**Exercise: Dictionaries and Sets**

*Topic title: Exercise: Dictionaries and Sets.*

Here are some questions for you to think about in this exercise. You'll first try and list some features of dictionaries in Python. You’ll then move on to describing the basic features of sets in Python, another complex data type.

And then compare and contrast, list some similarities and differences between the complex data types that you’ve understood, lists, dictionaries, and sets in Python. I'd suggest that you pause the video here, think about these questions for a while. Maybe, write down your answers before moving on to look at some sample answers.

*Solution*

Dictionaries in Python are a very commonly used and an extremely useful complex data type. Dictionaries in Python store information in the form of an unordered collection of key-value pairs. Every value has a corresponding key, and that value can be look up using that key. So, when you construct a dictionary, you’ll set up key-value pairs.

And there is no inherent ordering amongst the keys or the values. Each key in a dictionary is unique and values can be duplicates. Keys have to be unique because keys are what you use to look up corresponding values in a dictionary. Values are looked up by keys, so you can't go the other way round. So you can't specify a value and find the corresponding key. The mapping only goes in one direction.

Values can be either simple or complex data types. So you can have nested lists within dictionaries. You can have dictionaries within dictionaries. Tuples within dictionaries and so on. And, finally, dictionaries are mutable. Once created, you can add key-value pairs to a dictionary. You can change values corresponding to a key. You can delete key-value pairs from a dictionary as well.

Let's now move on to discussing the important features of sets in Python. Sets in Python are also an unordered collection. But they are made up of completely unique elements. All elements in a set have to be unique, duplicates are not allowed. If you add a duplicate value to a set, the set will automatically keep only one copy of the duplicate. So, for example, if you try to add the number ten multiple times, there'll be exactly one element ten within your set.

Now, sets in Python cannot contain other mutable complex data types. It cannot contain lists or dictionaries. In addition to the common operations of adding elements, removing elements from sets, sets also support other interesting operations. Such as union, intersection, difference, and so on. All operations that you might expect to perform with sets are available in the form of functions.

Now that we’ve understood the complex data types, let's move on to understanding similarities and differences between lists, dictionaries, and sets. Lists, as we spoke about, are ordered collections. Both dictionaries and sets are unordered. There is no intrinsic ordering of elements within dictionaries and sets. There is no constraint on how many times the same element can be added to a list. So you can have the element 10 present ten times if you want to.

Lists can contain duplicates, sets cannot contain duplicates, dictionaries cannot have duplicate keys. The non-duplicate constraint applies only to keys in a dictionary, not to its values. Lists can contain nested complex data types. So you can have a list of dictionaries, a list of sets, a list of tuples, and so on.

And values in dictionaries can also be complex data types. Values in dictionaries can be a nested dictionary, a nested list, a nested set, a nested tuple, anything. Sets, on the other hand, can contain only immutable tuples. It cannot contain other complex data types, such as lists or dictionaries because lists and dictionaries are mutable.

Here is one last point of comparison between lists, dictionaries, and sets. List elements can be looked up by index values starting at zero. There is a no corresponding look up for set elements. Sets don't have an intrinsic order. Dictionary values, on the other hand, can be looked up by key.

## Exercise: Shallow and Deep Copies

Topic title: Exercise: Shallow and Deep Copies.

In this exercise, we'll try and recall some of the things that we learned about shallow and deep copies. We'll start with trying to recall what shallow and deep copies mean for complex data types, such as lists, tuples, dictionaries, and so on.

You'll then try and explain how copying works with strings, that is the string data type. And, finally, move on to try and explain how copying works with lists. There are nuances involved across all of these, try and see if you can remember all of them.

I suggest you pause this video and try these questions out on your own, before moving on to sample answers. Just a heads up that the answers that you get here might be a little different. You may not have the exact same answers, but you should have similar points.

A shallow copy of contents are made when two variables with different variable names point to the same location in memory. So the actual contents of the data type are located somewhere in Python's memory, and we have two variables referring the same location.

Even though the names of the variables are different, because they point to the same location in memory, updates that you make using one variable are reflected when you try and access the same data using the other variable.

When you make updates to shallow copies using either variable, the updates are made to the same location in memory. Which is why the updates are accessible using both variables. The same complex data type has just two names with shallow copies.

That is, the two variables are the two names of the same data. In the case of deep copies, however, deep copies are made when the same data, when the same contents are copied over to an entirely new memory location.

