Complex Data Types in Python: Shallow & Deep Copies in Python

Explore copying operations on containers in Python in this 9-vdeo course, which examines the subtle distinction between shallow and deep copies. Changes made to shallow copies affect the original whereas with deep copies they do not. Learners begin by observing Jupyter notebook in Python, where you will be performing shallow and deep copies of Python strings. You will learn how to create shallow copies of lists, and then create deep copies of lists where changes to the copy do not affect the original. Following this, you will begin working with tuples, a process which you will discover is quite simple because tuples are immutable. So you will learn how to create shallow and deep copies of tuples. You will also learn how deep copies of dictionaries work, and perform shallow and deep copies of sets. In the closing exercise, learners are asked to recall how shallow and deep copies work for complex data types.

Table of Contents

Course Overview

[Video description begins] *Topic title: Course Overview* [Video description ends]

Hello there, welcome to this course Shallow and Deep Copies in Python. My name is Janani Ravi and I'll be your instructor for this course.

[Video description begins] *Your host for this session is Janani Ravi. She is a Software engineer and big data expert.* [Video description ends]

A little about myself first, I'm a co-founder at Loonycorn a studio for high-quality video content. Before founding Loonycorn, I worked at various companies including Google for seven years in New York and Singapore and Microsoft for three years in Redmond.

And before that, I attended grad school at Stanford. In this course, you will move on to an important concept in working with containers, such as lists, dictionaries, and tuples, that of shallow and deep copies.

This course assumes that you have an interactive Python development environment such as Jupyter, installed and are familiar with defining variables invoking simple built-in functions. You'll also need to be familiar with working with lists, dictionaries and sets.

In this course, you will work with copying operations on containers. You'll learn the important and subtle distinction between shallow and deep copies. Given a list variable, for instance, a shallow copy of that variable is another list variable that contains literally the same underlying elements.

So if you change the elements in the shallow copy, the data in the original variable changes as well. You will see how deep copies in contrast, actually go ahead and create copies of the data in the underlying complex data type as well.

Then changes made to the deep copy do not change elements in the original. Python has various built-in functions that support shallow and deep copying including versions that need deep copies nearly one level in.

Shallow and deep copying are important operations to understand from the point of view of functional programming and distributed processing. Advanced concepts that build on these basic building blocks.

There, copies of data are often made and shipped off to be processed in parallel. And it is important that these copies not interfere with each other's state. By the end of this course, you will be able to identify situations where either deep or shallow copies are required.

You'll know how to create shallow and deep copies, and also know the semantics of the different built-in Python functions that provide this functionality.

Copying Strings

[Video description begins] *Topic title: Copying Strings. Your host for this session is Janani Ravi*. [Video description ends]

Now that we've understood the various data types that Python supports, let's understand how copy operations work. This will allow you to avoid a few pitfalls when you're programming.

[Video description begins] *A jupyter notebook titled StringCopy displays in the interface. The Titlebar is present at the top of the interface which contains the file name. Below the Titlebar is the Notebook Menu bar containing the tabs File, Edit, View, Insert, and so on. Underneath the Notebook Menu bar is the Notebook toolbar which contains various icons. A blank notebook appears below. A cell in the form of an empty input box displays with the command prompt: In [ ] : to the left of it*. [Video description ends]

In this video we'll start off by focusing on string copy operations. Here we have a string called old_str that we've initialized to the string Python.

[Video description begins] *The host highlights the following two lines of code in the input box of the first code cell: Line1:old_str = "Python" Line 2: old_str. The number 1 now appears within the square brackets after the word 'In' in the command prompt. The following output displays below, after Out [1]: 'Python'*. [Video description ends]

I'll now perform a simple assignment operation where I assign old string to new string. Here the value present in old string is simply copied over to the new string.

[Video description begins] *She highlights the following code in the next code cell: new_str = old_str.* [Video description ends]

Both the old string and the new string now contain the string Python. Now let's try accessing one of these strings.

[Video description begins] *She highlights the following lines of code in the second code cell: print (old_str) print(new_str) The number 2 now appears within the square brackets after the word 'In' in the command prompt. The following two lines of output display below, after Out [2]: Line 1: Python Line 2: Python*. [Video description ends]

New_str at index position 2 gives us the character t. Remember, strings behave exactly like lists in this case.

[Video description begins] *She highlights the following code in the next code cell: new_str[2]. The number 3 now appears within the square brackets after the word 'In' in the command prompt. The following output displays below, after Out [3]: 't'*. [Video description ends]

But if you try to update the character at index position 2 with a new_str you'll encounter an error. That's because strings in Python are immutable. The assignment operation that we had performed earlier, new_str = old_str.

[Video description begins] *She enters the following code in the next input cell: new_str[2] = 'T'. The number 4 now appears within the square brackets after the word 'In' in the command prompt. The following line of error displays in the output: 'str' object does not support item assignment*. [Video description ends]

Both of these variables will refer to the same string Python under the hood. Because the string itself is immutable, you can't change the value of the string that both of these variables hold.

And as we've seen with variables if you set new string to some other string value this will not affect the value held in old_str. Now everything that we've discussed so far about strings are through of floating points and integer values as well.

Now I'm going to type in a Python statement which might be new for those who are learning Python as their first programming language, import copy.

[Video description begins] *She enters the following code in the next code cell: import copy. The number 5 now appears within the square brackets after the word 'In' in the command prompt*. [Video description ends]

What does this mean? We've already seen that Python contains a number of built-in functions for common use cases. In addition to these built-in functions, Python also comes along with a number of libraries that you can access based on your use case.

These libraries will have their own functions and classes that you might find useful. In order to access these libraries, you need to import them into your current program, which you'll do using the import keyword.

Here, I've used the import keyword to import the copy library that comes along with Python. Once this library has been imported into my current program, I'll be able to access useful functions from this copy module. I'll first set up a simple string variable first_str which contains the string Will.

[Video description begins] *She enters and executes the following code in the next code cell: Line 1: first_str = 'Will'. Line 2: first_str. The number 6 now appears within the square brackets after the word 'In' in the command prompt. The following output displays after Out [6]: 'Will'*. [Video description ends]

I'm now going to create a copy of the first string by calling the copy function within the copy module. So copy.copy. Here the first copy refers to the module.

[Video description begins] *She enters and executes the following lines of code in the next cell: Line 1: second_str = copy.copy(first_str) Line 2: print("First string: ", first_str) Line 3: print("Second string: ", second_str)*. [Video description ends]

And the copy with brackets is the function that I'm invoking within the module. And I've passed in the first string and this will create a copy and place the copy in second string. So if you print out first string and second string both contain the string Will.

[Video description begins] *She highlights the following output: Line 1: First string: Will Line 2: Second string: Will*. [Video description ends]

I'll now update the value stored in the second string to Smith, and you'll find that the first string remains unchanged. This would have been true even if you had used the assignment operator to assign second string to first string.

[Video description begins] *She enters and executes the following lines of code in the next cell: Line 1: second_str = 'Smith' Line 2: print("First string: ", first_str) Line 3: print("Second String: ", second_str). The number 8 now appears within the square brackets after the word 'In' in the command prompt. She highlights the following output: Line 1: First string: Will Line 2: Second string:Smith*. [Video description ends]

Because strings are immutable when you create a copy of the first string and assign it to the second string, it's the same as the assignment operator. I'll now assign a new value to the first string variable. This is the value Johnny.

[Video description begins] *She enters and executes the following lines of code in the next cell: Line 1: first_str = 'Johnny' Line 2: first_str. The number 9 now appears within the square brackets after the word 'In' in the command prompt. She highlights the following output: 'Johnny'*. [Video description ends]

This time I'll use a different function within the copy module, the deepcopy function. At this point in time, you don't need to worry about what exactly a deepcopy means.

[Video description begins] *She enters and executes the following lines of code in the next code cell: Line 1: second_str = copy.deepcopy(first_str) Line 2: print("First string: ", first_str) Line 3: print("Second String: ", second_str)*. [Video description ends]

It's the same as when you copy a string, the deepcopy of a string and the copy of a string is exactly the same in Python. You can see that both first_str and second_str are equal to Johnny.

[Video description begins] *The number 10 now appears within the square brackets after the word 'In' in the command prompt. She then highlights the following output: Line 1: First string: Johnny Line 2: Second string:Johnny*. [Video description ends]

Now, if you reassigned the value of the second_str to be equal to Depp, you'll find that the first_str remains unchanged.

