# Automate Cybersecurity Tasks with Python

# Module 1 - Introduction to Python

## How programming works

**Programming** is a process that can be used to create a specific set of instructions for a computer to execute tasks. Computer programs exist everywhere. Computers, cell phones, and many other electronic devices are all given instructions by computer programs.
There are multiple programming languages used to create computer programs. Python is one of these. Programming languages are converted to binary numbers, which are a series of 0s and 1s that represent the operations that the computer's central processing unit (CPU) should perform. Each instruction corresponds to a specific operation, such as adding two numbers or loading a value from memory.
It would be very time-consuming for humans to communicate this way. Programming languages like Python make it easier to write code because you can use less syntax when instructing computers to perform complex processes.

## Using Python to program

Python is a general purpose programming language that can be used to solve a variety of problems. For example, it can be used to build websites, perform data analysis, and automate tasks.
Python code must be converted through an interpreter before the computer can process it. An **interpreter** is a computer program that translates Python code into runnable instructions line by line.

### Python versions

There are multiple versions of Python. In this course, you are using Python 3. While using Python, it's important to keep track of the version you're using. There are differences in the syntax of each version. **Syntax** refers to the rules that determine what is correctly structured in a computing language.

## Python in cybersecurity

In cybersecurity, Python is used especially for automation. **Automation** is the use of technology to reduce human and manual effort to perform common and repetitive tasks. These are some specific areas of cybersecurity in which Python might be used to automate specific tasks:

- Log analysis
- Malware analysis
- Access control list management
- Intrusion detection
- Compliance checks
- Network scanning

You can run Python through a variety of environments. These environments include notebooks, integrated development environments (IDEs), and the command line. This reading will introduce you to these environments. It will focus primarily on notebooks because this is how you'll interact with Python in this course.

# Notebooks

One way to write Python code is through a notebook. In this course, you'll interact with Python through notebooks. A **notebook** is an online interface for writing, storing, and running code. They also allow you to document information about the code. Notebook content either appears in a code cell or markdown cell.

### Code cells

Code cells are meant for writing and running code. A notebook provides a mechanism for running these code cells. Often, this is a play button located within the cell. When you run the code, its output appears after the code.

### Markdown cells

Markdown cells are meant for describing the code. They allow you to format text in the markdown language. Markdown language is used for formatting plain text in text editors and code editors. For example, you might indicate that text should be in a certain header style.

### Common notebook environments

Two common notebook environments are Jupyter Notebook and Google Colaboratory (or Google Colab). They allow you to run several programming languages, including Python.

# Integrated development environments (IDEs)

Another option for writing Python code is through an **integrated development environment (IDE),** or a software application for writing code that provides editing assistance and error correction tools. Integrated development environments include a graphical user interface (GUI) that provides programmers with a variety of options to customize and build their programs.

# Command line

The command line is another environment that allows you to run Python programs. Previously, you learned that a **command-line interface (CLI)** is a text-based user interface that uses commands to interact with the computer. By entering commands into the command line, you can access all files and directories saved on your hard drive, including files containing Python code you want to run. You can also use the command line to open a file editor and create a new Python file.

# String

In Python, **string data** is data consisting of an ordered sequence of characters. Characters in a string may include letters, numbers, symbols, and spaces. These characters must be placed within quotation marks. These are all valid strings:

- `"updates needed"`
- `"20%"`
- `"5.0"`
- `"35"`
- `"**/**/**"`
- `""`

**Note:** The last item (`""`), which doesn't contain anything within the quotation marks, is called an empty string.
You can use the `print()` function to display a string. You can explore this by running this code:

```
print("updates needed")
```

The code prints `"updates needed"`.

You can place strings in either double quotation marks (`""`) or single quotation marks (`''`). The following code demonstrates that the same message prints when the string is in single quotation marks:

```
print('updates needed')
```

**Note:** Choosing one type of quotation marks and using it consistently makes it easier to read your code. This course uses double quotation marks.

# List

In Python, **list data** is a data structure that consists of a collection of data in sequential form. Lists elements can be of any data type, such as strings, integers, Booleans, or even other lists. The elements of a list are placed within square brackets, and each element is separated by a comma. The following lists contains elements of various data types:

- `[12, 36, 54, 1, 7]`
- `["eraab", "arusso", "drosas"]`
- `[True, False, True, True]`
- `[15, "approved", True, 45.5, False]`
- `[]`

**Note:** The last item `[]`, which doesn't contain anything within the brackets, is called an empty list.
You can also use the `print()` function to display a list:

```
print([12, 36, 54, 1, 7])
```

This displays a list containing the integers `12`, `36`, `54`, `1`, and `7`.

## Integer

In Python, **integer data** is data consisting of a number that does not include a decimal point. These are all examples of integer data:

- `-100`
- `-12`
- `-1`

- `0`
- `1`
- `20`
- `500`

Integers are not placed in quotation marks. You can use the `print()` function to display an integer. When you run this code, it displays `5`:

```
print(5)
```

You can also use the `print()` function to perform mathematical operations with integers. For example, this code adds two integers:

```
print(5 + 2)
```

The result is `7`. You can also subtract, multiply, or divide two integers.

# Float

**Float data** is data consisting of a number with a decimal point. All of the following are examples of float data:

- `-2.2`
- `-1.34`
- `0.0`
- `0.34`

Just like integer data, float data is not placed in quotation marks. In addition, you can also use the `print()` function to display float data or to perform mathematical calculations with float data. You can run the following code to review the result of this calculation:

```
print(1.2 + 2.8)
```

The output is `4.0`.
**Note:** Dividing two integer values or two float values results in float output when you use the symbol `/`:

```
print(1/4)
```

```
print(1.0/4.0)
```

The output of both calculations is the float value of `.25`.
If you want to return a whole number from a calculation, you must use the symbol `//` instead:

```
print(1//4)
```

```
print(1.0//4.0)
```

It will round down to the nearest whole number. In the case of `print(1//4)`, the output is the integer value of `0` because using this symbol rounds down the calculation from `.25` to the nearest whole number. In the case of `print(1.0//4.0)`, the output is the float value of `0.0`

because it maintains the float data type of the values in the calculation while also rounding down to the nearest whole number.

# Boolean

Boolean data is data that can only be one of two values: either **`True`** or **`False`**.
You should not place Boolean values in quotation marks. When you run the following code, it displays the Boolean value of **`True`**:

```
print(True)
```

You can also return a Boolean value by comparing numbers. Because **`9`** is not greater than **`10`**, this code evaluates to **`False`**:

```
print(9 > 10)
```

# Additional data types

In this course, you will work with the string, list, integer, float and Boolean data types, but there are other data types. These additional data types include tuple data, dictionary data, and set data.

