# Module 4 - Google IT Automation with Python Professional Certificate

## Strings, Lists and Dictionaries

In this module you'll dive into more advanced ways to manipulate strings using indexing, slicing, and advanced formatting. You'll also explore the more advanced data types: lists, tuples, and dictionaries. You'll learn to store, reference, and manipulate data in these structures, as well as combine them to store complex data structures.

### **Learning Objectives**

* Manipulate strings using indexing, slicing, and formatting
* Use lists and tuples to store, reference, and manipulate data
* Leverage dictionaries to store more complex data, reference data by keys, and manipulate data stored
* Combine the String, List, and Dictionary data types to construct complex data structures

## **Strings**

### Basic Structures Introduction

Now that we have the basics of syntax out of the way, we can start growing our Python knowledge which will let us do more and more interesting operations. Remember, one of our main goals in this course is to help you learn to write short Python scripts that automate actions. In the upcoming videos, we're going to learn a bunch of new super useful skills to add to your programming toolbox. We'll check out some data types provided by the Python language to help us solve common problems with our scripts.

In particular, we'll do a deep dive into strings, lists, and dictionaries. Heads-up, while we've used strings in our scripts already, we barely scratched the surface of all the things we can do with them in Python. We also ran into a few lists in some examples but there's a lot more of them we haven't seen yet. Dictionaries are a whole new data type for us to dig our teeth into. These are all data types or data structures that are super flexible. We're going to use them to write all kinds of scripts in Python. So it's a good idea to spend some time getting to know them, and learning when to use them, and how to make the most out of them. We've got a lot of new and exciting concepts to discover. So let's get right to it.

### Review: What is a string?

Introduction

This follow-along reading is organized to match the content in the video that follows. It contains the same code shown in the next video. These code blocks will provide you with the opportunity to see how the code is written, allow you to practice running it, and can be used as a reference to refer back to.

You can follow along in the reading as the instructor discusses the code or review the code after watching the video.

| name = "Sasha" color = 'Gold'  place = "Cambridge' #This will throw an error  pet = ""  name = "Sasha" color = 'Gold' print("Name: " + name + ", Favorite color: " + color)  "example" \* 3  pet = "loooooooooooooooooooooooooooooooog cat" len(pet) |
| --- |

### What is a string?

By now, we've used strings in a lot of examples, but we haven't spent time looking at them in detail yet. Before we dive into the nitty-gritty though, let's go over what we've seen so far and add a few more points. First, a quick refresher.

A string is a data type in Python that's used to represent a piece of text. It's written between quotes, either double quotes or single quotes.

| color = 'Gold' |
| --- |

It doesn't matter which type of quotes you use as long as they match. If we mix up double and single quotes, Python won't be too happy, and it'll return a syntax error, telling us it couldn't find the end of the string.

| #This will throw an error |
| --- |

A string can be as short as zero characters, usually called an empty string or really long.

| #This will throw an error pet = "" |
| --- |

We also learned that we can use strings to build longer strings using the plus sign and action called concatenating.

| print("Name: " + name + ", Favorite color: " + color) |
| --- |

A less common operation is to multiply the string by a number, which multiplies the content of the string that many times like this

| print("Name: " + name + ", Favorite color: " + color) "example" \* 3 |
| --- |

If we want to know how long a string is, we can use the len function which we saw in earlier videos. The len function tells us the number of characters contained in the string.

| len(pet) |
| --- |

We can use strings to represent a lot of different things. They can hold a username, the name of a machine, an email address, the name of a file, and any other text. A lot of the data that we'll interact with will be stored in strings, so it's important to know how to use them.

There are tons of things we could do with strings in our scripts. For example, we can check if files are named a certain way by looking at the filename and seeing if they match our criteria, or we can create a list of emails by checking out the users of our system and concatenating our domain.

I recently wrote a script that worked with a bunch of files and took different actions according to the name of each file. So the file ended in a certain extension, say, .TXT , then my script would print it. If the file had a certain string and the name, say, test, then my script would ignore it and move on to the next thing and so on. The contents of a text file are also strings.

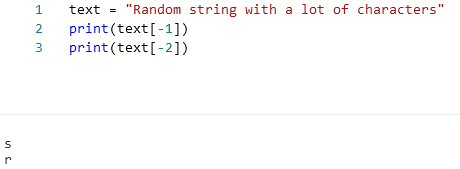
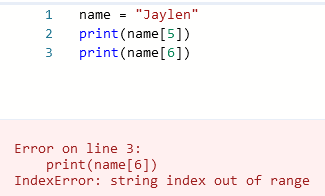
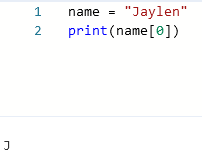
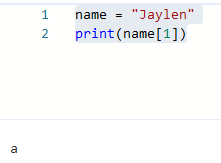
A few months ago, I had to change the default values for a bunch of configuration options from true to false. So I wrote a function that would find the string true in a file and replace it with false. You can probably think of more examples where your code needs to handle strings, but to use strings effectively, we need to know what options are available to us in Python. In the next few videos, we'll cover some of the operations we can perform over strings, including how to access parts of them and modify them.

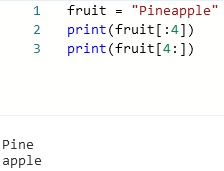
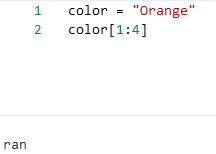
### Review: The parts of a string

Introduction

This follow-along reading is organized to match the content in the video that follows. It contains the same code shown in the next video. These code blocks will provide you with the opportunity to see how the code is written, allow you to practice running it, and can be used as a reference to refer back to.

You can follow along in the reading as the instructor discusses the code or review the code after watching the video.

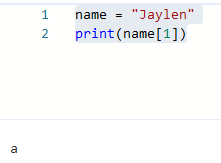




### The Parts of a String

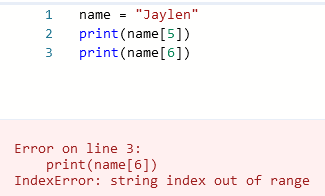
When we first came across the for loop, we called out that we can iterate over a string character by character. But what if we want to access just a specific character or characters? We might want to do this, for example, if we have a text that's too long to display and we want to show just a portion of it. Or if we want to make an acronym by taking the first letter of each word in a phrase. We can do that through an operation called string indexing.

This operation lets us access the character in a given position or index using square brackets and the number of the position we want. Like this.



This might seem confusing at first, like Python is acting up. We're asking for the first character, and it's giving us the second. What gives Python? Well, what's happening here is that Python starts counting indexes from 0 not 1. Just like it does with the range function.

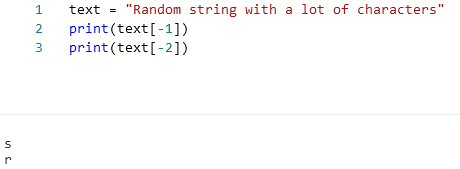
So if we want the first character, we need to access the one at index 0. Knowing that indexes start at 0, which one do you think will be the last index in the string? It'll always be one less than the length of the string. In this case, our string has six characters, so the last index will be 5. Let's try it out.



We see that the character in position five is the last character of the string. And if we try to access index six, we get an index error telling us that it's out of range.

We can only go up to length minus 1. What if you want to print the last character of a string but you don't know how long it is? You can do that using negative indexes. Let's see that in a different example.

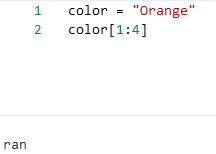
In this example, we don't know the length of the string, but it doesn't matter. Using negative indexes lets us access the positions in the string starting from the last.



On top of accessing individual characters, we can also access a slice of a string.

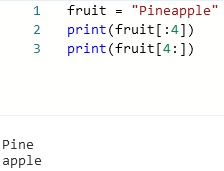
A slice is the portion of a string that can contain more than one character, also sometimes called a substring.

We do that by creating a range using a colon as a separator. Let's see an example of this.



The range we use when accessing a slice of a string works just like the one created by the range function. It includes the first number, but goes up to one less than the last number. In this case, we start with indexed one, the second letter of the string, and go up to index three, the fourth letter of the string.

Another option for the range is to include only one of the two indexes. In that case, it's assumed that the other index is either 0 for the first value or the length of the string for the second value. Check this out.



Accessing the slice from nothing to 4 takes the first four characters of the string, indexes 0 to 3. Accessing the slice from 4 to nothing takes everything from index four onward.

All of this indexing might seem confusing at first. Don't worry, we all took time to wrap our heads around it. Just like all the challenges we've come across so far, the key is to keep practicing until you master it. And there are a bunch of exercises ahead to help you with that. Now that we know how to select, slice, and access the parts of the string we want, we're going to learn how to modify them. That's coming up next.

### String Indexing and Slicing

String indexing allows you to access individual characters in a string. You can do this by using square brackets and the location, or index, of the character you want to access. It's important to remember that Python starts indexes at 0. So to access the first character in a string, you would use the index [0]. If you try to access an index that’s larger than the length of your string, you’ll get an **IndexError**. This is because you’re trying to access something that doesn't exist! You can also access indexes from the end of the string going towards the start of the string by using negative values. The index [-1] would access the last character of the string, and the index [-2] would access the second-to-last character.

You can also access a portion of a string, called a slice or a substring. This allows you to access multiple characters of a string. You can do this by creating a range, using a colon as a separator between the start and end of the range, like [2:5].

This range is similar to the range() function we saw previously. It includes the first number, but goes to one less than the last number. For example:

| *>>> fruit = "Mangosteen" >>> fruit[1:4] 'ang'* |
| --- |

The slice *includes* the character at index 1, and *excludes* the character at index 4. You can also easily reference a substring at the start or end of the string by only specifying one end of the range. For example, only giving the end of the range:

| *>>> fruit = "Mangosteen" >>> fruit[1:4] 'ang' >>> fruit[:5] 'Mango'* |
| --- |

This gave us the characters from the start of the string through index 4, *excluding* index 5. On the other hand this example gives is the characters *including* index 5, through the end of the string:

| *>>> fruit = "Mangosteen" >>> fruit[1:4] 'ang' >>> fruit[:5] 'Mango' >>> fruit[5:] 'steen'* |
| --- |

You might have noticed that if you put both of those results together, you get the original string back!

| *name = "Sasha" color = 'Gold' place = "Cambridge' #This will throw an error pet = "" name = "Sasha" color = 'Gold' print("Name: " + name + ", Favorite color: " + color) "example" \* 3 pet = "loooooooooooooooooooooooooooooooog cat" len(pet) >>> fruit = "Mangosteen" >>> fruit[1:4] 'ang' >>> fruit[:5] 'Mango' >>> fruit[5:] 'steen' >>> fruit[:5] + fruit[5:] 'Mangosteen'* |
| --- |

Cool!

## 

### Practice: Work with index values

### 

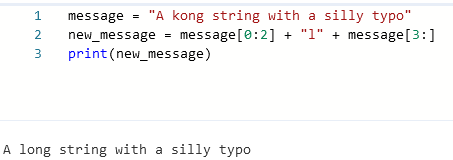
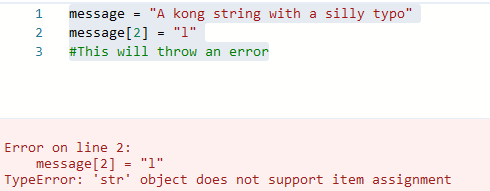
### Review: Creating new strings

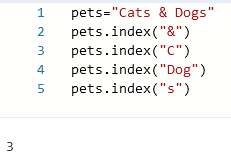
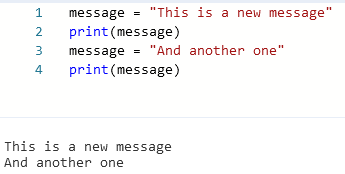
This reading contains the code used in the instructional videos from [**Creating new strings**](https://www.coursera.org/learn/python-crash-course/lecture/2OlJR/creating-new-strings)

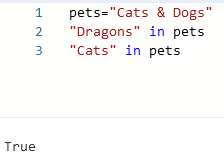
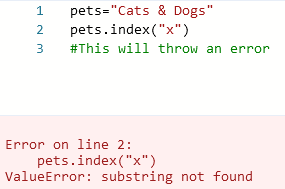
Introduction

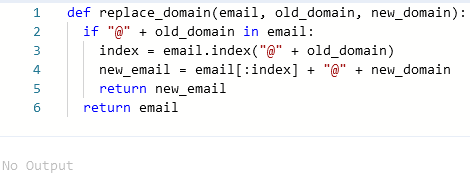
This follow-along reading is organized to match the content in the video that follows. It contains the same code shown in the next video. These code blocks will provide you with the opportunity to see how the code is written, allow you to practice running it, and can be used as a reference to refer back to.

You can follow along in the reading as the instructor discusses the code or review the code after watching the video.



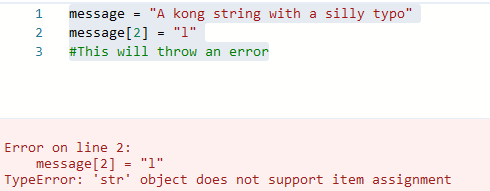






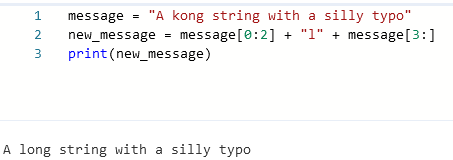
### Creating New Strings

In the last video, we saw how to access certain characters inside a string. Now, what if we wanted to change them? Imagine you have a string with a character that's wrong and you want to fix it, like this one.

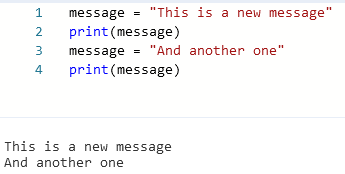


Taking into account what you learned about string indexing, you might be tempted to fix it by accessing the corresponding index and changing the character. Let's see what happens if we try that.

These pesky type errors, right? In this case, we're told that strings don't support item assignment. This means that we can't change individual characters because **strings in Python are immutable**, which is just a fancy word meaning they can't be modified. What we can do is create a new string based on the old one, like this.

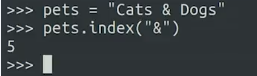


Nice, we fixed the typo. But does this mean the message variable can never change? Not really. We can assign a new value to the same variable. Let's do that a couple of times to see how it works.



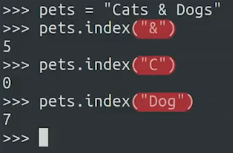
What we're doing here, is giving the message variable a whole new value. We're not changing the underlying string that was assigned to it before. We're assigning a whole new string with different content.

