**w07-01-ObjectOrientedDesign-2024**

0:00  
Great.

0:01  
So this week we're carrying on with our discussion of last week.

0:05  
Last week, you remember we started talking about object oriented programming and this time we're going to do a little bit on UML, which helpfully somebody mentioned last week where you'll see the UML class diagram next to a piece of code.

0:21  
Now where did we get to last week?

0:23  
Last week we stopped with inheritance.

0:27  
I know we went on and discussed a little bit of other stuff.

0:29  
We effectively stopped with inheritance.

0:31  
Now somebody helpfully mentioned super.

0:33  
So I've actually put in super into the example this time.

0:37  
So the function here, super finds the base class for the derive class.

0:44  
So this line here calls the constructor of map position class.

0:52  
Ah, thank you very much.

0:54  
Let's see what's wrong here.

1:02  
We'll try that again, see if that's better, right.

1:08  
So yeah, where we got to last time we were looking at inheritance.

1:13  
And here's the Super function to call the constructor in the base class.

1:17  
Now what I was talking about in the lab as well was public data members or public functions.

1:24  
So that's what we're going to discuss next.

1:27  
That is public, private, and protected.

1:30  
Now, protected in terms of how you use it is well defined for object oriented programming.

1:37  
However, it doesn't actually work as it should do in Python at the moment.

1:41  
And I don't quite know why that is, so I'm just going to describe how it should work and you can try out how it actually works.

1:50  
Anyway, the idea of public, private and protected is important for inheritance.

1:55  
So if something is public, then it is accessible outside the class.

2:02  
So if you have two objects of the same class, they are the same class.

2:06  
But if you're sitting outside that object and you try to get a data member or member function, you can as long as it is public.

2:16  
If it is private, you can't.

2:18  
And I've got an example of trying to do that in a minute and it basically tells you it doesn't know what the variable is, so it can't see that variable.

2:27  
Now the idea of protected is only important for inheritance.

2:32  
So a protected variable is available as if it is private within the derive class, and if it's defined in the base class, it's accessible there as well.

2:45  
So a protected variable behaves for something outside of the class, a bit like a private variable or private member function.

2:57  
The only time you want to really use it is if you have an inheritance.

3:01  
Now in Python, for some reason it doesn't exactly work how it should work, but I'm defining it how it should work.

3:09  
And with respect to other programming languages.

3:12  
All the object oriented programming languages I know such as C#, Java, C++, they all follow this same standard of public, private and protected.

3:24  
OK, So what is private?

3:26  
Private basically means that you cannot directly access that thing, whether it is a member function or a data member from outside of the class.

3:36  
And why is that sometimes useful?

3:38  
Well, it's sometimes useful because you want to so-called encapsulate things in a class.

3:44  
So you want to hide some internal data from somebody outside so that, for example, you could have a class which has a particular state and you don't want somebody else changing the state inside.

3:57  
Maybe it's a configuration setting or something like that.

4:00  
So you can hide it by making it private.

4:04  
All right.

4:05  
So in Python, strangely, they use underscores.

4:09  
Most languages you can see explicitly written as public, private, or protected.

4:15  
So there's a keyword and you'll see that in something like C++ or C# or Java or whatever.

4:23  
But in Python they use underscores.

4:25  
So if you have no underscores such as this variable here name, or you have no underscores at the front such as this function, public function, by default a Python data member such as this one or member function such as this one is public, meaning you can call it from outside the class and that's what you've been seeing in the lab.

4:48  
Most of the time we're just dealing with public member functions or public data members.

4:53  
Now if you want to make it protected, you put a single\_at the beginning.

4:57  
If you want to make it private, you put 2 underscores at the beginning.

5:00  
So the same is true for the functions.

5:03  
You can see here.

5:03  
This is a protected function and a private function.

5:06  
Now the function doesn't have to be called private function or protected function.

5:10  
I've just named it that just as a kind of reminder to you.