[Video description begins] *She enters and executes the following lines of code in the next code cell: Line 1: second_str = 'Depp' Line 2: print("First string: ", first_str) Line 3: print("Second String: ", second_str). The number 11 now appears within the square brackets after the word 'In' in the command prompt. She then highlights the following output: Line 1: First string: Johnny Line 2: Second string:Depp*. [Video description ends]

So note here, that there is no difference between the behavior of copy, deep_copy, and the assignment operator as far as strings are concerned in Python. F

[Video description begins] *She highlights the code in Line number 10 that contains the deepcopy function*. [Video description ends]

or other complex data types, there is indeed a difference and we'll see that in other videos that follow.

Performing Shallow Copies of Lists

[Video description begins] *Topic title: Performing Shallow Copies of Lists. Your host for this session is Janani Ravi*. [Video description ends]

As far as indexing and slicing operations go, we've seen that list and strings are very similar. But as far as copying is concerned you will find that there is a major difference and this comes from the fact that strings are immutable while lists are not.

[Video description begins] *A jupyter notebook titled ListShallowCopy displays in the interface*. [Video description ends]

In this video we'll understand how we can make shallow copies of a list, and also explain what shallow copies are in just a bit. Here is a list of names containing four different names.

[Video description begins] *The host enters and executes the following lines of code in the first code cell: Line 1:names = ["Sara", "David", "Warner", "Sandy"] Line 2: names. The number 1 now appears within the square brackets after the word 'In' in the command prompt. She highlights the following output that displays after Out [1]: ['Sara', 'David', 'Warner', 'Sandy']*. [Video description ends]

I'll now use the assignment operator to assign the names list to a new variable new_names.

[Video description begins] *She enters and executes the following code in the next code cell: Line 1: new_names = names Line 2: print("Original:" , names) Line 3: print("Copy:" , new_names)*. [Video description ends]

I'll now print out the original list of names in the names variable, and the new list in the new_names variable, and you can see that they are exactly the same. The same list of names are present in both.

[Video description begins] *The number 2 now appears within the square brackets after the word 'In' in the command prompt. She then highlights the following output: Line 1: Original: ['Sara', 'David', 'Warner', 'Sandy'] Line 2: Copy: ['Sara', 'David', 'Warner', 'Sandy']*. [Video description ends]

The name at index position 1 in the names list is the name David. Remember that index position 1 refers to the second element in the list.

[Video description begins] *She enters and executes the following code in the next code cell: names[1]. The number 3 now appears within the square brackets after the word 'In' in the command prompt. She highlights the following output: 'David'*. [Video description ends]

I'm now going to update this particular element in the new_names list, new_names[1] = Ridley. Observe that I performed this update on the new_names list not on names.

[Video description begins] *She enters and executes the following lines of code in the the next code cell: Line 1: new_names[1] = 'Ridley' Line 2: print("Original:", names) Line 3: print("Modified:", new_names)*. [Video description ends]

And if you take a look at the original list and the new_names list you'll find that the same update has applied to both.

[Video description begins] *The number 4 now appears within the square brackets after the word 'In' in the command prompt. She then highlights the following lines of output: Line 1: Original: ['Sara', 'Ridley', 'Warner', 'Sandy'] Line 2: Modified: ['Sara', 'Ridley', 'Warner', 'Sandy']*. [Video description ends]

Both names as well as new_names now contains Ridley as the second name in the list. How did this happen? This is because when we use the assignment operator with lists new_names = names, this created a shallow copy of the names list.

Both of these variables, new_names as well as names refer to the same list under the hood, which is why when we updated an element in the new_names list, the names list was also updated under the hood in memory this is the same list.

[Video description begins] *She refers to the code in Line number 2 which appears on the right-hand side of the interface*. [Video description ends]

This is what it means to make a shallow copy of a list. When we use the assignment operator with strings you couldn't update strings anyway so this does not matter.

[Video description begins] *She enters the following code in the next code cell: import copy. The number 5 now appears within the square brackets after the word 'In' in the command prompt.* [Video description ends]

Let's import the copy module as we did in the previous example, and let's set up a new list of companies. The company's list is simply a list of strings.

[Video description begins] *She enters and executes the following lines of code in the next code cell: Line 1: companies = ["hackerearth" , "google", "facebook"] Line 2: companies. The number 6 now appears within the square brackets after the word 'In' in the command prompt. She highlights the following output:['hackerearth' , 'google', 'facebook'].* [Video description ends]

I'll now create a copy of this company's list by invoking the copy function from the copy module, copy.copy passing companies and store the result in the variable, new_companies.

[Video description begins] *She enters and executes the following lines of code in the next code cell: Line 1: new_companies = copy.copy(companies) Line 2: print('Original:', companies) Line 3: print('Modified:', new_companies). The number 7 now appears within the square brackets after the word 'In' in the command prompt.* [Video description ends]

Print out the original list of companies and the new list of companies, and you'll see that they're exactly the same.

[Video description begins] *She then highlights the following lines of output: Line 1: Original: ['hackerearth' , 'google', 'facebook'] Line 2: Modified: ['hackerearth' , 'google', 'facebook'].* [Video description ends]

In the new_companies variable, let's print out the element at index position 1. And you can see that the result here is google.

[Video description begins] *She enters and executes the following code in the next code cell: new_companies[1]. The number 8 now appears within the square brackets after the word 'In' in the command prompt. She highlights the following output: 'google'.* [Video description ends]

Let's update this particular element to be equal to microsoft. I've referenced the variable new_companies, and updated the element at index position 1, and set it equal to microsoft. So what do the two lists contain?

[Video description begins] *She enters and executes the following code in the next code cell: Line 1: new_companies[1] ='microsoft' Line 2: print('Original:', companies) Line 3: print('Modified:', new_companies). The number 9 now appears within the square brackets after the word 'In' in the command prompt.* [Video description ends]

Well, you'll see that the original list, which we did not update still contains google at index position 1, whereas the modified list contains microsoft. This means that under the hood, new companies and companies are entirely different list.

[Video description begins] *She then highlights the following lines of output: Line 1: Original: ['hackerearth' , 'google', 'facebook'] Line 2: Modified: ['hackerearth' , 'microsoft', 'facebook'].* [Video description ends]

When we use the copy function in the copy module, this created a deep copy of a list. A deep copy essentially means that an entirely new list was created in memory, and changes to the copy will not affect the original.

[Video description begins] *She highlights the code in Line 7.* [Video description ends]

So we've seen two ways to make list copies so far, using the assignment operator which creates a shallow copy, and using the copy function in the copy module which created a deep copy.

[Video description begins] *The code in Lines 2 and 7 display on the right-hand side of the interface.* [Video description ends]

What if our lists were more complicated? What if we had a nested list structure? Here, old_list is a list of lists. Lists have been embedded in an outer list.

[Video description begins] *She enters and executes the following lines of code in the next code cell: Line 1: old_list = [[1, 2, 3], [4,5,6], [7, 8, 9]] Line 2: old_list. The number 10 now appears within the square brackets after the word 'In' in the command prompt. She highlights the following output:[[1, 2, 3], [4,5,6], [7, 8, 9]]* . [Video description ends]

I'll first use the copy function in the copy module to create a copy of the old list which I assign to the variable new_list. When I print out the old_list and the new_list no changes have been made to either list they contain the exact same content.

[Video description begins] *She enters and executes the following lines of code in the next code cell: Line 1: new_list = copy.copy(old_list) Line 2: print("Old list:", old_list) Line 3: print("New list:", new_list). The number 11 now appears within the square brackets after the word 'In' in the command prompt. She highlights the following lines of output: Line 1: Old list: [[1, 2, 3], [4,5,6], [7, 8, 9]] Line 2: New list: [[1, 2, 3], [4,5,6], [7, 8, 9]].* [Video description ends]

I'll now access one of the elements in one of the inner list in the old_list variable. So I access the list at index position 1, and access the element at index position 1 in the inner list, which gives me the element 5.

[Video description begins] *She enters and executes the following code in the next code cell: old_list [1][1]. The number 12 now appears within the square brackets after the word 'In' in the command prompt. She highlights the following output: 5. The following code in Line 10 gets highlighted: [4, 5, 6].* [Video description ends]

Using the old_list variable, I'm going to update this element in the inner list to be the string Five rather than the number 5. Observe that the update has been made using the old_list variable. Now, if I print out the contents of the old_list and the new_list, you will find that both have been updated.

