**w04-01-MutabilityAndFunctions-2024**

0:02  
In this morning's lecture, we're looking at mutability and functions.

0:08  
Now you can think of a variable as referring to a space in memory.

0:12  
If you look at the picture at the bottom right of our slide here, we have a space in memory, and then we've defined a variable and that is associated with this memory space.

0:22  
Now we can put a value into that memory space, or we can refer to the beginning of that memory space.

0:31  
So when we're referring to a memory space, the beginning of one, we might talk about a reference to the memory.

0:38  
OK, so a variable corresponds to a space in memory and we can refer to the value in the allocated memory or we can refer to the position using a so-called reference or pointer to that memory.

0:54  
It's an address, rather like we have an address with maybe a street name and a post code.

1:00  
A computer has an address for all memory allocations.

1:05  
So this reference or pointer refers to the beginning of the memory where the variable is.

1:12  
Now the size of the memory space is dependent on the type of the variable.

1:16  
For example, if it's an integer, that will be a fixed memory space.

1:22  
An integer can only be so big, it can't be infinitely large.

1:25  
If it becomes bigger than its memory space, strange things will happen, such as that it may become negative.

1:31  
Now if you have a different kind of variable that will be a different size of memory space.

1:38  
Now when we assign a value or make an assignment, we use the assignment operator or equals sign.

1:46  
Now the variables that are immutable and the variables that are mutable behave differently when we use the assignment operator.

1:57  
If we have an immutable variable, when we use the assignment operator, the value is copied into a separate memory space.

2:07  
Whereas when we have a mutable variable, we're not actually copying the value into the other memory space, we are just copying a reference to the original memory block into that other variable.

2:21  
So our second variable is really referring to the same space in memory.

2:27  
Now let's have a look at immutable types.

2:30  
First, immutable variables O.

2:32  
As we've already said, if we have an immutable variable, values are copied from 1 variable to another.

2:39  
The second variable, if we define one, occupies a different space in memory, and we can then copy and update the value in the other memory space.

2:50  
So here we have a little graphic at the bottom right and the hand of the screen, you can see that we've defined the variable.

2:56  
It's got a memory space, this orange thing here.

2:59  
And then if we say say for example, X = y and our first variable is Y and as new one is X, then the value that's in Y is copied into the variable Y.

3:15  
And that variable Y has oh, sorry, in this case it's an X.

3:20  
The the the other variable has a separate memory location.

3:23  
So the assignment operator is copying the value and there are two different memory spaces.

3:29  
So that means once we've copied the value, you can see it's a separate piece of memory.

3:33  
We can then update the value in here and it's not going to affect the value in the other memory space.

3:38  
Now examples of immutable types are integer, float, string, boolean, and tuple.

3:46  
Now, if you have a mutable variable, when we use the assignment operator, we're actually assigning a reference to the beginning of the memory space.

3:57  
Now mutable variables are generally used or they exist in Python And other languages where we have a larger block of memory.

4:06  
So we don't really want to be copying this large block of memory all the way around our programme all the time because that would require more CPU and memory.

4:15  
It would just generally slow our programme down.

4:18  
So instead we rely on copying the reference to this memory around.

4:23  
Now we have to be a bit careful in that when we design our programme, we need to think clearly where are we going to use the data?

4:30  
Are we going to update the data or not?

4:32  
Otherwise we could end up updating the data somewhere else when we think we've created a separate, separate variable.

4:40  
But it's not actually a separate variable, it's just a reference to the same memory block.

4:45  
So this kind of scheme of working that is immutable variable is present with lists, dictionaries, and objects.

4:53  
We haven't got to objects yet, but we will come back to them when we talk about object oriented programming.

4:59  
These three things are generally going to hold more data than a simple variable, so they're bigger pieces of memory and therefore copying them around all over the place is not a good idea.

5:15  
Now, objects actually can be defined to be immutable occasionally.

5:20  
Hardly ever, but occasionally you can do it.

5:23  
So for the Python gurus, I'm just saying on whole objects are normally mutable, but you can actually define any mutable object if you so wish.

5:35  
OK, so the references to the original memory are assigned to a new variable, and then the variable refers to the original instance.

5:45  
So if we modify that variable using the other variable, the original is changed.

5:53  
Now we can copy the data into a separate memory location, but we have to be explicit about this.

5:59  
We end up using the copy operator or we can use the copy module to do this.

6:07  
Now before we have a look at functions, which is the next part, I'm going to have a look at some code.

6:12  
So let's go back and have a look at some code.

6:16  
So let's go to our first example here.