5:18  
Double\_private, single\_protected.

5:19  
Now these functions which have double\_at the front and double\_at the back are particular.

5:25  
They are a pattern which is for a particular operation, OK.

5:31  
So this is an exception and there are a few of these and we'll come to them.

5:35  
In general, if you make your own function which starts with a double\_it is private OK.

5:42  
SO, does anybody ask any comments about sorry any, questions about public or private in general why?

5:52  
Would you want to use them otherwise they're?

5:54  
Going to go on yeah, So, public right, because?

5:59  
You can book the stores and stuff but, it doesn't actually work you.

6:03  
Just indicate the service.

6:06  
So what will happen is if you're sitting inside this class, you could have another function where let's say you've got this private name.

6:16  
You could have inside another member function self dot\_\_private name and you'd be absolutely fine.

6:21  
But if you were outside that class and you said let's create a my class thing, and then you try to ask lowercase my class dot private name equals something, you can't do that.

6:35  
It won't let you do it.

6:36  
That's the point of private.

6:37  
We'll see it in a minute.

6:38  
Yep, Yes, yes, that's right.

6:43  
The constructor always has this pattern.

6:46  
In other programming languages, the constructor function has the same name as the class name.

6:51  
So you'll see that in C or C or Java, it's always the same as the class name.

6:57  
But for whatever reason, Python decided to use this in it.

7:01  
I was saying to somebody in the lab, it's a little bit like, you know, having a plug where you plug in, you've got 3 pins, you've got to plug into it and that's it.

7:10  
Somebody's defined what it has to be.

7:12  
And you just have to use the standard.

7:15  
And then when you create an object, it will call this because it knows it's to call it.

7:21  
It's just, it's a design choice of the people who made the programming language.

7:25  
Anybody else?

7:28  
All right, let's go on.

7:31  
So accessor and mutator functions.

7:33  
Now you'll see these a lot in object oriented programming.

7:37  
And the idea is that you are providing access to data inside the object as you want to, meaning you can control how the data changed.

7:48  
Now here's an example.

7:50  
We've got the my class, we've instantiated it.

7:53  
So we've got an object here in this variable M And then what we can do is we can call the set name function.

8:02  
So you notice the set name function is public.

8:06  
A mutator or an accessor is public.

8:09  
The mutator is the one that changes it, and the accessor is the one that gets the value.

8:14  
And you notice these two functions are operating on a private data member.

8:20  
So that's actually private.

8:22  
And this is private, so you can decide to, for example, emit the mutator.

8:29  
So that means you just remove this and then what will happen is you'll be giving somebody a read only access to the data that are in\_\_name because if it's private they can't, change it from outside so you.

8:44  
Can sometimes or you may sometimes want to load a lot of data up and then give people a read only access to it through an accessor anyway that.

8:54  
Is, the general pattern of accessor and mutator functions now in.

8:58  
A lot of programming languages that's fine in Python.

9:02  
It's Not such a good idea because the the processing overhead, if you do this a lot, is actually significant.

9:11  
So when we have a private data member or protected even, we have these excessors and mutators potentially to either get or set the data.

9:24  
So that's what they are.

9:25  
They're sometimes referred to as getters and setters, but the technical name is excessor and mutator.

9:32  
Now Python programmers don't tend to use them.

9:34  
Why?

9:35  
Because as I already said, there's a little bit of an overhead, meaning the computer is actually having to go through these extra function calls.

9:42  
Now if it's another programming language like Java or C#, the compiler tries to reduce that overhead.

9:50  
So when it runs, it's not actually as bad as it might be.

9:54  
So when you're making Python programmes as as long as you really, really, really don't need a private data member, don't use them.

10:03  
If you think you do need one for a good reason, OK, use it.

10:07  
But it has an overhead in terms of you have to then have a an excess of potentially.

10:14  
Yes, that's all I was going to say about that.