[Video description begins] *She enters and executes the following lines of code in the next code cell: Line 1: old_list[1][1] = 'Five' Line 2: print("old list:", old_list) Line 3: print("New list:", new_list). The number 13 now appears within the square brackets after the word 'In' in the command prompt. She highlights the following lines of output: Line 1: Old list:[[1, 2, 3], [4,'Five',6], [7, 8, 9]] Line 2: New list: [[1, 2, 3], [4, 'Five',6], [7, 8, 9]].* [Video description ends]

The copy function that we used where we passed in a nested list created a deep copy of the outer list, but a shallow copy of list that were nested within the outer list. Which is why changes to any of the inner nested list are reflected in both the old_list as well as the new_list.

I'll now access the inner list at index 2 using the new_list variable, and to that inner list I'll append the string Ten. Now, when I print out the original as well as the new_list, you'll find that they contain the same updates.

[Video description begins] *She enters and executes the following lines of code in the next code cell: Line 1: new_list[2].append('Ten') Line 2: print ('Original:', old_list) Line 3: print('Modified:', new_list). The number 13 now appears within the square brackets after the word 'In' in the command prompt. She highlights the following lines of output: Line 1: Original:[[1, 2, 3], [4,'Five',6], [7, 8, 9, 'Ten']] Line 2: Modified: [[1, 2, 3], [4, 'Five',6], [7, 8, 9, 'Ten']].* [Video description ends]

The copy function only created a shallow copy of all of the inner list, which means they reference the same portion in Python memory. Changes that we make to one inner list are reflected in both the original as well as the copy.

I'll perform one last operation to really drive this point home. I call old_list.append and append the values 10, 11, and 12 to the old_list. Notice that I'm appending to the outer list. And you'll see that the updates have been applied only to the old_list and not to the new_list.

[Video description begins] *She enters and executes the following lines of code in the next code cell: Line 1: old_list.append([10, 11, 12]) Line 2: print ("old list:", old_list) Line 3: print("New list:", new_list). The number 15 now appears within the square brackets after the word 'In' in the command prompt. She highlights the following lines of output: Line 1: Old list:[[1, 2, 3], [4,'Five',6], [7, 8, 9, 'Ten'],[10, 11, 12]] Line 2: New list: [[1, 2, 3], [4, 'Five',6], [7, 8, 9, 'Ten']]*. [Video description ends]

We had spoken earlier of the fact that the copy function creates a deep copy of the outer list, so changes that you make to the outer list will not be reflected in the copy. But any change that you make to any of the inner nested list will be reflected in both.

Performing Deep Copies of Lists

[Video description begins] *Topic title: Performing Deep Copies of Lists. Your host for this session is Janani Ravi*. [Video description ends]

In the last video, we saw how the assignment operator produces a shallow copy of a list, and the copy function produces a deep copy, but not for nested lists.

[Video description begins] *A jupyter notebook titled ListDeepCopy displays in the interface*. [Video description ends]

In this video, we'll see how we can create deep copies of lists even when nested lists are involved. Let's set up a list of names once again, Sara, David, Warner, and Sandy.

[Video description begins] *The host enters and executes the following lines of code in the first code cell: Line 1: names = ["Sara", "David", "Warner", "Sandy"] Line 2: names. The number 1 now appears within the square brackets after the word 'In' in the command prompt. She highlights the following output: ['Sara', 'David', 'Warner', 'Sandy']*. [Video description ends]

In the last video, we explored the copy function from the copy module that we imported. It turns out that lists themselves have a copy function. Let's create a copy of our original names list and place it in the new names variable.

[Video description begins] *She enters and executes the following lines of code in the next code cell: Line 1: new_names = names.copy() Line 2: print("Original:" , names) Line 3: print("Modified:", new_names)*. [Video description ends]

Print out the original list of names and the new names, we'll now see whether this is a deep or a shallow copy.

[Video description begins] *She then highlights the following lines of output: Line 1: Original: ['Sara', 'David', 'Warner', 'Sandy'] Line 2: Modified: ['Sara', 'David', 'Warner', 'Sandy']*. [Video description ends]

Let's take a look at the name in index position 1 in the new_names list. It's David.

[Video description begins] *She enters and executes the following code in the next code cell: new_names[1]. The number 3 now appears within the square brackets after the word 'In' in the command prompt. She highlights the following output : 'David'*. [Video description ends]

I'll now update the new_names list to have the name Ridley at index position 1. So I've changed it.

[Video description begins] *She enters and executes the following lines of code in the next code cell: Line 1: new_names[1] = 'Ridley' Line 2: print('Original:', names) Line 3: print('Modified:', new_names).*' [Video description ends]

Now when you print out the original list and the modified list, you'll find that the change that we made to the new_names list is not reflected in the original. New_names contains the name Ridley at index 1.

[Video description begins] *She then highlights the following lines of output:Line 1: Original:['Sara', 'David ', 'Warner', 'Sandy'] Line 2: Modified: ['Sara', 'Ridley', 'Warner', 'Sandy']. She also refers to the code in the code given in code cell 2 which appears on the right-hand side of the interface*. [Video description ends]

The original list continues to have David at the same index position. That means the copy function invoked on a list creates a deep copy. An exercise that you can try out on your own is to see whether the copy function creates a deep copy of a list which contains a nested list within it.

Let's import the copy module once again and use a copy function in there.

[Video description begins] *She enters the following code in the next code cell: import copy. The number 5 now appears within the square brackets after the word 'In' in the command prompt*. [Video description ends]

I'll first create a list of companies. This is the same list that we had worked with earlier containing the companies hackerearth, google, and facebook.

[Video description begins] *She enters and executes the following code in the next cell: Line 1: companies = ["hackerearth" , "google", "facebook"] Line 2: companies. The number 6 now appears within the square brackets after the word 'In' in the command prompt. The following output displays:['hackerearth' , 'google', 'facebook']*. [Video description ends]

I'm now going to create a deep copy of the companies list and assign it to the variable new_companies.

[Video description begins] *She enters and executes the following code in the next code cell: Line 1: new_companies = copy.deepcopy(companies) Line 2: print('Original:', companies) Line 3: print('Modified:', new_companies)*. [Video description ends]

In addition to the copy function, the copy module also offers a deep copy function. So I've passed in a list of companies and I store the deep copy in the new_companies variable. I'll print out both of these. They contain these exact same elements at this point in time.

[Video description begins] *She then highlights the following output: Line 1: Original: ['hackerearth' , 'google', 'facebook'] Line 2: Modified: ['hackerearth' , 'google', 'facebook']*. [Video description ends]

I'll now print out the element at index 1 in the new_companies list. It is google.

[Video description begins] *She enters and executes the following code in the next cell: new_companies[1]. The number 8 now appears within the square brackets after the word 'In' in the command prompt. The following output displays: 'google'. The code in code cell 7 appears on the right-hand side of the interface*. [Video description ends]

Let's update the value of this particular element. I've assigned new_companies of 1 to be equal to microsoft. This is a change in the new_companies list.

And if you print out the original as well as the modified list, you'll find that the original list still contains google at index 1, the new list contains microsoft. The original remains unaffected with the change that we made to the new list.

[Video description begins] *She enters and executes the following code in the next cell: Line 1: new_companies[1] ='microsoft' Line 2: print('Original:', companies) Line 3: print('Modified:', new_companies). The number 9 now appears within the square brackets after the word 'In' in the command prompt. She then highlights the following lines of output: Line 1: Original: ['hackerearth' , 'google', 'facebook'] Line 2: Modified: ['hackerearth' , 'microsoft', 'facebook']*. [Video description ends]

The new list is indeed a deep copy. This means that the copy.copy function and the copy.deepcopy function both produced deep copies of non-nested lists. We did not have other lists nested within them.

Now let's check to see how deep copy works with nested lists. I'm going to set up an old list, which is a nested list, a list of lists.

[Video description begins] *She enters and executes the following code in the next cell: Line 1: old_list= [[1, 2, 3], [4,5,6], [7, 8, 9]] Line 2: old_list. The following output displays: [[1, 2, 3], [4,5,6], [7, 8, 9]]* . [Video description ends]

I'll now invoke the copy.deepcopy function, pass in the old list, and assign the result to the new_list variable. Let's print out both lists. They contain the exact same values at this point in time.

[Video description begins] *She enters and executes the following code in the next code cell: Line 1: new_list = copy.deepcopy(old_list) Line 2: print("Old list:", old_list) Line 3: print("New list:", new_list). She highlights the following output: Line 1: Old list: [[1, 2, 3], [4,5,6], [7, 8, 9]] Line 2: New list: [[1, 2, 3], [4,5,6], [7, 8, 9]].* [Video description ends]

Let´s check out what is at index [1][2] in the old list. We access the list at index 1 and access the element at index 2 within the inner list. This is the value 6.