## Tuple

**Tuple data** is a data structure that consists of a collection of data that cannot be changed. Like lists, tuples can contain elements of varying data types.
A difference between tuple data and list data is that it is possible to change the elements in a list, but it is not possible to change the elements in a tuple. This could be useful in a cybersecurity context. For example, if software identifiers are stored in a tuple to ensure that they will not be altered, this can provide assurance that an access control list will only block the intended software.
The syntax of a tuple is also different from the syntax of a list. A tuple is placed in parentheses rather than brackets. These are all examples of the tuple data type:

- **`("wjaffrey", "arutley", "dkot")`**
- **`(46, 2, 13, 2, 8, 0, 0)`**
- **`(True, False, True, True)`**
- **`("wjaffrey", 13, True)`**

**Pro tip:** Tuples are more memory efficient than lists, so they are useful when you are working with a large quantity of data.

## Dictionary

**Dictionary data** is data that consists of one or more key-value pairs. Each key is mapped to a value. A colon (`:`) is placed between the key and value. Commas separate key-value pairs from other key-value pairs, and the dictionary is placed within curly brackets (`{}`).
Dictionaries are useful when you want to store and retrieve data in a predictable way. For example, the following dictionary maps a building name to a number. The building name is the value, and the number is the key. A colon is placed after the key.

```
{ 1: "East",
```

```
  2: "West",
  3: "North",
  4: "South" }
```

### Set

In Python, **set data** is data that consists of an unordered collection of unique values. This means no two values in a set can be the same.
Elements in a set are always placed within curly brackets and are separated by a comma. These elements can be of any data type. This example of a set contains strings of usernames:
`{"jlanksy", "drosas", "nmason"}`

## What are variables?

In a programming language, a **variable** is a container that stores data. It's a named storage location in a computer's memory that can hold a value. It stores the data in a particular data type, such as integer, string, or Boolean. The value that is stored in a variable can change.
You can think of variables as boxes with labels on them. Even when you change the contents of a box, the label on the box itself remains the same. Similarly, when you change the value stored in a variable, the name of the variable remains the same.
Security analysts working in Python will use a variety of variables. Some examples include variables for login attempts, allow lists, and addresses.

## Working with variables

In Python, it's important to know both how to assign variables and how to reassign them.

### Assigning and reassigning variables

If you want to create a variable called `username` and assign it a value of `"nzhao"`, place the variable to the left of the equals sign and its value to the right:

```
# Assign 'username'
username = "nzhao"
```

If you later reset this username to `"zhao2"`, you still refer to that variable container as `username`.

```
# Reassign 'username'
username = "zhao2"
```

Although the contents have changed from `"nzhao"` to `"zhao2"`, the variable `username` remains the same.
**Note:** You must place `"nzhao"` and `"zhao2"` in quotation marks because they're strings. Python automatically assigns a variable its data type when it runs. For example, when the `username` variable contains the string `"nzhao"`, it's assigned a string data type.

### Assigning variables to variables

Using a similar process, you can also assign variables to other variables. In the following example, the variable `username` is assigned to a new variable `old_username`:

```
# Assign a variable to another variable
username = "nzhao"
old_username = username
```

Because **username** contains the string value of **"nzhao"** and **old_username** contains the value of **username**, **old_username** now contains a value of **"nzhao"**.

### Putting it together

The following code demonstrates how a username can be updated. The **username** variable is assigned an initial value, which is then stored in a second variable called **old_username**. After this, the **username** variable is reassigned a new value. You can run this code to get a message about the previous username and the current username:

```
username = "nzhao"

old_username = username

username = "zhao2"

print("Previous username:", old_username)

print("Current username:", username)
```

## Best practices for naming variables

You can name a variable almost anything you want, but there are a few guidelines you should follow to ensure correct syntax and prevent errors:

- Use only letters, numbers, and underscores in variable names. Valid examples: **date_3**, **username**, **interval2**
- Start a variable name with a letter or underscore. Do not start it with a number. Valid examples: **time**, **_login**
- Remember that variable names in Python are case-sensitive. These are all different variables: **time**, **Time**, **TIME**, **timE**.
- Don't use Python's built-in keywords or functions for variable names. For example, variables shouldn't be named **True**, **False**, or **if**.

Additionally, you should follow these stylistic guidelines to make your code easier for you and other security analysts to read and understand:

- Separate two or more words with underscores. Valid examples: **login_attempts**, **invalid_user**, **status_update**
- Avoid variables with similar names. These variables could be easily confused with one another: **start_time**, **starting_time**, **time_starting**.
- Avoid unnecessarily long names for variables. For instance, don't give variables names like **variable_that_equals_3**.
- Names should describe the data and not be random words. Valid examples: **num_login_attempts**, **device_id**, **invalid_usernames**

**Note**: Using underscores to separate multiple words in variables is recommended, but another convention that you might encounter is capitalizing the first letter of each word except the first word. Example: **loginAttempt**

## How conditional statements work

A **conditional statement** is a statement that evaluates code to determine whether it meets a specific set of conditions. When a condition is met, it evaluates to a Boolean value of `True` and performs specified actions. When the condition isn't met, it evaluates a Boolean value of `False` and doesn't perform the specified actions.
In conditional statements, the condition is often based on a comparison of two values. This table summarizes common comparison operators used to compare numerical values.

| operator | use |
|---|---|
| `>` | greater than |
| `<` | less than |
| `>=` | greater than or equal to |
| `<=` | less than or equal to |
| `==` | equal to |
| `!=` | not equal to |

**Note:** The equal to (`==`) and not equal to (`!=`) operators are also commonly used to compare string data.

# if statements

The keyword `if` starts a conditional statement. It's a necessary component of any conditional statement. In the following example, `if` begins a statement that tells Python to print an `"OK"` message when the HTTP response status code equals `200`:

```
if status == 200:
    print("OK")
```

This code consists of a header and a body.

## The header of an if statement

The first line of this code is the header. In the header of an `if` statement, the keyword `if` is followed by the condition. Here, the condition is that the `status` variable is equal to a value of `200`. The condition can be placed in parentheses:

```
if (status == 200):
    print("OK")
```

In cases like this one, placing parentheses around conditions in Python is optional. You might want to include them if it helps you with code readability. However, this condition will be processed the same way if written without parentheses.
In other situations, because Python evaluates the conditions in parentheses first, parentheses can affect how Python processes conditions. You will read more about one of these in the section of this reading on `not`.
**Note:** You must always place a colon (`:`) at the end of the header. Without this syntax, the code will produce an error.
**The body of an if statement**

After the header of an `if` statement comes the body of the `if` statement. This tells Python what action or actions to perform when the condition evaluates to `True`. In this example, there is just one action, printing `"OK"` to the screen. In other cases, there might be more lines of code with additional actions.

**Note:** For the body of the `if` statement to execute as intended, it must be indented further than the header. Additionally, if there are multiple lines of code within the body, they must all be indented consistently.

## Continuing conditionals with else and elif

In the previous example, if the HTTP status response code was not equal to `200`, the condition would evaluate to `False` and Python would continue with the rest of the program. However, it's also possible to specify alternative actions with `else` and `elif`.