10:17  
OK, so that's protected private and public functions.

10:22  
Now an interesting concept, one which you have already seen in action, although you may not have noticed it, is operator overloading.

10:30  
Now what?

10:30  
What does this mean?

10:32  
An operator is a function that will potentially add things together.

10:38  
That's the plus operator.

10:40  
It could append something to another.

10:42  
That's also the plus operator, but it's for a string.

10:46  
You could have multiplication.

10:48  
You could have all sorts of other operators, like division.

10:52  
You could have a moduli operator.

10:54  
Anyway, if you've got an operator with a class, you can define the associated function to the operator.

11:02  
So when you multiply together 2 objects, it will do something.

11:06  
If you don't implement the operator in the class, it won't know what to do.

11:12  
The only exception to that is the string representation.

11:16  
It will do something if you don't implement this string operator in the class.

11:22  
Now this is this subject area should we say is not really covered in Java.

11:27  
Java has one operator overloading principle, which refers to strings, which operates in a similar manner to Python.

11:35  
However, other languages such as C or C do allow full operator overloading, meaning anything you can think of as an operator assignment, whatever you like, you can define as a function.

11:50  
So it's normally, yeah, we are conditional, we'll see that in a minute, rather than an assignment.

11:57  
OK, so operators that you often see conversion to string, you've got an object and you want to turn it into a text string.

12:05  
What should that look like?

12:08  
Comparison.

12:08  
So you've got 2 objects and they might contain complicated data and you want to check are they the same thing.

12:16  
Now if you don't implement that function, it doesn't do anything mathematical operations.

12:20  
So maybe sometimes operators should be in there to allow you to add objects together and then it should do something sensible.

12:28  
Now the caveat here is it's a piece of code you write, so if you really want to be devious you could have the divide operator removing a file.

12:39  
I would suggest you don't do that.

12:40  
Try to use the operators to do things which are sensible to another developer.

12:47  
All right.

12:47  
One of the benefits of this is it potentially improves the code structure.

12:52  
Instead of having to carry around bits of function calls to turn things into text strings or whatever, you can put it into the class and then just use it wherever you want it so it's potentially more readable.

13:07  
OK, and what do they all look like?

13:09  
They all have these double underscores.

13:11  
There's a double\_at the front and a double\_at the back just like, our friend the constructor, the init, function so we've.

13:21  
Got double\_repr double\_so this.

13:24  
Is for string representation now there.

13:27  
Are in fact 2 string representation operators in Python the Second.

13:32  
One is for a readable string representation this is.

13:35  
More for debugging the top.

13:38  
One is the one I'm using So throughout.

13:40  
The module I mostly REFER to the repr function.

13:44  
Why?

13:45  
Because it will recursively return the string operations.

13:49  
It's actually more useful.

13:51  
So imagine you've got a list with objects inside, which then has another list inside.

13:56  
So when you ask for the thing to be turned into a string, it will go through each of the layers and then return this string.

14:04  
The STR won't do that.

14:06  
It's as I say for a readable string representation, so we're effectively ignoring that for the rest of the module.

14:13  
So yeah, less used you can have comparison.

14:16  
So you're asking to do 2 objects equal each other.

14:20  
So that's with the EQ.

14:21  
You can have the northeast function as well.

14:24  
So that's not equal.

14:25  
So it's the exclamation mark equal.

14:29  
You can have addition which is add, so you add 2 objects together and that could be used for example to add the data members together.

14:37  
If you're into maths, you could think of I've got 2 vectors, I'm going to add them together and I'm going to calculate the resultant vector.

14:43  
I'm going to return that as a new object.

14:46  
That could be perfectly fine in terms of the add function, subtract again that sub and multiply is MOL.

14:54  
Anyway, there's a link here for a few more of them on the reference page if you so want to have a look.

15:00  
Now the ones I'm using, they're even the Python programme examples or they're in this lecture.

