and Python
- Access a text file in Python
- Import files into Python
- Parse a text file in Python
- Work with files in Python
- Develop a parsing algorithm in Python
- Portfoleo project: Algorythm for file updates in Python
- Debugging strategies
- Matt: Learning from mistakes
- Apply debugging strategies
- Explore debugging techniques
- Wrap-up
- Reference guide: Python concepts from module 4
- Terms and definitions from Course 7, Module 4
- Course wrap-up
- Reference guide: Python concepts from Course 7
- Terms and definitions from Course 7

Welcome to module 4

We've learned a lot about Python together already, and we still have more to cover.
In this section, we're going to explore how a security analyst like yourself puts Python into practice.
As a security analyst, you will likely work with security logs that capture information on various system activities.
These logs are often very large and hard to quickly interpret.
But Python can easily automate these tasks and make things much more efficient.
So first, we'll focus on opening and reading files in Python.
This includes log files.
We'll then explore parsing files.
This means you'll be able to work with files in ways that provide you with the security-related information that you're targeting.
Finally, part of writing code is debugging code.
It's important to be able to interpret error messages to make your code work.
We'll cover common types of Python errors and ways to resolve them.
Overall, after completing this section, you'll have a better understanding of Python and how as a security analyst you can use it.
I can't wait to join you.

Automate cybersecurity tasks with Python

Automation is a key concern in the security profession.
For example, it would be difficult to monitor each individual attempt to access the system.
For this reason, it's helpful to automate the security controls put in place to keep malicious actors out of the system.
And it's also helpful to automate the detection of unusual activity.
Python is great for automation.
Let's explore three specific examples of this.
First, imagine you're a security analyst for a health care company that stores confidential patient records in a database server.
Your company wants to implement additional controls to protect this information.
In order to enhance the security of the records, you decide to implement a timeout policy that locks out a user if they spent more than three minutes logging into the database.
This is because it's possible that if a user is spending too much time, it could be that they are guessing the password.
To do this, you can use Python to identify when a user has entered a username and start tracking the time until this user enters the correct password.
Now, let's cover a different example.
This time, imagine you are a security analyst working at a law firm.
There have recently been some ongoing security attacks where threat actors hack into employee accounts and attempt to steal client information.
They then threaten to use this maliciously.
So the security team is working to target all security vulnerabilities that allow these attackers to break into the company's databases.
You personally are responsible for tracking all user logins by checking their login timestamp, IP address, and location of login.
For example, if a user logs in during the early hours of the morning, they should be flagged.
Also, if they are logging in from a location that's not one of the two established work zones, you must flag their account.
Finally, if a user is simultaneously logged in from two different IP addresses, you must flag their account.
Python can help you keep track of and analyze all of this different login information.
Let's consider one final example.
Imagine you are a security analyst working at a large organization.
Recently, this organization has increased security measures to make sure all customer-facing applications are better protected.
Since there is a password to access these applications, they want to monitor all password login attempts for suspicious activity.
One sign of suspicious activity is having several failed login attempts within a short amount of

time.
You need to flag users if they had more than three login failures in the last 30 minutes.
One way you could do this in Python is by parsing a static txt log file with all user login attempts to each machine.
Python could structure the information in this file, including the username, IP address, timestamp, and login status.
It could then use conditionals to determine if a user needs to be flagged.
These are just a few examples of how a security analyst might apply Python in their day-to-day work.
I hope you are as excited as I am to create solutions for security problems.

Essential Python components for automation

Throughout this course, you explored coding in Python. You've focused on variables, conditional statements, iterative statements, functions, and a variety of ways to work with strings and lists. In this reading, you will explore why these are all essential components when automating tasks through Python, and you'll be introduced to another necessary component: working with files.

Automating tasks in Python

Automation is the use of technology to reduce human and manual effort to perform common and repetitive tasks. As a security analyst, you will primarily use Python to automate tasks.

You have encountered multiple examples of how to use Python for automation in this course, including investigating logins, managing access, and updating devices.

Automating cybersecurity-related tasks requires understanding the following Python components that you've worked with in this course:

Variables

A **variable** is a container that stores data. Variables are essential for automation. Without them, you would have to individually rewrite values for each action you took in Python.

Conditional statements

A **conditional statement** is a statement that evaluates code to determine if it meets a specified set of conditions. Conditional statements allow you to check for conditions before performing actions. This is much more efficient than manually evaluating whether to apply an action to each separate piece of data.

Iterative statements

An **iterative statement** is code that repeatedly executes a set of instructions. You explored two kinds of iterative statements: *for* loops and *while* loops. In both cases, they allow you to perform the same actions a certain number of times without the need to retype the same code each time. Using a *for* loop allows you to automate repetition of that code based on a sequence, and using a

while loop allows you to automate the repetition based on a condition.

Functions

A **function** is a section of code that can be reused in a program. Functions help you automate your tasks by reducing the need to incorporate the same code multiple places in a program. Instead, you can define the function once and call it wherever you need it.

You can develop your own functions based on your particular needs. You can also incorporate the built-in functions that exist directly in Python without needing to manually code them.

Techniques for working with strings

String data is one of the most common data types that you'll encounter when automating cybersecurity tasks through Python, and there are a lot of techniques that make working with strings efficient. You can use bracket notation to access characters in a string through their indices. You can also use a variety of functions and methods when working with strings, including *str()*, *len()*, and *.index()*.

Techniques for working with lists

List data is another common data type. Like with strings, you can use bracket notation to access a list element through its index. Several methods also help you with automation when working with lists. These include *.insert()*, *.remove()*, *.append()*, and *.index()*.

Example: Counting logins made by a flagged user

As one example, you may find that you need to investigate the logins of a specific user who has been flagged for unusual activity. Specifically, you are responsible for counting how many times this user has logged in for the day. If you are given a list identifying the username associated with each login attempt made that day, you can automate this investigation in Python.

To automate the investigation, you'll need to incorporate the following Python components:

- A *for* loop will allow you to iterate through all the usernames in the list.
- Within the *for* loop, you should incorporate a conditional statement to examine whether each username in the list matches the username of the flagged user.
- When the condition evaluates to *True*, you also need to increment a counter variable that keeps track of the number of times the flagged user appears in the list.

Additionally, if you want to reuse this code multiple times, you can incorporate it into a function. The function can include parameters that accept the username of the flagged user and the list to iterate through. (The list would contain the usernames associated with all login attempts made that day.) The function can use the counter variable to return the number of logins for that flagged user.

Working with files in Python

One additional component of automating cybersecurity-related tasks in Python is understanding how to work with files. Security-related data will often be initially found in log files. A **log** is a record of events that occur within an organization's systems. In logs, lines are often appended to the record as time progresses.

Two common file formats for security logs are *.txt* files and *.csv* files. Both *.txt* and *.csv* files are types of text files, meaning they contain only plain text. They do not contain images and do not specify graphical properties of the text, including font, color, or spacing. In a *.csv* file, or a "comma-separated values" file, the values are separated by commas. In a *.txt* file, there is not a specific format for separating values, and they may be separated in a variety of ways, including spaces.

You can easily extract data from *.txt* and *.csv* files. You can also convert both into other file formats.

Coming up, you'll learn how to import, read from, and write to files. You will also explore how to structure the information contained in files.

Key takeaways

It is important for security analysts to be able to automate tasks in Python. This requires knowledge of fundamental Python concepts, including variables, conditional statements, iterative statements, and techniques for working with strings and lists. In addition, the ability to work with files is also essential for automation in Python.

Clancy: Continual learning and Python

My name is Clancy and I'm a Senior Security Engineer.
My team here at Google is part of an ongoing effort to protect Google's sensitive information, customer data, PII.
Everyday is different at my job, it allows me to use different skills, knowledge sets, and no day is alike.
By trade, I am not a engineer or software engineer at all.
I was actually in accounting.
Being affected by any type of cybersecurity attack definitely gives you a perspective on the opposite side.
You get to see how this affects users, how this affects people that were attacked.
Had I known when I first started out how big of a field cybersecurity really was, it would have allowed me to explore.
Python is a developmental language.
I use it very frequently at my role at Google.
One of my favorite things about Python is the power of the language.
You can use it to create very powerful scripts that you'll use in your day-to-day role.
When I first picked up Python, the trickiest part was learning how to say things the Pythonic way.
I used various resources online as well as books, as well as picking up side projects.
One of the best things about Python it's a very widely used language and you can find many resources online depending on your skill set.
Python, as well as any other developmental language, is constantly evolving.
Continue to take on projects, continue to stretch your knowledge and you'll continue to grow.
The advice I can give for a person starting to learn Python is make it fun.
I think once you find a learning a language to be fun, it allows you to be more engaged.
Develop a good baseline for what cybersecurity is.
Make yourself a little well-rounded in the beginning and then from there you can branch out and deep dive into subjects that are interesting to you.
When starting out, it can be very tough and you feel as if you're climbing up a big hill.
Persevere, continue to learn and it will be a very rewarding experience.