### else statements

The keyword `else` precedes a code section that only evaluates when all conditions that precede it within the conditional statement evaluate to `False`.

In the following example, when the HTTP response status code is not equal to `200`, it prints an alternative message of `"check other status"`:

```
if status == 200:
    print("OK")
else:
    print("check other status")
```

**Note:** Like with `if`, a colon (`:`) is required after `else`, and the body that follows the `else` header is indented.

### elif statements

In some cases, you might have multiple alternative actions that depend on new conditions. In that case, you can use `elif`. The `elif` keyword precedes a condition that is only evaluated when previous conditions evaluate to `False`. Unlike with `else`, there can be multiple `elif` statements following `if`.

For example, you might want to print one message if the HTTP response status code is `200`, one message if it is `400`, and one if it is `500`. The following code demonstrates how you can use `elif` for this:

```
if status == 200:
    print("OK")
elif status == 400:
    print("Bad Request")
elif status == 500:
    print("Internal Server Error")
```

Python will first check if the value of `status` is `200`, and if this evaluates to `False`, it will go onto the first `elif` statement. There, it will check whether the value of `status` is `400`. If that evaluates to `True`, it will print `"Bad Request"`, but if it evaluates to `False`, it will go on to the next `elif` statement.

If you want the code to print another message when all conditions evaluate to `False`, then you can incorporate `else` after the last `elif`. In this example, if it reaches the `else` statement, it prints a message to check the status:

```
if status == 200:
    print("OK")
elif status == 400:
    print("Bad Request")
elif status == 500:
    print("Internal Server Error")
else:
    print("check other status")
```

Just like with `if` and `else`, it's important to place a colon (`:`) after the `elif` header and indent the code that follows this header.

**Note:** Python processes multiple `elif` statements differently than multiple `if` statements. When it reaches an `elif` statement that evaluates to `True`, it won't check the following `elif` statements. On the other hand, Python will run all `if` statements.

## Logical operators for multiple conditions

In some cases, you might want Python to perform an action based on a more complex condition. You might require two conditions to evaluate to `True`. Or, you might require only one of two conditions to evaluate to `True`. Or, you might want Python to perform an action when a condition evaluates to `False`. The operators `and`, `or`, and `not` can be used in these cases.

### and

The `and` operator requires both conditions on either side of the operator to evaluate to `True`. For example, all HTTP status response codes between `200` and `226` relate to successful responses. You can use `and` to join a condition of being greater than or equal to `200` with another condition of being less than or equal to `226`:

```
if status >= 200 and status <= 226:
    print("successful response")
```

When both conditions are `True`, then the `"successful response"` message will print.

### or

The `or` operator requires only one of the conditions on either side of the operator to evaluate to `True`. For example, both a status code of `100` and a status code of `102` are informational responses. Using `or`, you could ask Python to print an `"informational response"` message when the code is either `100` or `102`:

```
if status == 100 or status == 102:
    print("informational response")
```

Only one of these conditions needs to be met for Python to print the message.

### not

The `not` operator negates a given condition so that it evaluates to `False` if the condition is `True` and to `True` if it is `False`. For example, if you want to indicate that Python should check the status code when it's something outside of the successful range, you can use `not`:

```
if not(status >= 200 and status <= 226):
    print("check status")
```
Python first checks whether the value of status is greater than or equal to `200` and less than or equal to `226`, and then because of the operator `not`, it inverts this. This means it will print the message if `status` is less than `200` or greater than `226`.
**Note:** In this case, the parentheses are necessary for the code to apply `not` to both conditions. Python will evaluate the conditions within the parentheses first. This means it will first evaluate the conditions on either side of the `and` operator and then apply `not` to both of them.

# How conditional statements work

A **conditional statement** is a statement that evaluates code to determine whether it meets a specific set of conditions. When a condition is met, it evaluates to a Boolean value of `True` and performs specified actions. When the condition isn't met, it evaluates a Boolean value of `False` and doesn't perform the specified actions.
In conditional statements, the condition is often based on a comparison of two values. This table summarizes common comparison operators used to compare numerical values.

| operator | use |
|---|---|
| > | greater than |
| < | less than |
| >= | greater than or equal to |
| <= | less than or equal to |
| == | equal to |
| != | not equal to |

**Note:** The equal to (==) and not equal to (`!=`) operators are also commonly used to compare string data.

# if statements

The keyword `if` starts a conditional statement. It's a necessary component of any conditional statement. In the following example, `if` begins a statement that tells Python to print an `"OK"` message when the HTTP response status code equals `200`:
```
if status == 200:
    print("OK")
```
This code consists of a header and a body.

## The header of an if statement

The first line of this code is the header. In the header of an `if` statement, the keyword `if` is followed by the condition. Here, the condition is that the `status` variable is equal to a value of `200`. The condition can be placed in parentheses:
```
if (status == 200):
```

```
    print("OK")
```
In cases like this one, placing parentheses around conditions in Python is optional. You might want to include them if it helps you with code readability. However, this condition will be processed the same way if written without parentheses.
In other situations, because Python evaluates the conditions in parentheses first, parentheses can affect how Python processes conditions. You will read more about one of these in the section of this reading on **`not`**.
**Note:** You must always place a colon (`:`) at the end of the header. Without this syntax, the code will produce an error.
**The body of an if statement**
After the header of an **`if`** statement comes the body of the **`if`** statement. This tells Python what action or actions to perform when the condition evaluates to **`True`**. In this example, there is just one action, printing **`"OK"`** to the screen. In other cases, there might be more lines of code with additional actions.
**Note:** For the body of the **`if`** statement to execute as intended, it must be indented further than the header. Additionally, if there are multiple lines of code within the body, they must all be indented consistently.

# Continuing conditionals with else and elif

In the previous example, if the HTTP status response code was not equal to **`200`**, the condition would evaluate to **`False`** and Python would continue with the rest of the program. However, it's also possible to specify alternative actions with **`else`** and **`elif`**.

## else statements

The keyword **`else`** precedes a code section that only evaluates when all conditions that precede it within the conditional statement evaluate to **`False`**.
In the following example, when the HTTP response status code is not equal to **`200`**, it prints an alternative message of **`"check other status"`**:
```
if status == 200:
    print("OK")
else:
    print("check other status")
```
**Note:** Like with **`if`**, a colon (`:`) is required after **`else`**, and the body that follows the **`else`** header is indented.

## elif statements

In some cases, you might have multiple alternative actions that depend on new conditions. In that case, you can use **`elif`**. The **`elif`** keyword precedes a condition that is only evaluated when previous conditions evaluate to **`False`**. Unlike with **`else`**, there can be multiple **`elif`** statements following **`if`**.
For example, you might want to print one message if the HTTP response status code is **`200`**, one message if it is **`400`**, and one if it is **`500`**. The following code demonstrates how you can use **`elif`** for this:
```
if status == 200:
    print("OK")
```

```
elif status == 400:
    print("Bad Request")
elif status == 500:
    print("Internal Server Error")
```

Python will first check if the value of `status` is `200`, and if this evaluates to `False`, it will go onto the first `elif` statement. There, it will check whether the value of `status` is `400`. If that evaluates to `True`, it will print `"Bad Request"`, but if it evaluates to `False`, it will go on to the next `elif` statement.