If this seems a bit complex, that's okay. You don't need to worry about this right now. We'll call this out whenever it's relevant for the programmer writing. So, we figured out how to create a new message from the old one. But how are we supposed to know which character to change? Let's try something different.



In this case, we're using a **method** to get the index of a certain character. **A method is a function associated with a specific class.**

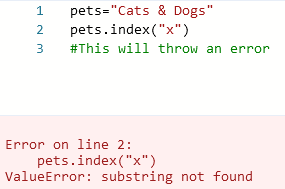
We'll talk a lot more about classes and methods later. For now, what you need to know is that this is a function that applies to a variable. And we can call it by following the variable with a dot. Let's try this a few more times.



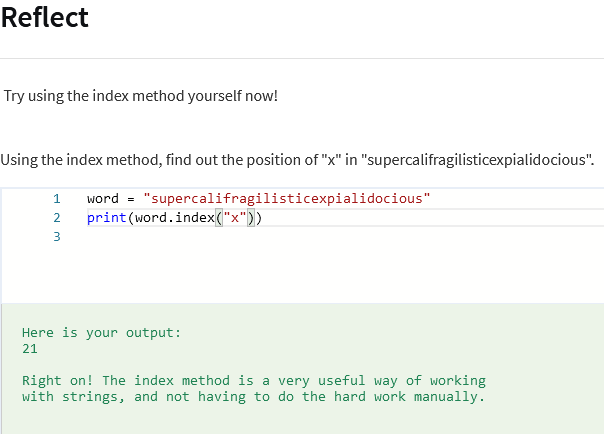
So the index method returns the index of the given substring, inside the string. The substring that we pass, can be as long or as short as we want. What if there's more than one of the substring?



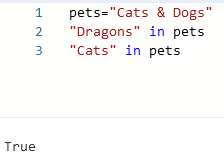
Here, we know there are two s characters, but we only get one value. That's because the index method returns just the first position that matches. And what happens if the string doesn't have the substring we're looking for?



The index method can't return a number because the substring isn't there, so we get a value error instead.



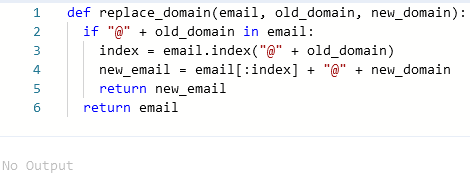
We said that if the substring isn't there, we would get an error. So how can we know if a substring is contained in a string to avoid the error? Let's check this out.



We can use the key word in to check if a substring is contained in a string. We came across the keyword in, when using four loops. In that case, it was used for iteration. Here, it's a conditional that can be either true or false. It'll be true if the substring is part of the string, and false if it's not. So here, the Dragons substring isn't part of the string, and sadly, we can't have a Dragon as a pet.

Let's put all the stuff together to solve a real-world problem.

Imagine that your company has recently moved to using a new domain, but a lot of the company email addresses are still using the old one. You want to write a program that replaces this old domain with the new one in any outdated email addresses. The function to replace the domain would look like this.



This function is a bit more complex than others, so let's go through it line by line. First, we define the replace\_domain function which accepts three parameters.

The email address to be checked, the old domain, and the new domain. Having all these values as parameters instead of directly in the code, makes our function reusable. We aren't just changing one domain to the other, we have a function that will work with all domains.

In the first line of the body of the function, we check if the concatenation of the @ sign and the old domain are contained in the email address, using the keyword in. We check this to make sure the email has "old domain" on the portion that comes after the @ sign.

If the condition is true, the email address needs to be updated. To do that, we first find out the index where the old domain, including the @ sign starts. We know that this index will be a valid number because we've already checked that the substring was present. So, using this index, we create the new email.

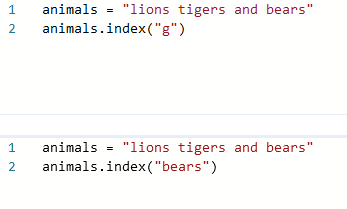
This is a string that contains the first portion of the old email, up until the index we had calculated, followed by the @ sign and the new domain. Finally, we return this new email. If the email didn't contain the new domain, then we can just return it, which is what we do in the last line.

Wow, that was a really complex function with a lot of new things in it. So don't worry if you're finding it a bit tricky. Re-watch the video and take your time. If there's a specific part that's tripping you up, remember, you can always ask your fellow learners for help in the discussion forum. You may even find that someone has asked and got the answer to the same question already.

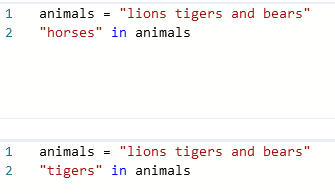
### Basic String Methods

In Python, strings are immutable. This means that they can't be modified. So if we wanted to fix a typo in a string, we can't simply modify the wrong character. We would have to create a new string with the typo corrected. We can also assign a new value to the variable holding our string.

If we aren't sure what the index of our typo is, we can use the string method *index* to locate it and return the index. Let's imagine we have the string **"lions tigers and bears"** in the variable **animals**. We can locate the index that contains the letter **g** using *animals.index("g")*, which will return the index; in this case 8. We can also use substrings to locate the index where the substring begins. *animals.index("bears")* would return 17, since that’s the start of the substring. If there’s more than one match for a substring, the index method will return the first match. If we try to locate a substring that doesn't exist in the string, we’ll receive a **ValueError** explaining that the substring was not found.



We can avoid a ValueError by first checking if the substring exists in the string. This can be done using the ***in*** keyword. We saw this keyword earlier when we covered *for* loops. In this case, it's a conditional that will be either True or False. If the substring is found in the string, it will be True. If the substring is not found in the string, it will be False. Using our previous variable **animals**, we can do **"horses"** **in animals** to check if the substring "horses" is found in our variable. In this case, it would evaluate to False, since horses aren’t included in our example string. If we did **"tigers" in animals**, we'd get True, since this substring is contained in our string.



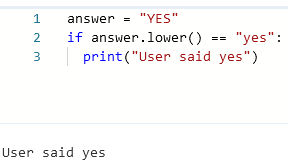
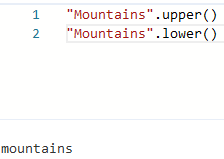
### Review: More string methods

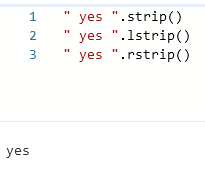
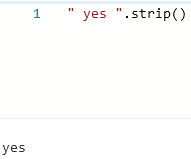
This reading contains the code used in the instructional videos from [**More string methods**](https://www.coursera.org/learn/python-crash-course/lecture/45PT2/more-string-methods)

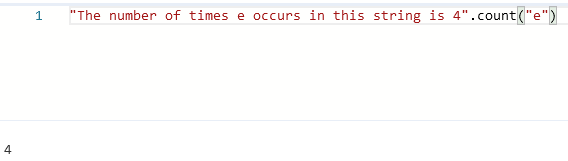
Introduction

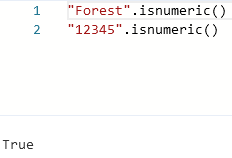
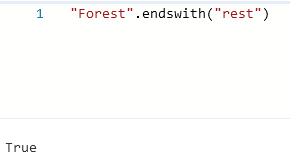
This follow-along reading is organized to match the content in the video that follows. It contains the same code shown in the next video. These code blocks will provide you with the opportunity to see how the code is written, allow you to practice running it, and can be used as a reference to refer back to.

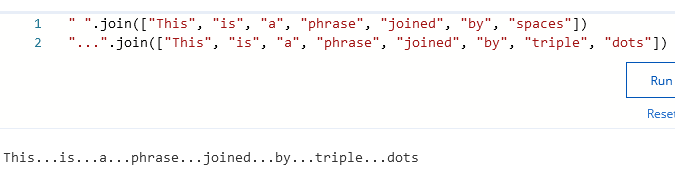
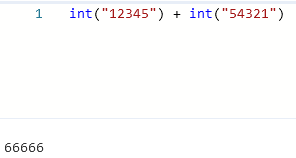
You can follow along in the reading as the instructor discusses the code or review the code after watching the video.

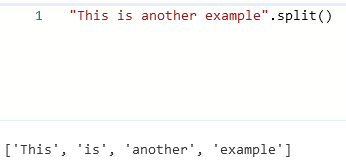








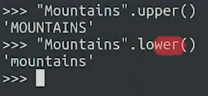




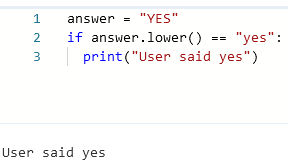
### More String Methods

So far, we've seen ways you can access portions of strings using the indexing technique, create new strings by slicing and concatenating, find characters and strings using the index method, and even test if one string contains another. On top of all this string processing power, the string class provides a bunch of other methods for working with text. Now, we'll show you how to use some of these methods. Remember, the goal is not for you to memorize all of this. Instead, we want to give you an idea of what you can do with strings in Python.

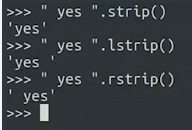
Some string methods let you perform transformations or formatting on the string text, like upper, and its opposite, lower.



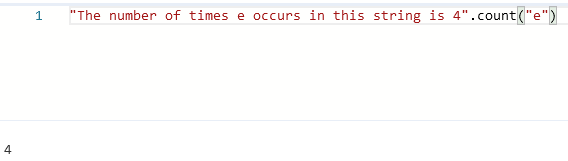
These methods are really useful when you're handling user input. Let's say you wanted to check if the user answered yes to a question. How would you know if the user typed it using upper or lower case? You don't need to, you just transform the answer to the case you want. Like this example.



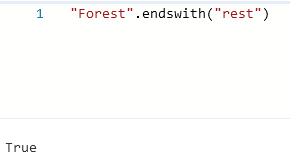
Another useful method when dealing with user input is the strip method. This method will get rid of surrounding spaces in the string. If we ask the user for an answer, we usually don't care about any surrounding spaces. So it's a good idea to use the strip method to get rid of any white space. This means that strip doesn't just remove spaces, it also removes tabs and new line characters, which are all characters we don't usually want in user-provided strings. There are two more versions of this method, lstrip rstrip, to get rid of the whitespace characters just to the left or to the right of the string instead of both sides.



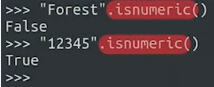
Other methods give you information about the string itself. The method count returns how many times a given substring appears within a string.



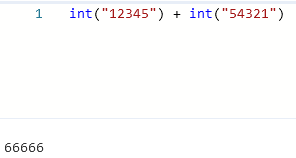
The method endswith returns whether the string ends with a certain substring.



The method isnumeric returns whether the string's made up of just numbers.

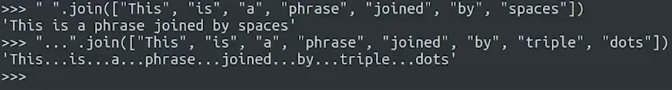


Adding to that, if we have a string that is numeric, we can use the int function to convert it to an actual number.



In earlier videos, we showed that we can concatenate strings using the plus sign. The join method can also be used for concatenating.

To use the join method, we have to call it on the string that'll be used for joining. In this case, we're using a string with a space in it. The method receives a list of strings and returns one string with each of the strings joined by the initial string. Let's check out another example.



Finally, we can also split a string into a list of strings.



The split method returns a list of all the words in the initial string and it automatically splits by any whitespace. It can optionally take a parameter and split the strings by another character, like a comma or a dot.

We've just learned a bunch of new methods. But there are tons more that you can use on strings. We've included a list with the ones we talked about, and some new ones in the next cheat sheet. You'll also find a link to the full Python documentation there, which gives you all the info on each available method.

As we've said before, don't worry about trying to memorize everything. You'll pick these concepts up with practice, and the documentation is always there if you need it. All right, last up in our string of string videos, we're going to check out how to format strings.

### Advanced String Methods

We've covered a bunch of String class methods already, so let's keep building on those and run down some more advanced methods.

The string method **lower** will return the string with all characters changed to lowercase. The inverse of this is the **upper** method, which will return the string all in uppercase. Just like with previous methods, we call these on a string using dot notation, like **"this is a string".upper()**. This would return the string **"THIS IS A STRING"**. This can be super handy when checking user input, since someone might type in all lowercase, all uppercase, or even a mixture of cases.

You can use the **strip** method to remove surrounding whitespace from a string. Whitespace includes spaces, tabs, and newline characters. You can also use the methods **lstrip** and **rstrip** to remove whitespace only from the left or the right side of the string, respectively.

The method **count** can be used to return the number of times a substring appears in a string. This can be handy for finding out how many characters appear in a string, or counting the number of times a certain word appears in a sentence or paragraph.

If you wanted to check if a string ends with a given substring, you can use the method **endswith**. This will return True if the substring is found at the end of the string, and False if not.

The **isnumeric** method can check if a string is composed of only numbers. If the string contains only numbers, this method will return True. We can use this to check if a string contains numbers before passing the string to the **int()** function to convert it to an integer, avoiding an error. Useful!

We took a look at string concatenation using the plus sign, earlier. We can also use the **join** method to concatenate strings. This method is called on a string that will be used to join a list of strings. The method takes a list of strings to be joined as a parameter, and returns a new string composed of each of the strings from our list joined using the initial string. For example, **" ".join(["This","is","a","sentence"])** would return the string **"This is a sentence"**.

The inverse of the join method is the **split** method. This allows us to split a string into a list of strings. By default, it splits by any whitespace characters. You can also split by any other characters by passing a parameter.

### Review: Formatting strings

Introduction

This follow-along reading is organized to match the content in the video that follows. It contains the same code shown in the next video. These code blocks will provide you with the opportunity to see how the code is written, allow you to practice running it, and can be used as a reference to refer back to.

You can follow along in the reading as the instructor discusses the code or review the code after watching the video.

| **name = "Manny" number = len(name) \* 3 print("Hello {}, your lucky number is {}".format(name, number))  Hello Manny, your lucky number is 15** |
| --- |

| **name = "Manny" print("Your lucky number is {number}, {name}.".format(name=name, number=len(name)\*3))  Your lucky number is 15, Manny.** |
| --- |

| **price = 7.5 with\_tax = price \* 1.09 print(price, with\_tax) print("Base price: ${:.2f}. With Tax: ${:.2f}".format(price, with\_tax))  7.5 8.175 Base price: $7.50. With Tax: $8.18** |
| --- |

| **def to\_celsius(x):  return (x-32)\*5/9  for x in range(0,101,10):  print("{:>3} F | {:>6.2f} C".format(x, to\_celsius(x)))   0 F | -17.78 C  10 F | -12.22 C  20 F | -6.67 C  30 F | -1.11 C  40 F | 4.44 C  50 F | 10.00 C  60 F | 15.56 C  70 F | 21.11 C  80 F | 26.67 C  90 F | 32.22 C 100 F | 37.78 C** |
| --- |

### Formatting Strings

Up to now we've been making strings using the plus sign to just concatenate the parts of the string we wanted to create.

