Automate Cybersecurity Tasks with Python

Module 1: Introduction to Python

Python Environments

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.

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`
- 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`

Module 1's Terms

Automation: The use of technology to reduce human and manual effort to perform common and repetitive tasks
Boolean data: Data that can only be one of two values: either `true` or `false`
Command-line interface: A text-based user interface that uses commands to interact with the computer
Comment: A note programmers make about the intention behind their code
Conditional statement: A statement that evaluates code to determine if it meets a specified set of conditions
Data type: A category for a particular type of data item
Dictionary data: Data that consists of one or more key-value pairs
Float data: Data consisting of a number with a decimal point
Integer data: Data consisting of a number that does not include a decimal point
Integrated development environment (IDE): A software application for writing code that provides editing assistance and error correction tools
Interpreter: A computer program that translates Python code into runnable instructions line by line
Iterative statement: Code that repeatedly executes a set of instructions
List data: Data structure that consists of a collection of data in sequential form
Loop variable: A variable that is used to control the iterations of a loop
Notebook: An online interface for writing, storing, and running code
Programming: A process that can be used to create a specific set of instructions for a computer to execute tasks
Set data: Data that consists of an unordered collection of unique values
String data: Data consisting of an ordered sequence of characters
Syntax: The rules that determine what is correctly structured in a computing language
Tuple data: Data structure that consists of a collection of data that cannot be changed
Type error: An error that results from using the wrong data type
Variable: A container that stores data

Module 2: Write Effective Python Code

Import Modules and Libraries in Python

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).

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

Run

Reset

```
mean: 35.25
```

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

Run

Reset

```
median: 20.5
```

Module 3: Work with Strings and Lists

Strings and the Security Analyst

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: `"h32rb1"`. The following table indicates the index for each character in this string:

Character	Index	Negative Index
h	0	-6
3	1	-5
2	2	-4
r	3	-3
b	4	-2
1	5	-1

Bracket notation

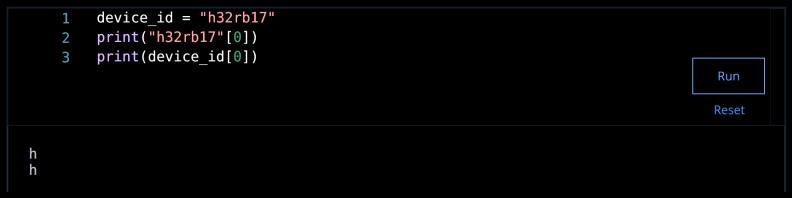
Bracket notation refers to the indices placed in square brackets. You can use bracket notation to extract a part of a string. For example, the first character of the device ID might represent a certain characteristic of the device. If you want to extract it, you can use bracket notation for this:

```
"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:

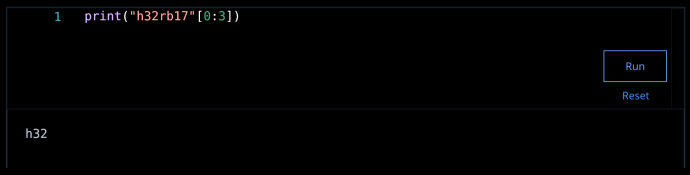
```
device_id = "h32rb17"

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:



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.

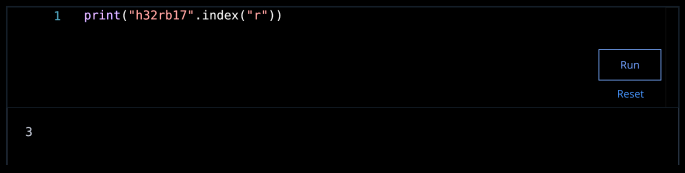
In the device ID example, you might need the first three characters to determine a particular quality of the device. To do this, you can take a slice of the string using bracket notation. You can run this line of code to observe that it outputs `"h32"`:



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`.

.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"`:



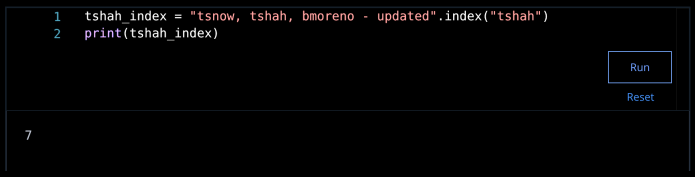
The `.index()` method returns `3` because the first occurrence of the character `"r"` is at index `3`.

Pro-Tip: If there's no instance of the term you are trying to search for, use an exception to avoid an error/exit.