Access a text file in Python

Security professionals are often tasked with reviewing log files.
These files may have thousands of entries, so it can be helpful to automate this process, and that's where Python comes in.
Let's start by importing a simple text file that just contains a few words and then restoring it as a string in Python.
All we need is the text file, its location, and the right Python keywords.
We're going to start by typing a "with" statement.
The keyword with handles errors and manages external resources.
When using with, Python knows to automatically release resources that would otherwise keep our system busy until the program finishes running.
It's often used in file handling to automatically close a file after reading it.
To open files and then read them, we write a statement that begins with the keyword with.
Then, we use the open() function.
Open() is a function that opens a file in Python.
The first parameter is the name of the text file on your computer or a link to it on the internet.
Depending on the Python environment, you might also need to include a path to this file.
Remember to include thetxt extension in the file name.
Now let's discuss the second parameter.
This parameter in the open() function tells Python what we want to do with the file.
In our case, we want to read a file, so we use the letter "r" between quotation marks.
If we wanted to write to a file, we would replace this "r" with a "w".
But here, we're focusing on reading.
Finally, file is a variable that contains the file information as long as we're inside the with statement.
Like with other types of statements, we end our with statement with a colon.
The code that comes after the colon will tell Python what to do with the content of the file.
Let's go into Python and use what we learned.
We're ready to open a text file in Python.
Now we'll type our with statement.
Next, we'll use Python's built-in read method.
The read method converts files into strings.
Now let's go back to our with statement.
Similar to a for loop, with statements start an indent on the next line.
This tells Python that this code is happening inside the with statement.
Inside of the statement, we're going to use the read() function to turn our file into a string and store that inside a new variable.
This new variable can be used outside of the with statement.
So let's exit the with statement by removing the indentation and print the variable.
Perfect!
The string from the text prints out.
Coming up, we're going to discuss parsing files so we are equipped to handle security logs in the

future.

Import files into Python

Previously, you explored how to open files in Python, convert them into strings, and read them. In this reading, you'll review the syntax needed for this. You'll also focus on why the ability to work with files is important for security analysts using Python, and you will learn about writing files.

Working with files in cybersecurity

Security analysts may need to access a variety of files when working in Python. Many of these files will be logs. A **log** is a record of events that occur within an organization's systems.

For instance, there may be a log containing information on login attempts. This might be used to identify unusual activity that signals attempts made by a malicious actor to access the system.

As another example, malicious actors that have breached the system might be capable of attacking software applications. An analyst might need to access a log that contains information on software applications that are experiencing issues.

Opening files in Python

To open a file called "*update_log.txt*" in Python for purposes of reading it, you can incorporate the following line of code:

with open("update_log.txt", "r") as file:

This line consists of the *with* keyword, the *open()* function with its two parameters, and the *as* keyword followed by a variable name. You must place a colon (*:*) at the end of the line.

with

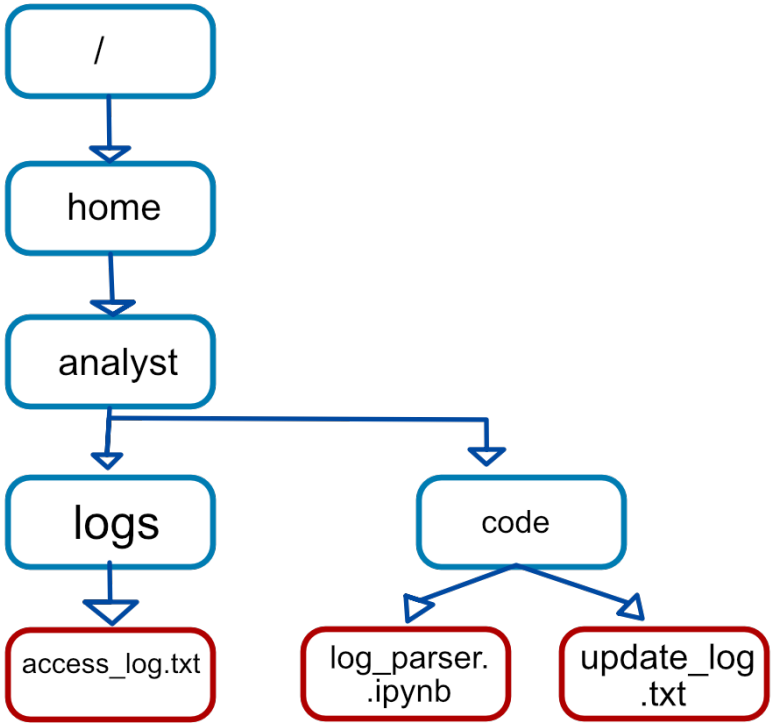
The keyword *with* handles errors and manages external resources when used with other functions. In this case, it's used with the *open()* function in order to open a file. It will then manage the resources by closing the file after exiting the *with* statement.

Note: You can also use the *open()* function without the *with* keyword. However, you should close the file you opened to ensure proper handling of the file.

open()

The *open()* function opens a file in Python.

The first parameter identifies the file you want to open. In the following file structure, "*update_log.txt*" is located in the same directory as the Python file that will access it, "*log_parser.ipynb*":



Because they're in the same directory, only the name of the file is required. The code can be written as *with open("update_log.txt", "r") as file:*.

However, "*access_log.txt*" is not in the same directory as the Python file "*log_parser.ipynb*". Therefore, it's necessary to specify its absolute file path. A **file path** is the location of a file or directory. An absolute file path starts from the highest-level directory, the root. In the following code, the first parameter of the *open()* function includes the absolute file path to "*access_log.txt*":

with open("/home/analyst/logs/access_log.txt", "r") as file:

Note: In Python, the names of files or their file paths can be handled as string data, and like all string data, you must place them in quotation marks.

The second parameter of the *open()* function indicates what you want to do with the file. In both of these examples, the second parameter is "*r*", which indicates that you want to read the file. Alternatively, you can use "*w*" if you want to write to a file or "*a*" if you want to append to a file.

as

When you open a file using *with open()*, you must provide a variable that can store the file while you are within the *with* statement. You can do this through the keyword *as* followed by this variable

name. The keyword *as* assigns a variable that references another object. The code *with open("update_log.txt", "r") as file:* assigns *file* to reference the output of the *open()* function within the indented code block that follows it.

Reading files in Python

After you use the code *with open("update_log.txt", "r") as file:* to import "*update_log.txt*" into the *file* variable, you should indicate what to do with the file on the indented lines that follow it. For example, this code uses the *.read()* method to read the contents of the file:

```
with open("update_log.txt", "r") as file:

    updates = file.read()

print(updates)
```

The *.read()* method converts files into strings. This is necessary in order to use and display the contents of the file that was read.

In this example, the *file* variable is used to generate a string of the file contents through *.read()*. This string is then stored in another variable called *updates*. After this, *print(updates)* displays the string.

Once the file is read into the *updates* string, you can perform the same operations on it that you might perform with any other string. For example, you could use the *.index()* method to return the index where a certain character or substring appears. Or, you could use *len()* to return the length of this string.

Writing files in Python

Security analysts may also need to write to files. This could happen for a variety of reasons. For example, they might need to create a file containing the approved usernames on a new allow list. Or, they might need to edit existing files to add data or to adhere to policies for standardization.

To write to a file, you will need to open the file with "*w*" or "*a*" as the second argument of *open()*.

You should use the "*w*" argument when you want to replace the contents of an existing file. When working with the existing file *update_log.txt*, the code *with open("update_log.txt", "w") as file:* opens it so that its contents can be replaced.

Additionally, you can use the "*w*" argument to create a new file. For example, *with open("update_log2.txt", "w") as file:* creates and opens a new file called "*update_log2.txt*".

You should use the "*a*" argument if you want to append new information to the end of an existing file rather than writing over it. The code *with open("update_log.txt", "a") as file:* opens "*update_log.txt*" so that new information can be appended to the end. Its existing information will not be deleted.

Like when opening a file to read from it, you should indicate what to do with the file on the indented lines that follow when you open a file to write to it. With both "*w*" and "*a*", you can use the *.write()* method. The *.write()* method writes string data to a specified file.

The following example uses the *.write()* method to append the content of the *line* variable to the file "*access_log.txt*".

```
line = "jrafael,192.168.243.140,4:56:27,True"

with open("access_log.txt", "a") as file:

    file.write(line)
```

Note: Calling the *.write()* method without using the *with* keyword when importing the file might result in its arguments not being completely written to the file if the file is not properly closed in another way.