And we've used the str function to convert numbers into strings so that we can concatenate them, too. This works, but it's not ideal, especially when the operations you want to do with the string or on the tricky side.

There's a better way to do this using the format method. Let's see a couple of examples.

| print("Hello {}, your lucky number is {}".format(name, number)) |
| --- |

**Hello Manny, your lucky number is 15**

In this example, we have two variables, name and number. We generate a string that has those variables in it

| print("Hello {}, your lucky number is {}".format(name, number)) number = len(name) \* 3 |
| --- |

by using the curly brackets placeholder to show where the variables should be written (name, number). We then pass the variables as a parameter to the format method .format(name, number)).

See how it doesn't matter that name is a string and number is an integer? The format method deals with that, so we don't have to.

The curly brackets aren't always empty. By using certain expressions inside those brackets, we can take advantage of the full power of the format expression.

One of the things we can put inside the curly brackets is the name of the variable we want in that position to make the whole string more readable. This is particularly relevant when the text can get rewritten or translated and the variables might switch places. In our earlier example, we could rewrite the message to make the variables appear in a different order. In that case, we'd need to pass the parameters to format in a slightly different way.

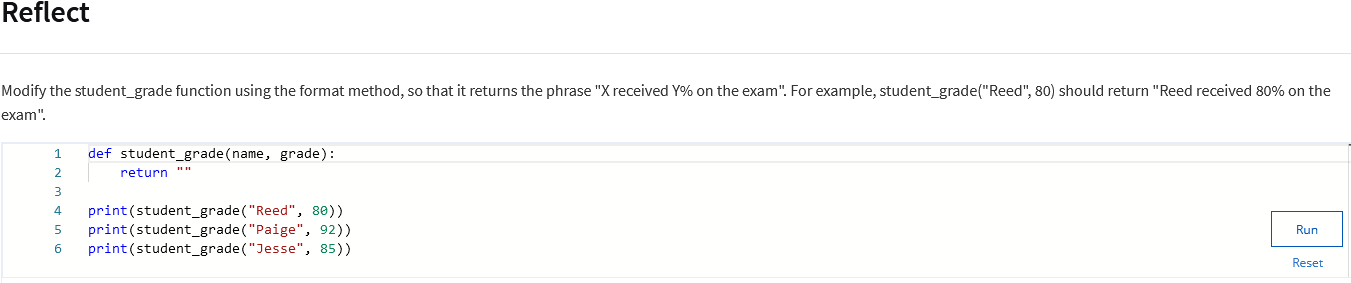
| print("Hello {}, your lucky number is {}".format(name, number)) number = len(name) \* 3 print("Your lucky number is {number}, {name}.".format(name=name, number=len(name)\*3)) |
| --- |

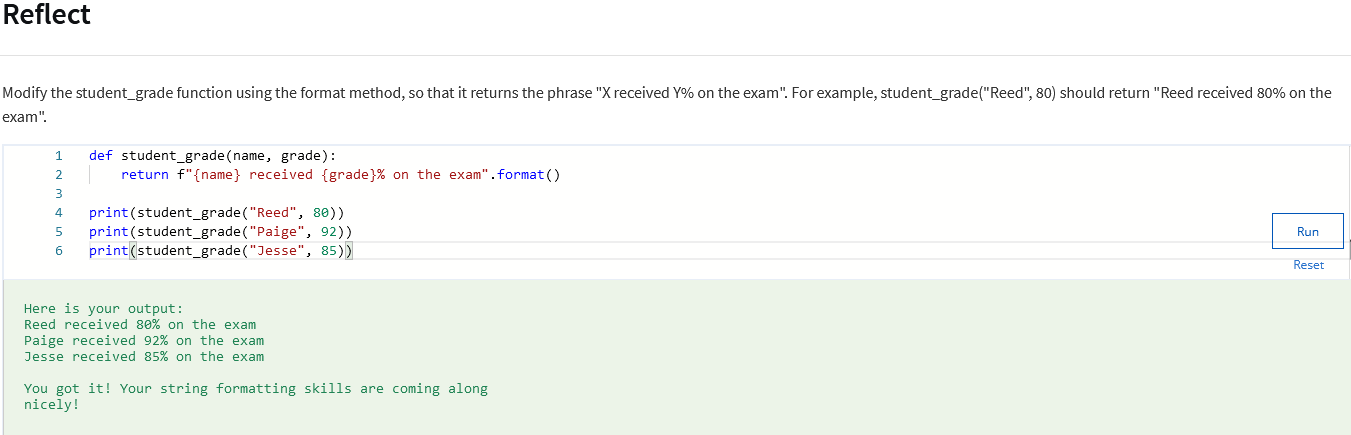
| name = "Manny" number = len(name) \* 3 print("Hello {}, your lucky number is {}".format(name, number)) name = "Manny" number = len(name) \* 3 name = "Manny" print("Your lucky number is {number}, {name}.".format(name=name, number=len(name)\*3)) Your lucky number is 15, Manny. |
| --- |

Because we're using placeholders with variable names, the order in which the variables are passed to the format function doesn't matter.

But for this to work, we need to set the names we're going to use and assign a value to them inside the parameters to format.

| name=name, number=len(name)\*3) |
| --- |





And that's just the tip of the iceberg of what we can do with the format method.

Let's say you want to output the price of an item with and without tax. Depending on what the tax rate is, the number might be a long number with a bunch of decimals. So if something costs $7.5 without tax and the tax rate is 9%, the price with tax would be $8.175.

| print(price, with\_tax |
| --- |

Since there's no such thing as half a penny anymore, that number doesn't make sense. So to fix this we can make the format function print only two decimals, like this.

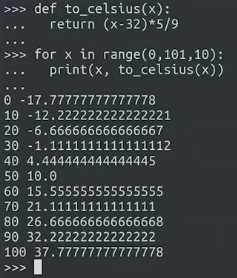
| **price = 7.5 with\_tax = price \* 1.09 print(price, with\_tax price = 7.5 with\_tax = price \* 1.09 print(price, with\_tax) print("Base price: ${:.2f}. With Tax: ${:.2f}".format(price, with\_tax))  7.5 8.175 Base price: $7.50. With Tax: $8.18** |
| --- |

In this case between the curly brackets we're writing a formatting expression{:.2f}. There are a bunch of different expressions we can write. These expressions are needed when we want to tell Python to format our values in a way that's different from the default.

The expression starts with a colon to separate it from the field name that we saw before. After the colon, we write .2f.

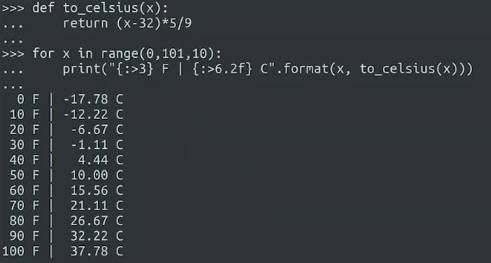
This means we're going to format a float number and that there should be two digits after the decimal dot. So no matter what the price is, our function always prints two decimals.

Remember when we did the table to convert from Fahrenheit to Celsius temperatures?



Our table looked kind of ugly because it was full of float numbers that had way too many decimal digits.

Using the format function, we can make it look a lot nicer.



| def to\_celsius(x):  return (x-32)\*5/9  for x in range(0,101,10):  print("{:>3} F | {:>6.2f} C".format(x, to\_celsius(x))) |
| --- |

In these two expressions we're using the greater than operator to align text to the right so that the output is neatly aligned{:>3} F | {:>6.2f}.

In the first expression we're saying we want the numbers to be aligned to the right for a total of three spaces{:>3}. In the second expression we're saying we want the number to always have exactly two decimal places and we want to align it to the right at six spaces {:>6.2f}. We can use string formatting like this to make the output of our program look nice and also to generate useful logging and debugging messages.

### String Formatting

You can use the **format** method on strings to concatenate and format strings in all kinds of powerful ways. To do this, create a string containing curly brackets, **{}**, as a placeholder, to be replaced. Then call the format method on the string using *.format()* and pass variables as parameters. The variables passed to the method will then be used to replace the curly bracket placeholders. This method automatically handles any conversion between data types for us.

If the curly brackets are empty, they’ll be populated with the variables passed in the order in which they're passed. However, you can put certain expressions inside the curly brackets to do even more powerful string formatting operations. You can put the name of a variable into the curly brackets, then use the names in the parameters. This allows for more easily readable code, and for more flexibility with the order of variables.

You can also put a formatting expression inside the curly brackets, which lets you alter the way the string is formatted. For example, the formatting expression **{:.2f}** means that you’d format this as a float number, with two digits after the decimal dot. The colon acts as a separator from the field name, if you had specified one. You can also specify text alignment using the greater than operator: **>**. For example, the expression **{:>3.2f}** would align the text three spaces to the right, as well as specify a float number with two decimal places. String formatting can be very handy for outputting easy-to-read textual output.

### String Reference Guide

<https://github.com/GaJoDev/Python/blob/main/learning_resources/links/format_string_ref_guide.md>

In Python, there are a lot of things you can do with strings. In this study guide, you’ll find the most common string operations and string methods.

#### String operations

| len(string) |
| --- |

Returns the length of the string

| for character in string |
| --- |

Iterates over each character in the string

| for character in string string[i] |
| --- |

Accesses the character at index i of the string, starting at zero

| print(len("abcde")) # prints 5 for character in string for c in "abcde": print(c) # prints "a", then "b", then "c", etc. if substring in string  Checks whether the substring is part of the string print("abc" in "abcde") # prints True print("def" in "abcde") # prints False string[i] print("abcde"[2]) # prints "c" print("abcde"[-1]) # prints "e" string[i:j] |
| --- |

Accesses the substring starting at index i, ending at index j minus 1. If i is omitted, its value defaults to 0. If j is omitted, Python returns everything from i to the end of the string.

#### String methods

string.lower() Returns a copy of the string with all lowercase characters

| print("AaBbCcDdEe".lower()) # prints "aabbccddee" |
| --- |

string.upper() Returns a copy of the string with all uppercase characters

| print("AaBbCcDdEe".upper()) # prints "AABBCCDDEE" |
| --- |

string.lstrip() Returns a copy of the string with the left-side whitespace removed

| print(" Hello ".lstrip()) # prints "Hello " |
| --- |

string.rstrip() Returns a copy of the string with the right-side whitespace removed

| print(" Hello ".rstrip()) # prints " Hello" |
| --- |

string.strip() Returns a copy of the string with both the left and right-side whitespace removed

| print(" Hello ".strip()) # prints "Hello" |
| --- |

string.count(substring) Returns the number of times substring is present in the string

| test = "How much wood would a woodchuck chuck"  print(test.count("wood")) # prints 2 |
| --- |

string.isnumeric() Returns True if there are only numeric characters in the string. If not, returns False.

| print("12345".isnumeric()) # prints True print("-123.45".isnumeric()) # prints False |
| --- |

string.isalpha() Returns True if there are only letters in the string. If not, returns False.

| print("xyzzy".isalpha()) # prints True |
| --- |

string.split() Returns a list of substrings that were separated by whitespace (whitespace can be a space, tab, or new line)

| test = "How much wood would a woodchuck chuck"  print(test.split()) # prints ['How', 'much', 'wood', 'would', 'a', 'woodchuck', 'chuck'] |
| --- |

string.split(delimiter) Returns a list of substrings that were separated by whitespace or another string

| test = "How-much-wood-would-a-woodchuck-chuck"  print(test.split("-")) # prints ['How', 'much', 'wood', 'would', 'a', 'woodchuck', 'chuck'] |
| --- |

string.split(old, new) Returns a new string where all occurrences of old have been replaced by new.

| test = "How much wood would a woodchuck chuck"  print(test.replace("wood", "plastic"))# prints "How much plastic would a plasticchuck chuck" |
| --- |

delimiter.join(list of strings) Returns a new string with all the strings joined by the delimiter

| test = "How much wood would a woodchuck chuck"  print("-".join(test.split())) # prints "How-much-wood-would-a-woodchuck-chuck" |
| --- |

#### String methods

The String methods page in the Python documentation has a more complete list of the available string methods.

<https://docs.python.org/3/library/stdtypes.html#string-methods>

### Formatting strings reference guide

I made a markdown document that is on my github:

[https://github.com/GaJoDev/Python/blob/main/format\_string\_ref\_guide.md#formatting-strings-reference-guide](https://github.com/GaJoDev/Python/blob/main/learning_resources/links/format_string_ref_guide.md)

Most programs eventually need to provide some kind of output or feedback to the user. Formatting the output makes it easier to read.

For example, imagine you are working in a farmer’s market and need to generate receipts for your customers. You need to weigh the items, calculate the total price for each item (weight times the price per pound), and then add sales tax to the subtotal. Your first attempt might look like this:

| # Here are the items in the customer's basket. Each item is a tuple # of (item name, weight, price per pound). # basket = [  ("Peaches", 3.0, 2.99),  ("Pears", 5.0, 1.66),  ("Plums", 2.5, 3.99) ]   # Calculate the total price for each item (weight times price per pound) # and add them up to get a subtotal. # subtotal = 0.00 for item in basket:  fruit, weight, unit\_price = item  subtotal += (weight \* unit\_price)   # Now calculate the sales tax and total bill. # tax\_rate = 0.06625 # 6.625% sales tax in New Jersey tax\_amt = subtotal \* tax\_rate total = subtotal + tax\_amt   # Print the receipt for the customer. # print("Subtotal:", subtotal) print("Sales Tax:", tax\_amt) print("Total:", total) |
| --- |

If you run the above code, you’ll notice the output looks a bit messy:

Subtotal: 27.245

Sales Tax: 1.8049812500000002

Total: 29.049981250000002

We’d much prefer the output to look like a real register receipt:

Subtotal: $27.25

Sales Tax: $ 1.80

Total: $29.05

The way to do this in Python is by formatting strings in your output.

Python offers different ways to format strings. In the Formatting Strings video , we explained the format() method. In this reading, we'll highlight three different ways of formatting strings. For this course you need to know only the format() method. But on the internet, you might find any of the three, so it's a good idea to know that the others exist.

#### Using the format() method

The format() method takes a string containing special placeholders, called the format string, and replaces the special placeholder characters {} with the value of the arguments you pass. The arguments are converted to strings if they weren’t already. The number of arguments you pass must exactly match the number of placeholders in the format string:

fruit = "peaches"

weight = 3.0

per\_pound = 2.99

output = "You are buying {} pounds of {} at {} per pound.".format(weight, fruit, per\_pound)

print(output)

You are buying 3.0 pounds of peaches at 2.99 per pound.