6:18  
Now in the first example here we're looking at a immutable variable.

6:24  
We have a value Y.

6:26  
Now this variable Y is an integer.

6:29  
You can see we've assigned 5 to it.

6:32  
And then at line 17 we are assigning the value to the variable X.

6:39  
And what has happened here is that we have copied the value into X and then this word pass is used simply because it allows me to stop the debugger.

6:53  
It doesn't actually do anything, but it's an executable line.

6:57  
OK, so we're going to go ahead and debug this so you can see it working.

7:01  
So let's say run debug current Python file, then the debugger will start and we will stop at the break point, which is currently at line 14.

7:27  
Here we are, we're at line 8 where I've left a previous break point incidentally.

7:34  
So let's hop over one step, one forward.

7:38  
So you can see that we've got a variable X and it is an integer, so it contains the value 10.

7:44  
And remember what we've already said that this is an immutable variable.

7:48  
So the memory space that's associated with X is not the same as the memory space that's associated with Y.

7:55  
So if we hop forward, so what's happened is we've now got two variables.

8:01  
So the variable Y contains the value 10.

8:05  
Now if we update the value that's in the variable Y, it won't update the value that's in the variable X because they are separate memory spaces.

8:15  
So let's hop over.

8:18  
So now you can see that, yes, the value of Y is five, and the value of X is still 10.

8:24  
Now, of course, we can update the value X by copying a value back into it, as we're doing here at line 17.

8:32  
And there we go.

8:34  
So now let's have a look at another example.

8:37  
We're going to have a look at something that's mutable.

8:41  
Now, as you said previously, lists and dictionaries are mutable.

8:45  
That means that when we use the assignment operator, it's actually copying a reference to the original memory block.

8:52  
So let's add a break point here at line 7 and we're going to run and debug.

9:05  
Here we are, line 7.

9:06  
Let's hop forward.

9:08  
So now you can see that we've got a list and it has one element.

9:12  
So the single element contains the value 10.

9:17  
Now when we say P equals values, what's going to happen is that they're reference to this original memory, which is this list of one.

9:28  
One value is going to be assigned to P It won't create a new list, it will just assign a reference.

9:36  
So if we hop forward, we now see that we've got P and it's also a list, and it also contains 10.

9:43  
But if we use P and we access the 1st element of P and assign 5, what's going to happen is because we've copied a reference to the original memory line 10, it will update the same list that values points to.

10:01  
So let's just have a hop forward and there you can see that P has been updated as we requested at line 13.

10:10  
But values has also been requested as because at line 10P is given a reference to the beginning of the memory that values refers to as well.

10:21  
So when we update it through P it's also been updated as far as we can see through values.

10:28  
OK, so there we are.

10:30  
That's simple explanation of what is a immutable and mutable variable and the differences.

10:37  
Now, in different languages, the choices of how this behaves, whether it's mutable or immutable, is slightly different.

10:45  
But you will see something, for example in C# or Java that there is a choice about how mutability works.

10:55  
OK, so we stop there.

10:57  
So let's go back to our slides.

11:02  
Now we're skipping onto functions.

11:04  
The reason why we're talking about functions and mutability together is when we pass something into a function, we are essentially using the assignment operator.

11:14  
We'll get to that in a minute.

11:16  
Now, a function is essentially a piece of Python programme in this case, and it can contain one or more lines.

11:26  
The reason for creating a function is so that we have a shorter piece of programme.

11:32  
We can debug it and understand it.

11:34  
And also we might want to repeatedly use a piece of programme.

11:39  
For example, there might be a particular function maybe to update some data or put something on the screen, or we want to use it in several places that programme.

11:47  
So the two reasons are the testability.

11:50  
We break the programme down into functions.

11:52  
And the other one is if we want to repeatedly use the code, OK, let's hop over to the next slide.

12:02  
So a function can have zero or more input parameters.

12:08  
Now you might have a function, for example, and you call it and it returns a random number.

12:14  
And you don't actually want to pass in anything.

12:16  
You're just going to call the function and it returns some random number, some number from somewhere.

12:21  
So yeah, you can have functions with have no inputs, but you can have one or more input parameters.

12:28  
Now, when you're programming, the things that you pass into a function are normally called a parameter or an argument.

12:36  
They are things that we pass in.

12:37  
And as I've already said, when we pass something in, it's, it's synonymous or equivalent to using the assignment operator.

12:46  
We're assigning a value to a variable that's inside the function.

12:51  
Now, the input parameters can be complicated.

12:54  
So for example, we could pass in a list of dictionaries or an object or anything else.