Key takeaways

It's important for security analysts to be able to import files into Python and then read from or write to them. Importing Python files involves using the *with* keyword, the *open()* function, and the *as* keyword. Reading from and writing to files requires knowledge of the *.read()* and *.write()* methods and the arguments to the *open()* function of "*r*", "*w*", and "*a*".

Parse a text file in Python

Now that you know how to import text files into Python, we're going to take this one step further and learn how to give them a structure.
This will allow us to analyze them more easily.
This process is often referred to as parsing.
Parsing is the process of converting data into a more readable format.
To do this, we're going to put together everything we learned about lists and strings and learn another method for working with strings in Python.
The method we need is the split method.
The split method converts a string into a list.
It does this by separating the string based on a specified character.
Or, if no argument is passed, every time it encounters a whitespace, it separates the string.
So, a split would convert the string "We are learning about parsing!" into this list.
We are using the split method to separate the strings into smaller chunks that we can analyze more easily than one big block of text.
In this video, we'll work with an example of a security log where every line represents a new data point.
To store these points in a list, we want to separate the text based on the new line.
Python considers a new line to be a type of whitespace.
We can use the split method without passing an argument.
We'll start with our code from the previous video.
Remember, we used this code to open a file and then read it into a string.
Now, let's split that string into a list using the split method and then print the output.
After we run it, Python outputs a list of usernames instead of one big string of them.
If we want to save this list, we would need to assign it to another variable.
For example, we can call the variable usernames.
And then we'll run it again.
And now this list can be reused in other code.
Congratulations!
You just learned the basics of parsing a text file in Python.
In the next videos, we're going to be exploring techniques that help us work more in depth with data in Python.
Now that you know how to import text files into Python,.

Work with files in Python

You previously explored how to open files in Python as well as how to read them and write to them. You also examined how to adjust the structure of file contents through the *.split()* method. In this reading, you'll review the *.split()* method, and you'll also learn an additional method that can help you work with file contents.

Parsing

Part of working with files involves structuring its contents to meet your needs. **Parsing** is the process of converting data into a more readable format. Data may need to become more readable in a couple of different ways. First, certain parts of your Python code may require modification into a specific format. By converting data into this format, you enable Python to process it in a specific way. Second, programmers need to read and interpret the results of their code, and parsing can also make the data more readable for them.

Methods that can help you parse your data include *.split()* and *.join()*.

.split()

The basics of .split()

The *.split()* method converts a string into a list. It separates the string based on a specified character that's passed into *.split()* as an argument.

In the following example, the usernames in the *approved_users* string are separated by a comma. For this reason, a string containing the comma (",") is passed into *.split()* in order to parse it into a list. Run this code and analyze the different contents of *approved_users* before and after the *.split()* method is applied to it:

```
approved_users = "elarson,bmoreno,tshah,sgilmore,eraab"
print("before .split():", approved_users)
approved_users = approved_users.split(",")
print("after .split():", approved_users)
```

```
before .split(): elarson,bmoreno,tshah,sgilmore,eraab
after .split(): ['elarson', 'bmoreno', 'tshah', 'sgilmore', 'eraab']
```

Before the *.split()* method is applied to *approved_users*, it contains a string, but after it is applied, this string is converted to a list.

If you do not pass an argument into *.split()*, it will separate the string every time it encounters a whitespace.

Note: A variety of characters are considered whitespaces by Python. These characters include spaces between characters, returns for new lines, and others.

The following example demonstrates how a string of usernames that are separated by space can be split into a list through the *.split()* method:

```
removed_users = "wjaffrey jsoto abernard jhill awilliam"
print("before .split():", removed_users)
removed_users = removed_users.split()
print("after .split():", removed_users)
```

```
before .split(): wjaffrey jsoto abernard jhill awilliam
after .split(): ['wjaffrey', 'jsoto', 'abernard', 'jhill', 'awilliam']
```

Because an argument isn't passed into *.split()*, Python splits the *removed_users* string at each space when separating it into a list.

Applying .split() to files

The *.split()* method allows you to work with file content as a list after you've converted it to a string through the *.read()* method. This is useful in a variety of ways. For example, if you want to iterate through the file contents in a *for* loop, this can be easily done when it's converted into a list.

The following code opens the "*update_log.txt*" file. It then reads all of the file contents into the *updates* variable as a string and splits the string in the *updates* variable into a list by creating a new element at each whitespace:

```
with open("update_log.txt", "r") as file:
    updates = file.read()
updates = updates.split()
```

After this, through the *updates* variable, you can work with the contents of the "*update_log.txt*" file in parts of your code that require it to be structured as a list.

Note: Because the line that contains *.split()* is not indented as part of the *with* statement, the file closes first. Closing a file as soon as it is no longer needed helps maintain code readability. Once a file is read into the *updates* variable, it is not needed and can be closed.

.join()

The basics of .join()

If you need to convert a list into a string, there is also a method for that. The *.join()* method concatenates the elements of an iterable into a string. The syntax used with *.join()* is distinct from the syntax used with *.split()* and other methods that you've worked with, such as *.index()*.

In methods like *.split()* or *.index()*, you append the method to the string or list that you're working with and then pass in other arguments. For example, the code *usernames.index(2)*, appends the *.index()* method to the variable *usernames*, which contains a list. It passes in *2* as the argument to indicate which element to return.

However, with *.join()*, you must pass the list that you want to concatenate into a string in as an argument. You append *.join()* to a character that you want to separate each element with once they are joined into a string.

For example, in the following code, the *approved_users* variable contains a list. If you want to join that list into a string and separate each element with a comma, you can use *",".join(approved_users)*. Run the code and examine what it returns:

```
approved_users = ["elarson", "bmoreno", "tshah", "sgilmore", "eraab"]
print("before .join():", approved_users)
approved_users = ",".join(approved_users)
print("after .join():", approved_users)
```

```
before .join(): ['elarson', 'bmoreno', 'tshah', 'sgilmore', 'eraab']
after .join(): elarson,bmoreno,tshah,sgilmore,eraab
```

Before *.join()* is applied, *approved_users* is a list of five elements. After it is applied, it is a string with each username separated by a comma.

Note: Another way to separate elements when using the *.join()* method is to use "*\n*", which is the newline character. The "*\n*" character indicates to separate the elements by placing them on new lines.

Applying .join() to files

When working with files, it may also be necessary to convert its contents back into a string. For example, you may want to use the *.write()* method. The *.write()* method writes string data to a file. This means that if you have converted a file's contents into a list while working with it, you'll need to convert it back into a string before using *.write()*. You can use the *.join()* method for this.

You already examined how *.split()* could be applied to the contents of the "*update_log.txt*" file once it is converted into a string through *.read()* and stored as *updates*:

```
with open("update_log.txt", "r") as file:
    updates = file.read()
updates = updates.split()
```

After you're through performing operations using the list in the *updates* variable, you might want to replace "*update_log.txt*" with the new contents. To do so, you need to first convert updates back into a string using *.join()*. Then, you can open the file using a *with* statement and use the *.write()* method to write the *updates* string to the file:

```
updates = " ".join(updates)
with open("update_log.txt", "w") as file:
    file.write(updates)
```

The code " "*.join(updates)* indicates to separate each of the list elements in *updates* with a space once joined back into a string. And because "*w*" is specified as the second argument of *open()*, Python will overwrite the contents of "*update_log.txt*" with the string currently in the *updates* variable.

Key takeaways

An important element of working with files is being able to parse the data it contains. Parsing means converting the data into a readable format. The *.split()* and *.join()* methods are both useful for parsing data. The *.split()* method allows you to convert a string into a list, and the *.join()* method allows you to convert a list into a string.

Develop a parsing algorithm in Python

We're now going to bring all of the pieces together to import a file, parse it, and implement a simple algorithm to help us detect suspicious login attempts.
In this video, we want to create a program that runs every time a new user logs in and checks if that user has had three or more failed login attempts.
First, let's discuss the structure of our inputs to build a strategy to develop our program.
We have a log file stored in atxt format that contains one username per line.
Each username represents a failed login attempt.
So when a user logs in, we want our program to check for their username and count how many times that username is in our log file.
If that username is repeated three or more times, the program returns an alert.
We'll start with code that imports the file of logging attempts, splits it, and stores it into a variable named usernames.
Let's try printing the variable user names to check for its contents.
We'll run this.
Perfect!
This is exactly what we expected.
The variable usernames is ready to be used in our algorithm.
Now let's develop a strategy for counting username occurrences in the list.
We'll start with the first eight elements of the usernames list.
We notice that there are two occurrences of the username "eraab" in the list, but how would we tell Python to count this?
We'll implement a for loop that iterates through every element.
Let's represent the loop variable with an arrow.
We'll also define a counter variable that starts at 0.
So, our for loop starts at the username "elarson." At every element, Python asks, "Is this element equal to the string 'eraab'?" If the answer is yes, the counter goes up by one.
If it isn't, then the counter stays the same.
Since "elarson" is not the same as "eraab," the counter remains 0.
Then, we move on to the next element.
We encounter our first occurrence of "eraab." At this point, the counter increases by 1.
As we move to the next element, we find another occurrence of "eraab," so we increase our counter by 1 again.
That means that our counter is now at 2.
We will continue this process for the rest of the list.
Now that we know the solution, let's talk about how to implement it in Python.
Solving the problem in Python will involve a for loop, a counter variable, and an if statement.
Let's get back into our code.
We'll create a function that counts a user's failed login attempts.