[Video description begins] *She enters and executes the following code in the next cell: old_list [1][2]. She highlights the following output: 6. The code in the code cell 10 appears on the right-hand side of the interface*. [Video description ends]

I'll now update the element in this position in the old list. Old_list of [1][2] = Six, the string rather than 6 the value. And when we print out the values of the old list and the new list, you will find that the old list has been updated to have the string six, but the new list remains unchanged.

The new list that we created using the copy.deepcopy function created deep copies of all of the nested lists as well. Under the hood, this means that the old_list variable and the new_list variable share no memory. Changes that you make to one will not be reflected in the other.

[Video description begins] *She enters and executes the following code in the next cell: Line 1: old_list[1][2] = 'Six' Line 2: print("old list:", old_list) Line 3: print("New list:", new_list). The following output displays: Line 1: Old list:[[1, 2, 3], [4, 5,'Six',], [7, 8, 9]]Line 2: New list: [[1, 2, 3], [4,5,6], [7, 8, 9]].* [Video description ends]

If you have a simple list that is a non-nested list, such as the companies list that we set up earlier, there are different ways you can use to make a deep copy of this list.

[Video description begins] *She enters and executes the following code in the next cell: companies. The following output displays: ['hackerearth' , 'google', 'facebook']*. [Video description ends]

In fact, the array slicing operation that we have studied earlier also makes deep copies.

[Video description begins] *She enters and executes the following code in the next cell: Line 1: new_companies = Companies(:) Line 2: new_companies*. [Video description ends]

We slice, the entire range of the companies list new_companies, is a deep copy of the original list.

[Video description begins] *The following output displays: ['hackerearth' , 'google', 'facebook']*. [Video description ends]

If I make a change to an element in the new_companies list, let's say I set new_companies[0] to skillsoft, and I take a look at the original as well as the modified list, the change skillsoft is not reflected in the original list. The new list is a deep copy of the original, and the new list was created using the slicing operation.

[Video description begins] *She enters and executes the following code in the next cell: Line 1: new_companies[0] = 'skillsoft' Line 2: print('Original:', companies) Line 3: print('Modified:', new_companies). The following output displays: Line 1: Original: ['hackerearth' , 'google', 'facebook'] Line 2: Modified: ['skillsoft' , 'google', 'facebook'].* [Video description ends]

Creating Shallow and Deep Copies of Tuples

[Video description begins] *Topic title: Creating Shallow and Deep Copies of Tuples. Your host for this session is Janani Ravi.* [Video description ends]

Now that we've understood how shallow and deep copies work with lists, let's move on to tuples.

[Video description begins] *A jupyter notebook titled TupleCopy displays in the interface.* [Video description ends]

Now, you'll find that working with tuples is rather simple, because tuples are immutable. Let's now create a tuple of fruits, apple, banana, and cherry.

[Video description begins] *The host enters and executes the following code in the first code cell: Line 1: fruits= ("apple", "banana", "cherry") Line 2: fruits. The number 1 now appears within the square brackets after the word 'In' in the command prompt next to this input box. The following output displays below: ('apple' , 'banana', 'cherry').* [Video description ends]

Once we have this original tuple assigned to the fruits variable, let's use the assignment operator to assign this tuple to the new_fruits variable. Let's print out both of the tuples, and you'll find that their contents are exactly the same. This is not a surprise.

[Video description begins] *The host enters and executes the following code in the Input Code Cell 2: new_fruits = fruits print("Original:", fruits) print("Modified:", new_fruits). The print result reads: Original: ('apple' , 'banana', 'cherry')Modified: ('apple' , 'banana', 'cherry')* [Video description ends]

Using the new_fruits variable, let's access the field at index 2, and this gives us the value cherry.

[Video description begins] *The host enters and executes the following code in the Input Code Cell 3: new_fruits[2]. The Output Code Cell displays the code: 'cherry'* [Video description ends]

This is the fruit at index 2 in the new_fruits variable. Let's now try and assign a new value at this index position, and because tuples are immutable, you'll find that Python raises an error.

[Video description begins] *She enters and executes the following code in the next cell: new_fruits[2]= "orange". The code in code cell 2 appears on the right-hand side of the interface. The following error displays: TypeError: 'tuple' object does not support item assignment.* [Video description ends]

Just like strings are immutable, tuples are immutable as well. Which means when you use the assignment operator to assign a tuple to a new variable both variable will refer to the same tuple in memory.

But because the tuple is immutable and cannot be updated once created, this does not matter. So even though the assignment operator creates a shallow copy, you can't change the tuple anyway. So there's nothing to really watch out for while you're coding. But what if your tuple contains a list within it?

Here we have a tuple of fruits. And notice that one of the fields within this tuple is a list which contains the fruit banana and cherry.

[Video description begins] *She enters and executes the following code in the next cell: Line 1: fruits= ("apple", ["banana", "cherry"], "blackcurrant") Line 2: fruits. The following output displays: ('apple' , ['banana',*

'cherry'], 'blackcurrant'). [Video description ends]

Lists are mutable even if tuples are not, and we have seen that lists within tuples can be updated. I'll use the assignment operator once again and assigned fruits to the new_fruits variable.

[Video description begins] *She enters and executes the following code in the next cell: Line 1: new_fruits = fruits Line 2= print("Original:", fruits) Line 3: print("Modified:", new_fruits). A section of the code in the print result of Input Code Cell 5 displays on the right-hand side of the interface*. [Video description ends]

When you use the assignment operator with complex data types, this always creates a shallow copy. Let's print out the original and the new_fruits contents. They are exactly the same at this point in time.

[Video description begins] *The following output displays: Line 1: Original: ('apple' , ['banana', 'cherry'], 'blackcurrant'). Line 2: Modified: ('apple' , ['banana', 'cherry'], 'blackcurrant')*. [Video description ends]

Using the new_fruits tuple, let's access an element within the list contained in that tuple.

[Video description begins] *She enters and executes the following code in the next cell: new_fruits[1][0]. The following output displays: 'banana'*. [Video description ends]

At index 1 within the tuple, we have a list and we access the index 0 within that list to get the fruit banana. Lists are mutable, even when contained within tuples. That means we can update the fruit within the inner list. I'm going to set it to be orange. Let's now take a look at the original tuple and the new tuple and see what their contents are.

[Video description begins] *The host enters and executes the following code in the next cell: Line 1: new_fruits[1] [0] ='orange' Line 2: print("Original:", fruits) Line 3: print("Modified:", new_fruits). The following output displays: Line 1: Original: ('apple' , ['orange', 'cherry'], 'blackcurrant'). Line 2: Modified: ('apple' , ['orange', 'cherry'], 'blackcurrant'). A section of the code in the print result of Input Code Cell 5 displays on the right-hand side of the interface*. [Video description ends]

The update that we made to the list using the new_fruits variable is reflected in the original tuple, fruits, as well. The assignment operator created a shallow copy of this tuple, and the nested list within the tuple is basically the same, whether we use the new_fruits variable or the fruits variable. I'll now use the copy function within the copy module.

[Video description begins] *She enters and executes the following code in the Input Code Cell 9: import copy* [Video description ends]

I'm going to set up a tuple, bikes, which contains a nested list within this tuple. You can see that Kawasaki and Ducati are the elements of a list within the outer tuple, which also contains the bikes Honda, Suzuki, and Triumph.

[Video description begins] *She enters and executes the following code in Input Code Cell 10 : Line 1: bikes = ('Honda', 'Suzuki', 'Triumph', ['Kawasaki', 'Ducati']) Line 2: bikes. The following output displays: ('Honda', 'Suzuki', 'Triumph', ['Kawasaki', 'Ducati']) Line 2: bikes*. [Video description ends]

I'll now use the copy function within the copy module to make a copy of the bikes tuple and assign this copy to the new_bikes variable. Let's print out the original bikes tuple and the new_bikes tuple, and let's see whether this copy function produced a deep or a shallow copy.

[Video description begins] *She enters and executes the following code in Input Code Cell 11: Line 1: new_bikes = copy.copy(bikes) Line 2: print('Original:', bikes) Line 3: print('Modified:', new_bikes). The following output displays: Line 1: Original: ('Honda', 'Suzuki', 'Triumph', ['Kawasaki', 'Ducati']) Line 2: Modified: ('Honda', 'Suzuki', 'Triumph', ['Kawasaki', 'Ducati'])*. [Video description ends]

Let's take a look at a list element within the new_bikes tuple. At field index 3 with a new_bikes, we have a list and the zeroth position of the list contains the bike Kawasaki.

