module 3

Welcome to module 3

As a security analyst,
you'll work with a lot of data.
Being able to develop solutions for
managing this data is very important.
What we're about to learn in
Python will help you with that.
Previously, we set
foundations for what we're going to do in this section.
We learned all about data types and variables.
We also covered conditional and iterative statements.
We learned about building
functions and even created our own functions.
Here, we'll build on that in a few different ways.
First, you'll learn more about
working with strings and lists.
We'll expand the ways that you
can work with these data types,
including extracting characters from
strings or items from lists.
Our next focus is on writing algorithms.
You'll consider a set of rules that can be applied in
Python to solve a security-related problem.
Finally, we'll further expand the ways we can search
for strings when we explore using regular expressions.
We're going to have a lot of fun, and you'll
start writing some really interesting code in Python.
I can't wait to get started.

String operations

Knowing how to work with
the string data in security is important.
For example, you might find yourself working with
usernames to find patterns in login information.
We're going to revisit
the string data type and learn
how to work with it in Python.

First, let's have a quick refresher on the strings.
We defined the string data as
data consisting of an ordered sequence of characters.
In Python, strings are
written in between quotation marks.
It's okay to use either double or single quotation marks,
but in this course, we've been
using double quotation marks.
As examples, we have the strings "Hello",
"123", and "Number 1!"

We also previously covered variables.
Here, the variable my_string
is currently storing the string "security".

You can also create a string from another data type,
such as an integer or a float.
To do that, we need to introduce
a new built-in function, the string function.
The string function is a function that
converts the input object into a string.
Converting objects to strings allows us to
perform tasks that are only possible for strings.
For example, we might convert an integer into
a string to remove elements from it or to re-order it.
Both are difficult for an integer data type.

Let's practice converting an integer to a string.
We'll apply the string function to the integer 123.
Now, the variable new_string
contains a string of three characters:
1, 2, and 3.
Let's print its type to check.

We'll run it. Perfect, it tells
us that we now have a string!
Awesome! So far,
we know different ways to create and store a string.
Now, let's explore how to
perform some basic string operations.

Our first example is the length function.
The length function is a function that
returns the number of elements in an object.
Using it on a string tells
us how many characters the string has.
Earlier in the program,
we learned that IP addresses have
two versions, IPv4 or IPv6.
IPv4 addresses have a maximum of 15 characters.
So a security professional might use
the length function to check if an IPv4 address is valid.
If its length is greater than 15 characters,
then we'd know that it's an invalid IPv4 address.

Let's use this function to print
the length of the string "Hello"

We'll nest the length function
within the print function because we
want to first calculate the length of
this string and then print it to the screen.
Okay, let's run this and check out
how many characters Python counts.
The output is 5,
one for each letter in the word Hello.

We can also use the addition operator on the strings.
This is called string concatenation.
The string concatenation is
the process of joining two strings together.
For example, we can add
the strings "Hello" and "world" together.
To concatenate strings, we can use the + symbol.
After we run it,
we get "Helloworld" with
no spaces in between the two strings.
It's important to note that
some operators don't work for strings.
For example, you cannot use
a minus sign to subtract the two strings.

Finally, we're going to talk about string methods.
A method is a function
that belongs to a specific data type.
So, using a string method on another data type,
like an integer, would cause an error.
Unlike other functions, methods appear after the string.
Two common string methods are
the upper and the lower methods.
The upper method returns a copy of
the string in all uppercase letters.

Let's apply the upper method to the string "Hello"

We'll place this inside of
a print function to output it to the screen.
Let's focus on the unique syntax of methods.
After our string "Hello",
we place a period or dot,
and then specify the method we want to use.
Here, that's upper()

Okay, now we're ready to run this.
HELLO is printed to the screen in all uppercase letters.

Similarly, the lower method returns
a copy of the string in all lowercase letters.
Let's apply the lower method on the "Hello" string.
Remember that we need to put the string and the method
inside of a print function to output the results.
And now, we have the string
printed in all lowercase letters.

Coming up, we're going to
learn a lot more about strings,
like indexing and splitting strings.
I'm looking forward to meeting you there!