15:06  
I'm not expecting you to go off and learn others if you so wish to go for it, but I'm just keeping to the most common ones.

15:15  
All right, here's the string representation.

15:18  
So what have we got here?

15:20  
We have a class, and in the class we have a constructor.

15:24  
Again, at this time, I've just decided to pass in a variable name and we're assigning the value of name to a data member self dot name.

15:32  
Great, now we've got here a string representation function.

15:38  
Now, rather like the init function is going to do one thing, it's going to return an object.

15:44  
The repr function has to have the same form always.

15:48  
Here we pass in nothing and we return a text string.

15:53  
So if we don't return a text string, Python will actually complain the repr function doesn't return text string.

16:00  
Well, you'll say that in a little bit more convoluted way, but effectively it'll tell you that.

16:04  
So you need to return a text string here.

16:06  
Now I've used just the lazy formatting of a text string.

16:09  
So you can see I've got my class name.

16:12  
I've got the variable that I'm going to pass in equals the value of it.

16:19  
Now it's a normal convention that we create an repr function that returns Python that can be evaluated, meaning if we run that, it will actually work.

16:31  
So you can see here we've got my class name equals some value.

16:37  
So that will work as valid Python.

16:39  
And then we can go ahead and use eval to test that if we so wish.

16:44  
Eval is not a very secure thing, but it's sometimes used to turn strings back into Python.

16:51  
What it does is it runs the text string as if you typed it into your Python programme.

16:58  
So don't use this a lot in your code because like I said, it's unsafe, but potentially is useful sometimes.

17:05  
And it's the way that the repr output should be standardised.

17:10  
So I've got this in a few unit tests you'll see later on to verify that my repr function has been written correctly.

17:18  
Anyway, here you go.

17:19  
We're creating an object of my class, and now if we call print, what does print actually do?

17:25  
Well, print sends a text string to the screen.

17:29  
So although we've not said here, turn my object into a string explicitly, print will do that.

17:37  
It will take the object, it will turn it into a text string, and then it will put it onto the screen.

17:42  
Here what we've what we've said is turn my object into a text string and then we're going to send it off to this eval function which is going to run that bit of text string as if we typed it in as Python.

17:52  
And lo and behold, that will return a new object which is completely separate from the other one because we're effectively calling the constructor again.

18:00  
So it's a different piece of memory.

18:03  
OK, Does anybody have any questions or comments about string representation or operators in general so far?

18:15  
No, OK, fine.

18:18  
We'll go on and have a little look at the next thing.

18:21  
So this is comparison.

18:23  
Now, a lot of times you want an equal function that's the opposite of a not equal function.

18:30  
So that's what I've done here.

18:32  
You may dream up a particular application when you don't want that to happen.

18:36  
I don't know why, but they give you the tools to allow you to do that with the Python programming language anyway.

18:43  
So here we go.

18:44  
We've got the constructor and we have passed in a variable.

18:50  
We're assigning that value to a data member and then we've got the equals and the not equals.

18:57  
Now the format here is that rather like the repr, we have to do exactly what it wants.

19:04  
We have to call it\_\_EQ\_\_We have to make it a data sorry make it a, member, function meaning we've got self, and we have to give it another variable which is going to receive the other object and then we're inside.

19:20  
Saying is for example here is this data member equal to the data member in the other object now you can do.

19:29  
That even if it's private because this other thing should be the same of the same class OK remember private is.

19:40  
ONLY, private is private within the class not within the object, so you could have.

19:45  
Here a protected or a private data member you're comparing anyway for the any I've got here a naughty, so I'm just returning the opposite of the EQ.

19:57  
Now how does this work?

19:59  
We've constructed 2 objects and then we simply go ahead and compare them like that.

20:05  
So that is this object equal to that one that will call this function.

20:10  
Well, what happens?

20:11  
Is the D object appears here in the self and the P object appears here in the other for the northeast version, or the not equal.