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:



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

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:

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

Run

Reset

```
tshah
```

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

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

Run

Reset

```
tshah
```

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:

```
1  username_list = ["elarson", "fgarcia", "tshah", "sgilmore"]
2  print(username_list[0:2])
```

Run

Reset

```
['elarson', 'fgarcia']
```

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"`.

```
1  username_list = ["elarson", "fgarcia", "tshah", "sgilmore"]
2  print("Before changing an element:", username_list)
3  username_list[1] = "bmoreno"
4  print("After changing an element:", username_list)
```

Run

Reset

```
Before changing an element: ['elarson', 'fgarcia', 'tshah', 'sgilmore']
After changing an element: ['elarson', 'bmoreno', 'tshah', 'sgilmore']
```

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)
```

Run

Reset

```
Before inserting an element: ['elarson', 'bmoreno', 'tshah', 'sgilmore']
After inserting an element: ['elarson', 'bmoreno', 'wjaffrey', 'tshah', 'sgilmore']
```

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)
```

Run

Reset

```
Before removing an element: ['elarson', 'bmoreno', 'wjaffrey', 'tshah', 'sgilmore']
After removing an element: ['bmoreno', 'wjaffrey', 'tshah', 'sgilmore']
```

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)
```

Run

Reset

```
Before appending an element: ['bmoreno', 'wjaffrey', 'tshah', 'sgilmore']
After appending an element: ['bmoreno', 'wjaffrey', 'tshah', 'sgilmore', 'btang']
```

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)
```

Run

Reset

```
Before appending a sequence of numbers: []
After appending a sequence of numbers: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

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)
```

Run

Reset

```
2
```

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.

Regular Expressions (regex)

Regular email expression: `re.findall (" \w+ @ \w+ .\ w+ ")`

Basics of regular expressions

A regular expression (regex) is a sequence of characters that forms a pattern. In Python, you use regex to efficiently search for complex patterns like IP addresses, emails, or device IDs within strings.

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:

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

Run

Reset

```
['ts', 'ts']
```

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"`.

```
1  import re
2  re.findall("\w", "h32rb17")
```

Run

Reset

```
['h', '3', '2', 'r', 'b', '1', '7']
```

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.

These symbols match a single character of a specific type.

Symbol	Description	Example Match
`\w`	Matches any **alphanumeric character** (A-z, 0-9) OR an **underscore** (_).	In "ID_A17", matches I,D,_,A,1,7.
`\d`	Matches any **single digit** (0-9).	In "ID_A17", matches 1,7.
`\s`	Matches any **single whitespace** character (space, tab, newline).	Matches the space in "user 1".
`.`	Matches **any character** (letters, digits, symbols, spaces), except for a newline.	
`\.`	Matches the **literal period character** (.). The backslash \ is necessary to escape the special meaning of the dot.	

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:

```
1  import re
2  re.findall("\d", "h32rb17")
```

Run

Reset

```
['3', '2', '1', '7']
```

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.

Symbol	Description	Example
`+`	**One or more** occurrences. (e.g., \d+ matches 1,12,12345).	
`*`	**Zero, one, or more** occurrences.	
`{n}`	**Exactly n** occurrences.	\d{4} matches four consecutive digits (e.g., 1234).
`{n,n}`	Between **m (minimum) and n (maximum)** occurrences.	\d{1,3} matches 1,12, or 123.

For example, the + symbol represents one or more consecutive occurrences of the preceding character or character type. When used with `\d+`, it finds matches of one or more digits in a row, such as 1, 12, or 123.

In the following example, the pattern places it after the \d symbol to find matches to one or more consecutive digits:

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

Run

Reset

```
['32', '17']
```

With the regular expression `"\d+"`, the list contains the two matches of `"32"` and `"17"`. Note that the `+` matches a *sequence* of different digits, not just one digit repeated.

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:

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

Run

Reset

```
['', '32', '', '', '17', '']
```

Because it also matches to zero occurrences, the list now contains empty strings for the characters that were not single digits, as well as an empty string at the end.

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:

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

Run

Reset

```
['32', '17', '82', '29', '94']
```

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 at the character immediately following the match. So in cases in which three digits appear in a row (e.g., 123), \d{2} would match 12, and the process would start again at the third digit (3).