[Video description begins] *She enters and executes the following code in the Input Code Cell 12: new_bikes[3][0]. The following output displays: 'Kawasaki'*. [Video description ends]

I'll now update the element in this inner list, remember that the list is mutable, to be Dodge, and I'm going to print out the original tuple and the new_bikes tuple. And as you might expect, the copy function did not create a deep copy of the nested inner list.

[Video description begins] *She enters and executes the following code in Input Code Cell 13: Line 1: new_bikes[3][0] = 'Dodge' Line 2: print('Original:', bikes) Line 3: print('Modified:', new_bikes). The following output displays: Line 1: Original: ('Honda', 'Suzuki', 'Triumph', ['Dodge', 'Ducati']) Line 2: Modified: ('Honda', 'Suzuki', 'Triumph', ['Dodge', 'Ducati']). The Input Code in Cell 11 displays on the right side of the interface*. [Video description ends]

The element Dodge is present not only within the inner list of the new_bikes tuple but also in the inner list of the original bikes tuple. This was a shallow copy of a tuple containing an inner list. If you want a deep copy you need to use the copy.deepcopy function.

[Video description begins] *She enters and executes the following code in the Input Code Cell 14: Line 1: new_bikes = copy.deepcopybikes) Line 2: print ('Original:', bikes) Line 2: print('Modified:', new_bikes)*. [Video description ends]

Here, we make a deep copy of the original bikes tuple and assign it to new_bikes. To start off with, the original bikes and new_bikes tuple contain the same values.

[Video description begins] *The following output displays: Line 1: Original: ('Honda', 'Suzuki', 'Triumph', ['Dodge', 'Ducati']) Line 2: Modified: ('Honda', 'Suzuki', 'Triumph', ['Dodge', 'Ducati'])*. [Video description ends]

Let's now update an element of the list contained within the new_bikes tuple. I'm going to change the first element of the list from Dodge to be Harley Davidson.

[Video description begins] *She enters and executes the following code in Input Code Cell 15: Line 1: new_bikes[3][0] = 'Harley Davidson' Line 2: print('Original:', bikes) Line 3: print('Modified:', new_bikes)*. [Video description ends]

And if you now see the contents of the original tuple and the modified tuple, you'll find that the modified tuple is a complete deep copy, including the inner list. Changes that we made to the list within the new_bikes tuple haven't affected the original bikes tuple.

[Video description begins] *The following output displays: Line 1: Original: ('Honda', 'Suzuki', 'Triumph', ['Dodge', 'Ducati'])Line 2: Modified: ('Honda', 'Suzuki', 'Triumph', ['Harley Davidson', 'Ducati'])*. [Video description ends]

So if you want to create a deep copy of a tuple which may contain nested lists within it use the copy.deepcopy function.

Creating Shallow Copies of Dictionaries

[Video description begins] *Topic title: Creating shallow Copies of Dictionaries. Your host for this session is Janani Ravi*. [Video description ends]

In this video, we'll see how we can make shallow copies of Python dictionaries. I'm going to initialize a dictionary named months here, which contains the keys jan, feb, march, and april.

And the corresponding values are the numbers associated with these months in a year jan being the 1st month and april being the 4th.

[Video description begins] *A jupyter notebook titled DictionaryShallowCopy displays in the interface. She refers to the following code in Input Code Cell 1: Line 1: months = { 'jan ' : 1, ' feb ': 2 , ' march ': 3 , 'april ': 4 } Line 2: months*. [Video description ends]

I'll now use the assignment operator to create a copy of this dictionary, new_months = months. The variable new_months will now contain the exact same contents that is present in the months variable the same dictionary.

[Video description begins] *She enters and executes the following code in Input Code Cell 2: Line 1: new_months = months Line 2: print("Original:" , months) Line 3: print("Modified:", new_months). The following output displays: Line 1: Original: { 'jan ' : 1, ' feb ': 2 , ' march ': 3 , 'april ': 4 } Line 2: Modified: { 'jan ' : 1, ' feb ': 2 , ' march ': 3 , 'april ': 4 }*. [Video description ends]

I'll now look up the value corresponding to the key feb in the new_months dictionary and the value is 2.

[Video description begins] *She enters and executes the following code in Input Code Cell 3: new_months['feb']. The Output Code Cell reads: 2*. [Video description ends]

Using the assignment operator with dictionaries just like with list, creates a shallow copy, and that's exactly what we'll see here. I'm going to update the value associated with the month feb to be FEB in all caps rather than a number.

And you'll find that this update is reflected in the original months dictionary as well.

[Video description begins] *She enters and executes the following code in Input Code Cell 4: Line 1: new_months['feb']= 'FEB' Line 2: print("Original:" , months) Line 3: print("Modified:", new_months). The print results for Input Code Cell 4 reads: Line 1: Original: { 'jan ' : 1, ' feb ': FEB, ' march ': 3 , 'april ': 4 Line 2: Modified: { 'jan ' : 1, ' feb ': FEB, ' march ': 3 , 'april ': 4 }. The Input Cell Code 2 appears and is highlighted on the right-hand side of the interface*. [Video description ends]

Just like with other complex data types, both variables refer to the same dictionary in Python memory. So changes that you make using one variable will be reflected in the other variable as well, as under the hood they're exactly the same. Now that we understand how variable assignments work using dictionaries, let's move on to using the copy function from the copy module.

[Video description begins] *She enters the following code in Input Code Cell 5: import copy* [Video description ends]

I'm going to set up a new dictionary to work with. The employees dictionary contains the names of employees and the year in which they started in an organization.

[Video description begins] *The Input Code Cell 6 reads: Line 1: employees = { "Alison ": 2007, "Bart" : 2008, "Cathy" : 2009, "Dan" : 2010 }Line 2: employees. The Output Code Cell 6 reads: { 'Alison ': 2007, 'Bart' : 2008, 'Cathy' : 2009, 'Dan' : 2010 }*. [Video description ends]

I'll pass this employees dictionary in as an input argument to the copy.copy function and store the result in the employees_copy dictionary. The contents of both of these dictionaries are exactly the same when we start off.

[Video description begins] *She enters and executes the following code in Input Code Cell 7: Line 1: employees_copy = copy.copy(employees) Line 2: print (employees) Line 3: print (employees _ copy) . The print*

result for Input Code Cell 7 reads: {'Alison':2007, 'Bart':2008, 'Cathy': 2009, 'Dan':2010}Line 2: {'Alison':2007, 'Bart':2008, 'Cathy': 2009, 'Dan':2010}. [Video description ends]

Let's look up the value associated with the employee named Alison, and you can see that she started in the year 2007.

[Video description begins] *She enters and executes the following code in Input Code Cell 8: employees ["Alison"]. The Output Code Cell 8 reads: 2007. The Input Code for cell 6 appears on the right-hand side of the interface*. [Video description ends]

Dictionaries in Python are mutable, I'll update the value associated with the key Alison to be 2005, Alison actually joined in 2005 and not in 2007. And you'll find that this change is not reflected in the original employees dictionary.

Employees still has Alison associated with the year 2007, whereas employees_copy has Alison associated with the year 2005. The copy function created a deep copy of the dictionary.

[Video description begins] *She enters and executes the following code in Input Code Cell 9: Line 1: employees_copy["Alison"] = 2005 Line 2: print (employees) Line 3: print (employees _ copy) . The print result for Input Code Cell 7 reads: {'Alison':2007, 'Bart':2008, 'Cathy': 2009, 'Dan':2010} Line 2: {'Alison':2005, 'Bart':2008, 'Cathy': 2009, 'Dan':2010}*. [Video description ends]

But what if this dictionary contained a nested, complex data type, such as a list? Remember a list is mutable, and here we have a dictionary called original dictionary. And you can see that the value associated with the key four is a list of numbers. Lists are mutable, this is something that you need to remember.

[Video description begins] *She enters and executes the following code in Input Code Cell 10: orig_dict = {'one':1, 'two':2, 'three':3, ' four ' : [1, 2, 3, 4.0] }orig_dict. The Output Code Cell 10 displays the code: {'one':1, 'two':2, 'three':3, ' four ' : [1, 2, 3, 4.0] }*. [Video description ends]

Let's now create a copy of this dictionary using the copy.copy function. I pass in the original dictionary and store the copy in the new dictionary variable. The contents of the original as well as the new dictionary, to start off with, are exactly the same, this is what we expect.