First, let's define our function.
We'll call it login_check().
It takes two parameters.
The first is called login_list.
This will be used for the list of failed login attempts.
The second is called current_user.
This will be used for the user who logs in.
Inside of this function, we start by defining the counter variable and set its value to 0.
Now we start the for loop.
We'll use i as our loop variable and iterate through the login list.
In other words, as the loop iterates, it will run through all the failed login attempts in the list.
Directly inside of the for loop, we start the if statement.
The if statement checks if our loop variable is equal to the current_user we're searching for.
If this condition is true, We want to add 1 to the counter.
We're almost done with our algorithm.
Now, we just need the final if-else statement to print the alert.
If the counter adds up to 3 or more, we need to tell the user that their account is locked so they can't log in.
We'll also type an else statement for users who can log in.
Our algorithm is complete!
Let's try out our new function on an example username.
We can pull out a few of the usernames in the list and try our function on them.
Let's use the first name in the list.
Let's run the code.
According to our code, this user can log in.
They have fewer than three failed login attempts.
Now let's go back to our user "eraab." Remember, they had two entries in the list of the first eight names in our failed login attempts.
Do you think they'll be able to log in?
When we run, we get an "account locked" message.
This means they had three or more failed login attempts.
Excellent work!
You've just developed your first security algorithm involving a log.
As you grow in your skills, you'll learn how to make this algorithm more efficient, but this solution works well for now.
In this video, we recapped everything we learned so far, from list operations to algorithm development, all the way to file parsing.
We did this while building an algorithm we can apply in a security context.

Portfoleo project: Algorythm for file updates in Python

Algorithm for file updates in Python

Project description

this script automates management of an ip allowlist. the allowlist is in a file, and the script reads and writes to file.

Open the file that contains the allow list

```
import_file = "allow_list.txt"

# Assign `remove_list` to a list of IP addresses that are no longer allowed to access restricted information.

remove_list = ["192.168.97.225", "192.168.158.170", "192.168.201.40", "192.168.58.57"]

# First line of `with` statement to open the file and store as variable file

with open(import_file, "r") as file:
```

Read the file contents

```
with open(import_file, "r") as file:

  # Use `.read()` to read the imported file and store it in a variable named `ip_addresses`

  ip_addr = file.read()
```

Convert the string into a list

```
with open(import_file, "r") as file:

  # Use `.read()` to read the imported file and store it in a variable named `ip_addresses`

  ip_addresses = file.read()

# Use `.split()` to convert `ip_addresses` from a string to a list

ip_addresses = ip_addresses.split("\n")

# Display `ip_addresses`

print(ip_addresses)
```

Iterate through the remove list

```
for element in ip_addresses:

    # Display `element` in every iteration

    print(element)
```

Remove IP addresses that are on the remove list

```
for element in ip_addresses:

  # Build conditional statement
  # If current element is in `remove_list`,

    if element in remove_list:

        # then current element should be removed from `ip_addresses`

        ip_addresses.remove(element)
        # add to ips renmoved list

# Display `ip_addresses`

print(ip_addresses)
```

Update the file with the revised list of IP addresses

```
# Assign `import_file` to the name of the file

import_file = "allow_list.txt"

# Assign `remove_list` to a list of IP addresses that are no longer allowed to access
restricted information.

remove_list = ["192.168.97.225", "192.168.158.170", "192.168.201.40", "192.168.58.57"]

# Build `with` statement to read in the initial contents of the file

with open(import_file, "r") as file:

  # Use `.read()` to read the imported file and store it in a variable named `ip_addresses`

  ip_addresses = file.read()

# Use `.split()` to convert `ip_addresses` from a string to a list

ip_addresses = ip_addresses.split()

# Build iterative statement
# Name loop variable `element`
# Loop through `ip_addresses`

for element in ip_addresses:

  # Build conditional statement
  # If current element is in `remove_list`,

    if element in remove_list:

        # then current element should be removed from `ip_addresses`

        ip_addresses.remove(element)
```

```
# Convert `ip_addresses` back to a string so that it can be written into the text file

ip_addresses = " ".join(ip_addresses)

# Build `with` statement to rewrite the original file

with  open(import_file, "w") as file:

  # Rewrite the file, replacing its contents with `ip_addresses`

    file.write(ip_addresses)
```

Final

create the allow-list as a file

```
# Assign `import_file` to the name of the file

import_file = "allow_list.txt"
allow_list = """ip_address
192.168.25.60
192.168.205.12
192.168.97.225
192.168.6.9
192.168.52.90
192.168.158.170
192.168.90.124
192.168.186.176
192.168.133.188
192.168.203.198
192.168.201.40
192.168.218.219
192.168.52.37
192.168.156.224
192.168.60.153
192.168.58.57
192.168.69.116"""
# Assign `remove_list` to a list of IP addresses that are no longer allowed to access
restricted information.
```

```
remove_list = ["192.168.97.225", "192.168.158.170", "192.168.201.40", "192.168.58.57"]

# Build `with` statement to read in the initial contents of the file

with open(import_file, "r") as file:

  # Use `.read()` to read the imported file and store it in a variable named `ip_addresses`

  ip_addr = file.read()

temp_file = "task_3_remove_list.txt"
import_file = temp_file

with open(import_file, "w") as import_file:
    import_file.write(ip_addr)



# Display `ip_addresses`

print(ip_addr)
```

modify and prune the allow-list

```
# Assign `import_file` to the name of the file

import_file = "allow_list.txt"

# Assign `remove_list` to a list of IP addresses that are no longer allowed to access
restricted information.

remove_list = ["192.168.97.225", "192.168.158.170", "192.168.201.40", "192.168.58.57"]

# Build `with` statement to read in the initial contents of the file

with open(import_file, "r") as file:

  # Use `.read()` to read the imported file and store it in a variable named `ip_addresses`
```

```
  ip_addresses = file.read()

# Use `.split()` to convert `ip_addresses` from a string to a list

ip_addresses = ip_addresses.split()

# Build iterative statement
# Name loop variable `element`
# Loop through `ip_addresses`

for element in ip_addresses:

  # Build conditional statement
  # If current element is in `remove_list`,

    if element in remove_list:

        # then current element should be removed from `ip_addresses`

        ip_addresses.remove(element)

# Convert `ip_addresses` back to a string so that it can be written into the text file

ip_addresses = " ".join(ip_addresses)

# Build `with` statement to rewrite the original file

with open(import_file, "w") as file:

  # Rewrite the file, replacing its contents with `ip_addresses`

  file.write(ip_addresses)

# Build `with` statement to read in the updated file

with open(import_file, "r") as file:

    # Read in the updated file and store the contents in `text`

    text = file.read()
```

```
# Display the contents of `text`


print(text)
```

Summary

in a hypothetical scenario where you needed to remove information from a large file this could save alot of time. this can save you lots of time.

Debugging strategies

As a security analyst, you might be required to read or write code.
One of the biggest challenges is getting it to run and function properly.
In fact, fixing complex errors in code can sometimes take up just as much, if not more, time than writing your code.
This is why it's important to develop this skill.
Now that you've learned the basics of coding in Python, it's important to learn how to deal with errors.
For that reason, we'll focus on debugging your code.
Debugging is the practice of identifying and fixing errors in code.
In this video, we'll explore some techniques for this.
There are three types of errors: syntax errors, logic errors, and exceptions.
Syntax errors involve invalid usage of the Python language, such as forgetting to add a colon after a function header.
Let's explore this type of error.
When we run this code, we get a message that indicates there's a syntax error.
Depending on the Python environment, it may also display additional details.
We'll typically get information about the error, like its location.
These syntax errors are often easy to fix since you can find exactly where the error happened.
They are similar to correcting simple grammar mistakes in an email.
Since their message tells us the problem is on the line that defines the function, let's go there.
In this case, we can add a colon to the header and resolve our error.
When we run it again, there's no longer an error message.
This is just one example of a syntax error.
Other examples include omitting a parenthesis after a function, misspelling a Python keyword, or not properly closing quotation marks for a string.
Next, let's focus on logic errors.
Logic errors may not cause error messages; instead, they produce unintended results.
A logic error could be as simple as writing the incorrect text within a print statement, or it might involve something like writing a less than symbol instead of less than or equal to symbol.
This change in operator would exclude a value that was needed for the code to work as intended.
For example, imagine that you reach out to a response team when the priority level of an issue is less than three instead of less than or equal to three.
This means all events classified as priority level 3 could go unnoticed and unresolved.
To diagnose a logic error that's difficult to find, one strategy is to use print statements.
You'll need to insert print statements throughout your code.
The print statements should describe the location in the code; for example, "print line 20" or "print line 55: inside the conditional".
The idea is to use these print statements to identify which sections of the code are functioning properly.
When a print statement doesn't print as expected, this helps you identify sections of the code with problems.