If you want the code to print another message when all conditions evaluate to `False`, then you can incorporate `else` after the last `elif`. In this example, if it reaches the `else` statement, it prints a message to check the status:

```
if status == 200:
    print("OK")
elif status == 400:
    print("Bad Request")
elif status == 500:
    print("Internal Server Error")
else:
    print("check other status")
```

Just like with `if` and `else`, it's important to place a colon (`:`) after the `elif` header and indent the code that follows this header.

**Note:** Python processes multiple `elif` statements differently than multiple `if` statements. When it reaches an `elif` statement that evaluates to `True`, it won't check the following `elif` statements. On the other hand, Python will run all `if` statements.

## Logical operators for multiple conditions

In some cases, you might want Python to perform an action based on a more complex condition. You might require two conditions to evaluate to `True`. Or, you might require only one of two conditions to evaluate to `True`. Or, you might want Python to perform an action when a condition evaluates to `False`. The operators `and`, `or`, and `not` can be used in these cases.

### and

The `and` operator requires both conditions on either side of the operator to evaluate to `True`. For example, all HTTP status response codes between `200` and `226` relate to successful responses. You can use `and` to join a condition of being greater than or equal to `200` with another condition of being less than or equal to `226`:

```
if status >= 200 and status <= 226:
    print("successful response")
```

When both conditions are `True`, then the `"successful response"` message will print.

### or

The `or` operator requires only one of the conditions on either side of the operator to evaluate to `True`. For example, both a status code of `100` and a status code of `102` are informational responses. Using `or`, you could ask Python to print an `"informational response"` message when the code is either `100` or `102`:

```
if status == 100 or status == 102:
    print("informational response")
```
Only one of these conditions needs to be met for Python to print the message.

### not

The `not` operator negates a given condition so that it evaluates to `False` if the condition is `True` and to `True` if it is `False`. For example, if you want to indicate that Python should check the status code when it's something outside of the successful range, you can use `not`:

```
if not(status >= 200 and status <= 226):
    print("check status")
```

Python first checks whether the value of status is greater than or equal to `200` and less than or equal to `226`, and then because of the operator `not`, it inverts this. This means it will print the message if `status` is less than `200` or greater than `226`.

**Note:** In this case, the parentheses are necessary for the code to apply `not` to both conditions. Python will evaluate the conditions within the parentheses first. This means it will first evaluate the conditions on either side of the `and` operator and then apply `not` to both of them.

## Module 2 - Write effective python code

# Functions in cybersecurity

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

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

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

# Defining a function

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

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

### Function header

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

```
def display_investigation_message():
```

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

The parentheses that follow the name of the function and the colon (`:`) at the end of the function header are also essential parts of the syntax.
**Pro tip**: When naming a function, give it a name that indicates what it does. This will make it easier to remember when calling it later.

### Function body

The body of the function is an indented block of code after the function header that defines what the function does. The indentation is very important when writing a function because it separates the definition of a function from the rest of the code.
To add a body to your definition of the **`display_investigation_message()`** function, add an indented line with the **`print()`** function. Your function definition becomes the following:

```
def display_investigation_message():
    print("investigate activity")
```

## Calling a function

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

```
display_investigation_message()
```

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

```
def display_investigation_message():

    print("investigate activity")

application_status = "potential concern"

email_status = "okay"

if application_status == "potential concern":

    print("application_log:")

    display_investigation_message()

if email_status == "potential concern":

    print("email log:")

    display_investigation_message()
```

The **`display_investigation_message()`** function is used twice within the code. It will print **`"investigate activity"`** messages about two different logs when the specified conditions

evaluate to `True`. In this example, only the first conditional statement evaluates to `True`, so the message prints once.

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

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

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

# Working with variables in functions

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

## Parameters

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

```
def remaining_login_attempts(maximum_attempts, total_attempts):
    print(maximum_attempts - total_attempts)
```

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

## Arguments

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

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

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

## Return statements

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

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

```
def remaining_login_attempts(maximum_attempts, total_attempts):
    return maximum_attempts - total_attempts
```

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

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

```
def remaining_login_attempts(maximum_attempts, total_attempts):

    return maximum_attempts - total_attempts

remaining_attempts = remaining_login_attempts(3, 3)

if remaining_attempts <= 0:

    print("Your account is locked")
```

In this example, the message prints because the calculation in the function results in `0`.
**Note:** When Python encounters a **`return`** statement, it executes this statement and then exits the function. If there are lines of code that follow the **`return`** statement within the function, they will not be run. The previous example didn't contain any lines of code after the **`return`** statement, but this might apply in other functions, such as one containing a conditional statement.

# Global and local variables

To better understand how functions interact with variables, you should know the difference between global and local variables.
When defining and calling functions, you're working with local variables, which are different from the variables you define outside the scope of a function.

## Global variables

A **global variable** is a variable that is available through the entire program. Global variables are assigned outside of a function definition. Whenever that variable is called, whether inside or outside a function, it will return the value it is assigned.
For example, you might assign the following variable at the beginning of your code:
**`device_id = "7ad2130bd"`**
Throughout the rest of your code, you will be able to access and modify the **`device_id`** variable in conditionals, loops, functions, and other syntax.

## Local variables

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

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

```
def greet_employee(name):
    total_string = "Welcome" + name
    return total_string
```

The variable **`total_string`** is a local variable because it's assigned inside of the function. The parameter **`name`** is a local variable because it is also created when the function is defined.
Whenever you call a function, Python creates these variables temporarily while the function is running and deletes them from memory after the function stops running.
This means that if you call the **`greet_employee()`** function with an argument and then use the **`total_string`** variable outside of this function, you'll get an error.

## Best practices for global and local variables

When working with variables and functions, it is very important to make sure that you only use a certain variable name once, even if one is defined globally and the other is defined locally.
When using global variables inside functions, functions can access the values of a global variable. You can run the following example to explore this:

```
username = "elarson"

def identify_user():

    print(username)

identify_user()
```

The code block returns **`"elarson"`** even though that name isn't defined locally. The function accesses the global variable. If you wanted the **`identify_user()`** function to accommodate other usernames, you would have to reassign the global username variable outside of the function. This isn't good practice. A better way to pass different values into a function is to use a parameter instead of a global variable.
There's something else to consider too. If you reuse the name of a global variable within a function, it will create a new local variable with that name. In other words, there will be both a global variable with that name and a local variable with that name, and they'll have different values. You can consider the following code block:

```
username = "elarson"

print("1:" + username)

def greet():

    username = "bmoreno"

    print("2:" + username)

greet()

print("3:" + username)
```