[Video description begins] *She enters and executes the following code in Input Code Cell 11: Line 1: new_dict = copy.copy(orig_dict) Line 2: print ('Original:', orig_dict) Line 3: print ('Modified:', new_dict). The print results read: Original:{'one':1, 'two':2, 'three':3, ' four ' : [1, 2, 3, 4.0] }Modified: {'one':1, 'two':2, 'three':3, ' four ' : [1, 2, 3, 4.0] }*. [Video description ends]

Let's now access one of the elements of the list within this dictionary. Using the new dictionary, I look up the key four and access the list element at index 3 and this will give me the value 4.

[Video description begins] *She enters and executes the following code in Input Code Cell 12: new _ dict [' four '] [3] . The Output Code Cell 12 reads: 4.0. The Input Code in Cell 11 displays on the right- hand side of the interface*. [Video description ends]

I'll now update this particular value within the new dictionary. I'm going to set this list element to be the string FOUR rather than the number 4. Let's print out the original dictionary and the new dictionary, and you'll see that the same change is available in both.

[Video description begins] *She enters and executes the following code in Input Code Cell 13: new_dict ['four'] [3]='FOUR'print ('Original:', orig_dict) print ('Modified:', new_dict). The print results read: Original: {'one':1,'two':2, 'three':3, ' four ' : [1, 2, 3, 'FOUR'] }Modified: {'one':1, 'two':2 , 'three':3, ' four ' : [1, 2, 3, 'FOUR'] }*. [Video description ends]

Using the copy function from the copy module did not produce a deep copy of a list nested within the outer dictionary. The copy function produces a deep copy of the outer dictionary, as we've seen earlier, but a shallow copy of any nested complex data types within the outer dictionary.

Let's say we access the list using the new_dict variable, so I'll look up the value corresponding to the key four and append the element 5. This change is reflected in both the original dictionary as well as the new dictionary.

[Video description begins] *She enters and executes the following code in Input Code Cell 14: new_dict ['four'].append(5)print ('Original:', orig_dict) print ('Modified:', new_dict). The print results read: Original: {'one':1,'two':2, 'three':3, ' four ' : [1, 2, 3, 'FOUR',5] }Modified: {'one':1, 'two':2 , 'three':3, ' four ' : [1, 2, 3, 'FOUR',5] }.* [Video description ends]

What if we have an inner dictionary nested within an outer dictionary? Here is our original dictionary. Corresponding to the key three, we have another dictionary containing the keys zero and one. This is an example of having a dictionary within a dictionary.

[Video description begins] *She enters and executes the following code in Input Code Cell 15: orig_dict = {'one':1, 'two':2, 'three':{'zero':0, 'one':1}, 'four':4} orig_dict. The Output Code Cell 15 reads:{'one':1, 'two':2, 'three':{'zero':0, 'one':1}, 'four':4}.* [Video description ends]

Let's take a look at how the copy function works in this case. I'm going to copy over the original dictionary and assign the result to the new dictionary variable. Both of these variables now have the same content.

[Video description begins] *She enters and executes the following code in Input Code Cell 16: new_dict = copy.copy(orig_dict) print (orig_dict) print (new_dict). The print results read:{'one':1, 'two':2, 'three':{'zero': 0, 'one':1}, ' four ' :4}{'one':1, 'two':2, 'three':{'zero': 0, 'one':1}, ' four ' :4}.* [Video description ends]

Let's look up a value in the inner dictionary. So new_dict of three, zero, will give us the value 0, that's what is currently present. I'm now going to update this value.

[Video description begins] *She enters and executes the following code in Input Code Cell 17: new _ dict['three '] [' zero ']. The Output Code Cell reads: 0.* [Video description ends]

I'm going to set this value to the string ZERO, rather than the number 0. And let's print out the original as well as the new dictionary, and you'll find that the same change is available in both.

[Video description begins] *She enters and executes the following code in Input Code Cell 18: new_dict ['three '] [' zero '] = ' ZERO'print (orig_dict) print (new_dict). The print results read:{'one':1, 'two':2 , 'three':{'zero': 'ZERO', 'one':1}, ' four ' :4}{'one':1, 'two':2, 'three':{'zero': 'ZERO', 'one':1}, ' four ' :4}. The Input Code in Cell 16 displays on the right-hand side of the interface.* [Video description ends]

This is proof, once again, that the copy function does not produce a deep copy of any nested data types, complex data type. You can try this one more time with a different example. Here we have a set nested within a dictionary corresponding to the key three, we have a set containing the values 0, 1, and 2.

[Video description begins] *She enters and executes the following code in Input Code Cell 19: orig_dict = { 'one':1, 'two':2, 'three': { 0, 1, 2},'four' :4} orig_dict. The Output Code Cell 19 reads: {'one': 1, 'two': 2, 'three': { 0, 1, 2},'four' :4} .* [Video description ends]

Once again, I use the copy.copy function to create a copy of this dictionary and assign it to the new_dict variable. The contents are exactly the same to start off with.

[Video description begins] *She enters and executes the following code in Input Code Cell 20: new_dict = copy.copy(orig_dict) print ('Original:', orig_dict) print ('Modified:', new_dict). The print results read: Original: {'one':1, 'two':2, 'three':{0, 1, 2}, ' four ' :4}Modified: {'one':1, 'two':2, 'three':{0, 1, 2}, ' four ' :4}.* [Video description ends]

The original value corresponding to the key three is the set containing the elements 0, 1, and 2.

[Video description begins] *She enters and executes the following code in Input Code Cell 21: new_dict[' three] . The Output Code Cell 21 reads:{ 0, 1, 2}.* [Video description ends]

Let's make a change to this inner set, I'm going to add the element 3 to this inner set. And this change that I've made to the inner nested set is reflected in both the original dictionary as well as the new dictionary.

[Video description begins] *She enters and executes the following code in Input Code Cell 22: new_dict[' three] .add(3) print ('Original:', orig_dict) print ('Modified:', new_dict). The print result reads: Original:{'one':1, 'two':2, 'three': {0, 1, 2,3}, ' four ' :4}Modified :{'one':1, 'two':2, 'three': {0, 1, 2,3}, ' four ' :4}.* [Video description ends]

Nested complex data types are not deep copied when you use the copy function, this just drives the point home.

Creating Deep Copies of Dictionaries

[Video description begins] *Topic title: Creating Deep Copies of Dictionaries. Your host for this session is Janani Ravi.* [Video description ends]

So how do we create deep copies of dictionaries which contain other complex data types nested within it? That's exactly what we'll study here in this video.

[Video description begins] *A blank jupyter notebook titled DictionaryDeepCopy displays in the interface.* [Video description ends]

Once again, I'll create a dictionary that we've seen before, the months dictionary. This is a dictionary containing the names of months and their corresponding values. Dictionaries also contain a copy function, just like list.

[Video description begins] *She enters and executes the following code in Input Code Cell 1: months = { 'jan ' : 1, ' feb ': 2 , ' march ': 3, 'april ': 4 }months. The Output Code Cell 1 reads: { 'jan ' : 1, ' feb ': 2 , ' march ': 3, 'april ': 4 }* [Video description ends]

I'm going to make a copy of the months dictionary and assign this copy to the variable new_months. We haven't made any changes yet. To start off with, both of these dictionaries contain the same key value pairs.

[Video description begins] *She enters and executes the following code in Input Code Cell 2: new_months = months.copy() print("Original:" , months) print("Modified:", new_months). The print results Input Code Cell 2 reads: Original: { 'jan ' : 1, ' feb ': 2 , ' march ': 3 , 'april ': 4 } Modified: { 'jan ' : 1, ' feb ': 2 , ' march ': 3 , 'april ': 4 }* [Video description ends]

Let's now take a look at the value associated with the feb key in the new_months dictionary. This is the value 2.

[Video description begins] *She enters and executes the following code in Input Code Cell 3: new_months['feb']. The Output Code Cell reads: 2* [Video description ends]

This is what I'm going to update. I'm going to set the value of February to be equal to the string FEB in all capital letters. Let's look at the original dictionary and the copy. And you'll find that the change that we made to the copy is not reflected in the original dictionary.

The value associated with the string feb in the new_months dictionary is the string in all uppercase, FEB. Whereas in the original dictionary, it's still the number 2. For a simple dictionary that contains no complex nested data types within it, the copy function on the dictionary creates a deep copy of the dictionary.