Another option for identifying logic errors is to use a debugger.
A debugger will let you insert breakpoints into your code.
Breakpoints allow you to segment your code into sections and run just one portion at a time.
Just like with the print statements, running these sections independently can help isolate problems in the code.
Let's move on to our last type of error: an exception.
Exceptions happen when the program doesn't know how to execute code even though there are no problems with the syntax.
Exceptions occur for a variety of reasons.
For example, they can happen when something is mathematically impossible, like asking the code to divide something by 0.
Exceptions might also happen when you ask Python to access index values that don't exist or when Python doesn't recognize variable or function names.
Exceptions also may occur when you use an incorrect data type.
Let's demonstrate an exception.
Let's say you have a variable called my_string that contains the word "security".
Since this string has 8 characters, we can successfully print any index less than 8.
Index 0 contains "s." Index 1 contains "e." And index 2 contains "c." But, if you try to access the character at index 100, you'll get an error.
Let's run this and explore what happens.
After it successfully prints the first three statements, we get an error message: "string index out of range." For exception errors, you can also make use of debuggers and print statements to figure out the potential source of error.
Errors and exceptions can be expected when working in Python.
The important thing is to know how to deal with them.
Hopefully, this video provided some valuable insight about debugging code.
This will help ensure that the code that you write is functional.

Matt: Learning from mistakes

My name is Matt.
I'm a software engineer working in cybersecurity.
When I was in high school, what I really want to do is music, I was a musician.
I went to music school for jazz trombone.
Partway through that process, I realized, no, bandleader in their right mind, looks at a group and thinks, we need a trombone player.
This is what makes us better.
I grew up watching movies like The Matrix, and it's not very realistic to what cybersecurity work actually is, but it is inspiring.
If you dig through the work at specifics of what you do and take the broad step back like you are that cool guy with sunglasses, you're a hacker.
When I first started writing code, I would look at coding errors as a sign that I have done bad.
But as I grew older, a little bit more mature, I realized everyone has coding errors.
Literally the very best software engineer I know writes code and it has errors.
Errors represent a moment where you can take a step back and say, what did I do wrong?
It's a learning opportunity.
Now, I see these like moments where I can look at some problems that I don't understand and be like, why?
Dive into it and expand my knowledge of computer science, which is my whole job.
I see this as a learning process and it's a little bit fun.
One of the really messy coding areas that I've run into in my time here at Google involved fingerprinting when we find a vulnerability.
If we find the same vulnerability later, we don't want to have two vulnerabilities, we don't want to poke someone twice and be like, fix this thing if it's the exact same thing.
We do this thing called fingerprinting where we say this vulnerability has a specific fingerprint and if we find another vulnerability that has the same fingerprint, we're not going to store them separately or treat them separately.
They're effectively the same.
I was running into these errors where things would not fingerprint in the way that I expected them to.
I was literally grinding my gears for weeks, trying to figure out what is going on with this thing?
But when I found it was very satisfying like, that's it.
When you're in the middle of this mess, all of that self-doubt creeps into your brain.
You're like, maybe I'm not as good at this thing as I thought I was.
What I would go back and tell myself: A), it's not endless - it gets better.
Once you figure it out, that feeling of reward is incredible.
But B), also, it's okay to rope people in if you are struggling with something, I always advocate for asking for help.
Most people are really excited to go help you out with this thing, especially when it's a convoluted problem.
I am so thrilled I ended up in cybersecurity.

Cybersecurity is having its moment.
People are realizing, waking up to the amount of data that they're putting out into the world and they're starting to care about it.
Every day there's something new.
Every day there's something exciting for me to do and yeah, get into it.
Cybersecurity is the way.

Apply debugging strategies

Let's say our co-workers need some help getting their code to work, and we've offered to debug their code to make sure it runs smoothly.
First, we need to know about the purpose of the code.
In this case, the purpose of the code is to parse a single line from a log file and return it.
The log file we're using tracks potential issues with software applications.
Each line in the log contains the HTTP response status codes, the date, the time, and the application name.
When writing this code, our co-workers consider whether all these status codes needed to be parsed.
Since 200 signals a successful event, they concluded that lines with this status code shouldn't be parsed.
Instead, Python should return a message indicating that parsing wasn't needed.
To start the debugging process, let's first run the code to identify what errors appear.
Our first error is a syntax error.
The error message also tells us the syntax error occurs in a line that defines a function.
So let's just scroll to that part of the code.
As you might recall, these function headers should end with a colon.
Let's go ahead and add that to the code.
Now, the syntax error should go away.
Let's run the code again.
Now our syntax error is gone, which is good news, but we have another error, a "name error."
"Name error" is actually a type of exception, meaning we've written valid syntax, but Python can't process the statement.
According to the error, the interpreter doesn't understand the variable application_name at the point where it's been added to the parsed_line list.
Let's examine that section of code.
This error means we haven't assigned a variable name properly.
So now let's go back to where it was first assigned and determine what happened.
We find that this variable is misspelled.
There should be two p's in application_name, not one.
Let's correct the spelling.
Now that we've fixed it, it should work.
So let's run the code.
Great!
We fixed an error and an exception.
And we no longer have any error messages.
But this doesn't mean our debugging work is done.
Let's make sure the logic of the program works as intended by examining the output.
Our output is a parsed line.
In most cases, this would be what we wanted.
But as you might recall, if the status code is 200, our code shouldn't parse the line.

Instead, it should print a message that no parsing is needed.
And when we called it with a status code of 200, there was a logic error because this message wasn't displayed.
So let's go back to the conditional we used to handle the status code of 200 and investigate.
To find the source of the issue, let's add print statements.
In our print statements, we'll include the line number and the description of the location.
We'll add one print statement before the line of code containing "return parsed_list".
We'll add another above the if statement that checks for the 200 status code to determine if it reaches the if statement.
We'll add one more print statement inside the if statement to determine whether the program even enters it.
Now, let's run the code and review what gets printed.
Only the first print statement printed something.
The other two print statements after these didn't print.
This means the program didn't even enter the if statement.
The problem occurred somewhere before the line that returns the parsed_line variable.
Let's investigate.
When Python encounters the first return statement which sends back the parsed list, it exits the function.
In other words, it returns the list before it even checks for a status code value of 200.
To fix this, we must move the if statement, checking for the status code somewhere before "return parsed line".
Let's first delete our print statements.
This makes the program more efficient because it doesn't run any unneeded lines of code.
Now, let's move the if statement.
We'll place it right after the line of code that involves parsing the status code from the line.
Let's run our code and confirm that this fixed our issue.
Yes!
It printed "Successful event - no parsing needed." Great work!
We've fixed this logic error.
I enjoyed debugging this code with you.
I hope this video has strengthened your understanding of some helpful debugging strategies and gave you an idea of some errors you might encounter.

Explore debugging techniques

Previously, you examined three types of errors you may encounter while working in Python and explored strategies for debugging these errors. This reading further explores these concepts with additional strategies and examples for debugging Python code.

Types of errors

It's a normal part of developing code in Python to get error messages or find that the code you're running isn't working as you intended. The important thing is that you can figure out how to fix errors when they occur. Understanding the three main types of errors can help. These types include syntax errors, logic errors, and exceptions.

Syntax errors

A **syntax error** is an error that involves invalid usage of a programming language. Syntax errors occur when there is a mistake with the Python syntax itself. Common examples of syntax errors include forgetting a punctuation mark, such as a closing bracket for a list or a colon after a function header.

When you run code with syntax errors, the output will identify the location of the error with the line number and a portion of the affected code. It also describes the error. Syntax errors often begin with the label "*SyntaxError:*" . Then, this is followed by a description of the error. The description might simply be "*invalid syntax*" . Or if you forget a closing parentheses on a function, the description might be "*unexpected EOF while parsing*". "*EOF*" stands for "end of file."

The following code contains a syntax error. Run it and examine its output:

```
message = "You are debugging a syntax error
print(message)
```

```
Error on line 1:
    message = "You are debugging a syntax error
                                              ^
SyntaxError: EOL while scanning string literal
```

This outputs the message "*SyntaxError: EOL while scanning string literal*". "*EOL*" stands for "end of line". The error message also indicates that the error happens on the first line. The error occurred because a quotation mark was missing at the end of the string on the first line. You can fix it by

adding that quotation mark.