String indices and slices

In security, there are a variety
of reasons we might need to search through a string.
For example, we might need to
locate a username in a security log.
Or, if we learn that
a certain IP address is associated with malware,
we might search for this address in a network log.
And, the first step in being able to use Python in
these ways is learning
about the index of characters in a string.
The index is a number assigned to
every element in a sequence that indicates its position.
In this video, we are discussing strings.
So, the index is
the position of each character in a string.

Let's start with the string "HELLO."
Every character in the string is assigned an index.
In Python, we start counting indices from 0.
So, the character "H" has an index of 0,
and "E" has an index of 1, and so on.
Let's take this into Python and practice using indices.

Placing an index in square brackets after
a string returns the character at that index.
Let's place the index 1 in
square brackets after "HELLO" and run it.
This returned the character "E."
Remember, indices start at 0,
so an index of 1 isn't the first character in the word.

But what if we want it to
return more than just one character?
We can extract a larger part of
a string by specifying a set of indices.
This is called a slice.
When taking a slice from a string,
we specify where the slice
starts and where the slice ends.
So we provide two indices.
The first index is the beginning,

which is included in the output.
The second index is the end,
but it's not included in the final output.
Instead, Python stops
the slice at the element before the second index.
For example, if we wanted to take the letters E-L-L
from "HELLO,"
we would start the interval from the index 1,
but we'd end before the index 4.

Let's try this example and
extract a slice from a string in Python.
Let's type in the string and take the slice starting
at index 1 and ending before index 4.
Now, let's run the code and examine the output.
There's the slice we wanted.

Now that we know how to describe
the location of a character in a string,
let's learn how to search in a string.
To do this, we need to use the index method.
The index method finds the first occurrence of
the input in a string and returns its location.
Let's practice using the index method in Python.

Let's say we want to use the index method
to find the character "E" in the string "HELLO."
We'll locate the first instance of
the character "E." Let's examine this line in more detail.
After writing the string and the index method,
we use the character we want to
find as the argument of the index method.
Remember, the strings in Python are case-sensitive,
so we need to make sure
we use the appropriate case with the index method.
Let's run this code now.
This returned the number 1.
This is because "E" has an index value of 1.

Now, let's explore an example
where a character repeats multiple times in the string.
Let's try searching for
the character "L." We start with similar code as before,
passing the argument "L" instead of "E" to the index method.
Now, let's run this code and investigate the result.
The result is the index 2.
This tells us that the method only

identified the first occurrence of
the character "L" and not the second.
This is an important detail to
notice when working with the index method.

As a security analyst,
learning how to work with indices
allows you to find certain parts in a string.
For example, if you need to find
the location of the @ symbol in an email,
you can use the index method to find
what you're looking for with one line of code.

Now let's turn our attention
to an important property of the strings.
Have you ever heard the expression
"some things never change"?
It might be said about
the comfortable feeling you have with a good friend,
even when you haven't seen them for a long time.
Well, in Python, we can also say this about strings.
Strings are immutable.
In Python, "immutable" means that it
cannot be changed after it's
created and assigned a value.
Let's break this down with an example.

Let's assign the string "HELLO" to the variable my_string.
Now, if we want to change the character "E" to an "A"
so my_string has the value "HALLO,"
then we might be inclined to use index notation.
But here we get an error.
My_string is immutable, so we cannot
make changes like this. And there you have it!

You've just learned how to index and slice into strings.
You've also seen that strings are immutable.
You cannot reassign characters
after a string has been defined.
Coming up, we'll learn
about list operations and see that
lists can be changed with index notation. Meet you there.

Strings and the security analyst

The ability to work with strings is important in the cybersecurity profession. Previously, you were introduced to several ways to work with strings, including functions and methods. You also learned how to extract elements in strings using bracket notation and indices. This reading reviews these concepts and explains more about using the *.index()* method. It also highlights examples of string data you might encounter in a security setting.

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*

character	index
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])
```

```
h
h
```

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])
```

```
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*.

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.")
```

```
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"))
```

```
3
```

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"))
```

```
0
```

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)
```

```
7
```

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*".

Key takeaways

As a security analyst, you will work with strings in a variety of ways. First, you might need to use bracket notation to work with string indices. Two functions you will likely use are *str()*, which converts an input into a string, and *len()*, which finds the length of a string. You can also use string methods, functions that only work on strings. These include *.upper()*, which converts all letters in a string into uppercase letters, *.lower()*, which converts all letters in a string into lowercase letters, and *.index()*, which returns the index of the first occurrence of its input within a string.