The contents can be anything, a list, a dictionary, when the entire contents is copied to another memory location. So we have the same contents now in two bits of Python memory, that's what is considered a deep copy.

The main difference between shallow and deep copies is this copying over of contents. So changes made to the deep copy are not reflected in the original memory. Once the deep copy is made, the two contents are entirely separated from one another and changes can be made to one without affecting the other.

Let's talk a little bit about how copying works with strings in Python. Strings are immutable in Python, so once you've created a string, you can't really update that string. So strings, once created, cannot be updated.

So any changes that you seem to make to string variables, are actually creating entirely new strings. Let's say you have a string stored in variable a, and you use an assignment operator to assign this string to b. Both variables, a and b, point to the same memory location where that immutable string is located.

But that location cannot be changed that the contents of that location cannot be updated. Now, if you were to perform an assignment where you change the variable a to have a new string value, that is an entirely new string, a now refers to that new string in memory.

You've not actually changed the memory location's contents. Let's move on to talking about lists. Let's say you have two lists, list\_a and list\_b, and you set up a statement like this, list\_b = list\_a. What you're doing here is creating a shallow copy of list\_a.

Whatever the contents of list\_a, list\_b points to the same contents. So changes that you make using list\_b will be reflected in the variable list\_a. Now, if you set up something like this, list\_b = list\_a[:], what you're creating here is a deep copy of a list.

Using the slicing operator in this manner creates a deep copy. There are other ways to create deep copies as well, list\_b = copy.copy(list\_a). This creates a deep copy of the outer list, but shallow copies of any nested complex data types within list\_a.

So you need to be a little aware when you're using the copy function of the copy module. This creates a deep copy only of the outer list. If you want a deep copy of the list and its entire contents, you'll use copy.deepcopy. This creates a deep copy of the outer list, as well as any nested complex data types within the outer list.

## Exercise: Advanced Operations in for Loops

Topic title: Exercise: Advanced Operations in for Loops. The presenter is Kishan Iyer.

In this exercise, you will be writing some code in order to implement some for loops. The first for loop which you will write will iterate over a random list of numbers and then will print those numbers except if the number happens to be divisible by 7. And that is not all, because there is one more condition which you need to incorporate in your for loop, specifically that the iterations will stop if the number 315 is encountered within the list you are iterating over. So there are different courses of action which you can take depending upon the value which you are iterating over in the list. You will need to make use of a couple of the statements which we encountered during this course.

For the next task in the exercise, you will once again start with a random list of numbers. Using that list, you will create another list using list comprehension. And within the new list, you will include the cubes of the numbers from the original list but only if that number happens to be divisible by 3. So just to clarify, you will need to evaluate which of the numbers in your original list is divisible by 3, and you will need to include the cubes of those numbers in your new list. And the creation of the new list will occur in a single line using list comprehension. All of the skills which you need in order to solve each of these problems were covered during this course.

So please pause this video and then spend a few minutes to write the for loops required for this exercise on your own. For the first task in the exercise, you needed to iterate over a given list and then print out all of the values which are not divisible by 7, and also stop the iterations once you encounter the number 315. To incorporate all of those conditions, you need to make use of the break and the continue statements. So let us see how this works. We start off with a list of numbers which include multiples of 7 and also the number 315. So this is something we can use in order to test out our for loop. So let us start iterating over each of the elements within this list\_of\_nums with the for loop.

Since we will terminate this loop when the number 315 is encountered, we need to perform a check for that number first. So during our iterations, if you find that the value of i is equal to 315, then we stop the iterations by breaking from the loop. Note that since the number 315 is divisible by 7, we will not have to print out its value. In fact, the next thing we do within our for loop is to check whether i is divisible by 7. And if it is, we don't really need to do anything and can just head over to the next iteration in the for loop. So we invoke the continue statement. And then finally, we get to a point where the number is neither 315 and nor is it divisible by 7, in which case, we print out the number.

So once you run this code, you'll get an output similar to this. The numbers 238 and 301 are skipped because they're divisible by 7. And then once the number 315 is encountered, we break out of the for loop, which is why 361 and 399 are never evaluated. Moving along now to the last task in the exercise, which was to create a list containing all of the cubes for the integers in another list which are divisible by 3. So let us consider that our original\_list contains all of these elements, the elements 3, 6, 12, and 18 are all multiples of 3.