Note: You will sometimes encounter the error label "*IndentationError*" instead of "*SyntaxError*". "*IndentationError*" is a subclass of "*SyntaxError*" that occurs when the indentation used with a line of code is not syntactically correct.

Logic errors

A **logic error** is an error that results when the logic used in code produces unintended results. Logic errors may not produce error messages. In other words, the code will not do what you expect it to do, but it is still valid to the interpreter.

For example, using the wrong logical operator, such as a greater than or equal to sign (>=) instead of greater than sign (>) can result in a logic error. Python will not evaluate a condition as you intended. However, the code is valid, so it will run without an error message.

The following example outputs a message related to whether or not a user has reached a maximum number of five login attempts. The condition in the *if* statement should be *login_attempts > 5*, but it is written as *login_attempts >= 5*. A value of *5* has been assigned to *login_attempts* so that you can explore what it outputs in that instance:

```
login_attempts = 5
if login_attempts >= 5:
    print("User has not reached maximum number of login attempts.")
else:
    print("User has reached maximum number of login attempts.")
```

```
User has not reached maximum number of login attempts.
```

The output displays the message "*User has not reached maximum number of login attempts.*" However, this is not true since the maximum number of login attempts is five. This is a logic error.

Logic errors can also result when you assign the wrong value in a condition or when a mistake with indentation means that a line of code executes in a way that was not planned.

Exceptions

An **exception** is an error that involves code that cannot be executed even though it is syntactically correct. This happens for a variety of reasons.

One common cause of an exception is when the code includes a variable that hasn't been assigned or a function that hasn't been defined. In this case, your output will include "*NameError*" to indicate that this is a name error. After you run the following code, use the error message to determine which variable was not assigned:

```
username = "elarson"
month = "March"
total_logins = 75
failed_logins = 18
print("Login report for", username, "in", month)
print("Total logins:", total_logins)
print("Failed logins:", failed_logins)
print("Unusual logins:", unusual_logins)
```

```
Error on line 8:
    print("Unusual logins:", unusual_logins)
NameError: name 'unusual_logins' is not defined
```

The output indicates there is a "*NameError*" involving the *unusual_logins* variable. You can fix this by assigning this variable a value.

In addition to name errors, the following messages are output for other types of exceptions:

- "*IndexError*": An index error occurs when you place an index in bracket notation that does not exist in the sequence being referenced. For example, in the list *usernames = ["bmoreno", "tshah", "elarson"]*, the indices are *0*, *1*, and *2*. If you referenced this list with the statement *print(usernames[3])*, this would result in an index error.
- "*TypeError*": A type error results from using the wrong data type. For example, if you tried to perform a mathematical calculation by adding a string value to an integer, you would get a type error.
- "*FileNotFound*": A file not found error occurs when you try to open a file that does not exist in the specified location.

Debugging strategies

Keep in mind that if you have multiple errors, the Python interpreter will output error messages one at a time, starting with the first error it encounters. After you fix that error and run the code again, the interpreter will output another message for the next syntax error or exception it encounters.

When dealing with syntax errors, the error messages you receive in the output will generally help you fix the error. However, with logic errors and exceptions, additional strategies may be needed.

Debuggers

In this course, you have been running code in a notebook environment. However, you may write Python code in an Integrated Development Environment (IDE). An **Integrated Development Environment (IDE)** is a software application for writing code that provides editing assistance and error correction tools. Many IDEs offer error detection tools in the form of a debugger. A **debugger**

is a software tool that helps to locate the source of an error and assess its causes.

In cases when you can't find the line of code that is causing the issue, debuggers help you narrow down the source of the error in your program. They do this by working with breakpoints. Breakpoints are markers placed on certain lines of executable code that indicate which sections of code should run when debugging.

Some debuggers also have a feature that allows you to check the values stored in variables as they change throughout your code. This is especially helpful for logic errors so that you can locate where variable values have unintentionally changed.

Use print statements

Another debugging strategy is to incorporate temporary print statements that are designed to identify the source of the error. You should strategically incorporate these print statements to print at various locations in the code. You can specify line numbers as well as descriptive text about the location.

For example, you may have code that is intended to add new users to an approved list and then display the approved list. The code should not add users that are already on the approved list. If you analyze the output of this code after you run it, you will realize that there is a logic error:

```
new_users = ["sgilmore", "bmoreno"]
approved_users = ["bmoreno", "tshah", "elarson"]
def add_users():
    for user in new_users:
        if user in approved_users:
            print(user,"already in list")
        approved_users.append(user)
add_users()
print(approved_users)
```

```
bmoreno already in list
['bmoreno', 'tshah', 'elarson', 'sgilmore', 'bmoreno']
```

Even though you get the message "*bmoreno already in list*", a second instance of "*bmoreno*" is added to the list. In the following code, print statements have been added to the code. When you run it, you can examine what prints:

```
new_users = ["sgilmore", "bmoreno"]
approved_users = ["bmoreno", "tshah", "elarson"]
def add_users():
    for user in new_users:
```

```
        print("line 5 - inside for loop")
        if user in approved_users:
            print("line 7 - inside if statement")
            print(user,"already in list")
        print("line 9 - before .append method")
        approved_users.append(user)
add_users()
print(approved_users)
```

```
line 5 - inside for loop
line 9 - before .append method
line 5 - inside for loop
line 7 - inside if statement
bmoreno already in list
line 9 - before .append method
['bmoreno', 'tshah', 'elarson', 'sgilmore', 'bmoreno']
```

The print statement "*line 5 - inside for loop*" outputs twice, indicating that Python has entered the *for* loop for each username in *new_users*. This is as expected. Additionally, the print statement "*line 7 - inside if statement*" only outputs once, and this is also as expected because only one of these usernames was already in *approved_users*.

However, the print statement "*line 9 - before .append method*" outputs twice. This means the code calls the *.append()* method for both usernames even though one is already in *approved_users*. This helps isolate the logic error to this area. This can help you realize that the line of code *approved_users.append(user)* should be the body of an *else* statement so that it only executes when *user* is not in *approved_users*.

Key takeaways

There are three main types of errors you'll encounter while coding in Python. Syntax errors involve invalid usage of the programming language. Logic errors occur when the logic produced in the code produces unintended results. Exceptions involve code that cannot be executed even though it is syntactically correct. You will receive error messages for syntax errors and exceptions that can help you fix these mistakes. Additionally, using debuggers and inserting print statements can help you identify logic errors and further debug exceptions.

Wrap-up

Great work in this section!
We focused on a few new topics that will help you put Python into practice in the security profession.
First, we explored opening and reading files in Python.
Security analysts work with a lot of log files, so the ability to do this is essential.
Next, we covered parsing files.
Log files can be very long.
For this reason, a structure in these files to make them more readable helps you automate your tasks and get the information you need.
And last, we focused on debugging code.
Knowing how to debug your code can save you a lot of time, especially as your code increases in complexity.
Overall, I hope you feel proud of what you've accomplished in this section.
Addressing security issues through Python is exciting, and the information we covered will allow you to do that.

Reference guide: Python concepts from module 4

File operations

The following functions, methods, and keywords are used with operations involving files.

`with`

Handles errors and manages external resources

> `with open("logs.txt", "r") as file:`
> Used to handle errors and manage external resources while opening a file; the variable `file` stores the file information while inside of the `with` statement; manages resources by closing the file after exiting the `with` statement

`open()`

Opens a file in Python

> `with open("login_attempts.txt", "r") as file:`
> Opens the file `"login_attempts.txt"` in order to read it `("r")`
> `with open("update_log.txt", "w") as file:`
> Opens the file `"update_log.txt"` into the variable `file` in order to write over its contents `("w")`

> `with open(import_file, "a") as file:`
> Opens the file assigned to the `import_file` variable into the variable `file` in order to append information to the end of it `("a")`

`as`

Assigns a variable that references another object

`with open("logs.txt", "r") as file:`
Assigns the `file` variable to reference the output of the `open()` function

`.read()`

Converts files into strings; returns the content of an open file as a string by default

```
with open("login_attempts.txt", "r") as file:
    file_text = file.read()
```

Converts the file object referenced in the `file` variable into a string and then stores this string in the `file_text` variable

`.write()`

Writes string data to a specified file

```
with open("access_log.txt", "a") as file:
        file.write("jrafael")
```

Writes the string "jrafael" to the `"access_log.txt"` file; because the second argument in the call to the `open()` function is `"a"`, this string is appended to the end of the file

Parsing

The following methods are useful when parsing data.