List operations in Python

Another data type we discussed previously is the list.
Lists are useful because they allow you to store multiple pieces of data in a single variable.
In the security profession, you will work with a variety of lists.
For example, you may have a list of IP addresses that have accessed a network, and another list might hold information on applications that are blocked from running on the system.
Let's recap how to create a list in Python.
In this case, the items in our list are the letters A through E.
We separate them by commas and surround them with square brackets.
We can also assign our list to a variable to make it easier to use later.
Here, we've named our variable my_list.
When we access specific elements from lists, we use syntax similar to when we access the specific elements from strings.
We place its index value in brackets after the variable that stores the list.
So this would access the second item in the list.
This is because in Python, we start counting the elements in the list at zero and not at one.
So the index for the first element is zero and the index for the second element is one.
Let's try extracting some elements from a list.
We'll extract the second element by putting 1 in brackets after the variable.
We place this in a print() function to output the results, and after we run it, Python outputs the letter "b".
Similar to strings, we can also concatenate lists with the plus sign.
List concatenation is combining two lists into one by placing the elements of the second list directly after the elements of the first list.
Let's work with this in Python.
Let's concatenate two lists.
First, we define the same list as in the previous example and store it in the variable my_list.
Now, let's define an additional list with the numbers 1 through 4.
Finally, let's concatenate the two lists with a plus sign and print out the result.
And when we run it, we have a final concatenated list.
Having discussed the similarities, let's now explore the differences between lists and strings.
We mentioned earlier that strings are immutable, meaning after they are defined, they cannot be changed.
Lists, on the other hand, do not have this property, and we can freely change, add, and remove list values.
So, for example, if we have a list of malicious IP addresses, then every time a new malicious IP address is identified, we can easily add it to the list.
Let's first try changing a specific element in a list in Python.
We start with the list used in the previous example.
To change an element in a list, We combine what we learned about bracket notation with what we learned about variable assignment.

Let's change the second element in my_list, which is the string "b", to the number 7.
We place the object we want to change on the left-hand side of the variable assignment.
In this case, we'll change the second element in my_list.
Then we place an equals sign to indicate we are reassigning this element of the list.
Finally, we place the object to take its place on the right-hand side.
Here, we'll reassign the second list element to a value of 7.
Now let's print out the list and run the code to examine the change.
Perfect!
The letter "b" is now changed to the number 7.
Now, let's take a look at methods for inserting and removing elements in lists.
The first method we'll work with in this video is the insert method.
The insert method adds an element in a specific position inside a list.
The method takes two arguments: the first is the position we're adding the element to, and the second is the element we want to add.
Let's use the insert method.
We'll start with the list we defined in our my_list variable.
Then we type my_list.insert and pass in two arguments.
The first argument is the position where we want to insert the new information.
In this case, we want to insert into index 1.
The second argument is the information we want to add to the list; in this case, the integer 7.
Now let's print my_list.
Our list still begins with "a", the element with an index of 0, and now, we have the integer 7 in the next position, the position represented with an index of 1.
Notice that the letter "b", which was originally at index 1, did not get replaced like when we used bracket notation.
With the insert method, every element beyond index 1 is simply shifted down by one position.
The index of "b" is now 2.
Sometimes we might want to remove an element that is no longer needed from a list.
To do this, we can use the remove method.
The removed method removes the first occurrence of a specific element in the list.
Unlike insert, the argument of removed is not an index value.
Instead, you directly type the element you want to remove.
The remove method removes the first instance of it in the list.
Let's use the remove method to delete the letter "d" from our list.
We'll type the name of our variable my_list, then add the remove method.
We want to remove "d"from this list.
So, we'll place it in quotation marks as our argument.
Then we'll print my_list.
And let's run this.
Perfect!
"d" has now been removed from the list.
Just like with strings, being able to search through lists is a necessary skill for security analysts.
I'm looking forward to expanding our understanding as we move forward in this course.
nother data type we discussed previously is the list.
Lists are useful because they allow you to store multiple pieces of data in a single variable.
In the security profession, you will work with a variety of lists.

For example, you may have a list of IP addresses that have accessed a network, and another list might hold information on applications that are blocked from running on the system.

Let's recap how to create a list in Python.