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

# print()

The **`print()`** function outputs a specified object to the screen. The **`print()`** function is one of the most commonly used functions in Python because it allows you to output any detail from your code.
To use the **`print()`** function, you pass the object you want to print as an argument to the function. The **`print()`** function takes in any number of arguments, separated by a comma, and prints all of them. For example, you can run the following code that prints a string, a variable, another string, and an integer together:

```
month = "September"

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

# type()

The **`type()`** function returns the data type of its argument. The **`type()`** function helps you keep track of the data types of variables to avoid errors throughout your code.
To use it, you pass the object as an argument, and it returns its data type. It only accepts one argument. For example, you could specify **`type("security")`** or **`type(7)`**.

## Passing one function into another

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

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

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

# max() and min()

The **`max()`** function returns the largest numeric input passed into it. The **`min()`** function returns the smallest numeric input passed into it.
The **`max()`** and **`min()`** functions accept arguments of either multiple numeric values or of an iterable like a list, and they return the largest or smallest value respectively.
In a cybersecurity context, you could use these functions to identify the longest or shortest session that a user logged in for. If a specific user logged in seven times during a week, and you stored their access times in minutes in a list, you can use the **`max()`** and **`min()`** functions to find and print their longest and shortest sessions:

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

print(min(time_list))

print(max(time_list))
```

## sorted()

The **`sorted()`** function sorts the components of a list. The **`sorted()`** function also works on any iterable, like a string, and returns the sorted elements in a list. By default, it sorts them in ascending order. When given an iterable that contains numbers, it sorts them from smallest to largest; this includes iterables that contain numeric data as well as iterables that contain string data beginning with numbers. An iterable that contains strings that begin with alphabetic characters will be sorted alphabetically.
The **`sorted()`** function takes an iterable, like a list or a string, as an input. So, for example, you can use the following code to sort the list of login sessions from shortest to longest:

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

print(sorted(time_list))
```

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

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

print(sorted(time_list))

print(time_list)
```

The first **`print()`** function displays the sorted list. However, the second **`print()`** function, which does not include the **`sorted()`** function, displays the list as assigned to **`time_list`** in the first line of code.
One more important detail about the **`sorted()`** function is that it cannot take lists or strings that have elements of more than one data type. For example, you can't use the list **`[1, 2, "hello"]`**.

## Key takeaways

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

## Resources for more information

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

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

# The Python Standard Library

The **Python Standard Library** is an extensive collection of Python code that often comes packaged with Python. It includes a variety of modules, each with pre-built code centered around a particular type of task.
For example, you were previously introduced to the following modules in the Python Standard Library:

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

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

# How to import modules from the Python Standard Library

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

## Importing an entire module

To import an entire Python Standard Library module, you use the `import` keyword. The `import` keyword searches for a module or library in a system and adds it to the local Python environment. After `import`, specify the name of the module to import. For example, you can specify `import statistics` to import the `statistics` module. This will import all the functions inside of the `statistics` module for use later in your code.
As an example, you might want to use the `mean()` function from the `statistics` module to calculate the average number of failed login attempts per month for a particular user. In the following code block, the total number of failed login attempts for each of the twelve months is stored in a list called `monthly_failed_attempts`. Run this code and analyze how `mean()` can be used to calculate the average of these monthly failed login totals and store it in `mean_failed_attempts`:

```
import statistics

monthly_failed_attempts = [20, 17, 178, 33, 15, 21, 19, 29, 32, 15, 25,
19]
```

```
mean_failed_attempts = statistics.mean(monthly_failed_attempts)

print("mean:", mean_failed_attempts)
```

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

```
import statistics

monthly_failed_attempts = [20, 17, 178, 33, 15, 21, 19, 29, 32, 15, 25,
19]

median_failed_attempts = statistics.median(monthly_failed_attempts)

print("median:", median_failed_attempts)
```

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

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

## Importing specific functions from a module

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

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

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

```
from statistics import mean, median

monthly_failed_attempts = [20, 17, 178, 33, 15, 21, 19, 29, 32, 15, 25,
19]

mean_failed_attempts = mean(monthly_failed_attempts)

print("mean:", mean_failed_attempts)

median_failed_attempts = median(monthly_failed_attempts)
```

```
print("median:", median_failed_attempts)
```

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

# External libraries

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

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

```
%pip install numpy
```

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

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

```
import numpy
```

# Comments

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

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

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

## Single-line comments

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

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

```
# Print elements of 'computer_assets' list
computer_assets = ["laptop1", "desktop20", "smartphone03"]
for asset in computer_assets:
    print(asset)
```

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

## Multi-line comments

Multi-line comments are used when you need more than 79 characters in a single comment. For example, this might occur when defining a function if the comment describes its inputs and their data types as well as its output.
There are two commonly used ways of writing multi-line comments in Python. The first is by using the hashtag (`#`) symbol over multiple lines:

```
# remaining_login_attempts() function takes two integer parameters,
# the maximum login attempts allowed and the total attempts made,
# and it returns an integer representing remaining login attempts
def remaining_login_attempts(maximum_attempts, total_attempts):
    return maximum_attempts - total_attempts
```

Another way of writing multi-line comments is by using documentation strings and not assigning them to a variable. Documentation strings, also called docstrings, are strings that are written over multiple lines and are used to document code. To create a documentation string, use triple quotation marks (`""" """`).
You could add the comment to the function in the previous example in this way too:

```
"""
remaining_login_attempts() function takes two integer parameters,
the maximum login attempts allowed and the total attempts made,
and it returns an integer representing remaining login attempts
"""
```

# Correct indentation

**Indentation** is space added at the beginning of a line of code. In Python, you should indent the body of conditional statements, iterative statements, and function definitions. Indentation is not only necessary for Python to interpret this syntax properly, but it can also make it easier for you and other programmers to read your code.
The PEP 8 style guide recommends that indentations should be four spaces long. For example, if you had a conditional statement inside of a `while` loop, the body of the loop would be indented four spaces and the body of the conditional would be indented four spaces beyond that. This means the conditional would be indented eight spaces in total.

```
count = 0
login_status = True
while login_status == True:
    print("Try again.")
    count = count + 1
    if count == 4:
        login_status = False
```

# Maintaining correct syntax

Syntax errors involve invalid usage of the Python language. They are incredibly common with Python, so focusing on correct syntax is essential in ensuring that your code runs. Awareness of common errors will help you more easily fix them.
Syntax errors often occur because of mistakes with data types or in the headers of conditional or iterative statements or of function definitions.

## Data types

Correct syntax varies depending on data type:

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

### Colons in headers

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

```
def remaining_login_attempts(maximum_attempts, total_attempts):
    return maximum_attempts - total_attempts
```

## Resources for more information

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

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

# Module 3 - Work with strings and lists

## String data in a security setting

As an analyst, string data is one of the most common data types you will encounter in Python. **String data** is data consisting of an ordered sequence of characters. It's used to store any type of information you don't need to manipulate mathematically (such as through division or subtraction). In a cybersecurity context, this includes IP addresses, usernames, URLs, and employee IDs.