You can also consume the arguments to format() in any order you want by specifying the index inside the placeholder. As with lists and arrays, the index always starts with 0. You can even use an index more than once. Here you can see we’re using the second argument twice.

output = "{1} are {2} per pound, and you have {0} pounds of {1}.".format(weight, fruit, per\_pound)

print(output)

Peaches are 2.99 per pound, and you have 3.0 pounds of peaches.

A third option for the placeholders is to use field names instead of indexes. This can make your code much more readable.

output = "{fruit} are {price} per pound, and you have {weight} pounds of {fruit}.".format(weight=weight, fruit=fruit, price=per\_pound)

print(output)

Peaches are 2.99 per pound, and you have 3.0 pounds of peaches.

Subtotal: $ 27.25 Sales Tax: $ 1.80 Total: $ 29.05

Everything inside the placeholder after the “:” colon is part of the formatting expression. The expression “:10,.2f” means “make the output 10 characters wide, use digit separators if the amount is over 1000, output no more than 2 digits after the decimal, and expect the input to be a floating-point decimal number”.

The following table gives you some more examples of formatting expressions:

#### Formatting expressions

### 

| Expression | Meaning | Example |
| --- | --- | --- |
| {:d} | integer value | "{0:.0f}".format(10.5) → '10' |
| {:.2f} | floating point with that many decimals | '{:.2f}'.format(0.5) → '0.50' |
| {:.2s} | string with that many characters | '{:.2s}'.format('Python') → 'Py' |
| {:<6s} | string aligned to the left that many spaces | '{:<6s}'.format('Py') → 'Py ' |
| {:>6s} | string aligned to the right that many spaces | '{:>6s}'.format('Py') → ' Py' |
| {:^6s} | string centered in that many spaces | '{:^6s}'.format('Py') → ' Py ' |

### Official Documentation: [Common string operations — Python 3.13.1 documentation](https://docs.python.org/3/library/string.html#format-specification-mini-language)

#### Formatted string literals (Optional)

The formatted string literal feature, added in Python 3.6, isn’t used a lot yet. Again, it's included here in case you run into it in the future, but it's not needed for this or any upcoming courses.

A formatted string literal or f-string is a string that starts with 'f' or 'F' before the quotes. These strings might contain {} placeholders using expressions like the ones used for format() method strings.

The important difference between f-strings and the format() method is that f-strings take the value of the variables from the current context, instead of taking the values from parameters.

Examples:

>>> name = "Micah"

>>> print(f'Hello {name}')

Hello Micah

>>> item = "Purple Cup"

>>> amount = 5

>>> price = amount \* 3.25

>>> print(f'Item: {item} - Amount: {amount} - Price: {price:.2f}')

Item: Purple Cup - Amount: 5 - Price: 16.25

Official documentation for f-strings: <https://docs.python.org/3/reference/lexical_analysis.html#f-strings>

#### Old string formatting (Optional)

The format() method was introduced in Python 2.6. Before that, the % (modulo) operator could be used to get a similar result. Although this method is no longer recommended for new code, you might come across it in someone else's code. This is what it looks like:

"base string with %s placeholder" % variable

The % (modulo) operator returns a copy of the string where the placeholders indicated by % followed by a formatting expression are replaced by the variables after the operator.

To replace more than one value, you need to supply the values as a tuple. The formatting expression must match the value type.

"base string with %d and %d placeholders" % (value1, value2)

Variables can also be replaced by name using a dictionary syntax (you’ll learn about dictionaries in an upcoming video).

print("%(var1)d %(var2)d" % {"var1":value1, "var2":value2})

The formatting expressions are mostly the same as those of the format() method.

"Item: %s - Amount: %d - Price: %.2f" % (item, amount, price)

official documentation for old string formatting <https://docs.python.org/3/library/stdtypes.html#old-string-formatting>

### 

### Study Guide: Strings

<https://github.com/GaJoDev/Python/blob/main/learning_resources/links/strings_study_guide.md>

This study guide provides a quick-reference summary of what you learned in this lesson and serves as a guide for the upcoming practice quiz. The string readings in this section are great syntax guides to help you on the Strings Practice Quiz.

In the Strings segment, you learned about the parts of a string, string indexing and slicing, creating new strings, string methods and operations, and formatting strings.

#### Knowledge

##### String Operations and Methods

* **.format()** - String method that can be used to concatenate and format strings.
  + **{:.2f}** - Within the .format() method, limits a floating point variable to 2 decimal places. The number of decimal places can be customized.
* **len(string)** - String operation that returns the length of the string.
* **string[x]** - String operation that accesses the character at index [x] of the string, where indexing starts at zero.
* **string[x:y]** - String operation that accesses a substring starting at index [x] and ending at index [y-1]. If x is omitted, its value defaults to 0. If y is omitted, the value will default to len(string).
* **string.replace(old, new)** - String method that returns a new string where all occurrences of an old substring have been replaced by a new substring.
* **string.lower()** - String method that returns a copy of the string with all lowercase characters.

##### Coding skills

###### Skill Group 1

* Use a **for** loop to iterate through each letter of a string.
* Add a character to the front of a string.
* Add a character to the end of a string.
* Use the **.lower()** string method to convert the case (uppercase/lowercase) of the letters within a string variable. *This method is often used to eliminate cases as a factor when comparing two strings. For example, all lowercase “cat” is not equal to “Cat” because “Cat” contains an uppercase letter. To be able to compare the two strings to see if they are the same word, you can use the .lower() string method to remove capitalization as a factor in the string comparison.*

| # This function accepts a given string and checks each character of  # the string to see if it is a letter or not. If the character is a # letter, that letter is added to the end of the string variable # "forwards" and to the beginning of the string variable "backwards". def mirrored\_string(my\_string):   # Two variables are initialized as string data types using empty   # quotes. The variable "forwards" will hold the "my\_string"  # minus any character that is not a letter. The "backwards"   # variable will hold the same letters as "forwards", but in   # in reverse order.  forwards = ""  backwards = ""   # The for loop iterates through each character of the "my\_string"  for character in my\_string:   # The if-statement checks if the character is not a space.  if character.isalpha():   # If True, the body of the loop adds the character to the  # to the end of "forwards" and to the front of  # "backwards".   forwards += character  backwards = character + backwards   # If False (meaning the character is not a letter), no action  # is needed. This coding approach results prevents any   # non-alphabetical characters from being written to the  # "forwards" and "backwards" variables. The for loop will   # restart until all characters in "my\_string" have been  # processed.    # The final if-statement compares the "forwards" and "backwards"  # strings to see if the letters are the same both forwards and  # backwards. Since Python is case sensitive, the two strings will   # need to be converted to use the same case for this comparison.   if forwards.lower() == backwards.lower():  return True  return False   print(mirrored\_string("12 Noon")) # Should be True print(mirrored\_string("Was it a car or cat I saw")) # Should be False print(mirrored\_string("'eve, Madam Eve")) # Should be True |
| --- |

###### Skill Group 2

* Use the **format()** method, with **{}** placeholders for variable data, to create a new string.
* Use a formatting expression, like **{:.2f}**, to format a float variable and configure the number of decimal places to display for the float.

# This function converts measurement equivalents. Output is formatted

# as, "x ounces equals y pounds", with y limited to 2 decimal places.

def convert\_weight(ounces):

# Conversion formula: 1 pound = 16 ounces

pounds = ounces/16

# The result is composed using the .format() method. There are two

# placeholders in the string: the first is for the "ounces"

# variable and the second is for the "pounds" variable. The second

# placeholder formats the float result of the conversion

# calculation to be limited to 2 decimal places.

result = "{} ounces equals {:.2f} pounds".format(ounces,pounds)

return result

print(convert\_weight(12)) # Should be: 12 ounces equals 0.75 pounds

print(convert\_weight(50.5)) # Should be: 50.5 ounces equals 3.16 pounds

print(convert\_weight(16)) # Should be: 16 ounces equals 1.00 pounds

###### Skill Group 3

* Within the **format()** parameters, select characters at specific index [ ] positions from a variable string.
* Use the **format()** method, with **{}** placeholders for variable data, to create a new string.

# This function generates a username using the first 3 letters of a

# user’s last name plus their birth year.

def username(last\_name, birth\_year):

# The .format() method will use the first 3 letters at index

# positions [0,1,2] of the "last\_name" variable for the first

# {} placeholder. The second {} placeholder concatenates the user’s

# "birth\_year" to that string to form a new string username.

return("{}{}".format(last\_name[0:3],birth\_year))

print(username("Ivanov", "1985"))

# Should display "Iva1985"

print(username("Rodríguez", "2000"))

# Should display "Rod2000"

print(username("Deng", "1991"))

# Should display "Den1991"

###### Skill Group 4

* Use the **.replace()** method to replace part of a string.
* Use the **len()** function to get the number of index positions in a string.
* Slice a string at a specific index position.

# This function checks a given schedule entry for an old date and, if

# found, the function replaces it with a new date.

def replace\_date(schedule, old\_date, new\_date):

# Check if the given "old\_date" appears at the end of the given

# string variable "schedule".

if schedule.endswith(old\_date):

# If True, the body of the if-block will run. The variable "p" is

# used to hold the slicing index position. The len() function

# is used to measure the length of the string "old\_date".

p = len(old\_date)

# The "new\_schedule" string holds the updated string with the

# old date replaced by the new date. The schedule[:-p] part of

# the code trims the "old\_date" substring from "schedule"

# starting at the final index position (or right-side) counting

# towards the left the same number of index positions as

# calculated from len(old\_date). Then, the code schedule[-p:]

# starts the indexing position at the slot where the first

# character of the "old\_date" used to be positioned. The

# .replace(old\_date, new\_date) code inserts the "new\_date" into

# the position where the "old\_date" used to exist.

new\_schedule = schedule[:-p] + schedule[-p:].replace(old\_date, new\_date)

# Returns the schedule with the new date.

return new\_schedule

# If the schedule does not end with the old date, then return the

# original sentence without any modifications.

return schedule

print(replace\_date("Last year’s annual report will be released in March 2023", "2023", "2024"))

# Should display "Last year’s annual report will be released in March 2024"

print(replace\_date("In April, the CEO will hold a conference", "April", "May"))

# Should display "In April, the CEO will hold a conference"

print(replace\_date("The convention is scheduled for October", "October", "June"))

# Should display "The convention is scheduled for June"

###### Python practice information

For additional Python practice, the following links will take you to several popular online interpreters and codepads:

* [Welcome to Python](https://www.python.org/shell/)
* [Online Python Interpreter](https://www.onlinegdb.com/online_python_interpreter)
* [Create a new Repl](https://repl.it/languages/python3)
* [Online Python-3 Compiler (Interpreter)](https://www.tutorialspoint.com/execute_python3_online.php)
* [Compile Python 3 Online](https://rextester.com/l/python3_online_compiler)
* [Your Python Trinket](https://trinket.io/python3)

### Practice Quiz: Strings

<https://github.com/GaJoDev/Python/blob/main/coursera_quizzes/strings_quiz.md>



Fill in the blanks to complete the is\_palindrome function. This function checks if a given string is a palindrome. A palindrome is a string that contains the same letters in the same order, whether the word is read from left to right or right to left. Examples of palindromes are words like kayak and radar, and phrases like "Never Odd or Even". The function should ignore blank spaces and capitalization when checking if the given string is a palindrome. Complete this function to return True if the passed string is a palindrome, False if not.

def is\_palindrome(input\_string):

# Two variables are initialized as string date types using empty

# quotes: "reverse\_string" to hold the "input\_string" in reverse

# order and "new\_string" to hold the "input\_string" minus the

# spaces between words, if any are found.

new\_string = ""

reverse\_string = ""

# Complete the for loop to iterate through each letter of the

# "input\_string"

for \_\_\_:

# The if-statement checks if the "letter" is not a space.

if letter != " ":

# If True, add the "letter" to the end of "new\_string" and

# to the front of "reverse\_string". If False (if a space

# is detected), no action is needed. Exit the if-block.

new\_string = \_\_\_

reverse\_string = \_\_\_

# Complete the if-statement to compare the "new\_string" to the

# "reverse\_string". Remember that Python is case-sensitive when

# creating the string comparison code.

if \_\_\_:

# If True, the "input\_string" contains a palindrome.

return True

# Otherwise, return False.

return False

print(is\_palindrome("Never Odd or Even")) # Should be True

print(is\_palindrome("abc")) # Should be False

print(is\_palindrome("kayak")) # Should be True

Solution

def is\_palindrome(input\_string):

# Two variables are initialized as string date types using empty

# quotes: "reverse\_string" to hold the "input\_string" in reverse

# order and "new\_string" to hold the "input\_string" minus the

# spaces between words, if any are found.

new\_string = ""

reverse\_string = ""

# Complete the for loop to iterate through each letter of the

# "input\_string"

for letter in input\_string:

# The if-statement checks if the "letter" is not a space.

if letter != " ":

# If True, add the "letter" to the end of "new\_string" and

# to the front of "reverse\_string". If False (if a space

# is detected), no action is needed. Exit the if-block.

new\_string += letter

reverse\_string = letter + reverse\_string

# Complete the if-statement to compare the "new\_string" to the

# "reverse\_string". Remember that Python is case-sensitive when

# creating the string comparison code.

if new\_string.lower() == reverse\_string.lower():

# If True, the "input\_string" contains a palindrome.

return True

# Otherwise, return False.

return False

print(is\_palindrome("Never Odd or Even")) # Should be True

print(is\_palindrome("abc")) # Should be False

print(is\_palindrome("kayak")) # Should be True

Using the format method, fill in the gaps in the convert\_distance function so that it returns the phrase "X miles equals Y km", with Y having only 1 decimal place. For example, convert\_distance(12) should return "12 miles equals 19.2 km".

def convert\_distance(miles):

km = miles \* 1.6

result = "{} miles equals {\_\_\_} km".\_\_\_

return result

print(convert\_distance(12)) # Should be: 12 miles equals 19.2 km

print(convert\_distance(5.5)) # Should be: 5.5 miles equals 8.8 km

print(convert\_distance(11)) # Should be: 11 miles equals 17.6 km

Solution

def convert\_distance(miles):

km = miles \* 1.6

result = "{miles} miles equals {km:.1f} km".format(miles=miles, km=km)

return result

print(convert\_distance(12)) # Should be: 12 miles equals 19.2 km

print(convert\_distance(5.5)) # Should be: 5.5 miles equals 8.8 km

print(convert\_distance(11)) # Should be: 11 miles equals 17.6 km

If we have a string variable named Weather = "Rainfall", which of the following will print the substring "Rain" or all characters before the "f"?