12:58  
It doesn't matter.

12:59  
We pass in something or or nothing if we wish, and the function can either return nothing at all, which in Python is None, or it can return a value and now this value again Can be some kind of complicated data structure.

13:14  
You can return a list or a dictionary and integer in other programming languages.

13:19  
Typically you only return the simple types like immutable variables, but in Python you can return more complicated things because they are treated mutably and you're actually copying a reference.

13:32  
You do have to be careful, as I'm going to mention in a minute, because if you update the data that's been returned and it's mutable, then you're actually updating the copy that's inside the function itself.

13:44  
OK, so let's think about functions and immutable variables.

13:50  
Now if we have immutable variables and we're going to pass them into a function that is they are input parameters.

13:57  
What happens is the value is copied into a variable that's defined inside the function.

14:05  
So that inside that function if we do some changes to the other variable, nothing happens to the variable that the storing a value outside the function.

14:19  
If we return immutable variables from our function, then likewise the value is copied or assigned into the other variable outside of the function and so if we change it, the variable value inside the function is not changed.

14:41  
Now if we use mutable variables and we pass them in as input parameters, what happens is if we update them inside the function, they're actually referring to the same memory as we saw before we were talking about functions and so if we update them inside the function, they remain updated outside the function too.

15:03  
Likewise, if we return a mutable variable as a return value, then if we update them outside the function, that will Causeway what's inside the function to be updated.

15:16  
Now, this is more important when you're dealing with object orientation than if you're just creating a list, for example, inside a function and returning it, that a new list is going to be created each time inside that function.

15:30  
So it's fairly harmless to return it and nothing strange will happen.

15:34  
But if you are using objects, and we'll come back to this, if you return a list and you access it, update it, you are going to have updated or accessed something which belongs to the object.

15:48  
OK, so let us go and have a look at some examples.

15:52  
We're going to go back to our code and we'll look at the next one here.

16:00  
Now, before we go on to functions and the function examples, I'm going to mention briefly copying.

16:08  
Now, as I said earlier, if you have something immutable, you can copy it and there are two ways of copying it.

16:16  
You can use the the variable and then you can say dot copy and this will create a shallow copy or you can use the copy module as I'm doing so in this example.

16:28  
Now the copy module here gives you 2 functions.

16:31  
It gives you copy and it also gives you deep copy.

16:35  
Now the copy, what it does is it copies the top layer of the mutable thing so that we have a separate copy of the top layer.

16:46  
Now if you had a mutable thing inside a mutable thing, the second layer, so for example, a list inside a list is not copied.

16:55  
So although you had two outside lists that are different, the reference that is inside to the other list isn't copied.

17:04  
OK, let's have a look at the simple shallow copy.

17:08  
So we're going to use a break point again and we're going to run and debug and you'll see that we're creating another list and it's called values and it contains one element that has the number 10 in it.

17:30  
And now we are going to use copy.

17:33  
So you notice we've got the assignment operator, but this time, because we've used copy, it is going to create a separate instance.

17:43  
There you go.

17:43  
So we've copied it.

17:44  
So now you can see we've got P and values, and now we're going to assign the value 5 as we did previously to the first element of the list P Having done that, you can see that the value 5 is in P, but values remains unchanged.

18:04  
And that's because we explicitly copied the mutable variable and so they are two separate pieces of memory.

18:12  
Now you don't want to do this too much because I said previously if we copy around large box of memory, that will use computer time and memory.

18:21  
But occasionally you do want to make a copy because maybe you want to genuinely separate data structure for some reason, OK, but don't do this too much.

18:30  
So let's stop that now.

18:33  
We're going to go on and have another look at another example.

18:37  
This is using the deep copy operator, and we've got the copy module again, but this time we're going to create a deep copy.

18:45  
Now, the deep copy implies that all layers of the mutable thing are copied to make a separate thing.

18:55  
If you want a sort of a mixture of shallow and deep, you're going to have to implement it yourself as a function, which you might need in occasions perhaps.

19:05  
OK, so we've got a list values, which is actually containing a list which has got the number 10 in it.

19:13  
Now we can then access the element to the first list and the element to the second list inside it and assign the value 5.

19:21  
And because we have a deep copy, you'll see that we don't update the value that's in the original.

19:28  
So let's give it a go, and we can then refer back to a shallow copy to demonstrate the difference.

19:36  
So we're gonna create a list of a list.

19:40  
And now we're creating a deep copy, meaning it's copying all layers, which in this case are two.

19:46  
So there you go, we'll hop forward.