`.split()`

Converts a string into a list; separates the string based on the character that is passed in as an argument; if an argument is not passed in, it will separate the string each time it encounters whitespace characters such as a space or return

```
approved_users = "elarson,bmoreno,tshah".split(",")
```

Converts the string `"elarson,bmoreno,tshah"` into the list `["elarson","bmoreno","tshah"]` by splitting the string into a separate list element at each occurrence of the `","` character

```
removed_users = "wjaffrey jsoto abernard".split()
```

Converts the string `"wjaffrey jsoto abernard"` into the list `["wjaffrey","jsoto","abernard"]` by splitting the string into a separate list element at each space

`.join()`

Concatenates the elements of an iterable into a string; takes the iterable to be concatenated as an argument; is appended to a character that will separate each element once they are joined into a string

```
approved_users = ",".join(["elarson", "bmoreno", "tshah"])
```

Concatenates the elements of the list `["elarson","bmoreno","tshah"]` into the string `"elarson,bmoreno,tshah"` , separating each element with the `","` character within the string

Terms and definitions from Course 7, Module 4

Glossary terms from module 4

Automation: The use of technology to reduce human and manual effort to perform common and repetitive tasks

Conditional statement: A statement that evaluates code to determine if it meets a specified set of conditions

Debugger: A software tool that helps to locate the source of an error and assess its causes

Debugging: The practice of identifying and fixing errors in code

Exception: An error that involves code that cannot be executed even though it is syntactically correct

File path: The location of a file or directory

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

Integrated development environment (IDE): A software application for writing code that provides editing assistance and error correction tools

Iterative statement: Code that repeatedly executes a set of instructions

Log: A record of events that occur within an organization's systems

Logic error: An error that results when the logic used in code produces unintended results

Parsing: The process of converting data into a more readable format

Syntax error: An error that involves invalid usage of a programming language

Variable: A container that stores data

Course wrap-up

As we wrap up this course, I want to congratulate you for your commitment to learning Python. You should feel accomplished having explored a programming language that's useful in the security field.
Let's recap some of what we've learned.
First, we covered basic programming concepts in Python.
We discussed variables, data types, conditional statements, and iterative statements.
These topics provided important foundations for what we explored later in the course.
And our next focus was on writing effective Python code.
We learned how we can reuse functions in our programs to improve efficiency.
We explored built-in functions and even created our own user-defined functions.
Another topic was modules and libraries.
The pre-packaged functions and variables they contain can make our work easier.
Last, we learned ways to ensure our code is readable.
In the next section, we focused on working with strings and lists.
We learned a variety of methods that can be applied to these data types.
We also learned about their indices and how to slice characters from a string or elements from a list.
We put all of these together to write a simple algorithm, and then we explored how regular expressions can be used to find patterns in the strings.
And last, we wrapped up the course with a focus on putting Python into practice.
We learned how to open, read, and parse files.
With these skills, you can work with a variety of logs you will encounter in a security setting.
We also learned how to debug code.
This is an important skill for all programmers.
Wow!
You learned a lot about Python in this course, so great job!
I hope soon you'll join me in using Python in the security profession.
In the meantime, I encourage you to practice, and feel free to rewatch these videos whenever you like.
The more you study these concepts, the easier they will become.
Thanks again for joining me as we explored Python.

Reference guide: Python concepts from Course 7

Google Cybersecurity Certificate

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

Conditional statements

The following keywords and operators are used in conditional statements.

`if`

Starts a conditional statement

`if device_id != "la858zn":`

Starts a conditional statement that evaluates whether the `device_id` variable contains a value that is not equal to `"la858zn"`

`if user in approved_list:`

Starts a conditional statement that evaluates if the `user` variable contains a value that is also found in the `approved_list` variable

`elif`

Precedes a condition that is only evaluated when previous conditions evaluate to `False`; previous conditions include the condition in the `if` statement, and when applicable, conditions in other `elif` statements

`elif status == 500:`

When previous conditions evaluate to `False`, evaluates if the `status` variable contains a value that is equal to `500`

`else`

Precedes a code section that only evaluates when all conditions that precede it within the conditional statement evaluate to `False`; this includes the condition in the `if` statement, and when applicable, conditions in `elif` statements

`else:`

When previous conditions evaluate to `False`, Python evaluates this `else` statement

`and`

Requires both conditions on either side of the operator to evaluate to `True`

`if username == "bmoreno" and login_attempts < 5:`

Evaluates to `True` if the value in the `username` variable is equal to `"bmoreno"` and the value in the `login_attempts` variable is less than `5`

`or`

Requires only one of the conditions on either side of the operator to evaluate to `True`

`if status == 100 or status == 102:`

Evaluates to `True` if the value in the `status` variable is equal to `100` or the value in the `status` variable is equal to `102`

`not`

Negates a given condition so that it evaluates to `False` if the condition is `True` and to `True` if it is `False`

```
if not account_status == "removed"
```

Evaluates to `False` if the value in the `account_status` variable is equal to `"removed"` and evaluates to `True` if the value is the `account_status` variable is not equal to `"removed"`

Iterative statements

The following keywords are used in iterative statements.

`for`

Signals the beginning of a `for` loop; used to iterate through a specified sequence

```
for username in ["bmoreno", "tshah", "elarson"]:
```

Signals the beginning of a `for` loop that iterates through the sequence of elements in the list `["bmoreno", "tshah", "elarson"]` using the loop variable `username`

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

Signals the beginning of a `for` loop that iterates through a sequence of numbers created by `range(10)` using the loop variable `i`

`while`

Signals the beginning of a `while` loop; used to iterate based on a condition

```
while login_attempts < 5:
```

Signals the beginning of a `while` loop that will iterate as long as the condition that the value of `login_attempts` is less than `5` evaluates to `True`

`break`

Used to break out of a loop

`continue`

Used to skip a loop iteration and continue with the next one

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.

`print()`

Outputs a specified object to the screen

```
print("login success")
```

Outputs the string `"login success"` to the screen

```
print(9 < 7)
```

Outputs the Boolean value of `False` to the screen after evaluating whether the integer `9` is less than the integer `7`

`type()`

Returns the data type of its input

`print(type(51.1))`

Returns the data type of float for the input of `51.1`

`print(type(True))`

Returns the data type of Boolean for the input of `True`

`range()`

Generates a sequence of numbers

`range(0, 5, 1)`

Generates a sequence with a start point of `0`, a stop point of `5`, and an increment of `1`; because the start point is inclusive but the stop point is exclusive, the generated sequence is `0`, `1`, `2`, `3`, and `4`

`range(5)`

Generates a sequence with a stop point of `5`; when the start point is not specified, it is set at the default value of `0`, and when the increment is not specified, it is set at the default value of `1`; the generated sequence is `0`, `1`, `2`, `3`, and `4`

`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

`str()`

Converts the input object to a string

```
str(10)
```

Converts the integer `10` to the string `"10"`

`len()`

Returns the number of elements in an object

```
print(len("security"))
```

Returns and displays `8`, the number of characters in the string `"security"`

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

String methods

The following methods can be applied to strings in Python.

`.upper()`

Returns a copy of the string in all uppercase letters

```
print("Security".upper())
```

Returns and displays a copy of the string `"Security"` as `"SECURITY"`

`.lower()`

Returns a copy of the string in all lowercase letters

```
print("Security".lower())
```

Returns and displays a copy of the string `"Security"` as `"security"`

`.index()`

Finds the first occurrence of the input in a string and returns its location

```
print("Security".index("c"))
```

Finds the first occurrence of the character `"c"` in the string `"Security"` and returns and displays its index of 2

List methods

The following methods can be applied to lists in Python.

`.insert()`

Adds an element in a specific position inside the list

```
username_list = ["elarson", "fgarcia", "tshah"]
 username_list.insert(2,"wjaffrey")
```

Adds the element `"wjaffrey"` at index 2 to the `username_list`; the list becomes `["elarson", "fgarcia", "wjaffrey", "tshah"]`

`.remove()`

Removes the first occurrence of a specific element inside a list

```
username_list = ["elarson", "bmoreno", "wjaffrey", "tshah"]
username_list.remove("elarson")
```

Removes the element `"elarson"` from the `username_list`; the list becomes `["fgarcia", "wjaffrey", "tshah"]`

`.append()`

Adds input to the end of a list

```
username_list = ["bmoreno", "wjaffrey", "tshah"]
username_list.append("btang")
```

Adds the element `"btang"` to the end of the `username_list`; the list becomes `["fgarcia", "wjaffrey", "tshah", "btang"]`

`.index()`

Finds the first occurrence of an element in a list and returns its index

```
username_list = ["bmoreno", "wjaffrey", "tshah", "btang"]
print(username_list.index("tshah"))
```

Finds the first occurrence of the element `"tshah"` in the `username_list` and returns and displays its index of 2

Additional syntax for working with strings and lists

The following syntax is useful when working with strings and lists.