[ x ] print(Weather[:4])

[ ] print(Weather[4:])

[ ] print(Weather[1:4])

[ ] print(Weather[:"f"])

Fill in the gaps in the nametag function so that it uses the format method to return first\_name and the first initial of last\_name followed by a period. For example, nametag("Jane", "Smith") should return "Jane S."

def nametag(first\_name, last\_name):

return("\_\_\_.".format(\_\_\_))

print(nametag("Jane", "Smith"))

# Should display "Jane S."

print(nametag("Francesco", "Rinaldi"))

# Should display "Francesco R."

print(nametag("Jean-Luc", "Grand-Pierre"))

# Should display "Jean-Luc G."

Solution

def nametag(first\_name, last\_name):

return("{first\_name} {last\_name}.".format(first\_name=first\_name, last\_name=last\_name[:1]))

print(nametag("Jane", "Smith"))

# Should display "Jane S."

print(nametag("Francesco", "Rinaldi"))

# Should display "Francesco R."

print(nametag("Jean-Luc", "Grand-Pierre"))

# Should display "Jean-Luc G."

The replace\_ending function replaces a specified substring at the end of a given sentence with a new substring. If the specified substring does not appear at the end of the given sentence, no action is performed and the original sentence is returned. If there is more than one occurrence of the specified substring in the sentence, only the substring at the end of the sentence is replaced. For example, replace\_ending("abcabc", "abc", "xyz") should return abcxyz, not xyzxyz or xyzabc. The string comparison is case-sensitive, so replace\_ending("abcabc", "ABC", "xyz") should return abcabc (no changes made).

def replace\_ending(sentence, old, new):

# Check if the old substring is at the end of the sentence

if \_\_\_:

# Using i as the slicing index, combine the part

# of the sentence up to the matched string at the

# end with the new string

i = \_\_\_

new\_sentence = \_\_\_

return new\_sentence

# Return the original sentence if there is no match

return sentence

print(replace\_ending("It's raining cats and cats", "cats", "dogs"))

# Should display "It's raining cats and dogs"

print(replace\_ending("She sells seashells by the seashore", "seashells", "donuts"))

# Should display "She sells seashells by the seashore"

print(replace\_ending("The weather is nice in May", "may", "april"))

# Should display "The weather is nice in May"

print(replace\_ending("The weather is nice in May", "May", "April"))

# Should display "The weather is nice in April"

Solution

def replace\_ending(sentence, old, new):

# Check if the old substring is at the end of the sentence

if sentence.endswith(old):

# Using i as the slicing index, combine the part

# of the sentence up to the matched string at the

# end with the new string

i = len(old)

new\_sentence = sentence[:-i] + new

return new\_sentence

# Return the original sentence if there is no match

return sentence

print(replace\_ending("It's raining cats and cats", "cats", "dogs"))

# Should display "It's raining cats and dogs"

print(replace\_ending("She sells seashells by the seashore", "seashells", "donuts"))

# Should display "She sells seashells by the seashore"

print(replace\_ending("The weather is nice in May", "may", "april"))

# Should display "The weather is nice in May"

print(replace\_ending("The weather is nice in May", "May", "April"))

# Should display "The weather is nice in April"

## **Lists**

This reading contains the code used in the instructional videos from [**What is a list?**](https://www.coursera.org/learn/python-crash-course/lecture/DoC7j/what-is-a-list)

Introduction

This follow-along reading is organized to match the content in the video that follows. It contains the same code shown in the next video. These code blocks will provide you with the opportunity to see how the code is written, allow you to practice running it, and can be used as a reference to refer back to.

You can follow along in the reading as the instructor discusses the code or review the code after watching the video.

| x = ["Now", "we", "are", "cooking!"] ['Now', 'we', 'are', 'cooking!'] |
| --- |

| x = ["Now", "we", "are", "cooking!"] type(x)  <class 'list'> |
| --- |

| x = ["Now", "we", "are", "cooking!"] print(x) ['Now', 'we', 'are', 'cooking!'] |
| --- |

| x = ["Now", "we", "are", "cooking!"] len(x)  4 |
| --- |

| x = ["Now", "we", "are", "cooking!"] "are" in x  True |
| --- |

| x = ["Now", "we", "are", "cooking!"] "Today" in x  False |
| --- |

| x = ["Now", "we", "are", "cooking!"] print(x[0]) print(x[3])  Now cooking! |
| --- |

| x = ["Now", "we", "are", "cooking!"] print(x[4]) #This last line will throw an error  Error on line 2: print(x[4]) IndexError: list index out of range |
| --- |

| x = ["Now", "we", "are", "cooking!"]  print(x[1:3])  print(x[:2])  print(x[2:])  # Output  ['we', 'are']  ['Now', 'we']  ['are', 'cooking!'] |
| --- |

### Review: What is a list?

As you know by now, Python comes with a lot of ready-to-use data types. We've seen integers, floats, Booleans, and strings in detail. But those data types can only take you so far. Eventually in your scripts, you want to develop code that manipulates collections of items like a list of strings representing all the file names in a directory or a list of integers representing the size of network packets. This is where the list data type comes in handy. You can think of lists as long boxes with the space inside the box divided up into different slots. Each slot can contain a different value. Like we mentioned earlier when we first came across the list, in Python, we use square brackets to indicate where the list starts and ends. Let's check out an example. Here, we've created a new variable called x and set its contents to be a list of strings. We can check the type of x using the type function we saw a little while ago. Nice. Python tells us this is a list. In the same way, we've done with other variables, we can show the contents of the whole list using the print function. The length of the list is how many elements it has. To get that value, we'll use the same len function we used for strings. That's right. Our list has four elements. When calling Len for the list, it doesn't matter how long each string is on its own. What matters is how many elements the list has. To check if a list contains a certain element, you can use the keyword "in" like in these examples. Again, like when we use this with strings, the result of this check is a Boolean, which we can use as a condition for branching or looping. We can also use indexing to access individual elements depending on their position in the list. To do that, we use the square brackets and the index we want to access, exactly like we did with strings. Remember that the first element is given the index zero. This means the last index of the list will be the length of the list minus one. What happens if we try to access an element after the end of the list? You might have seen this coming. We get an index error. We can't go over the end of the list. Remember that because list indexes start at zero, accessing the item at index four means we're trying to access the fifth element in the list. There are only four elements. So we're out of range if we try to access the index number four. Does this seem a bit confusing? If it does, this visualization might help you out. As you can see, index four doesn't point at anything since there's no slot four in our list. As with strings, we can also use indexes to create a slice of the list. For this, we use ranges of two numbers separated by a colon. Again, the second element isn't included in the slice. So the range goes to the second index minus one. Here, we start at index one and go up to one less than three, which is two. We can also leave out one of the range indexes empty. The first value defaults to zero and the second value to the length of the list. Makes sense? If all this sounds really familiar to what we said about strings, then this course is working as intended. That's because strings and lists are very similar data types. In Python, strings and lists are both examples of sequences. There are other sequences too, and they all share a bunch of operations like iterating over them using for-loops, indexing using the len function to know the length of the sequence, using plus to concatenate two sequences and using in to verify if the sequence contains an element. So this is great news. While understanding indexing is hard, once you know it for one data type, you've pretty much mastered it for every data type. So you actually know way more than you thought. Wow, now, we're really cooking. Next up, we're going to look at some more list operations. This time, actually specific to lists.

### What is a list?

As you know by now, Python comes with a lot of ready-to-use data types. We've seen integers, floats, Booleans, and strings in detail. But those data types can only take you so far.

Eventually in your scripts, you want to develop code that manipulates collections of items like a list of strings representing all the file names in a directory or a list of integers representing the size of network packets.

This is where the list data type comes in handy. You can think of lists as long boxes with the space inside the box divided up into different slots. Each slot can contain a different value.



Like we mentioned earlier when we first came across the list, **in Python, we use square brackets to indicate where the list starts and ends**. Let's check out an example.

Here, we've created a new variable called x and set its contents to be a list of strings. We can check the type of x using the type function we saw a little while ago.

| x = ["Now", "we", "are", "cooking!"] type(x)  <class 'list'> |
| --- |

Nice. Python tells us this is a list.

In the same way, we've done with other variables, we can show the contents of the whole list using the print function.

| x = ["Now", "we", "are", "cooking!"] print(x) ['Now', 'we', 'are', 'cooking!'] |
| --- |

The length of the list is how many elements it has. To get that value, we'll use the same len function we used for strings.

| x = ["Now", "we", "are", "cooking!"] len(x)  4 |
| --- |

That's right. Our list has four elements.

When calling Len for the list, it doesn't matter how long each string is on its own. What matters is how many elements the list has. To check if a list contains a certain element, you can use the keyword "in" like in these examples.

| x = ["Now", "we", "are", "cooking!"] "are" in x  True |
| --- |

| x = ["Now", "we", "are", "cooking!"] "Today" in x  False |
| --- |

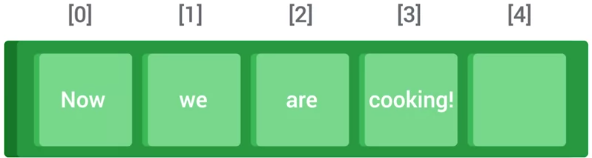
Again, like when we use this with strings, the result of this check is a Boolean, which we can use as a condition for branching or looping. We can also use indexing to access individual elements depending on their position in the list. To do that, we use the square brackets and the index we want to access, exactly like we did with strings.

| x = ["Now", "we", "are", "cooking!"] print(x[0]) print(x[3])  Now cooking! |
| --- |

Remember that the first element is given the index zero. This means the last index of the list will be the length of the list minus one. What happens if we try to access an element after the end of the list?

| x = ["Now", "we", "are", "cooking!"] print(x[4]) #This last line will throw an error  Error on line 2: print(x[4]) IndexError: list index out of range |
| --- |

You might have seen this coming. We get an index error. We can't go over the end of the list. Remember that because list indexes start at zero, accessing the item at index four means we're trying to access the fifth element in the list. There are only four elements. So we're out of range if we try to access the index number four. Does this seem a bit confusing? If it does, this visualization might help you out.



As you can see, index four doesn't point at anything since there's no slot four in our list.

As with strings, we can also use indexes to create a slice of the list. For this, we use ranges of two numbers separated by a colon. Again, the second element isn't included in the slice. So the range goes to the second index minus one.

Here, we start at index one and go up to one less than three, which is two. We can also leave out one of the range indexes empty. The first value defaults to zero and the second value to the length of the list. Does it make sense?

| x = ["Now", "we", "are", "cooking!"]  print(x[1:3])  print(x[:2])  print(x[2:])  # Output  # ['we', 'are']  # ['Now', 'we']  # ['are', 'cooking!'] |
| --- |

If all this sounds really familiar to what we said about strings, then this course is working as intended.

**Reflect**

*Using the "split" string method from the preceding lesson, complete the get\_word function to return the {n}th word from a passed sentence. For this example, run the print statements one at a time. Delete the other 3 print statements to do this. For example, get\_word("This is a lesson about lists", 4) should return "lesson", which is the 4th word in this sentence. Hint: remember that list indexes start at 0, not 1.*

def get\_word(sentence, n):

# Only proceed if n is positive

if n > 0:

words = \_\_\_

# Only proceed if n is not more than the number of words

if n <= len(words):

return(\_\_\_)

return("")

print(get\_word("This is a lesson about lists", 4)) # Should print: lesson

print(get\_word("This is a lesson about lists", -4)) # Nothing

print(get\_word("Now we are cooking!", 1)) # Should print: Now

print(get\_word("Now we are cooking!", 5)) # Nothing

**Solution**

def get\_word(sentence, n):

# Only proceed if n is positive

if n > 0:

words = sentence.split() # Split the sentence into a list of words

# Only proceed if n is not more than the number of words

if n <= len(words):

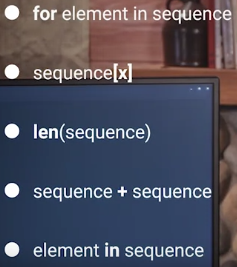
return (words[n-1]) # Return the nth word (n-1 for zero-based index)

return "" # Return an empty string if n is not valid

***GJ NOTE***

***The above really threw me, primarily because I was trying to do everything within the outer if, what was also very confusing is that the nested return statement*** return(\_\_\_)***did not make it clear at all because of the brackets. Primarily because of my lack of understanding of control flow and not breaking down the question correctly***

That's because strings and lists are very similar data types. In Python, strings and lists are both examples of sequences. There are other sequences too, and they all share a bunch of operations like iterating over them using for-loops, indexing using the len function to know the length of the sequence, using plus to concatenate two sequences and using in to verify if the sequence contains an element.



So this is great news. While understanding indexing is hard, once you know it for one data type, you've pretty much mastered it for every data type. So you actually know way more than you thought. Next up, we're going to look at some more list operations. This time, actually specific to lists.

### Lists Defined

Lists in Python are defined using square brackets, with the elements stored in the list separated by commas:

| my\_list = ["This", "is", "a", "list"] |
| --- |

You can use the **len()** function to return the number of elements in a list: **len(my\_list)** would return **4**. You can also use the **in** keyword to check if a list contains a certain element. If the element is present, it will return a True boolean. If the element is not found in the list, it will return False. For example, **"This" in my\_list** would return True in our example.

Similar to strings, lists can also use indexing to access specific elements in a list based on their position.

You can access the first element in a list by doing **my\_list[0]**, which would allow you to access the string **"This"**.

In Python, lists and strings are quite similar. They’re both examples of sequences of data. Sequences have similar properties, like:

(1) being able to iterate over them using **for loops**

(2) support indexing

(3) using the **len()** function to find the length of the sequence

(4) using the plus operator **+** in order to concatenate

(5) using the **in** keyword to check if the sequence contains a value.

Understanding these concepts allows you to apply them to other sequence types as well.

### Review: Modifying the contents of a list

This reading contains the code used in the instructional videos from [**Modifying the contents of a list**](https://www.coursera.org/learn/python-crash-course/supplement/v07vB/modifying-lists)

Introduction

This follow-along reading is organized to match the content in the video that follows. It contains the same code shown in the next video. These code blocks will provide you with the opportunity to see how the code is written, allow you to practice running it, and can be used as a reference to refer back to.