20:22  
Again, the D object appears here in self, the P in other, and that's just the recipe so you know how to connect the thing together.

20:30  
You can put whatever you like inside of that EQ or NE function.

20:36  
Anyway, the output of running this sort of programme is shown on the right.

20:39  
It just prints true if they're equal and false because they're not equal.

20:44  
They've got exactly the same, sorry, false for not equal because they have the same data member value inside.

20:52  
All right, OK, now I'm going to go on to something else.

20:58  
Yes, yes, yes.

21:13  
Very, very fractionally more resource, yes, because I'm calling another function.

21:18  
Generally speaking, whenever you call a function in whatever programme language you're using, there is a little bit of an overhead calling function.

21:25  
So if you call a function vast numbers of times, yeah, it does affect it.

21:30  
But this, this is probably so small, I don't really care.

21:34  
All right, great.

21:38  
So there you go.

21:39  
That's comparisons now.

21:42  
Yeah, let's go ahead and demystify it.

21:45  
Maybe I should do that first.

21:48  
I'll bring up my code here.

21:52  
So first of all, we're gonna go and have a quick look at the private data member.

21:58  
So this bit of code is the code that we had on the lecture slide.

22:04  
And what we're gonna do here is we're gonna create an object, we're going to happily access the private data member, and then we're gonna try to access the private data member.

22:15  
OK, so let's do that.

22:16  
We'll just run and debug here now.

22:22  
We'll see what happens.

22:25  
Yeah.

22:30  
In other languages, the error is slightly more meaningful.

22:34  
In Python, it just simply says it doesn't know it.

22:45  
Right.

22:46  
So we're gonna go back here.

22:47  
Let's see if I can get this down a bit.

22:50  
Go on, do that.

22:54  
I wanna minimise.

22:57  
OK, fine.

22:58  
We'll just step forward.

22:59  
Fine.

22:59  
So what we've got here is we've got an object and it has a data member inside called name and that has value my class.

23:09  
So we can go ahead and print it and this line doesn't cause any errors.

23:12  
But when we do this, oh Oh dear, it says my class object has no attributes\_\_private name.

23:21  
It's a little bit confusing because it does have an attribute called\_\_private name.

23:25  
It's just that we cannot see it because we're sat outside of the class.

23:30  
OK, so that's that one, the accessor.

23:36  
Yeah.

23:36  
OK, so what's happening here?

23:38  
Well, it's pretty obvious.

23:39  
We're just operating on a private.

23:41  
You can play with that yourself.

23:42  
Here's the string 1.

23:45  
So what are we doing?

23:47  
Let's put a break point in here and we'll run.

23:54  
Let's run down to there.

24:03  
Yeah, why not?

24:04  
We'll just go and put one there.

24:05  
Actually, why not, right?

24:10  
So we stopped here at this print function.

24:12  
Now, as I told you before, if you give print something that isn't a string, print will try to turn it into a string before it prints it.

24:19  
So let's step in, see what happens.

24:22  
Oh, look, we're here in the ropr function.

24:24  
So just like we did with the constructor, if we try to turn it into a string Python, we'll go and find that function if it exists and do whatever you want it to do.

24:34  
So we'll hop over and now we're here, so explicitly asking for it to be turned into a text string.

24:40  
And lo and behold, we appear back here where we do whatever we've been asked to do.

24:45  
And there this in this constructed call is being created, or sorry, it's being called by eval, which is taking the piece of text we said we gave it and creating another object.

24:59  
OK, let's go back and have a look at the other one.

25:03  
Here's the operator overloading example.

25:06  
So we'll run and debug down to here.

25:10  
Debug.

25:14  
So we're creating 2 objects and they've got the same value inside them.

25:19  
That's what you saw I was telling you.

25:22  
So we're gonna go down to there, right?

25:24  
OK, great.

25:25  
So what's the object D has inside a variable X which has got 10 and the object P has got 10.