In this case, the items in our list are the letters A through E.

We se.

Write a simple algorithm

In our everyday lives, we frequently follow rules for solving problems.
As a simple example, imagine you want a cup of coffee.
If you've made coffee many times, then you likely follow a process to make it.
First, you grab your favorite mug.
Then, you put water into the coffee maker and add your coffee grounds.
You press the start button and wait a few minutes.
Finally, you enjoy your fresh cup of coffee.
Even if you have a different approach to making coffee or don't drink coffee at all, you will likely follow a set of rules for completing similar everyday tasks.
When you complete these routine tasks, you're following an algorithm.
An algorithm is a set of rules that solve a problem.
In more detail, an algorithm is a set of steps that takes an input from a problem, uses this input to perform tasks, and returns a solution as an output.
Let's explore how algorithms can be used to solve problems in Python.
Imagine that you, as a security analyst, have a list of IP addresses.
You want to extract the first three digits of each IP address, which will tell you information about the networks that these IP addresses belong to.
To do this, we're going to write an algorithm that involves multiple Python concepts that we've covered so far: loops, lists, and strings.
Here's a list with IP addresses that are stored as strings.
For privacy reasons, in our example, we're not showing the full IP addresses.
Our goal is to extract the first three numbers of each address and store them in a new list.
Before we write any Python code, let's break down an approach to solving this problem with an algorithm.
What if you had one IP address instead of an entire list?
Well, then the problem becomes much simpler.
The first step in solving the problem will be to use string slicing to extract the first three digits from one IP address.
Now let's consider how to apply these to an entire list.
As the second step, we'll use a loop to apply that solution to every IP address on the list.
Previously, you learned about string slicing, so let's write some Python code to solve the problem for one IP address.
Here we're starting with one IP address that begins with 198.567.
And we'll write a few lines of code to extract the first three characters.
We'll use the bracket notation to slice the string.
Inside the print statement, we have the address variable, which contains the IP address we want to slice.
Remember that Python starts counting at 0.
To get the first three characters, we start our slice at index 0 and then continue all the way until index 3.

Remember, that Python excludes the final index.

In other words, Python will return the characters at index 0, 1, and 2.

Now, let's run this.

We get the first three digits of the address: 198.

Now that we're able to solve this problem for one IP address, we can put this code into a loop and apply it to all IP addresses in the original list.

Before we do this, let's introduce one more method that we'll be using in this code: the append method.

The append method adds input to the end of a list.

For example, let's say that my list contains 1, 2, and 3.

With this code, we can use the append method to add 4 to this list.

First, we are given the IP list.

Now, we're ready to extract the first three characters from each element in this list.

Let's create an empty list to store the first three characters of each IP from the list.

Now we can start the for loop.

Let's break this down.

The word "for" tells Python that we're about to start a for loop.

We then choose address as the variable inside of the for loop, and we specify the list called IP as the iterable.

As the loop runs, each element from the IP list will be stored temporarily in the address variable.

Inside the for loop, we have a line of code to add the slice from address to the networks list.

Breaking this down, we use the code we wrote earlier to get the first three characters of an IP address.

We'll use our append method to add an item to the end of a list.

In this case, we're adding to the networks list.

Finally, let's print the networks list and run the code.

The variable networks now contains a list of the first three digits of each IP address in the original list: IP.

That was a lot of information.

Designing algorithms can be challenging.

It's a good idea to break them down into smaller problems before jumping into writing your code.

We'll continue to practice this idea in the upcoming videos.

Meet you there.

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])
```

```
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:

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

```
['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 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)
```

```
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)
```

```
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)
```

```
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)
```

```
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)
```

```
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)
```

```
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.

Key takeaways

Python offers a lot of ways to work with lists. Bracket notation allows you to extract elements and slices from lists and also to alter them. List methods allow you to alter lists in a variety of ways. The *.insert()* and *.append()* methods add elements to lists while the *.remove()* method allows you to remove them. The *.index()* method allows you to find the index of an element in a list.

Regular expressions in Python

We've already learned a lot about working with strings.
This includes working with their positional indices and slicing them.
In the previous video, we applied these to extract the first three digits from a list of IP addresses.
In this video, we're going to focus on a more advanced way to search through strings.
We'll learn about searching for patterns in strings through regular expressions.
A regular expression, shortened to regex, is a sequence of characters that forms a pattern.
This pattern can be used when searching within log files.
We can use them to search for any kind of pattern.
For example, we can find all strings that start with a certain prefix, or we can find all strings that are a certain length.
We can apply this to a security context in a variety of ways.
For example, let's say we needed to find all IP addresses with a network ID of 184.
Regular expressions would allow us to efficiently search for this pattern.
We'll examine another example throughout this video.
Let's say that we want to extract all the email addresses containing a log.
If we try to do this through the index method, we would need the exact email addresses we were searching for.
As security analysts, we rarely have that kind of information.
But if we use a regular expression that tells Python how an email address is structured, it would return all the strings that have the same elements as an email address.
Even if we were given a log file with thousands of lines and entries, we could extract every email in the file by searching for the structure of an email address through a regular expression.
We wouldn't need to know the specific emails to extract them.
Let's explore the regular expression symbols that we need to do this.
To begin, let's learn about the plus sign.
The plus sign is a regular expression symbol that represents one or more occurrences of a specific character.
Let's explain that through an example pattern.
The regular expression pattern a+ matches a string of any length in which "a" is repeated.
For example, just a single "a", three "a's" in a row, or five "a's" in a row.
It could even be 1000 "a's" in a row.
We can start working with a quick example to see which strings this pattern would extract.
Let's start with this string of device IDs.
These are all the instances of the letter "a" written once or multiple times in a row.
The first instance has one "a", the second has two "a's", the third one has one "a", and the fourth has three "a's".
So, if we told Python to find matches to the a+ sign regular expression, it would return this list of

"a's".

The other building block we need is the \w symbol.

This matches with any alphanumeric character, but it doesn't match symbols.

"1", "k", and "i" are just three examples of what "\w" matches.

Regular expressions can easily be combined to allow for even more patterns in a search.

Before we apply this to our email context, let's explore the patterns we can search for if we combine the "\w" with the plus sign.

"\w" matches any alphanumeric character, and the plus sign matches any number of occurrences of the character before it.

This means that the combination of "\w+" matches an alphanumeric string of any length.

"\w" provides flexibility in the alphanumeric characters that this regular expression matches, and the plus sign provides flexibility in the length of the string that it matches.

The strings "192", "abc123", and "security" are just three possible strings that match to "\w+".

Now let's apply these to extracting email addresses from a log.

Email addresses consist of text separated by certain symbols, like the @ symbol and the period.

Let's learn how we can represent this as a regular expression.

To start, let's think about the format of a typical email address; for example, user1@email1.com.

The first segment of an email address contains alphanumeric characters, and the number of alphanumeric characters may vary in length.

We can use our regular expression "\w+" for this portion to match to an alphanumeric string of any length.

The next segment in an email address is the @ symbol.

This segment is always present.

We'll enter this directly in our regular expression.

Including this is essential for ensuring that Python distinguishes email addresses from other strings.

After the @ symbol is the domain name.

Just like the first segment, this one varies depending on the email address, but it always contains alphanumeric characters, so we can use "\w+" again to allow for this variation.

Next, just like the @ symbol, a period is always part of an email address.

But unlike the @ symbol, in regular expressions, the period has a special meaning.

For this reason, we need to use backslash period here.

When we add a backslash in front of it, we let Python know that we are not intending to use it as an operator, and that our pattern should include a period in this location.

For the last segment, we can also use "\w+".

This final part of an email address is often "com" but might be other strings like "net." When we put the pieces together, we get the regular expression we'll use to find email addresses in our row.

This pattern will match all email addresses.

It will exclude everything else in our string.

This is because we've included the @ symbol and the period where they appear in the structure of an email address.

Let's bring this into Python.

We'll use regular expressions to extract email addresses from a string.

Regular expressions can be used when the re module is imported into Python, so we begin with that step.

Later, we'll learn how to import and open files like logs.

But for now, we've restored our log as a string variable named email_log.

Because this is a multi-line string, we're using three sets of quotation marks instead of just one.
Next, we'll apply the findall() function from the re module to a regular expression.
re.findall() returns a list of matches to a regular expression.
Let's use this with the regular expression we created earlier for email addresses.
The first argument is the pattern that we want to match.
Notice that we place it in quotation marks.
The second argument indicates where to search for the pattern.
In this case, we're searching through the string contained within the email log variable.
When we run this, we get a list of all the emails in the string.
Imagine applying this to a log with thousands of entries.
Pretty useful, right?
This was just an introduction to the power of regular expressions.
There are many more symbols you can use.
I encourage you to explore regular expressions on your own and learn more.

More about regular expressions

You were previously introduced to regular expressions and a couple of symbols that you can use to construct regular expression patterns. In this reading, you'll explore additional regular expression symbols that can be used in a cybersecurity context. You'll also learn more about the *re* module and its *re.findall()* function.

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")
```

```
['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*".

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

```
['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.