You can follow along in the reading as the instructor discusses the code or review the code after watching the video

|  |
| --- |

|  |
| --- |

| fruits = ["Pineapple", "Banana", "Apple", "Melon"]  fruits.insert(0, "Orange")  fruits.insert(25, "Peach")  print(fruits)  ['Orange', 'Pineapple', 'Banana', 'Apple', 'Melon', 'Peach'] |
| --- |

fruits = ["Pineapple", "Banana", "Apple", "Melon"]

fruits.insert(0, "Orange")

fruits.insert(25, "Peach")

fruits.remove("Melon")

print(fruits)

['Orange', 'Pineapple', 'Banana', 'Apple', 'Peach']

fruits = ["Pineapple", "Banana", "Apple", "Melon"]

fruits.remove("Pear")

#This will throw an error

Error on line 2:

fruits.remove("Pear")

ValueError: list.remove(x): x not in list

fruits = ["Pineapple", "Banana", "Apple", "Melon"]

fruits.insert(0, "Orange")

fruits.insert(25, "Peach")

fruits.remove("Melon")

fruits.pop(3)

print(fruits)

['Orange', 'Pineapple', 'Banana', 'Peach']

fruits = ["Pineapple", "Banana", "Apple", "Melon"]

fruits.insert(0, "Orange")

fruits.insert(25, "Peach")

fruits.remove("Melon")

fruits.pop(3)

fruits[2] = "Strawberry"

print(fruits)

['Orange', 'Pineapple', 'Strawberry', 'Peach']

### Modifying the Contents of a List

One of the ways that lists and strings are different is that lists are **mutable, meaning that they can change**. This means we can add, remove, or modify elements in a list.

Let's go back to our example of thinking of a list as a long box. Changing the list means we keep the same box and we add, remove, or change the elements inside that box.



We'll now go through the methods that let us modify the list one by one. If all these details seem a little overwhelming, there will be a cheat sheet at the end and you'll have lots of chances to practice each of these methods as we go along.

We'll start with the simplest change; adding an element to a list using the append method. Let's check this out in the tastiest example yet.

| fruits = ["Pineapple", "Banana", "Apple", "Melon"]  fruits.append("Kiwi")  print(fruits) # Output: ['Pineapple', 'Banana', 'Apple', 'Melon', 'Kiwi'] |
| --- |

The append method adds a new element at the end of the list. It doesn't matter how long the list is. The element always gets added to the end. You could start with an empty list and add all of its items using append.

If you want to insert an element in a different position, instead of at the end, you can use the insert method.

| fruits.insert(0, "Orange")  print(fruits) # Output: ['Orange', 'Pineapple', 'Banana', 'Apple', 'Melon', 'Kiwi'] |
| --- |

The insert method takes an index as the first parameter and an element as the second parameter. It adds the element at that index in the list. To add it as the first element, we use index zero and we can use any other number. What happens if we use a number larger than the length of the list?

| fruits.insert(25, "Peach")  print(fruits) # Output: ['Orange', 'Pineapple', 'Banana', 'Apple', 'Melon', 'Kiwi', 'Peach'] |
| --- |

No errors.

If we use an index higher than the current length, the element just gets added to the end. You can pass any number to insert but usually, you either add at the beginning using insert at the zero index or at the end using append. We can also remove elements from the list. We can do it using the value of the element we want to remove. Can you guess what method we would use? You got it, use the remove method.

| fruits.remove("Melon")  print(fruits) # Output: ['Orange', 'Pineapple', 'Banana', 'Apple', 'Kiwi', 'Peach'] |
| --- |

The remove method removes from the list the first occurrence of the element we pass to it. What happens if the element is not in the list?

| fruits.remove("Pear")  print(fruits)  # Output: fruits.remove("Pear") ValueError: list.remove(x): x not in list |
| --- |

We got a value error, telling us the element isn't in the list.

Another way we can remove elements is by using the pop method, which receives an index.

| fruits.pop(3)  print(fruits) # Output: ['Orange', 'Pineapple', 'Banana', 'Kiwi', 'Peach'] |
| --- |

The pop method returns the element that was removed at the index that was passed.

The last way to modify the contents of a list is to change an item by assigning something else to that position, like this.

| fruits[2] = "Strawberry"  print(fruits) # Output: ['Orange', 'Pineapple', 'Strawberry', 'Kiwi', 'Peach'] |
| --- |

Wow, the contents of our fruits variable have changed a lot since we started this video. But it's always the same variable, the same box. We've just modified what's inside. Modifying the contents of lists will come up in tons of scripts as we operate with them. If the list contains hosts on a network, you could add or remove hosts as they come online or offline. If the list contains users authorized to run a certain process, you could add or remove users when permissions are granted or removed and so on.

You've now seen a number of methods that let us modify the contents of a list, adding, removing, and changing the elements that are stored inside the list. Whenever you need to write a program that'll handle a variable amount of elements, you'll use a list. What if you need a sequence of a fixed amount of elements? That's coming up in our next video.

### Review: Lists and tuples

Introduction

This follow-along reading is organized to match the content in the video that follows. It contains the same code shown in the next video. These code blocks will provide you with the opportunity to see how the code is written, allow you to practice running it, and can be used as a reference to refer back to.

def convert\_seconds(seconds):

hours = seconds // 3600

minutes = (seconds - hours \* 3600) // 60

remaining\_seconds = seconds - hours \* 3600 - minutes \* 60

return hours, minutes, remaining\_seconds

result = convert\_seconds(5000)

print(type(result))

def convert\_seconds(seconds):

hours = seconds // 3600

minutes = (seconds - hours \* 3600) // 60

remaining\_seconds = seconds - hours \* 3600 - minutes \* 60

return hours, minutes, remaining\_seconds

result = convert\_seconds(5000)

print(result)

def convert\_seconds(seconds):

hours = seconds // 3600

minutes = (seconds - hours \* 3600) // 60

remaining\_seconds = seconds - hours \* 3600 - minutes \* 60

return hours, minutes, remaining\_seconds

result = convert\_seconds(5000)

hours, minutes, seconds = result

print(hours, minutes, seconds)

def convert\_seconds(seconds):

hours = seconds // 3600

minutes = (seconds - hours \* 3600) // 60

remaining\_seconds = seconds - hours \* 3600 - minutes \* 60

return hours, minutes, remaining\_seconds

hours, minutes, seconds = convert\_seconds(1000)

print(hours, minutes, seconds)

### Lists and Tuples

As we called out before, there are a number of data types in Python that are all sequences.

**Strings** are sequences of characters and are immutable.

**Lists** are sequences of elements of any type and are mutable.

A third data type that's a sequence and also closely related to lists is the tuple.

**Tuples** are sequences of elements of any type that are immutable. We write tuples in parentheses instead of square brackets.

| fullname = ('Grace', 'M', 'Hopper') |
| --- |

You might be wondering, why do we even need another sequence type? Yes, lists are great. They can hold any number of elements and we can add, remove and modify their contents as much as we want, but there are cases when we want to make sure an element in a certain position or index refers to one specific thing and won't change. In these situations, lists won't help us.

In our example, we have a tuple that represents someone's full name. The first element of the tuple is the first-name. The second element is the middle initial, and the third element is the last-name. If we add another element somewhere in there, what would that element represent? It would just be confusing and our code wouldn't know what to do with it, and that's why modifying isn't allowed.

In other words, when using tuples the position of the elements inside the tuple have meaning.

Tuples are used for lots of different things in Python. One common example is the return value of functions. When a function returns more than one value, it's actually returning a tuple.

Remember the function to convert seconds to hours, minutes, and seconds that we saw a while back?

This function returns three values. In other words, it returns a tuple of three elements. Let's give it a try.

| def convert\_seconds(seconds):  hours = seconds // 3600  minutes = (seconds - hours \* 3600) // 60  remaining\_seconds = seconds - hours \* 3600 - minutes \* 60  return hours, minutes, remaining\_seconds  result = convert\_seconds(5000)  print(type(result)) # Output: <class 'tuple'>  print(result) # Output: (1, 23, 20) |
| --- |

We see the result is a tuple. What if we print it? We see that it has the three elements we expect it to have. Remember, since this is a tuple, the order matters. The first element represents the hours, the second one represents the minutes, and the third represents the seconds.

One interesting thing we can do with tuples is unpack them. This means that we can turn a tuple of three elements into three separate variables. Because the order won't change, we know what those variables are present, like this.

| def convert\_seconds(seconds):  hours = seconds // 3600  minutes = (seconds - hours \* 3600) // 60  remaining\_seconds = seconds - hours \* 3600 - minutes \* 60  return hours, minutes, remaining\_seconds  result = convert\_seconds(5000)  hours, minutes, seconds = result  print(hours, minutes, seconds) # Output : 1 23 20 |
| --- |

So now we've split the tuple into three separate values. We've seen before that we can also do this directly when calling the function without the intermediate result variable.

| def convert\_seconds(seconds):  hours = seconds // 3600  minutes = (seconds - hours \* 3600) // 60  remaining\_seconds = seconds - hours \* 3600 - minutes \* 60  return hours, minutes, remaining\_seconds  hours, minutes, seconds = convert\_seconds(1000)  print(hours, minutes, seconds) # Output: 0 16 40 |
| --- |

In Python, it's really common to use tuples to represent data that has more than one value and that needs to be kept together.

For example, you could use a tuple to have a filename and it's size, or you could store the name and email address of a person, or a date and time and the general health of the system at any point in time.

Can you see how these different data types could help you automate some of your IT work.

Knowing when to use tuples and when to use lists can seem a little fuzzy at first, but don't worry, it'll get clearer as we tackle more examples.

Reflect

*Let's use tuples to store information about a file: its name, its type and its size in bytes. Fill in the gaps in this code to return the size in kilobytes (a kilobyte is 1024 bytes) up to 2 decimal places.*

| def file\_size(file\_info):  \_\_\_, \_\_\_, \_\_\_= file\_info  return("{:.2f}".format(\_\_\_ / 1024))  print(file\_size(('Class Assignment', 'docx', 17875))) # Should print 17.46  print(file\_size(('Notes', 'txt', 496))) # Should print 0.48  print(file\_size(('Program', 'py', 1239))) # Should print 1.21 |
| --- |

Solution

| def file\_size(file\_info):  name, type, size = file\_info  return("{:.2f}".format(size / 1024))  print(file\_size(('Class Assignment', 'docx', 17875))) # Should print 17.46  print(file\_size(('Notes', 'txt', 496))) # Should print 0.48  print(file\_size(('Program', 'py', 1239))) # Should print 1.21  # Output:  # 17.46  # 0.48  # 1.21 |
| --- |

### Tuples

As we mentioned earlier, strings and lists are both examples of sequences.

Strings are sequences of characters, and are immutable.

| my\_string = "This is a String" |
| --- |

Lists are sequences of elements of any data type, and are mutable

| my\_list = [1, "hello", 3.14, [True, False]] |
| --- |

The third sequence type is the tuple. Tuples are like lists, since they can contain elements of any data type. But unlike lists, tuples are immutable. They’re specified using parentheses instead of square brackets.

| my\_tuple = (1, "hello", 3.14, (True, False)) |
| --- |

You might be wondering why tuples are a thing, given how similar they are to lists. Tuples can be useful when we need to ensure that an element is in a certain position and will not change. Since lists are mutable, the order of the elements can be changed on us.

Since the order of the elements in a tuple can't be changed, the position of the element in a tuple can have meaning. A good example of this is when a function returns multiple values. In this case, what gets returned is a tuple, with the return values as elements in the tuple.

The order of the returned values is important, and a tuple ensures that the order isn’t going to change. Storing the elements of a tuple in separate variables is called unpacking. This allows you to take multiple return values from a function and store each value in its own variable.

### Review: Iterating over lists and tuples

This reading contains the code used in the instructional videos from [**Iterating over lists and tuples**](https://www.coursera.org/learn/python-crash-course/lecture/M7Dbr/iterating-over-lists-and-tuples)

Introduction

This follow-along reading is organized to match the content in the video that follows. It contains the same code shown in the next video. These code blocks will provide you with the opportunity to see how the code is written, allow you to practice running it, and can be used as a reference to refer back to.

You can follow along in the reading as the instructor discusses the code or review the code after watching the video.

| animals = ["Lion", "Zebra", "Dolphin", "Monkey"]  chars = 0  for animal in animals:  chars += len(animal)  print("Total characters: {}, Average length: {}".format(chars, chars/len(animals)))  # Output Total characters: 22, Average length: 5.5 |
| --- |

| winners = ["Ashley", "Dylan", "Reese"]  for index, person in enumerate(winners):  print("{} - {}".format(index + 1, person))  # Output  # 1 - Ashley  # 2 - Dylan  # 3 - Reese |
| --- |

| def full\_emails(people):  result = []  for email, name in people:  result.append("{} <{}>".format(name, email))  return result  print(full\_emails([("alex@example.com", "Alex Diego"), ("shay@example.com", "Shay Brandt")]))  # Output ['Alex Diego <alex@example.com>', 'Shay Brandt <shay@example.com>'] |
| --- |

### Iterating over Lists and Tuples

When we looked at for loops, we said they iterate over a sequence of elements. One of the examples we checked out was iterating over a list.

| animals = ["Lion", "Zebra", "Dolphin", "Monkey"]  chars = 0  for animal in animals:  chars += len(animal)  print("Total characters: {}, Average length: {}".format(chars, chars/len(animals)))  # Output: Total characters: 22, Average length: 5.5 |
| --- |

In this code, we're iterating over a list of strings. For each of the strings, we get its length and add it to the total amount of characters. At the end we print the total and the average which we get by dividing the total by the length of the list. You can see we're using the len function twice, once to get the length of the string chars += len(animal) and then again to get the amount of elements in the list chars += len(animal).

What if you wanted to know the index of an element while going through the list?

* You could use a range function and then use indexing to access the elements at the index that range just returned.
* Or you could just use the enumerate function.

| winners = ["Ashley", "Dylan", "Reese"]  for index, person in enumerate(winners):  print("{} - {}".format(index + 1, person))  # Output:  #1 - Ashley  #2 - Dylan  #3 - Reese |
| --- |

The enumerate function returns a tuple for each element in the list. The first value in the tuple is the index of the element in the sequence. The second value in the tuple is the element in the sequence person.

Let's use all of this now to solve a slightly more interesting problem. Say you have a list of tuples containing two strings each. The first string is an email address and the second is the full name of the person with that email address. You want to write a function that creates a new list containing one string per person including their name and the email address between angled brackets. the format usually used in emails like this.



We'll start by defining a function that receives a list of people.

| def full\_emails(people):  result = []  for email, name in people:  result.append("{} <{}>".format(name, email))  return result  print(full\_emails([("alex@example.com", "Alex Diego"), ("shay@example.com", "Shay Brandt")]))  # Output: ['Alex Diego <alex@example.com>', 'Shay Brandt <shay@example.com>'] |
| --- |

Remember, people is a list of tuples where the first element is the email address and the second one is the full name. So in our function, we'll first create the variable that we'll use as a return value which will be a list and we'll call it result = [].