25:33  
They're two separate objects in memory because we called the constructor twice.

25:38  
Now what happens here is if we step in, we appear here in the EQ function and you can see that we've got A1 variable.

25:47  
This is X = 10.

25:49  
If you look at the memory address, this memory address is C-150 at the end, and the other one is CE 10.

26:00  
So you can see that the two objects are actually just two separate objects in memory, although the variable values the same.

26:08  
Anyway, that returns true, obviously.

26:12  
And then in this case, we step in here so we're inside and that is gonna call the equals again and return and then return and we're finished.

26:24  
OK, Hopefully that's clear enough.

26:28  
If you want to run it yourself afterwards, please do so.

26:32  
So object oriented issues.

26:34  
Now, if you have grown up as a computer scientist, maybe in the 90s, you would have believed the mantra which is object oriented coding or design is always best.

26:49  
And in fact, if you go onto YouTube, you find all sorts of people say, Oh yes, it's always best.

26:53  
It's always best so I can categorically say this is not true.

26:58  
Why is this not true?

27:00  
It's not true because in the real programming environment you need to change things, and to start with, you don't necessarily know everything either.

27:11  
So you have a problem and you are trying to find a solution and your first attempt might be slightly wrong.

27:19  
So what issues does this 'cause you?

27:22  
Well, if you are trying to define a class with some functions inside it, you're trying to think about how it's going to be used, all the things that you might need to do with that class, what functions, what data need to go in there.

27:36  
Now, yes, if you get it slightly wrong, that's OK, because you can come back and always add another function, add another data member.

27:42  
But if you are completely wrong, then it's actually more work to create a new class and split up.

27:50  
Or maybe you have to create two classes and split up the data members and the member functions.

27:54  
Essentially, by creating something that's more complicated, changing it becomes more difficult.

28:01  
So if you are starting potential development, you may have limited understanding of what needs to be in a class.

28:08  
Now there will be some classes which are more known like the data model classes we talked about last time, where you can see what data are going to be carried around through the application.

28:20  
And defining those is a little bit safer.

28:23  
But I'm warning you that if you encapsulate things, that is if you put things into a class in such a way and you think oh this looks so nice, it looks great, show it to my fellow developer, they think it looks great.

28:36  
You then may have to make a change at some point and you end up doing more work.

28:40  
So it can be, especially with inheritance, costly to rewrite.

28:46  
So object oriented programming is nice because it does actually conceptualise things together.

28:52  
But the caveat is that it can be hard to rewrite later.

28:58  
The other problem that you have with people who love object oriented programming is that they tend to copy data around all over the place.

29:07  
So remember, an object has inside data.

29:11  
It can have some or lots or very little data, but it has data inside.

29:17  
So as you pass those objects around, you're passing around functions, but you're also passing around data that's inside the object.

29:23  
Now bad code design is where we have Pat brought in some data, We create some object, it has some data, then it goes somewhere else.

29:32  
And so you just have this data copying all the way around.

29:35  
And then somebody changes data in one place and now it's out of sync with respect to the original data.

29:42  
Imagine if you were taking some measurements from a sensor and you copied them into an object and then you gave the object to some processing algorithm and then that copied into another object.

29:52  
And then there was a bug somewhere else and you're like, OK, where did that data change?

29:58  
So it just becomes a tangled mess.

30:00  
Now, you can avoid this by being careful and you can say, right, my data are just going to be stored in these model classes.

30:10  
I'm not going to change them.

30:12  
The data that are that are in the model classes, I may be going to create a copy if I'm going to change something, but I'm going to be very careful with my programming and that's OK.

30:22  
But I've seen even commercial projects do this and it's an absolute mess to debug to understand what's going on.

30:30  
So please don't do that.

30:32  
OK?

30:33  
That's the warning label.

30:35  
It's not a panacea, nor is lack of classes, but you just have to be careful.

30:42  
Now on to something else, UML class diagrams.