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:

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

Run

Reset

```
['32', '17', '825', '0', '299', '4']
```

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

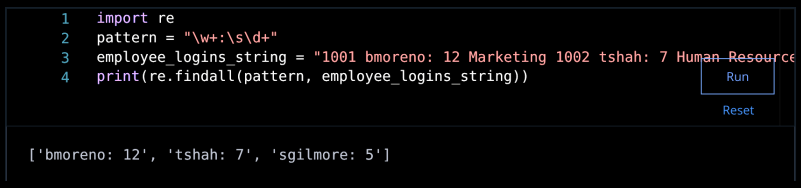
Constructing a pattern

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:



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 CI/CD

Imagine manually checking every piece of code for problems, or manually testing every version of software for security issues before it's released. That would be really slow and have a lot of errors. Python is a great tool to automate these security tasks in CI/CD because it's flexible and has many helpful tools - like libraries.

Using Python to automate security tasks in your CI/CD pipeline is beneficial for a few reasons: :

- Increases Speed and Efficiency: Python scripts for security checks are fast and work well as part of your pipeline. This keeps your software releases quick and secure at the same time.
- Finds Problems Early: Python can help find security problems early on when software is being developed. This makes problems easier and less expensive to fix.
- Remains Consistent: Python scripts make sure security checks are done the same way every time you build and release software. This lowers the chance of human error.
- Reduces workload for Security Teams: Python frees up security teams from repetitive tasks and allows them to work on larger security problems, planning, or creating better Python scripts for security automation.
- Supports a culture or security: Python-based automation helps put security into the CI/CD process. This helps create a DevSecOps culture where everyone thinks about security, not just the security team.

What Security Tasks Can You Automate in CI/CD with Python?

You can use Python to automate many kinds of security tasks in CI/CD pipelines. Here are some main tasks:

Security Testing

- Static Application Security Testing (**SAST**): Python scripts can be written to start SAST tools that look at your code for weaknesses *before* it gets built. Python can also be used to understand the SAST results, create reports, and automatically stop the process if serious security problems are found.
- Dynamic Application Security Testing (**DAST**): Python can be used to automatically run DAST tools to test software while it's running in a test area. Then, Python scripts can look at the DAST results and give feedback in the CI/CD pipeline.
- Software Composition Analysis (**SCA**): Python can work with SCA tools to check your software's dependencies for weaknesses. Dependencies are things like open source code and components from other companies. Scripts can control the SCA process, report problems, and set rules based on the severity of weaknesses.

Automated Vulnerability Scanning

Python scripts can organize vulnerability scans of things like container images, infrastructure settings, and the CI/CD pipeline itself. You can use Python to schedule these scans, collect the results, and send alerts when new vulnerabilities are discovered.

Compliance Checks

Python scripts can automatically check for compliance. For example, scripts can check if code follows secure coding rules or if infrastructure settings meet security guidelines. You can then use Python to make reports about compliance and ensure security standards are followed.

Secrets Management Automation

Python is key for automating secure secrets management. Scripts can be used to review through code and stop private credentials from being directly written in the code. Also, Python scripts can work with secret management tools (like HashiCorp Vault) to safely get and put secrets into applications during automated releases.

Policy Enforcement

"Policy as Code" and Python scripts work together to automatically enforce security policies. Python can be used to define and understand security policies. Then, scripts can check pipeline steps against these policies. If policies are broken (for example, if too many vulnerabilities are found), Python can automatically stop releases.

How Python Works with CI/CD Tools

Python is even more helpful for CI/CD security automation because it works well with popular CI/CD tools. Tools like Jenkins, GitLab CI, and CircleCI let you easily run Python scripts as part of your release process.

Here's how Python fits in:

- Run Scripts: CI/CD systems let you set up release steps that run commands or scripts. You can easily set up steps to run Python scripts that do security tasks.
- API Connections: Many CI/CD tools and security tools have APIs (Application Programming Interfaces). Python is excellent at using APIs. You can write Python scripts to use CI/CD system APIs to manage the release process, start jobs, get software build files, and connect to security tool APIs to start scans and get results.
- Add-ons and Extensions: Some CI/CD systems have add-ons or extensions made in Python or that can easily use Python scripts. This makes it even simpler to add security automation based on Python.

Using Python to Set Up Environments, Check Code Quality, and Secure Releases

Besides security testing, Python scripts can automate other important CI/CD tasks while adding security best practices:

- Set Up Environments: Python can automate staging areas. Scripts can make sure these areas are set up securely, with good network settings and security controls.
- Code Quality Checks: Python can be used to run code quality tools (**linters**). Scripts can check code for style problems and possible security errors. This helps make sure code quality standards are followed early in development.
- Automate Secure Releases: Python scripts are very useful for automating releases to staging and production areas securely. Python can manage release processes and ensure releases follow security best practices. This includes using secure settings and moving software files securely.