[Video description begins] *She enters and executes the following code in Input Code Cell 4: new_months['feb']= 'FEB' print("Original:" , months) The print results for Input Code Cell 4 reads: Original: { 'jan ' : 1, ' feb ': 2, ' march ': 3 , 'april ': 4Modified: { 'jan ' : 1, ' feb ': FEB, ' march ': 3 , 'april ': 4 }* [Video description ends]

We'll now use the copy module to create a deep copy of our dictionary. And you'll see that this deep copy works when the dictionary contains nested complex data types as well.

[Video description begins] *She enters the following code in Input Code Cell 5: import copy* [Video description ends]

Let's set up a simple dictionary here, comprising of employees and the year that they started in an organization. This is a simple dictionary with no nested complex types.

[Video description begins] *The Input Code Cell 6 reads: employees = { "Alison ": 2007, "Bart " : 2008, "Cathy" : 2009, "Dan" : 2010 }employees. The Output Code Cell 6 reads: { 'Alison ': 2007, 'Bart' : 2008, 'Cathy' : 2009, 'Dan' : 2010 }* [Video description ends]

I'm now going to use the copy.deepcopy function, pass in employees to this deepcopy function, and create a copy of the original dictionary. When we first create the copy, the contents of the original dictionary and its copy are exactly the same.

[Video description begins] *She enters and executes the following code in Input Code Cell 7: employees_copy = copy.deepcopy(employees) print (employees_copy) . The print result for Input Code Cell 7 reads: {'Alison':2007, 'Bart': 2008, 'Cathy': 2009, 'Dan':2010} {'Alison':2007, 'Bart':2008, 'Cathy': 2009, 'Dan':2010}* [Video description ends]

Let's take a look at the value associated with the key Alison. It is the year 2007. I'm now going to update this value in the employees dictionary.

[Video description begins] *She enters and executes the following code in Input Code Cell 8: employees ["Alison"]. The Output Code Cell 8 reads: 2007.* [Video description ends]

I'm going to set the value to be 2011. Let's say Alison joined in the year 2011. If you look at employees and the copy of the dictionary, you will find that the changes that we made to the employees dictionary are not reflected in the copy.

[Video description begins] *She enters and executes the following code in Input Code Cell 9: employees["Alison"] = 2011 print (employees) print (employees _ copy) . The print result for Input Code Cell 9 reads: {'Alison':2011, 'Bart':2008, 'Cathy': 2009, 'Dan':2010} {'Alison':2007{'Alison':2007, 'Bart':2008, 'Cathy': 2009, 'Dan':2010} {'Alison':2005* [Video description ends]

The copy is indeed a deep copy. But what if the original dictionary contained a nested list? You can see the value corresponding to the key four is a list of numbers. Here is our original dictionary.

[Video description begins] *She enters and executes the following code in Input Code Cell 10: orig_dict = {'one':1, 'two':2, 'three':3, ' four ' : [1, 2, 3, 4.0] }orig_dict. The Output Code Cell 10 displays the code: {'one':1, 'two':2, 'three':3, ' four ' : [1, 2, 3, 4.0] }* [Video description ends]

I'm now going to use the copy.deepcopy function to create a deep copy of this original dictionary, and assign it to the new_dict variable. In this new dictionary, look up the list corresponding to the key four,

[Video description begins] *She enters and executes the following code in Input Code Cell 11: new_dict = copy.deepcopy(orig_dict) print ('Original:', orig_dict) print ('Modified:', new_dict). The print results read: Original:{'one':1, 'two':2, 'three':3, ' four ' : [1, 2, 3,4.0] }Modified: {'one':1, 'two':2, 'three':3, ' four ' : [1, 2, 3, 4.0] }* [Video description ends]

and find the element at index three in this list. It is the number 4.

[Video description begins] *She enters and executes the following code in Input Code Cell 12: new _ dict [' four '] [3] . The Output Code Cell 12 reads: 4.0. The Input Code in Cell 11 displays on the right- hand side of the interface*. [Video description ends]

I'm going to update this particular element to be the string FOUR, rather than the number 4. And let's print out the contents of both dictionaries, the original as well as the copy. And the result here should show you that the deep copy function created a deep copy of the nested list as well.

The new dictionary contains the string FOUR in the inner list. Whereas the original dictionary continues to have the floating point number 4 in the inner list.

[Video description begins] *She enters and executes the following code in Input Code Cell 13: new_dict ['four'] [3]='FOUR'print ('Original:', orig_dict) print ('Modified:',new_dict). The print results read: Original: {'one':1,'two':2, 'three':3, ' four ' : [1, 2, 3, 4.0] }Modified: {'one':1, 'two':2 , 'three':3, ' four ' : [1, 2, 3, 'FOUR'] }* [Video description ends]

I'll now append a new element to this inner list in the new dictionary. And you'll find that this change applies only to the inner list within the new dictionary, and not in the original dictionary. The new dictionary is indeed a deep copy.

[Video description begins] *She enters and executes the following code in Input Code Cell 14: new_dict ['four'].append(5)print ('Original:', orig_dict) print ('Modified:', new_dict). The print results read: Original: {'one':1,'two':2, 'three':3, ' four ' : [1, 2, 3, 4.0] }Modified: {'one':1, 'two':2 , 'three':3, ' four ' : [1, 2, 3, 'FOUR',5] }* [Video description ends]

We'll find that the deep copy function creates a deep copy of all nested complex data types. Here, once again, we associate a nested dictionary within our outer dictionary.

[Video description begins] *She enters and executes the following code in Input Code Cell 15: orig_dict = {'one':1, 'two':2, 'three':{'zero':0, 'one':1}, 'four':4} orig_dict. The Output Code Cell 15 reads:{'one':1, 'two':2, 'three':{'zero':0, 'one':1}, 'four':4}* [Video description ends]

I'm now going to create a deep copy of the original dictionary and assign it to the new_dict variable. They start off exactly the same.

[Video description begins] *She enters and executes the following code in Input Code Cell 16 new_dict = copy.deepcopy(orig_dict) print (orig_dict) print (new_dict).The print results read:{'one':1, 'two':2, 'three': {'zero': 0, 'one':1}, ' four ' :4}{'one':1, 'two':2, 'three':{'zero': 0, 'one':1}, ' four ' :4}* [Video description ends]

Let's look up a particular value in this nested inner dictionary corresponding to the key zero.

[Video description begins] *She enters and executes the following code in Input Code Cell 17: new _ dict['three '] [' zero ']. The Output Code Cell reads: 0* [Video description ends]

The value is the number 0. I'll now update this value to be the string ZERO, where the string ZERO is all in uppercase. If you print out the contents of the original dictionary and the new dictionary, you'll find that the change is reflected only in the new dictionary, where we performed the update. The original dictionary remains unchanged.

[Video description begins] *She enters and executes the following code in Input Code Cell 18: new_dict ['three '] [' zero '] = ' ZERO'print (orig_dict) print (new_dict). The print results read:{'one':1, 'two':2 , 'three':{'zero': 0, 'one':1}, ' four ' :4}{'one':1, 'two':2, 'three':{'zero': 'ZERO', 'one':1}, ' four ' :4}.* [Video description ends]

Let's take a look at one last example of a nested complex data type within a dictionary. Here we have a set corresponding to the key three within our original dictionary.

[Video description begins] *She enters and executes the following code in Input Code Cell 19: orig_dict = { 'one':1, 'two':2, 'three': { 0, 1, 2},'four' :4} orig_dict. The Output Code Cell 19 reads: {'one': 1, 'two': 2, 'three': { 0, 1, 2},'four' :4}* [Video description ends]

I'm going to create a deep copy of this original dictionary, assign it to the new_dict variable. The contents are exactly the same to start off with. Let's take a look at the set corresponding to the key three.

[Video description begins] *She enters and executes the following code in Input Code Cell 20: new_dict = copy.deepcopy(orig_dict) print ('Original:', orig_dict) print ('Modified:', new_dict). The print results read: Original:{'one':1, 'two':2, 'three':{0, 1, 2}, ' four ' :4}Modified: {'one':1, 'two':2, 'three':{0, 1, 2}, ' four ' :4}* [Video description ends]

The set contains the elements 0, 1, and 2. I'm now going to append a new element to the set using the new dictionary variable. We'll now compare the original as well as the deep copy, and we'll find that only the copy has been modified.

[Video description begins] *She enters and executes the following code in Input Code Cell 21: new_dict[' three] . The Output Code Cell 21 reads:{ 0, 1, 2}* [Video description ends]

Changes to a nested complex data type within the copy did not affect the original dictionary.