You'll need to work with these strings in a variety of ways. For example, you might extract certain parts of an IP address, or you might verify whether usernames meet required criteria.

## Working with indices in strings

### Indices

An **index** is a number assigned to every element in a sequence that indicates its position. With strings, this means each character in the string has its own index.

Indices start at `0`. For example, you might be working with this string containing a device ID: `"h32rb17"`. The following table indicates the index for each character in this string:

| character | index |
| --- | --- |
| `h` | `0` |

| | |
|---|---|
| 3 | 1 |
| 2 | 2 |
| r | 3 |
| b | 4 |
| 1 | 5 |
| 7 | 6 |

You can also use negative numbers as indices. This is based on their position relative to the last character in the string:

| character | index |
|---|---|
| h | -7 |
| 3 | -6 |
| 2 | -5 |
| r | -4 |
| b | -3 |
| 1 | -2 |
| 7 | -1 |

## Bracket notation

**Bracket notation** refers to the indices placed in square brackets. You can use bracket notation to extract a part of a string. For example, the first character of the device ID might represent a certain characteristic of the device. If you want to extract it, you can use bracket notation for this:

`"h32rb17"[0]`

This device ID might also be stored within a variable called `device_id`. You can apply the same bracket notation to the variable:

```
device_id = "h32rb17"
device_id[0]
```

In both cases, bracket notation outputs the character `h` when this bracket notation is placed inside a `print()` function. You can observe this by running the following code:

```
device_id = "h32rb17"

print("h32rb17"[0])

print(device_id[0])
```

You can also take a slice from a string. When you take a slice from a string, you extract more than one character from it. It's often done in cybersecurity contexts when you're only interested in a specific part of a string. For example, this might be certain numbers in an IP address or certain parts of a URL.
In the device ID example, you might need the first three characters to determine a particular quality of the device. To do this, you can take a slice of the string using bracket notation. You can run this line of code to observe that it outputs **`"h32"`**:

```
print("h32rb17"[0:3])
```

**Note:** The slice starts at the **`0`** index, but the second index specified after the colon is excluded. This means the slice ends one position before index **`3`**, which is at index **`2`**.

# String functions and methods

The **`str()`** and **`len()`** functions are useful for working with strings. You can also apply methods to strings, including the **`.upper()`**, **`.lower()`**, and **`.index()`** methods. A **method** is a function that belongs to a specific data type.

## str() and len()

The **`str()`** function converts its input object into a string. As an analyst, you might use this in security logs when working with numerical IDs that aren't going to be used with mathematical processes. Converting an integer to a string gives you the ability to search through it and extract slices from it.
Consider the example of an employee ID **`19329302`** that you need to convert into a string. You can use the following line of code to convert it into a string and store it in a variable:

```
string_id = str(19329302)
```

The second function you learned for strings is the **`len()`** function, which returns the number of elements in an object.
As an example, if you want to verify that a certain device ID conforms to a standard of containing seven characters, you can use the **`len()`** function and a conditional. When you run the following code, it will print a message if **`"h32rb17"`** has seven characters:

```
device_id_length = len("h32rb17")

if device_id_length == 7:

    print("The device ID has 7 characters.")
```

## .upper() and .lower()

The **`.upper()`** method returns a copy of the string with all of its characters in uppercase. For example, you can change this department name to all uppercase by running the code **`"Information Technology".upper()`**. It would return the string **`"INFORMATION TECHNOLOGY"`**.
Meanwhile, the **`.lower()`** method returns a copy of the string in all lowercase characters. **`"Information Technology".lower()`** would return the string **`"information technology"`**.

### .index()

The `.index()` method finds the first occurrence of the input in a string and returns its location. For example, this code uses the `.index()` method to find the first occurrence of the character `"r"` in the device ID `"h32rb17"`:

```
print("h32rb17".index("r"))
```

The `.index()` method returns `3` because the first occurrence of the character `"r"` is at index `3`. In other cases, the input may not be found. When this happens, Python returns an error. For instance, the code `print("h32rb17".index("a"))` returns an error because `"a"` is not in the string `"h32rb17"`.

Also note that if a string contains more than one instance of a character, only the first one will be returned. For instance, the device ID `"r45rt46"` contains two instances of `"r"`. You can run the following code to explore its output:

```
print("r45rt46".index("r"))
```

The output is `0` because `.index()` returns only the first instance of `"r"`, which is at index `0`. The instance of `"r"` at index `3` is not returned.

### Finding substrings with .index()

A **substring** is a continuous sequence of characters within a string. For example, `"llo"` is a substring of `"hello"`.

The `.index()` method can also be used to find the index of the first occurrence of a substring. It returns the index of the first character in that substring. Consider this example that finds the first instance of the user `"tshah"` in a string:

```
tshah_index = "tsnow, tshah, bmoreno - updated".index("tshah")

print(tshah_index)
```

The `.index()` method returns the index `7`, which is where the substring `"tshah"` starts.

**Note:** When using the `.index()` method to search for substrings, you need to be careful. In the previous example, you want to locate the instance of `"tshah"`. If you search for just `"ts"`, Python will return `0` instead of `7` because `"ts"` is also a substring of `"tsnow"`.

# Lists and the security analyst

Previously, you examined how to use bracket notation to access and change elements in a list and some fundamental methods for working with lists. This reading will review these concepts with new examples, introduce the `.index()` method as it applies to lists, and highlight how lists are used in a cybersecurity context.

## List data in a security setting

As a security analyst, you'll frequently work with lists in Python. **List data** is a data structure that consists of a collection of data in sequential form. You can use lists to store multiple elements in a single variable. A single list can contain multiple data types.

In a cybersecurity context, lists might be used to store usernames, IP addresses, URLs, device IDs, and data.

Placing data within a list allows you to work with it in a variety of ways. For example, you might iterate through a list of device IDs using a **`for`** loop to perform the same actions for all items in the list. You could incorporate a conditional statement to only perform these actions if the device IDs meet certain conditions.

# Working with indices in lists

## Indices

Like strings, you can work with lists through their indices, and indices start at `0`. In a list, an index is assigned to every element in the list.
This table contains the index for each element in the list **`["elarson", "fgarcia", "tshah", "sgilmore"]`**:

| element | index |
| --- | --- |
| **`"elarson"`** | **`0`** |
| **`"fgarcia"`** | **`1`** |
| **`"tshah"`** | **`2`** |
| **`"sgilmore"`** | **`3`** |

# Bracket notation

Similar to strings, you can use bracket notation to extract elements or slices in a list. To extract an element from a list, after the list or the variable that contains a list, add square brackets that contain the index of the element. The following example extracts the element with an index of **`2`** from the variable **`username_list`** and prints it. You can run this code to examine what it outputs:

```
username_list = ["elarson", "fgarcia", "tshah", "sgilmore"]

print(username_list[2])
```

This example extracts the element at index **`2`** directly from the list:

```
print(["elarson", "fgarcia", "tshah", "sgilmore"][2])
```

## Extracting a slice from a list

Just like with strings, it's also possible to use bracket notation to take a slice from a list. With lists, this means extracting more than one element from the list.
When you extract a slice from a list, the result is another list. This extracted list is called a sublist because it is part of the original, larger list.
To extract a sublist using bracket notation, you need to include two indices. You can run the following code that takes a slice from a list and explore the sublist it returns:

```
username_list = ["elarson", "fgarcia", "tshah", "sgilmore"]

print(username_list[0:2])
```

The code returns a sublist of `["elarson", "fgarcia"]`. This is because the element at index `0`, `"elarson"`, is included in the slice, but the element at index `2`, `"tshah"`, is excluded. The slice ends one element before this index.