Conclusion: Python - Your Automation Ally for Secure CI/CD

Using Python to automate security tasks is key to making your CI/CD pipeline secure and fast. By using Python's abilities and connecting it to your CI/CD tools, you can find and fix security problems early, do less manual work, enforce security rules, and make your software more secure overall.

By making Python automation a main part of your CI/CD security plan, you'll be ready to create and release secure software, quickly and with confidence. Now you know *how* Python helps automate security in CI/CD. Next, you'll learn about the specific parts of Python that make this possible. You'll learn about variables, conditional statements, iterative statements, functions, and working with files. These are the basic pieces for creating your own powerful Python scripts to automate security in CI/CD.

Resources:

1. Best Python Libraries for Cybersecurity in 2024. (Poorly written by AI, verify all information) https://medium.com/@Scofield_Idehen/best-python-libraries-for-cybersecurity-in-2024-037a870f39d1
2. Vulnerability Scanning for Secure Python Development. https://safetycli.com/
3. Article 3 - OWASP Dependency-Check and Vulnerability Scanning. https://www.linkedin.com/pulse/article-3-owasp-dependency-check-vulnerability-scanning-adorsys-p73fe
4. Python library for Hashicorp Vault implementation. https://discuss.hashicorp.com/t/python-library-for-hashicorp-vault-implementation/55805
5. Continuous Integration With Python: An Introduction. https://realpython.com/python-continuous-integration/
6. Python for DevOps: An Ultimate Guide. https://code-b.dev/blog/python-devops
7. Building Custom Cybersecurity Tools with Python: A Guide for Beginners. https://www.linkedin.com/pulse/building-custom-cybersecurity-tools-python-bi6if
8. Secure Coding in Python: Essential Practices for Data Engineers. https://www.linkedin.com/pulse/secure-coding-python-essential-practices-data-engineers-priyanka-sain-wewkc

Essential Python Concepts for Automation

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.

Import Files into Python

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.

Work with Files in Python

Parsing

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

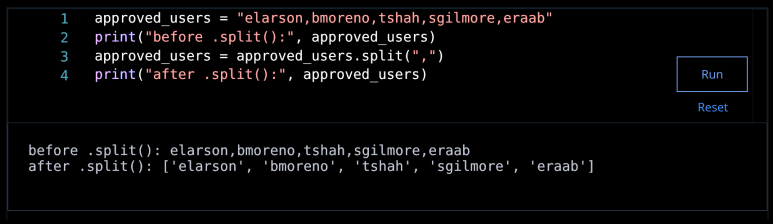
Methods that can help you parse your data include `.split()` and `.join()`.

.split()

The basics of .split()

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

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



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

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

Applying .split() to files

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

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

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

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

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

.join()

The basics of .join()

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

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

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

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

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

Run

Reset

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

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

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

Applying .join() to files

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

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

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

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

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

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

Explore Debugging Techniques

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."

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.

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.

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.

Recent advancements in AI have opened up many opportunities for enhancing IDEs with powerful, context-aware coding assistance. These are tools that are integrated directly into the IDE or coding environment to provide a more seamless coding experience. For example, Gemini Code Assist is a free AI tool that integrates into popular IDEs like Visual Studio Code and JetBrains. It functions as an assistant that can help analyze code and find errors, suggest modifications, and also interact in a conversational way to answer many other questions, whether they're technical or more conceptual.

These tools are quickly becoming indispensable to coders and cybersecurity professionals as they facilitate and accelerate workflows, but if you choose to use them, please remember that this technology is still evolving. The suggestions, code, or explanations provided may not always be perfectly accurate, optimal, or secure. Always review and validate any AI-generated output before executing programs or relying on the information. Treat the AI assistance as a helpful co-pilot, but maintain oversight and responsibility for your final code.

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:

```
1  new_users = ["sgilmore", "bmoreno"]
2  approved_users = ["bmoreno", "tshah", "elarson"]
3  def add_users():
4      for user in new_users:
5          if user in approved_users:
6              print(user,"already in list")
7          approved_users.append(user)
8  add_users()
9  print(approved_users)
```

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

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

```
1  new_users = ["sgilmore", "bmoreno"]
2  approved_users = ["bmoreno", "tshah", "elarson"]
3  def add_users():
4      for user in new_users:
5          print("line 5 - inside for loop")
6          if user in approved_users:
7              print("line 7 - inside if statement")
8              print(user,"already in list")
9          print("line 9 - before .append method")
10         approved_users.append(user)
11 add_users()
12 print(approved_users)
```

Run

Reset

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

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

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