Now, let us create a cubes\_list where we iterate over all of the elements in the original\_list and check whether the element is divisible by 3. If it is, then the cube of that element will be included in our cubes\_list. So within this single line, we have defined what our list of cubes will look like. And to confirm your logic, you can print out the contents of cubes\_list, and it should look something like this. Where the cubes of 3, 6, 12, and 18 will be included.

## Exercise: While Loops in Python

Topic title: Exercise: While Loops in Python. Your host for this session is Kishan Iyer.

In this exercise, you will start off by comparing for and while loops. During this course, we saw that while loops have a lot of similarities with for loops. However, each of them is suited for slightly different use cases. You will identify exactly what type of operations both of these loops can be used for while highlighting their commonalities as well as their differences.

Next you will need to write some code specifically using a while loop. But this code will repeatedly prompt the user to enter the capital of Egypt and it will keep asking the user this question until one of the following two things happens: The user could either enter the correct answer, which is Cairo, or they could just give up and then type in the word quit.

So the important thing here is that the user has no way of getting out of being repeatedly asked the question until they enter one of these two values. These topics were covered during this course. So please pause this video and then spend a few minutes to do this exercise on your own. For the first task in the exercise, you needed to compare both for and while loops.

A solution banner appears at the bottom.

Let us begin by pointing out the big similarity between these two control structures. So they are both used in order to run sections of code repeatedly. So the code written within the body of a for or a while loop represent a single iteration of those looping structures and that code will be executed repeatedly. But how they cease execution is what distinguishes both for and while loops.

So in the case of a while loop, the iterations will go on until a specific condition is met. So this could be the condition which is evaluated after the while keyword or one which is defined within the body of the while loop within an if block and then a break is triggered. In any case, the while loop will continue iterating until this condition is satisfied.

So this is specifically suited for situations where the number of iterations is not known up front. This could be because the condition depends upon some code which is executed within the body of the while loop or some value which is input by the user or pretty much any other case where the outcome is hard to determine beforehand.

In the case of for loops, however, this is specifically suited in order to iterate over a sequence of values. So these could be iterable objects which have a known length such as a list, a set, a tuple, and so on. In any of these cases, the for loop will continue iterating and then executing the code defined within its body until all of the elements in the sequence it is iterating over have been processed. Once this happens, the for loop will terminate on its own.

Do keep in mind that an iterable object can be defined using the range function as well. This can create a sequence of integers with a given interval. In addition to these points, you could also say that some point of similarity between these two control structures is that their execution can be controlled by using the break and continue statements. Moving along now to the last task in this exercise where you needed to prompt the user to enter the capital of Egypt until the user either enters the correct answer or gives up and types in quit.

Screen title: Run Till Explicit Condition.   That is a while loop will continue iterating until an explicit condition is met. Given this is a coding problem, there are multiple solutions. But I'll just present one possible option. A terminal is open.

To start off with, we initialize a variable called response, which will be used to store the response given by the user, and this is initialized to a blank string. The presenter pastes the following line of code: response = "".  And then in order to ensure that the user is repeatedly prompted for an answer, we will make use of a while loop. He pastes the following line of code: while True:.

And given that we can break out of the while loop only if the user either types in quit or enters the correct answer, we can set this while loop to run indefinitely until one of those two responses comes in. So the first thing we do within the body of our while loop is to ask the user the question, what is the capital of Egypt? And then their response will be captured in the response variable. He pastes the following line of code: response = input("\nWhat is the capital of Egypt? ").

We check whether the response is equal to the string "quit", and if it is, we will give them the correct answer. He pastes the following two lines of code: Line 1: if response == "quit": and line 2: print("The correct answer is Cairo. Better luck next time!").  And then following that, we will end the while loop by invoking the break statement. He pastes the following line of code: break

If the user has not decided to quit, we will check the value of their answer and see if it is the correct one. He pastes the following two lines of code: Line 1: if response.upper() == "CAIRO": and line 2: print("That is the correct answer!").  So note here, we are allowing them to type in the answer in any case. We just convert their response to all uppercase and then compare it with CAIRO, which is also in all uppercase.

So if the correct answer has been input, then we print out the fact that they have answered correctly and we break out of the while loop by invoking break. If the user response is not the correct answer, we will convey this to them and then we will ask them to try again, in which case we go over to the next iteration of the while loop where once again they will be asked the question, what is the capital of Egypt? He pastes the following two lines of code: Line 1: else: and line 2: print("That is not the correct answer. Try again...").