## Changing the elements in a list

Unlike strings, you can also use bracket notation to change elements in a list. This is because a string is **immutable** and cannot be changed after it is created and assigned a value, but lists are not immutable.
To change a list element, use similar syntax as you would use when reassigning a variable, but place the specific element to change in bracket notation after the variable name. For example, the following code changes the element at index `1` of the `username_list` variable to `"bmoreno"`.

```
username_list = ["elarson", "fgarcia", "tshah", "sgilmore"]

print("Before changing an element:", username_list)

username_list[1] = "bmoreno"

print("After changing an element:", username_list)
```

This code has updated the element at index `1` from `"fgarcia"` to `"bmoreno"`.

## List methods

List methods are functions that are specific to the list data type. These include the `.insert()` , `.remove()`, `.append()` and `.index()`.

### .insert()

The `.insert()` method adds an element in a specific position inside a list. It has two parameters. The first is the index where you will insert the new element, and the second is the element you want to insert.
You can run the following code to explore how this method can be used to insert a new username into a username list:

```
username_list = ["elarson", "bmoreno", "tshah", "sgilmore"]

print("Before inserting an element:", username_list)

username_list.insert(2,"wjaffrey")

print("After inserting an element:", username_list)
```

Because the first parameter is `2` and the second parameter is `"wjaffrey"`, `"wjaffrey"` is inserted at index `2`, which is the third position. The other list elements are shifted one position in the list. For example, `"tshah"` was originally located at index `2` and now is located at index `3`.

### .remove()

The **`.remove()`** method removes the first occurrence of a specific element in a list. It has only one parameter, the element you want to remove.
The following code removes **`"elarson"`** from the **`username_list`**:

```
username_list = ["elarson", "bmoreno", "wjaffrey", "tshah", "sgilmore"]

print("Before removing an element:", username_list)

username_list.remove("elarson")

print("After removing an element:", username_list)
```

This code removes **`"elarson"`** from the list. The elements that follow **`"elarson"`** are all shifted one position closer to the beginning of the list.
**Note:** If there are two of the same element in a list, the **`.remove()`** method only removes the first instance of that element and not all occurrences.

## .append()

The **`.append()`** method adds input to the end of a list. Its one parameter is the element you want to add to the end of the list.
For example, you could use **`.append()`** to add **`"btang"`** to the end of the **`username_list`**:

```
username_list = ["bmoreno", "wjaffrey", "tshah", "sgilmore"]

print("Before appending an element:", username_list)

username_list.append("btang")

print("After appending an element:", username_list)
```

This code places **`"btang"`** at the end of the **`username_list`**, and all other elements remain in their original positions.
The **`.append()`** method is often used with **`for`** loops to populate an empty list with elements. You can explore how this works with the following code:

```
numbers_list = []

print("Before appending a sequence of numbers:", numbers_list)

for i in range(10):

    numbers_list.append(i)

print("After appending a sequence of numbers:", numbers_list)
```

Before the **`for`** loop, the **`numbers_list`** variable does not contain any elements. When it is printed, the empty list is displayed. Then, the **`for`** loop iterates through a sequence of numbers and uses the **`.append()`** method to add each of these numbers to **`numbers_list`**. After the loop, when the **`numbers_list`** variable is printed, it displays these numbers.

## .index()

Similar to the `.index()` method used for strings, the `.index()` method used for lists finds the first occurrence of an element in a list and returns its index. It takes the element you're searching for as an input.

**Note:** Although it has the same name and use as the `.index()` method used for strings, the `.index()` method used for lists is not the same method. Methods are defined when defining a data type, and because strings and lists are defined differently, the methods are also different.

Using the `username_list` variable, you can use the `.index()` method to find the index of the username `"tshah"`:

```
username_list = ["bmoreno", "wjaffrey", "tshah", "sgilmore", "btang"]

username_index = username_list.index("tshah")

print(username_index)
```

Because the index of `"tshah"` is `2`, it outputs this number.

Similar to the `.index()` method used for strings, it only returns the index of the first occurrence of a list item. So if the username `"tshah"` were repeated twice, it would return the index of the first instance, and not the second.

# Basics of regular expressions

A **regular expression (regex)** is a sequence of characters that forms a pattern. You can use these in Python to search for a variety of patterns. This could include IP addresses, emails, or device IDs.

To access regular expressions and related functions in Python, you need to import the `re` module first. You should use the following line of code to import the `re` module:

```
import re
```

Regular expressions are stored in Python as strings. Then, these strings are used in `re` module functions to search through other strings. There are many functions in the `re` module, but you will explore how regular expressions work through `re.findall()`. The `re.findall()` function returns a list of matches to a regular expression. It requires two parameters. The first is the string containing the regular expression pattern, and the second is the string you want to search through.

The patterns that comprise a regular expression consist of alphanumeric characters and special symbols. If a regular expression pattern consists only of alphanumeric characters, Python will review the specified string for matches to this pattern and return them. In the following example, the first parameter is a regular expression pattern consisting only of the alphanumeric characters `"ts"`. The second parameter, `"tsnow, tshah, bmoreno"`, is the string it will search through. You can run the following code to explore what it returns:

```
import re

re.findall("ts", "tsnow, tshah, bmoreno")
```

The output is a list of only two elements, the two matches to `"ts"`: `['ts', 'ts']`.

If you want to do more than search for specific strings, you must incorporate special symbols into your regular expressions.

# Regular expression symbols

## Symbols for character types

You can use a variety of symbols to form a pattern for your regular expression. Some of these symbols identify a particular type of character. For example, **`\w`** matches with any alphanumeric character.

**Note:** The **`\w`** symbol also matches with the underscore ( **`_`** ).

You can run this code to explore what **`re.findall()`** returns when applying the regular expression of **`"\w"`** to the device ID of **`"h32rb17"`**.

```
import re

re.findall("\w", "h32rb17")
```

Because every character within this device ID is an alphanumeric character, Python returns a list with seven elements. Each element represents one of the characters in the device ID.

You can use these additional symbols to match to specific kinds of characters:

- **`.`** matches to all characters, including symbols
- **`\d`** matches to all single digits [0-9]
- **`\s`** matches to all single spaces
- **`\.`** matches to the period character

The following code searches through the same device ID as the previous example but changes the regular expression pattern to **`"\d"`**. When you run it, it will return a different list:

```
import re

re.findall("\d", "h32rb17")
```

This time, the list contains only four elements. Each element is one of the numeric digits in the string.