19:48  
So now you can see we've got two variables, P and values.

19:53  
We update the value that is in the first element of the internal list and we give it 5.

20:00  
And look, the value that is in values is not updated.

20:06  
OK, so I'm gonna change the programme to make the point of what the deep copy is doing.

20:10  
So we're gonna stop that and we're gonna change this back to copy, save this, and we're gonna run it again.

20:20  
And if we go, so this time, remember, we're copying the outer layer of the list values.

20:27  
We're not copying this inner list, which means although we have two separate lists, the inner one actually has a reference to the same list.

20:36  
So let's hop forward how we're creating a copy.

20:41  
Now we're going to assign a value to the first element of the inner list.

20:45  
Oh, look, the value of the other list has been updated.

20:49  
And that's not surprising because the inner list is actually the same thing.

20:54  
It's a reference to the same memory, correct?

20:57  
So I'm going to put that back to how it was before.

21:02  
OK, so now we're going to have a look at some function examples.

21:06  
So we'll take that break point away.

21:10  
So what can we see when we have a function?

21:13  
Well, functions have in front the word deaf.

21:16  
You can think of deaf as being a shortening for define.

21:20  
So we're going to define a function and then they have a name.

21:24  
So here I've called my function fun, which is not a very good name.

21:28  
It's a good idea to give functions and variables meaningful names.

21:35  
So if you have a function that is going to save some data, you might want to call it save\_data a meaningful name.

21:43  
Likewise, unless you're doing some kind of maths, you probably don't want to stick with variables called X&Y.

21:49  
You might want to give them more meaningful names like file\_name or or something else like that.

21:55  
Anyway that you can see, the construct is death, the function name, and then we have rounded brackets.

22:01  
And inside the rounded brackets we have the parameters that we're going to pass into the function.

22:08  
You can see at the end of the line we have a colon, and the contents of the function is indented in the same way that the contents of a for loop is indented.

22:18  
So what we have here is our function.

22:21  
We've got 2 input parameters X&Y.

22:24  
Now it's a good idea to have a comment inside of our function where we say what the function does.

22:31  
We can then have any number of lines inside our function and then at the end of the function we can return something or not.

22:38  
In this case we are returning the value that is in the immutable variable Z.

22:45  
Now print is not the same as return.

22:49  
What print does is it takes the value and prints it onto the screen, whereas return takes the value and returns it back to the programme outside.

23:03  
So anyway, what's happening here is we call the function.

23:06  
We pass in two numbers, so 10 and three.

23:09  
Those two values go into the variable X&Y respectively.

23:14  
We've now got these two variables X&Y which are defined inside the function.

23:19  
They're not defined outside the function.

23:23  
We've then got a calculation so Z = X + y and we finally return the value Z.

23:31  
So let's give it a go and debug and see what happens.

23:34  
So I'm gonna put a break point there.

23:36  
We're gonna run and debug, right?

23:45  
So here we are at the function.

23:47  
We can now if we were debugging, we can step over, meaning we skip over the function and we are going to collect the result or we can step into which is what this thing does.

24:00  
So we're going to step into which we can click on this button or press F11.

24:05  
So inside the function, you can see we've now got 2 local variables X&Y and they contain the value 10 and three.

24:14  
So the values here were assigned to this separate piece of memory associated with X&Y.

24:21  
In the same way we're assigning a immutable variable value with the equals assign.

24:27  
The assignment operator that's happened where we passed these in as parameters.

24:32  
So now we can update them.

24:33  
Yes, we can print.

24:34  
You can see the print out down here.

24:36  
And finally, we can return the value.

24:40  
Now you can think of as a function as being a separate piece of programme off to one side.

24:45  
And so the main programme calls it, it sends the parameters off to the function.

24:50  
The function does some stuff, and then it returns back a value.

24:54  
So let's go forward and we can see that the value is returned from here, line 14.

25:01  
And then it's assigned to the variable R And there you can see we've got 13, which is what we expected.

25:08  
It's just 10 + 3.

25:10  
OK, let's have a look at another example.

25:14  
This one is going to use a global variable.

25:19  
Now, global variables can be useful sometimes where we want to have a variable that sometimes may be used as a cache inside a Python file.

25:31  
We don't want to use this too often because it can confuse ourselves.

25:35  
Why is a global variable confusing?

25:37  
Well, because we're not actually passing it into the function, so we can change it in different functions in different places, and then it can become really confusing as to how the data are being treated and where they're being treated.

25:52  
So don't use globals very much, and probably only for specific use cases like maybe a cache for a file content or or something like that.