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")
```

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

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")
```

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

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")
```

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

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")
```

```
['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. 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")
```

```
['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:

```
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))
```

```
['bmoreno: 12', 'tshah: 7', 'sgilmore: 5']
```

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.

Key takeaways

Regular expressions allow you to search through strings to find matches to specific patterns. You can use regular expressions by importing the *re* module. This module contains multiple functions, including *re.findall()*, which returns all matches to a pattern in the form of a list. To form a pattern, you use characters and symbols. Symbols allow you to specify types of characters and to quantify how many repetitions of a character or type of character can occur in the pattern.

Wrap-up

Congratulations!
We accomplished a lot together.
Let's take time to quickly go through all the new concepts we covered.
We started this course by focusing on working with strings and lists.
We learned methods that work specifically with these data types.
We also learned to work with indices and extract information we need.
Next, we focused on writing algorithms.
We wrote a simple algorithm that sliced the network ID from a list of IP addresses.
Finally, we covered using regular expressions.
Regular expressions allow you to search for patterns, and this provides expanded ways to locate what you need in logs and other files.
These are complex concepts, and you're always welcome to visit the videos again whenever you like.
With these concepts, you took a big step towards being able to work with data and write the algorithms that security professionals need.
Throughout the rest of this course, you're going to get more practice with Python and what it can offer to security analysts.

Reference guide: Python concepts from module 3; Terms and definitions from Course 7, Module 3

Built-in functions

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

`str()`

Converts the input object to a string

```
str(10)
```

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

`len()`

Returns the number of elements in an object

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

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

String methods

The following methods can be applied to strings in Python.

`.upper()`

Returns a copy of the string in all uppercase letters

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

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

`.lower()`

Returns a copy of the string in all lowercase letters

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

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

`.index()`

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

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

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

List methods

The following methods can be applied to lists in Python.

`.insert()`

Adds an element in a specific position inside the list

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

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

`.remove()`

Removes the first occurrence of a specific element inside a list

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

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

`.append()`

Adds input to the end of a list

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

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

.index()

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

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

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

Additional syntax for working with strings and lists

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

+ (concatenation)

Combines two strings or lists together

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

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

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

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

[] (bracket notation)

Uses indices to extract parts of a string or list

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

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

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

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

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

Extracts the element at index `2`, which is (`"tshah"`), from the `username_list`

Regular expressions

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

re.findall()

Returns a list of matches to a regular expression

```
import re
    re.findall("a53", "a53-32c .E")
```

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

\w

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

```
import re
    re.findall("\w", "a53-32c .E")
```

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

.

Matches to all characters, including symbols

```
import re
    re.findall(".", "a53-32c .E")
```

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

\d

Matches to all single digits

```
import re
    re.findall("\d", "a53-32c .E")
```

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

`\s`

Matches to all single spaces

```
import re
    re.findall("\d", "a53-32c .E")
```

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

`\.`

Matches to the period character

```
import re
    re.findall("\.", "a53-32c .E")
```

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

`+`

Represents one or more occurrences of a specific character

```
import re
    re.findall("\w+", "a53-32c .E")
```

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

`*`

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

```
import re
    re.findall("\w*", "a53-32c .E")
```

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

{ }

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

```
import re
    re.findall("\w{3}", "a53-32c .E")
```

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

Glossary terms from module 3

Algorithm: A set of rules that solve a problem

Bracket notation: The indices placed in square brackets

Debugging: The practice of identifying and fixing errors in code

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

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

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

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

Method: A function that belongs to a specific data type

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

String concatenation: The process of joining two strings together

String data: Data consisting of an ordered sequence of characters

Substring: A continuous sequence of characters within a string