We'll then iterate over the list of people. We know this list contains tuples of two strings each. So we'll unpack the values directly when iterating in variables for email, name in people:.

Now, our result variable is a list and it should contain strings. So we'll append the resulting string to the results list result.append.

The string that will append will be formatted in the way we want. To do that, we'll use the format method with the two variables of our iteration ("{} <{}>".format(name, email)).

Once we're done with the iteration, we'll return the list which should now contain all the necessary emails return result.

Will this work? What do you think? Let's try it out print(full\_emails([("alex@example.com", "Alex Diego"), ("shay@example.com", "Shay Brandt")])).

This worked as expected # Output: ['Alex Diego <alex@example.com>', 'Shay Brandt <shay@example.com>'].

Before we move on, a quick word of **caution** about some common errors when dealing with lists in Python.

Because we use the range function so much with for loops, you might be tempted to use it for iterating over indexes of a list and then to access the elements through indexing. You could be particularly inclined to do this if you're used to other programming languages. Because in some languages, the only way to access an element of a list is by using indexes.

This works but looks ugly. It's more idiomatic in Python to iterate through the elements of the list directly or using enumerate when you need the indexes like we've done so far. There are some specific cases that do require us to iterate over the indexes, for example, when we're trying to modify the elements of the list we're iterating. By the way, if you're iterating through a list and you want to modify it at the same time, you need to be very careful. If you remove elements from the list while iterating, you're likely to end up with an unexpected result. In this case, it might be better to use a copy of the list instead. We've now seen a bunch of different things we can do with lists, and hopefully you're starting to see how they can be a very powerful tool in your IT specialist toolkit. Next up, we're going to learn a powerful technique for creating lists.

**Reflect**

Try out the enumerate function for yourself in this quick exercise. Complete the skip\_elements function to return every other element from the list, **this time using the enumerate function** to check if an element is in an even position or an odd position.

| def skip\_elements(elements):  # code goes here    return \_\_\_  print(skip\_elements(["a", "b", "c", "d", "e", "f", "g"]))  # Should be ['a', 'c', 'e', 'g']  print(skip\_elements(['Orange', 'Pineapple', 'Strawberry', 'Kiwi', 'Peach'])) # Should be ['Orange', 'Strawberry', 'Peach'] |
| --- |

### Solution

| def skip\_elements(elements):  new\_list = []  for index, element in enumerate(elements):  if index % 2 == 0:  new\_list.append(element)  return new\_list  print(skip\_elements(["a", "b", "c", "d", "e", "f", "g"]))  # Should be ['a', 'c', 'e', 'g']  print(skip\_elements(['Orange', 'Pineapple', 'Strawberry', 'Kiwi', 'Peach']))  # Should be ['Orange', 'Strawberry', 'Peach']  # Output  # ['a', 'c', 'e', 'g']  # ['Orange', 'Strawberry', 'Peach'] |
| --- |

### Iterating Over Lists Using Enumerate

When we covered *for* loops, we showed the example of iterating over a list. This lets you iterate over each element in the list, exposing the element to the for loop as a variable. But what if you want to access the elements in a list, along with the index of the element in question? You can do this using the **enumerate()** function. The enumerate() function takes a list as a parameter and returns a tuple for each element in the list. The first value of the tuple is the index and the second value is the element itself.

### Review: List Comprehensions

This reading contains the code used in the instructional videos from [**List comprehensions**](https://www.coursera.org/learn/python-crash-course/lecture/J9gYw/list-comprehensions)

**Introduction**

This follow-along reading is organized to match the content in the video that follows. It contains the same code shown in the next video. These code blocks will provide you with the opportunity to see how the code is written, allow you to practice running it, and can be used as a reference to refer back to.

You can follow along in the reading as the instructor discusses the code or review the code after watching the video.

| multiples = []  for x in range(1,11):  multiples.append(x\*7)  print(multiples) |
| --- |

| multiples = [x\*7 for x in range(1,11)]  print(multiples) |
| --- |

| languages = ["Python", "Perl", "Ruby", "Go", "Java", "C"]  lengths = [len(language) for language in languages]  print(lengths) |
| --- |

| z = [x for x in range(0,101) if x % 3 == 0]  print(z) |
| --- |

### List Comprehensions

We're almost done with our deep dive into Python lists. But before we continue to our next data structure, let's talk about creating lists in a shorter way. Say we wanted to create a list with multiples of seven from seven to 70. We could do it like this.

| multiples = []  for x in range(1,11):  multiples.append(x\*7)  print(multiples)  # Output [7, 14, 21, 28, 35, 42, 49, 56, 63, 70] |
| --- |

This works fine and is a good way of solving it but because creating lists based on sequences is such a common task, Python provides a technique called list comprehension that lets us do it in just one line. This is how we would do the same with list comprehension.

| multiples = [x\*7 for x in range(1,11)]  print(multiples)  # Output [7, 14, 21, 28, 35, 42, 49, 56, 63, 70] |
| --- |

**List comprehensions let us create new lists based on sequences or ranges.**

So we can use this technique whenever we wanna create a list based on a range, like in this example, or based on the contents of a list, a tuple, a string, or any other Python sequence.

The syntax tries to copy how you would express these concepts with natural language, although it can still be confusing sometimes. Let's check out a different example. Say we have a list of strings with the names of programming languages and we want to generate a list of the lengths of the strings. We could iterate over the list and add them using append like we did before or we could use a list comprehension like this.

| languages = ["Python", "Perl", "Ruby", "Go", "Java", "C"]  lengths = [len(language) for language in languages]  print(lengths)  # Output[6, 4, 4, 2, 4, 1] |
| --- |

List comprehensions also let us use a conditional clause. Say we wanted all the numbers that are divisible by three between zero and 50. We could create a list like this.

| z = [x for x in range(0,51) if x % 3 == 0]  print(z)  # Output [0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48] |
| --- |

In this case, we just want the element x to be a part of the list, but we only want the numbers where the remainder of the division by three is zero. So we add the conditional clause after the range.

Using list comprehensions when programming in Python is totally optional. Sometimes it can make the code look nicer and more readable.

**Reflect**

The odd\_numbers function returns a list of odd numbers between 1 and n, inclusively. Fill in the blanks in the function, using list comprehension. Hint: remember that list and range counters start at 0 and end at the limit minus 1.

| def odd\_numbers(n):  return [x for x in \_\_\_ if \_\_\_]  print(odd\_numbers(5)) # Should print [1, 3, 5]  print(odd\_numbers(10)) # Should print [1, 3, 5, 7, 9] |
| --- |

**Solution**

| def odd\_numbers(n):  return [x for x in range(n+1) if x % 2 != 0]  print(odd\_numbers(5)) # Should print [1, 3, 5]  print(odd\_numbers(10)) # Should print [1, 3, 5, 7, 9]  print(odd\_numbers(11)) # Should print [1, 3, 5, 7, 9, 11]  print(odd\_numbers(1)) # Should print [1]  print(odd\_numbers(-1)) # Should print [] |
| --- |

At other times, it can have the opposite effect, especially if we try to pack too much information together. In general, it's a good idea to know that list comprehensions exist, especially when you're trying to understand someone else's code.

All right, we've now seen a bunch of different methods we can use to operate with lists and tuples, and Python provides even more of them that we didn't get to talk about. In our next reading, you'll find the list of the most common operations and links to the official documentation in case you want to learn more.List comprehension vs for loops

### List comprehension vs for loops

The first time we talked about list comprehensions, we showed you how to create a list using a for loop. Then we showed you how to create that same list using a list comprehension.

You are probably wondering why there are two ways to do the same thing. So how do you know which one to use?

In this video, we're going to talk about when to use a for loop and when to use a list comprehension.

Firstly, a quick review. We said:

For loop iterates, or repeats over a sequence of values like numbers, letters, and so on.

A list comprehension lets us create new lists based on sequences or ranges.

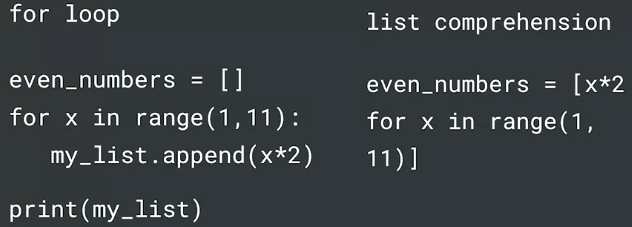
It's basically a shorthand way to build a new list based on the contents of an existing list.

The list comprehension has the for loop, the compilation, and builds the list all in a single line of code.

List comprehensions and for loops both do the same thing, so when should you use which one?

Both methods are essentially interchangeable so there are no standards or best practices. Deciding when to use a list comprehension instead of a for loop or vice versa is totally up to you. That said, there are considerations that many programmers make when deciding whether to use a list comprehension or a for loop.

Programmers tend to use a list comprehension when it's fairly simple code, such as filtering a list to pull out items that match or capitalizing words in a list. If it can fit on a single line, I'd probably make it a list comprehension which you can see in this example.



For more complex code, it's usually a good idea to use a for loop.

A for loop spells out the process better which will make it easier for you and other programmers to read and maintain the code you write.

For example, let's say you're working for a company that maintains a list of invoices. You need to calculate the sales tax and balance due for all the items on the list. You wanna start with updating the status of past due invoices.

You could accomplish this with a list comprehension but it would be complicated and likely confusing to read and understand. Let's go over an example of what the list comprehension would look like for updating the status of past due invoices.

In this code, we create a list called Past Due that contains all of the invoices in the invoices list that are more than 30 days old and have not yet been paid



**‘i’** is used in the list comprehension to iterate over the invoices list.

Then for **‘j’** in past\_due iterates over the past due list and sets the status of each invoice to past due. **‘j’** is used in the for loop to iterate over the past due list.

In this case, a for loop would be preferable because it clearly lays out all the steps the code is going through so other programmers can read and understand it quickly.

Here's an example of what that looks like.



This for loop iterates over the invoices list and sets the attribute of each invoice to past due if the invoice is more than 30 days old and has not been paid yet. This code will have the same result as the list comprehension we just went over but it's more straightforward. This makes it easier to code and understand.

So let's loop back to the question we asked earlier. When should you use a for loop and when should you use a list comprehension? Here are some questions to ask yourself to guide your decision.

Which one will make my code both clear and concise and which one will make my code easier for other people to read and understand?

It's important to keep in mind that most developers like their code short, sweet, and to the point. Finally, remember that whatever code writing strategy you choose, use meaningful names for all your variables, data sets and lists. Always assume others will see your code.

### List Comprehension Examples

You can create a list from an iterable using a for loop, which is a useful way to iterate over an iterable:

| new\_list = []  for thing in list\_of\_things:  new\_list.append(do\_something(thing)) |
| --- |

There is also a more streamlined way to do this by using a list comprehension. A list comprehension allows you to create a new list from an iterable in a single line. This line achieves the same result as the previous for loop, but in a more concise way:

| new\_list = [do\_something(thing) for thing in list\_of\_things] |
| --- |

List comprehensions can be used with tuples in Python, too. Here is an example of a list of tuples. In this example each tuple will contain the numbers 1, 2, and 3. This code will create 5 sets of (1,2,3).

| # Create a list of tuples where each tuple contains the numbers 1, 2, and 3.  numbers = [(1, 2, 3) for \_ in range(5)]  print(numbers)  # Output [(1, 2, 3), (1, 2, 3), (1, 2, 3), (1, 2, 3), (1, 2, 3)] |
| --- |

*For* loop vs. list comprehension

The two code blocks below compare performing the same process using a list comprehension and a *for* loop. The line ***[x\*2 for x in range(1,11)]*** is a simple list comprehension. This single line of code iterates over a range from 1 to 10, multiplies each element in the range by 2, and ultimately creates a new list from all multiples of 2 from 2 to 20.

The second block of code below is a for loop designed to carry out the same function as the list comprehension. The *for* loop, however, requires three lines of code

| ### Simple List Comprehension  print("List comprehension result:")  # The following list comprehension compacts several lines  # of code into one line:  print([x\*2 for x in range(1,11)])  ### Long form for loop  print("Long form code result:")  # The list comprehension above accomplishes the same result as  # the long form version of the code shown below:  my\_list = []  for x in range(1,11):  my\_list.append(x\*2)  print(my\_list)  # Select Run to compare the two results.  # Output List comprehension result: [2, 4, 6, 8, 10, 12, 14, 16, 18, 20] Long form code result: [2, 4, 6, 8, 10, 12, 14, 16, 18, 20] |
| --- |

**List comprehension with conditional statement**

You can also use conditionals with list comprehensions to build even more complex and powerful statements. You can do this by appending an *if* statement to the end of the list comprehension. For example, ***[x for x in range(1,101) if x % 10 == 0]*** generates a new list containing all the integers divisible by 10 from 1 to 100. The *if* statement evaluates each value in the range from 1 to 100 to check if it’s evenly divisible by 10. If it is, the number is added to a new list.

| print("List comprehension result:")  print([x for x in range(1,101) if x % 10 == 0])  # The list comprehension above accomplishes the same result as  # the long form version of the code:  print("Long form code result:")  my\_list = []  for x in range(1,101):  if x % 10 == 0:  my\_list.append(x)  print(my\_list)  # Output List comprehension result: [10, 20, 30, 40, 50, 60, 70, 80, 90, 100] Long form code result: [10, 20, 30, 40, 50, 60, 70, 80, 90, 100] |
| --- |

List comprehensions can be really powerful, but they can also be complex, resulting in code that’s hard to read. Be careful when using them, because they might make it more difficult for someone else looking at your code to easily understand what the code is doing. It is a best practice to add descriptive comments about any list comprehensions used in your code. This helps to communicate the purpose of list comprehensions to other coders. Comments will also help you remember the goal of the code when performing future code additions and maintenance.

**Practice exercise**

This exercise walks you through how to write a list comprehension to create a list of squared numbers (n\*n or n\*\*2). It needs to return a list of squares of consecutive numbers between “start” and “end” *inclusively*. For example, squares(2, 3) should return a list containing [4, 9].