26:02  
OK, So what have we done?

26:04  
We've defined a global variable at the top.

26:07  
Now again, a function here has two input parameters X&Y.

26:13  
Now inside the function, we need to tell it that this is in fact a global.

26:19  
And so Python won't panic to say, hang on a minute, are going to create a new variable G there.

26:27  
It will just use the existing G If we omitted global G here, what it would do is it would create a local variable G and assign these values to it.

26:38  
So we say global G We then go ahead and update the value of G and what happens is the value will be updated outside.

26:48  
So the other thing we've done in this function is you can see we've got no return statement, which implies that this function will return None.

26:57  
So let's go ahead and run this.

26:59  
We'll run it down to the break point and we'll see what we have.

27:08  
So there we go, We're down to the break point, and you can see that R does in fact contain None.

27:15  
And you can also see that the variable G does in fact contain 13, which is just the sum of these two values.

27:25  
OK, so that's global.

27:28  
So let's go back and have another look at a different example.

27:33  
Now in this example, we're passing in again two values.

27:38  
You can see we've got two variables here, K&P.

27:42  
Now, these are integer variables, so that means they're immutable, and they will contain a value in a separate memory space to something we copy the value to.

27:55  
So here we are, we've got K&P.

27:58  
We're passing them to fun.

28:00  
So the value that's in K is copied into a new variable X, which is a separate memory block, and a new variable Y, which again is a separate memory block from P here.

28:14  
So when we update X&Y, they're updated in the local function.

28:19  
They're not updated outside here because the K&P values were copied in, the references weren't copied in.

28:28  
So let's go ahead and do it.

28:29  
We'll put a breakpoint there and we'll run down to line 20.

28:44  
OK, so we've got two variables, K&P.

28:49  
Now we're gonna hop inside the function.

28:52  
You can see that we've now got local variables, X&Y.

28:55  
They've received the values, but when we update these two, so we're gonna update them.

29:00  
You can see that update here, the local variables.

29:03  
We then leave the function.

29:05  
The local variables have gone away because the function's finished, and you can see that K&P still contain 10 and three.

29:12  
That's because, as we've said already, we copied the values.

29:16  
We didn't copy the references into this function.

29:19  
OK, let's stop that.

29:21  
We'll go to our last example.

29:25  
Now.

29:25  
In this example, we're using a list and we've already said a list is a mutable type.

29:34  
And So what are we doing here?

29:36  
At line 15, we're defining an empty list.

29:39  
We're then passing the list in as a parameter.

29:42  
So as I've already said, if you pass something as a parameter, it behaves as if you've used the assignment operator, meaning it is going to send a reference to the original list into our function.

29:56  
So this thing, my values will contain the reference, the same memory.

30:02  
So when we update it inside the function, which is what we've done here, it will remain updated.

30:08  
You can see this function doesn't return anything.

30:11  
Now it's a choice.

30:12  
You can have functions where you use mutable things, you pass them in, and everybody understands that that function is going to put some data in the mutable thing.

30:25  
You could make the name of the mutable thing a bit more meaningful, like it could be data read or something like that.

30:33  
So instead of returning values, you could update a mutable thing.

30:37  
But you have to make it clearer to a developer what the function is doing.

30:41  
Don't just pass in some input data and have things be changed and leave it to the imagination of another developer as to what's going on.

30:50  
You have to be a bit more careful.

30:51  
Anyway, in this example, we are gonna go ahead and see what happens with a mutable variable.

31:00  
So run and debug.

31:05  
We'll run down to line 18.

31:18  
So here we are.

31:18  
We've got a variable values and it is holding an empty list at the moment.

31:25  
And then we're gonna go into the function and you can see that now we've got my values.

31:31  
Remember this is referring to the same memory as values is referring to.

31:36  
It's an empty list.

31:38  
So we're gonna hop over and we've left the function, but notice the values here has got a list which contains the number 7, and there is an illustration of how mutable variables are updated inside functions.

31:59  
OK, let's go back to our slides.

32:03  
So basically that's the end of our presentation.

32:07  
Now it's a good idea when you are looking at mutable or immutable variables to just try it out.

32:15  
Take some of these programmes we've been discussing, change some of the values, run it again, and verify to yourself how it's working.

32:23  
As I've already said, mutability and immutability behave slightly differently in different programming languages, so there's a different choice in C and C++ and Java and all these other languages.

32:36  
So you probably just going to have to read about what is mutable and how mutability is assumed in different languages.

32:43  
Anyway, hope you've enjoyed this lecture and I look forward to seeing you in the lab.