## Symbols to quantify occurrences

Other symbols quantify the number of occurrences of a specific character in the pattern. In a regular expression pattern, you can add them after a character or a symbol identifying a character type to specify the number of repetitions that match to the pattern.

For example, the **`+`** symbol represents one or more occurrences of a specific character. In the following example, the pattern places it after the **`"\d"`** symbol to find matches to one or more occurrences of a single digit:

```
import re

re.findall("\d+", "h32rb17")
```

With the regular expression **`"\d+"`**, the list contains the two matches of **`"32"`** and **`"17"`**.

Another symbol used to quantify the number of occurrences is the **`*`** symbol. The **`*`** symbol represents zero, one, or more occurrences of a specific character. The following code substitutes the **`*`** symbol for the **`+`** used in the previous example. You can run it to examine the difference:

```
import re

re.findall("\d*", "h32rb17")
```

Because it also matches to zero occurrences, the list now contains empty strings for the characters that were not single digits.
If you want to indicate a specific number of repetitions to allow, you can place this number in curly brackets (`{ }`) after the character or symbol. In the following example, the regular expression pattern `"\d{2}"` instructs Python to return all matches of exactly two single digits in a row from a string of multiple device IDs:

```
import re

re.findall("\d{2}", "h32rb17 k825t0m c2994eh")
```

Because it is matching to two repetitions, when Python encounters a single digit, it checks whether there is another one following it. If there is, Python adds the two digits to the list and goes on to the next digit. If there isn't, it proceeds to the next digit without adding the first digit to the list.
**Note:** Python scans strings left-to-right when matching against a regular expression. When Python finds a part of the string that matches the first expected character defined in the regular expression, it continues to compare the subsequent characters to the expected pattern. When the pattern is complete, it starts this process again. So in cases in which three digits appear in a row, it handles the third digit as a new starting digit.
You can also specify a range within the curly brackets by separating two numbers with a comma. The first number is the minimum number of repetitions and the second number is the maximum number of repetitions. The following example returns all matches that have between one and three repetitions of a single digit:

```
import re

re.findall("\d{1,3}", "h32rb17 k825t0m c2994eh")
```

The returned list contains elements of one digit like `"0"`, two digits like `"32"` and three digits like `"825"`.

## Constructing a pattern

Constructing a regular expression requires you to break down the pattern you're searching for into smaller chunks and represent those chunks using the symbols you've learned. Consider an example of a string that contains multiple pieces of information about employees at an organization. For each employee, the following string contains their employee ID, their username followed by a colon (`:`), their attempted logins for the day, and their department:

```
employee_logins_string = "1001 bmoreno: 12 Marketing 1002 tshah: 7 Human Resources 1003 sgilmore: 5 Finance"
```

Your task is to extract the username and the login attempts, without the employee's ID number or department.
To complete this task with regular expressions, you need to break down what you're searching for into smaller components. In this case, those components are the varying number of characters in a username, a colon, a space, and a varying number of single digits. The corresponding regular expression symbols are `\w+`, `:`, `\s`, and `\d+` respectively. Using these symbols as your regular expression, you can run the following code to extract the strings:

```
import re

pattern = "\w+:\s\d+"
```

```
employee_logins_string = "1001 bmoreno: 12 Marketing 1002 tshah: 7 Human
Resources 1003 sgilmore: 5 Finance"

print(re.findall(pattern, employee_logins_string))
```

**Note:** Working with regular expressions can carry the risk of returning unneeded information or excluding strings that you want to return. Therefore, it's useful to test your regular expressions.

## Module 4 - Python in practice

# 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.

## 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

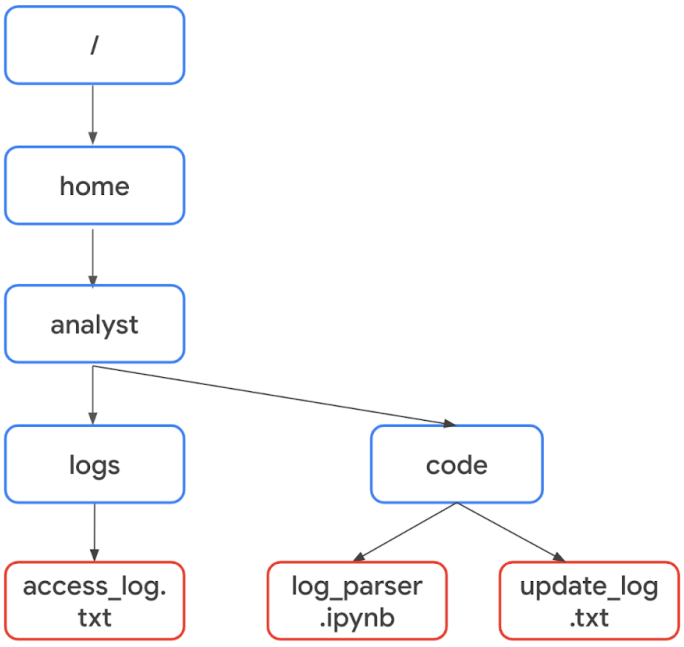
The keyword `with` handles errors and manages external resources when used with other functions. In this case, it's used with the `open()` function in order to open a file. It will then manage the resources by closing the file after exiting the `with` statement.

**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.

# 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.

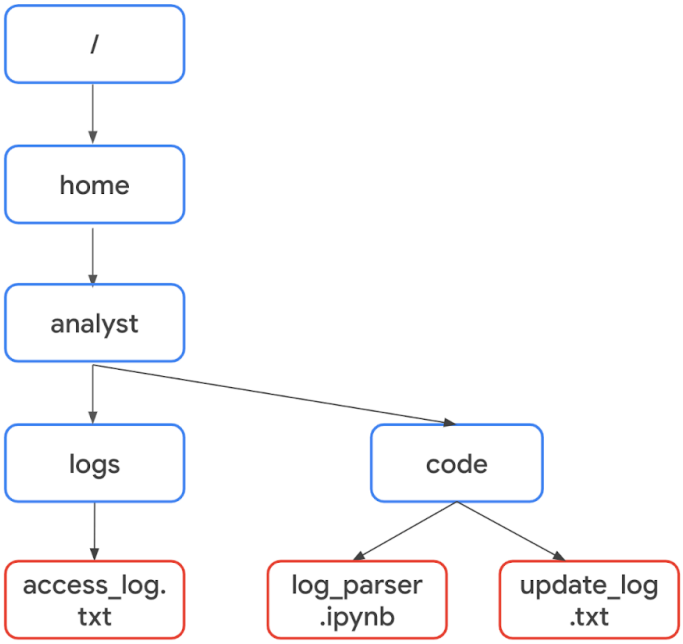
### with

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.

# 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)
```

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.")
```

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)
```

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)
```

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)
```

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`**.