1. The function receives the variables “start” and “end” through the function parameters.
2. In the *return* line, start by entering the list brackets [ ] that will contain the list comprehension.
3. Between the brackets [ ]:
   1. Enter the arithmetic expression to square a variable “n”.
   2. To the right of the square expression, write a *for* loop that iterates over “n” in a *range()* from the “start” to “end” variables.
   3. Ensure the “end” range value is included in the *range()* by adding 1 to it.
4. Run your code to see if it works! If needed, the solution to this code is included in the “Study Guide: List Operations and Methods” reading under “Skill Group 2” (list comprehensions).

| def squares(start, end):  return \_\_\_  print(squares(2, 3)) # Should print [4, 9]  print(squares(1, 5)) # Should print [1, 4, 9, 16, 25]  print(squares(0, 10)) # Should print [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100] |
| --- |

| def squares(start, end):  return [n \*\*2 for n in range(start, end+1)]  print(squares(2, 3)) # Should print [4, 9]  print(squares(1, 5)) # Should print [1, 4, 9, 16, 25]  print(squares(0, 10)) # Should print [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]  # Output  # [4, 9]  # [1, 4, 9, 16, 25]  # [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100] |
| --- |

### Study Guide: List Operations and Methods

Github markdown document: <https://github.com/GaJoDev/Python/blob/main/learning_resources/links/Study%20Guide%20-%20List%20Operations%20and%20Methods.md>

This study guide provides a quick-reference summary of what you learned in this lesson and serves as a guide for the upcoming practice quiz.

In the [Lists and Tuples video](https://www.coursera.org/learn/python-crash-course/lecture/jxAkv/lists-and-tuples)

, you learned about the differences between lists and tuples, how to modify the contents of a list, how to iterate over lists and tuples, how to use the enumerate() function, and how to create list comprehensions.

**Knowledge**

**Common sequence operations**

Lists and tuples are both sequences and they share a number of sequence operations. The following common sequence operations are used by both lists and tuples:

* **len(sequence)** - Returns the length of the sequence.
* **for element in sequence** - Iterates over each element in the sequence.
* **if element in sequence** - Checks whether the element is part of the sequence.
* **sequence[x]** - Accesses the element at index [x] of the sequence, starting at zero
* **sequence[x:y]** - Accesses a slice starting at index [x], ending at index [y-1]. If [x] is omitted, the index will start at 0 by default. If [y] is omitted, the len(sequence) will set the ending index position by default.
* **for index, element in enumerate(sequence)** - Iterates over both the indices and the elements in the sequence at the same time.

Because integers are not iterable, they need to be converted to a range as such:

| someList = ["An", "Example", "List"]  for x in range(len(someList)):  print(x)  # this will print out a numerical value up to the length of the original string  # Output  # 0  # 1  # 2 |
| --- |

**List-specific operations and methods**

One major difference between lists and tuples is that lists are mutable (changeable) and tuples are immutable (not changeable). There are a few operations and methods that are specific to changing data within lists:

* *list[index] = x* - Replaces the element at index [n] with x.
* *list.append(x)* - Appends x to the end of the list.
* *list.insert(index, x)* - Inserts x at index position [index].
* *list.pop(index)* - Returns the element at [index] and removes it from the list. If [index] position is not in the list, the last element in the list is returned and removed.
* *list.remove(x)* - Removes the first occurrence of x in the list.
* *list.sort()* - Sorts the items in the list.
* *list.reverse()* - Reverses the order of items of the list.
* *list.clear()* - Deletes all items in the list.
* *list.copy()* - Creates a copy of the list.
* *list.extend(other\_list)* - Appends all the elements of other\_list at the end of list
* *map(function, iterable)* - Applies a given function to each item of an iterable (such as a list) and returns a map object with the results
* *zip(\*iterables)* - Takes in iterables as arguments and returns an iterator that generates tuples, where the i-th tuple contains the i-th element from each of the argument iterables.

**Tuple use cases**

Remember, there are a number of cases where using a tuple might be more suitable than other data types:

* Protecting data: Because tuples are immutable, they can be used in situations where you want to ensure the data you have cannot be changed. This can be particularly helpful when dealing with sensitive or important information that should remain constant throughout the execution of a program.
* Hashable keys: Because they're immutable, tuples can be used as keys on dictionaries, which can be useful for complex keys.
* Efficiency: Tuples are generally more memory-efficient than lists, making them advantageous when dealing with large datasets.

**The *tuple()* operator**

The *tuple()* operator is used to convert an iterable (like a list, string, or set) into a tuple. Let's see an example:

| # Convert a list to a tuple  my\_list = [1, 2, 3, 4]  my\_tuple = tuple(my\_list)  print(my\_tuple) # Outputs: (1, 2, 3, 4)  # Remember that although parentheses are often used to define a tuple,  # they're not always necessary. The following syntax is also valid:  my\_tuple = 1, 2, 3, 4  print(my\_tuple) # Outputs: (1, 2, 3, 4) |
| --- |

This can sometimes lead to confusion, particularly when tuples are used in function calls or with operators that also use parentheses.

**Tuples with mutable objects**

Although tuples themselves are immutable, they can contain mutable objects, such as lists. This means that although the tuple cannot be modified (for example, you can't add or remove elements), the mutable elements within the tuple can be changed.

| # A tuple with a list as an element  my\_tuple = (1, 2, ['a', 'b', 'c'])  # You can't change the tuple itself  # my\_tuple[0] = 3 # This would raise a TypeError  # But you can modify the mutable elements within the tuple  my\_tuple[2][0] = 'x'  print(my\_tuple) # Outputs: (1, 2, ['x', 'b', 'c']) |
| --- |

**Returning multiple values from functions**

One of the most useful aspects of tuples in Python is their ability to return multiple values from a function. This allows a function to produce more than one result, providing flexibility and improving code readability.

Here's an example:

| def calculate\_numbers(a, b):  return a+b, a-b, a\*b, a/b  result = calculate\_numbers(10, 2)  print(result) # Outputs: (12, 8, 20, 5.0) |
| --- |

In the above function, the four arithmetic calculations are returned as a tuple, which can be assigned to a single variable (result), or "unpacked" into multiple variables:

| def calculate\_numbers(a, b):  return a+b, a-b, a\*b, a/b  add\_result, sub\_result, mul\_result, div\_result = calculate\_numbers(10, 2)  print(add\_result) # Outputs: 12  print(sub\_result) # Outputs: 8 |
| --- |

This is a powerful feature that makes Python functions extremely versatile.

**List comprehensions**

A list comprehension is an efficient method for creating a new list from a sequence or a range in a single line of code. It is a best practice to add descriptive comments about any list comprehensions used in your code, as their purpose can be difficult to interpret by other coders.

* **[expression for variable in sequence]** - Creates a new list based on the given sequence. Each element in the new list is the result of the given expression.
  + Example: *my\_list = [ x\*2 for x in range(1,11) ]*
* **[expression for variable in sequence if condition]** - Creates a new list based on a specified sequence. Each element is the result of the given expression; elements are added only if the specified condition is true.
  + Example: *my\_list = [ x for x in range(1,101) if x % 10 == 0 ]*

Note that tuples do not have comprehensions but a similar functionality can be achieved with:

*tuple(i for i in (1, 2, 3))*

**When to use *for* loops or list comprehensions**

In Python, list comprehensions are generally used for creating new lists from existing ones in a concise and readable manner, especially when the task involves simple transformations or filtering of elements.

*for* loops are more versatile and are preferred when the operation is more complex, requires multiple lines of code, involves statements other than expression (like *print*, *pass*, *continue*, *break*), or when you need to iterate over a list without creating a new one.

**Coding skills**

**Skill Group 1**

* Use a *for* loop to modify elements of a list.
* Use the *list.append()* method.
* Use the *string.endswith()* and *string.replace()* methods to modify the elements within a list.

| # This block of code changes the year on a list of dates.  # The "years" list is given with existing elements.  years = ["January 2023", "May 2025", "April 2023", "August 2024", "September 2025", "December 2023"]  # The variable "updated\_years" is initialized as a list data type  # using empty square brackets []. This list will hold the new list  # with the updated years.  updated\_years = []  # The for loop checks each "year" element in the list "years".  for year in years:  # The if-statement checks if the "year" element ends with the  # substring "2023".  if year.endswith("2023"):  # If True, then a temporary variable "new" will hold the  # modified "year" element where the "2023" substring is  # replaced with the substring "2024".  new = year.replace("2023","2024")  # Then, the list "updated\_years" is appended with the changed  # element held in the temporary variable "new".  updated\_years.append(new)  # If False, the original "year" element will be appended to the  # the "updated\_years" list unchanged.  else:  updated\_years.append(year)  print(updated\_years)  # Output ["January 2024", "May 2025", "April 2024", "August 2024", "September 2025", "December 2024"] |
| --- |

**Skill Group 2**

* Use a list comprehension to return values

| # This list comprehension creates a list of squared numbers (n\*n). It  # accepts two integer variables through the function’s parameters.  def squares(start, end):  # The list comprehension calculates the square of a variable integer  # "n", where "n" ranges from the "start" to "end" variables inclusively.  # To be inclusive in a range(), add +1 to the end of range variable.  return [n\*n for n in range(start,end+1)]  print(squares(2, 3)) # Should print [4, 9]  print(squares(1, 5)) # Should print [1, 4, 9, 16, 25]  print(squares(0, 10)) # Should print [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100] |
| --- |

**Skill Group 3**

* Use a list comprehension to modify elements of a list.
* Use the *string.replace()* method within a list comprehension.
* Use the *string[index]* method within a list comprehension.

| # This block of code also changes the year on a list of dates using a  # different approach than demonstrated in Skill Group 1. By using a  # list comprehension, you can see how it is possible to refactor the  # code to a shorter, more efficient code block.  # The "years" list is given with existing elements.  years = ["January 2023", "May 2025", "April 2023", "August 2024", "September 2025", "December 2023"]  # The list comprehension below creates a new list "updated\_years" to  # hold the command to replace the "2023" substring of the "year"  # element with the substring "2024". This action will be executed if  # the last 4 indices of the "year" string is equal to the substring  # "2023". If false (else), the "year" element will be included in the  # new list "updated\_years" unchanged.  updated\_years = [year.replace("2023","2024") if year[-4:] == "2023" else year for year in years]  print(updated\_years)  # Should print ["January 2024", "May 2025", "April 2024", "August 2024", "September 2025", "December 2024"] |
| --- |

**Skill Group 4**

* Use the *string.split()* method to separate a string into a list of individual words.
* Iterate over the new list using a for loop.
* Modify each element in the list by slicing the element’s string at a given [1:] index position and appending the substring to the end of the element.
* Convert a list back into a string.

| # This function splits a given string into a list of elements. Then, it  # modifies each element by moving the first character to the end of the  # element and adds a dash between the element and the moved character.  # For example, the element "2two" will be changed to "two-2". Finally,  # the function converts the list back to a string, and returns the  # new string.  def change\_string(given\_string):  # Initialize "new\_string" as a string data type by using empty quotes.`  new\_string = ""  # Split the "given\_string" into a "new\_list", with each "element"  # holding an individual word from the string.  new\_list = given\_string.split()  # The for loop iterates over each "element" in the "new\_list".  for element in new\_list:  # Convert the list into a "new\_string" by using the assignment  # operator += to concatenate the following items:  # + Each list "element" (starting at index position [1:]),  # + a dash "-",  # + append the first character of the "element" (using the index  # [0]) to the end of the "element", and finally,  # + a space " " to separate each "element" in the "new\_string".  new\_string += element[1:] + "-" + element[0] + " "  # Return the list that has been converted back into a string.  return new\_string  print(change\_string("1one 2two 3three 4four 5five")) # Should print "one-1 two-2 three-3 four-4 five-5" |
| --- |

**Skill Group 5**

* Use the *string.join()* method to concatenate a string that provides a list name and its elements.

| # This function accepts a list name and a list of elements, and returns  # a string with the format: "The "list\_name" list includes: element1,  # element2, element3".  def list\_elements(list\_name, elements):  # This task can be completed in a single line of code. The  # concatenation of strings, "list\_name", and the list "elements" can  # occur on the return line. In this case, the string "The " is added  # to the "list\_name", plus the string " list includes: ", then the  # "elements" are joined using a comma to separate each element of the  # list.  return "The " + list\_name + " list includes: " + ", ".join(elements)  print(list\_elements("Printers", ["Color Printer", "Black and White Printer", "3-D Printer"]))  # Should print "The Printers list includes: Color Printer, Black and White Printer, 3-D Printer" |
| --- |

**Skill Group 6**

* Use *map()* and convert the map object to a list so we can print all the results at once.

| result = map(add\_one, numbers)  # Convert the map object to a list to print the result  print(list(result))  # Outputs: [2, 3, 4, 5, 6] |
| --- |

**Skill Group 7**

* Use *zip()* to combine a list of names and ages into a list of tuples, and print all the tuples at once.

| # A simple function to add 1 to a given number  def add\_one(number):  return number + 1  # A list of numbers  numbers = [1, 2, 3, 4, 5]  # Use map to apply the function to each element in the list  # Two lists  names = ["Alice", "Bob", "Charlie"]  ages = [25, 30, 35]  # Use zip to combine the lists  combined = zip(names, ages)  # Convert the zip object to a list to print the result  print(list(combined))  # Outputs: [('Alice', 25), ('Bob', 30), ('Charlie', 35)] |
| --- |

**Resources**

For additional information about list and tuple operations and methods, please visit:

* [Common Sequence Operations](https://docs.python.org/3/library/stdtypes.html#sequence-types-list-tuple-range) - Official python.org documentation for list, tuple, and range sequence operations.
* [Lists](https://docs.python.org/3/library/stdtypes.html#lists) - Official python.org documentation for list operations and methods.
* [Tuples](https://docs.python.org/3/library/stdtypes.html#tuples) - Official python.org documentation for tuple operations and methods

### Practice Quiz: Lists

## **Dictionaries**

|  |
| --- |

|  |
| --- |

|  |
| --- |

|  |
| --- |

|  |
| --- |

|  |
| --- |

|  |
| --- |

|  |
| --- |

|  |
| --- |

|  |
| --- |

|  |
| --- |

|  |
| --- |

|  |
| --- |

|  |
| --- |

|  |
| --- |

|  |
| --- |

|  |
| --- |

|  |
| --- |

|  |
| --- |

### Review: What is a dictionary?

### 

### What is a dictionary?

### 

### Dictionaries Defined

### 

### Review: Iterating over the contents of a dictionary

### 

### Iterating over the Contents of a Dictionary

### 

### Use while loops and if else statements for dictionaries

### 

### Dictionaries vs. Lists

### 

### Connect: Types of iterables

### 

### Study Guide: Dictionary Methods

### 

### Practice Quiz: Dictionaries

## **Object-oriented Programming (Optional)**

### OOP Introduction (Optional)

### 

### What Is a Method?

### 

### Constructors and Other Special Methods (Optional)

### 

### Special Methods

### 

### Instance Methods (Optional)

### 

### Methods as special operators

### 

### Study guide: Classes and methods (optional)

### 

### Basic Structures Wrap Up

### 

### In Marga's Words: My Most Challenging Script

### 

### Glossary terms from course 1, module 4

### 

### Study Guide: Module 4 Graded Quiz