`+` (concatenation)

Combines two strings or lists together

```
device_id = "IT"+"nwp12"
```

Combines the string `"IT"` with the string `"nwp12"` and assigns the combined string of `"ITnwp12"` to the variable `device_id`

```
users = ["elarson", "bmoreno"] + ["tshah", "btang"]
```

Combines the list `["elarson", "bmoreno"]` with the list `["tshah", "btang"]` and assigns the combined list of `["elarson", "bmoreno", "tshah", "btang"]` to the variable `users`

`[]` (bracket notation)

Uses indices to extract parts of a string or list

```
print("h32rb17"[0])
```
Extracts the character at index `0`, which is (`"h"`), from the string `"h32rb17"`

```
print("h32rb17"[0:3])
```
Extracts the slice `[0:3]`, which is (`"h32"`), from the string `"h32rb17"`; the first index in the slice (`0`) is included in the slice but the second index in the slice (3) is excluded

```
username_list = ["elarson", "fgarcia", "tshah"]
print(username_list[2])
```
Extracts the element at index 2, which is (`"tshah"`), from the `username_list`

Regular expressions

The following `re` module function and regular expression symbols are useful when searching for patterns in strings.

`re.findall()`

Returns a list of matches to a regular expression

```
import re
```
`re.findall("a53", "a53-32c .E")`
Returns a list of matches to the regular expression pattern `"a53"` in the string `"a53-32c .E"`; returns the list `["a53"]`

`\w`

Matches with any alphanumeric character; also matches with the underscore (_)

```
import re
```
`re.findall("\w", "a53-32c .E")`
Returns a list of matches to the regular expression pattern `"\w"` in the string `"a53-32c .E"`; matches to any alphanumeric character and returns the list `["a", "5", "3", "3", "2", "c", "E"]`

`.`

Matches to all characters, including symbols

```
import re
```

`re.findall(".", "a53-32c .E")`
Returns a list of matches to the regular expression pattern `"."` in the string `"a53-32c .E"` ; matches to all characters and returns the list `["a", "5", "3", "-", "3", "2", "c", " ", ".", "E"]`

`\d`

Matches to all single digits

```
import re
```

`re.findall("\d", "a53-32c .E")`
Returns a list of matches to the regular expression pattern `"\d"` in the string `"a53-32c .E"`; matches to all single digits and returns the list `["5", "3", "3", "2"]`

`\s`

Matches to all single spaces

```
import re
```

`re.findall("\d", "a53-32c .E")`
Returns a list of matches to the regular expression pattern `"\s"` in the string `"a53-32c .E"`; matches to all single spaces and returns the list `[" "]`

`\.`

Matches to the period character

```
import re
```

`re.findall("\.", "a53-32c .E")`
Returns a list of matches to the regular expression pattern `"\."` in the string `"a53-32c .E"`; matches to all instances of the period character and returns the list `["."]`

`+`

Represents one or more occurrences of a specific character

```
import re
```

`re.findall("\w+", "a53-32c .E")`
Returns a list of matches to the regular expression pattern `"\w+"` in the string `"a53-32c .E"`; matches to one or more occurrences of any alphanumeric character and returns the list `["a53", "32c", "E"]`

`*`

Represents, zero, one or more occurrences of a specific character

```
import re
```

`re.findall("\w*", "a53-32c .E")`
Returns a list of matches to the regular expression pattern `"\w*"` in the string `"a53-32c .E"`; matches to zero, one or more occurrences of any alphanumeric character and returns the list `["a53", " ", "32c", " ", " ", "E"]`

`{ }`

Represents a specified number of occurrences of a specific character; the number is specified within the curly brackets

```
import re
```

`re.findall("\w{3}", "a53-32c .E")`
Returns a list of matches to the regular expression pattern `"\w{3}"` in the string `"a53-32c .E"`; matches to exactly three occurrences of any alphanumeric character and returns the list `["a53","32c"]`

File operations

The following functions, methods, and keywords are used with operations involving files.

`with`

Handles errors and manages external resources

```
with open("logs.txt", "r") as file:
```

Used to handle errors and manage external resources while opening a file; the variable `file` stores the file information while inside of the `with` statement; manages resources by closing the file after exiting the `with` statement

`open()`

Opens a file in Python

```
with open("login_attempts.txt", "r") as file:
```

Opens the file `"login_attempts.txt"` in order to read it `("r")`

```
with open("update_log.txt", "w") as file:
```

Opens the file `"update_log.txt"` into the variable `file` in order to write over its contents `("w")`

```
with open(import_file, "a") as file:
```

Opens the file assigned to the `import_file` variable into the variable `file` in order to append information to the end of it `("a")`

as

Assigns a variable that references another object

```
with open("logs.txt", "r") as file:
```

Assigns the `file` variable to reference the output of the `open()` function

.read()

Converts files into strings; returns the content of an open file as a string by default

```
with open("login_attempts.txt", "r") as file:
    file_text = file.read()
```

Converts the file object referenced in the `file` variable into a string and then stores this string in the `file_text` variable

.write()

Writes string data to a specified file

```
with open("access_log.txt", "a") as file:
        file.write("jrafael")
```

Writes the string "jrafael" to the `"access_log.txt"` file; because the second argument in the call to the `open()` function is `"a"`, this string is appended to the end of the file

Parsing

The following methods are useful when parsing data.

.split()

Converts a string into a list; separates the string based on the character that is passed in as an argument; if an argument is not passed in, it will separate the string each time it encounters whitespace characters such as a space or return

```
approved_users = "elarson,bmoreno,tshah".split(",")
```

Converts the string `"elarson,bmoreno,tshah"` into the list `["elarson","bmoreno","tshah"]` by splitting the string into a separate list element at

each occurrence of the `","` character

`removed_users = "wjaffrey jsoto abernard".split()`
Converts the string `"wjaffrey jsoto abernard"` into the list `["wjaffrey","jsoto","abernard"]` by splitting the string into a separate list element at each space

`.join()`

Concatenates the elements of an iterable into a string; takes the iterable to be concatenated as an argument; is appended to a character that will separate each element once they are joined into a string

`approved_users = ",".join(["elarson", "bmoreno", "tshah"])`
Concatenates the elements of the list `["elarson","bmoreno","tshah"]` into the string `"elarson,bmoreno,tshah"` , separating each element with the `","` character within the string

Terms and definitions from Course 7

A

Algorithm: A set of rules that solve a problem

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

Automation: The use of technology to reduce human and manual effort to perform common and repetitive tasks

B

Boolean data: Data that can only be one of two values: either `True` or `False`

Bracket notation: The indices placed in square brackets

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

C

Command-line interface: A text-based user interface that uses commands to interact with the computer

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

Conditional statement: A statement that evaluates code to determine if it meets a specified set of conditions

D

Data type: A category for a particular type of data item

Debugger: A software tool that helps to locate the source of an error and assess its causes

Debugging: The practice of identifying and fixing errors in code

Dictionary data: Data that consists of one or more key-value pairs

E

Exception: An error that involves code that cannot be executed even though it is syntactically correct

F

File path: The location of a file or directory

Float data: Data consisting of a number with a decimal point

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

G

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

I

Immutable: An object that cannot be changed after it is created and assigned a value

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

Index: A number assigned to every element in a sequence that indicates its position

Integer data: Data consisting of a number that does not include a decimal point

Integrated development environment (IDE): A software application for writing code that provides editing assistance and error correction tools

Interpreter: A computer program that translates Python code into runnable instructions line by line

Iterative statement: Code that repeatedly executes a set of instructions

L

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

List concatenation: The concept of combining two lists into one by placing the elements of the second list directly after the elements of the first list

List data: Data structure that consists of a collection of data in sequential form

Local variable: A variable assigned within a function

Log: A record of events that occur within an organization's systems

Logic error: An error that results when the logic used in code produces unintended results

Loop variable: A variable that is used to control the iterations of a loop

M

Method: A function that belongs to a specific data type

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

N

Notebook: An online interface for writing, storing, and running code

P

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

Parsing: The process of converting data into a more readable format

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

Programming: A process that can be used to create a specific set of instructions for a computer to execute tasks

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

R

Regular expression (regex): A sequence of characters that forms a pattern

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

S

Set data: Data that consists of an unordered collection of unique values

String concatenation: The process of joining two strings together

String data: Data consisting of an ordered sequence of characters

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

Substring: A continuous sequence of characters within a string

Syntax: The rules that determine what is correctly structured in a computing language

Syntax error: An error that involves invalid usage of a programming language

T

Tuple data: Data structure that consists of a collection of data that cannot be changed

Type error: An error that results from using the wrong data type

U

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

V

Variable: A container that stores data