[Video description begins] *She enters and executes the following code in Input Code Cell 22: new_dict[' three] .add('THREE') print ('Original:', orig_dict) print ('Modified:', new_dict). The print result reads: Original: {'one':1, 'two':2, 'three': {0, 1, 2,3}, ' four ' :4}Modified :{'one':1, 'two':2, 'three': {0, 1, 2,'THREE', ' four ' :4}* [Video description ends]

Creating Shallow and Deep Copies of Sets

[Video description begins] *Topic title: Creating Shallow and Deep Copies of Sets. Your host for this session is Janani Ravi.* [Video description ends]

We'll now round out our discussion of shallow and deep copies of complex data types by taking a look at sets. I'm going to instantiate a new set here, called set_1. This is a set of popular programming languages, C++, PHP, Java, and Python.

[Video description begins] *A jupyter notebook titled SetCopy displays in the interface. The Input Code Cell 1 contains a code that reads: set _1 = { 'c++', 'php', ' java', 'python' }set_1 The Output Code Cell 1 reads: { 'c++', 'java', ' php', 'python' }* [Video description ends]

I'll now use the assignment operator to set the value of set_1 to the variable set_2. We haven't made any updates or changes yet, which means that both of these sets contain the same elements.

[Video description begins] *She enters and executes the following code in Input Code Cell 2: set_2 = set_1 print("Original:", set_1) print ("Modified:", set_2). The print result reads: Original: {'java', 'python', 'c++', 'php'}Modified:{'java', 'python', 'c++', 'php'}* [Video description ends]

I'll now modify the contents of set_2 by adding a new language to set_2, I'll add in SQL. And you will find that this modification affects both the original set, as well as the copy. As with other complex data types, when you use the assignment operator, both variables refer to the same set in memory.

[Video description begins] *She enters and executes the following code in Input Code Cell 3: set_2.add('sql') print("Original:", set_1) print ("Modified:", set_2). The print result reads: Original: { 'c++', 'php',' java',*

'python','sql'}Modified:{ 'c++', 'php',' java', 'python','sql'} [Video description ends]

So changes that you make using one variable are reflected in the contents of the other variable as well. I'm going to make one more change using the set_2 variable, I'm going to pop an element at random from this set.

The element c++ happens to have been popped from my set. It's possible that when you're coding the element might be different. If you now print out the contents of set_1 as well as set_2, you'll find that they are the same.

[Video description begins] *She enters and executes the following code in Input Code Cell 4: set_2.pop(). The Output code Cell reads: 'c++'* [Video description ends]

The element c++ is not present in either of the sets. Both of these sets refer to the same set under the hood in Python memory. If you remember our earlier discussion on sets, elements of a set cannot be accessed using index values, because sets have no intrinsic ordering.

[Video description begins] *She enters and executes the following code in Input Code Cell 5: print("Original:", set_1) print ("Modified:", set_2). The print result reads: Original: { 'php', 'java', 'python','sql'}Modified: { 'php', 'java', 'python','sql'}* [Video description ends]

So accessing elements of a set using index values or trying to assign new values to a set using indexes are both not allowed.

[Video description begins] *She enters and executes the following code in Input Code Cell 6, which reads: set _ 2 [2]. An error message displays which reads: TypeError: 'set' object does not support indexing*. [Video description ends]

[Video description begins] *She enters and executes the following code in Input Code Cell 7: set _ 2 [2]= 'ruby'. An error message displays which reads: TypeError: 'set' object does not support indexing*. [Video description ends]

We've also seen that sets cannot contain lists nested within the set. Sets can contain within them only immutable data structures such as tuples. A list is mutable which is why you can't enclose a nested list within a set.

[Video description begins] *She enters and executes the following code in Input Code Cell 8: set_1 = {'c++', 'php', [' java ', ' python '] ,' sql'} set_1. An error message displays which reads: TypeError:unhasable type: 'list'* [Video description ends]

A tuple is an immutable data structure which means you can have set contain tuples. You can use the assignment operator to assign set_1 to set_2.

[Video description begins] *She enters and executes the following code in Input Code Cell 9: set_1 ={ 'c++','php', (' java', 'python'),'sql'}set_1. The Output Code Cell 9 reads: {(' java', 'python'),'c++', 'php','sql'}* [Video description ends]

We've already seen earlier that this just creates a shallow copy of set_1. Let's import the copy module and see if we can create a different kind of copy.

[Video description begins] *She enters and executes the following code in Input Code Cell 10: set_2= set_1 print("Original:", set_1) print ("Modified:", set_2). The print result reads: Original: : {('java', 'python'), 'c++', 'php'}Modified:{('java', 'python'), 'c++', 'php'}* [Video description ends]

I'll instantiate a teacher's set comprising of names of teachers,

[Video description begins] *She enters the following code in Input Code Cell 11: import copy She enters and executes the following code in Input Code Cell 12: teachers _ set = { " Ava ", " Mia ", "Jacob ", " Daniel " }. teachers_set. The Output Code Cell 12 reads: {' Ava ',' Daniel ','Jacob', 'Mia' }*. [Video description ends]

Ava, Daniel, Jacob, and Mia. Use the copy function to create a copy of this teachers set and assign it to the new_teachers_set variable. Both of these sets are exactly the same. I'll now make a change to the original teachers set,

[Video description begins] *She enters and executes the following code in Input Code Cell 13: new_teachers_set =copy . copy (teachers _ set)print(teachers_set) print(new_teachers_set). The print result reads: { 'Jacob', 'Daniel', 'Mia', 'Ava'}{ 'Jacob', 'Daniel', 'Mia', 'Ava'}* [Video description ends]

I'll add in the name Emma. And this change is reflected only in the teachers_set, the copy remains unaffected.

[Video description begins] *She enters and executes the following code in Input Code Cell 14: teachers _ set.add("Emma") print(teachers_set) print(new_teachers_set). The print results read: {'Ava', 'Emma', 'Daniel', 'Jacob', 'Mia'}{ 'Jacob', 'Daniel','Mia','Ava'} . The Input Code in Cell 13 displays on the right-hand side of the interface*. [Video description ends]

I'll now remove the name Ava from the copy, new_teachers_ set.remove("Ava"). And this change is reflected only in the copy, the original set still contains the name Ava. The copy function has created a deep copy of the set. Changes made to the copy are not reflected in the original and vice versa.

[Video description begins] *She enters and executes the following code in Input Code Cell 15: new_teachers _ set.remove("Ava") print(teachers_set) print(new_teachers_set). The print results read: {'Ava', 'Emma', 'Daniel', 'Jacob', 'Mia'}{ 'Jacob', 'Daniel','Mia'} .* [Video description ends]

Let's instantiate the teachers_set to four names once again, Ava, Mia, Jacob, and Daniel. This time,

[Video description begins] *She enters and executes the following code in Input Code Cell 16: new_teachers_set = { " Ava ", " Mia ", "Jacob ", " Daniel " }teachers _set. The Output Code Cell 16 reads: {' Ava ',' Daniel ','Jacob', 'Mia' }*. [Video description ends]

I'll use the copy.deepcopy function to create the new teachers_set. We'll find that with the set, the copy and the deepcopy, works in exactly the same way.

[Video description begins] *She enters and executes the following code in Input Code Cell 17: new_teachers_set =copy.deepcopy(teachers_set) print(teachers_set) print(new_teachers_set). The print result reads: { 'Jacob', 'Daniel', 'Mia', 'Ava'}{ 'Mia', 'Daniel', 'Jacob', 'Ava'}* [Video description ends]

I've added Emma to the teachers_set, the change is reflected only in the teachers_set and not in the deepcopy.

[Video description begins] *She enters and executes the following code in Input Code Cell 18: teachers _ set.add("Emma") print(teachers_set) print(new_teachers_set). The print results read: {'Ava', 'Emma', 'Daniel', 'Jacob', 'Mia'}{ 'Mia', 'Daniel', 'Jacob', 'Ava'}* [Video description ends]

And when I remove Ava from the teachers_set, the change once again is reflected only in the teachers_set and not in the copy.

[Video description begins] *She enters and executes the following code in Input Code Cell 19: teachers _ set.remove("Ava") print(teachers_set) print(new_teachers_set). The print results read: {'Emma', 'Daniel', 'Jacob', 'Mia' }{ 'Mia', 'Daniel','Jacob', 'Ava'} .* [Video description ends]

Exercise: Shallow and Deep Copies

[Video description begins] *Topic title: Exercise: Shallow and Deep Copies. Your host for this session is Janani Ravi*. [Video description ends]

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.