30:47  
So the idea of our design here is we want to describe what we're going to build before we actually build it.

30:55  
Now in the week two, we discussed pseudo code and flow charts where we're trying to puzzle out what functions do we have, what actions are we going to do?

31:07  
And that can still be very useful for just run-of-the-mill functions.

31:11  
In this case, we're trying to puzzle out what classes do we have.

31:15  
Now we are not writing the code here, we're just saying, OK, we've got a class.

31:21  
In this case, I've called the class Data Element and then inside I've decided I've got some data members.

31:28  
So you can see here I've got a data member and I've called it public data.

31:34  
This one's called protected data and this one's called Private data.

31:37  
Now the naming convention for UML is not the same as the Python naming convention.

31:44  
So here I'm using the UML naming convention.

31:48  
Now in UML, you've got the plus meaning protected, the hash meaning sorry, plus meaning public, hash protected, and minus meaning private.

32:00  
You can also say the type of the variable and you can give it a default.

32:04  
Now, a lot of times you're not doing this because you're brainstorming.

32:08  
So you might come up with a class name, you might come up with a few data members that you know that you need, and you might maybe have the type in mind.

32:17  
So maybe it's a basic type or an object or something.

32:21  
Remember this is a more a brainstorming tool when you're doing the early stages of design.

32:27  
So you're unlikely to be able to say what the default is, although you can.

32:31  
It's part of the standard anyway.

32:33  
So over the left here we can see this.

32:35  
These are the UML names.

32:36  
So we have it's so-called attribute in programming terms.

32:40  
This is a data member and we have visibility, meaning is it public, private, or protected, which is this symbol here?

32:49  
If it is a function in UML speak it is an operation, whereas in our implementation we always called it a member function.

32:58  
Anyway, So what does this look like for a piece of code?

33:03  
Let's have a look.

33:06  
So on the top left I've got a piece of code and the bottom right I have the UML version.

33:12  
Now, I haven't put anything useful inside the functions just so that I can get it onto the screen.

33:17  
It's more about what's on the outside that goes in the UML diagram.

33:21  
So you can see that my code here has a class called Data Element and my UML has a class called Data Element.

33:30  
And then I have here I've got a lovely variable which is in this case public data, right?

33:39  
Different naming convention for Python.

33:41  
It has to be in snake case.

33:44  
And then I have protected data, protected data and private data and private data.

33:50  
And then I've got the multiply numbers.

33:52  
Here you can see there's multiply numbers.

33:54  
It's a public function and I've got another public function called test something.

33:59  
Again, this is in snake case because it's Python, And this is not in Pascal case because it's UML.

34:07  
You can see it returns a bull, it returns a bull, it returns a float, it returns a float.

34:11  
Anyway, you get the idea.

34:13  
This is helpful before coding.

34:16  
You know, you're just trying to think what classes do I need?

34:20  
We'll come back to coding and design for classes in 993 next semester, but this is just an introduction to it.

34:30  
OK, Now if you've got other classes, you might want to say 1 is inheriting from another and you can do that with UML.

34:40  
So the way to say it inherits is this funny arrow, this rather extreme looking arrow, which says this class derived class is inheriting from this class base class.

34:53  
So you can define that.

34:54  
Now, normally when you're documenting things, you don't bother putting the attributes that are in the base class again into the drive class.

35:03  
You just think, OK, they're inherited.

35:06  
So somebody's going to read my diagram and remember it's inheriting those from the other one.

35:11  
So you don't put those in, but you can also define relationships between classes.

35:17  
So you have already seen some of this where we put the solar panels inside the solar array.

35:25  
So one can contain zero more of the other.

35:29  
So here in my case I've got a data set which can contain 0 or as many as I like in this data slices.

35:37  
So the data set object could exist with an empty list inside it, or the list could have some number of data slices.

35:46  
Now you can also say in UML the other way around.

35:50  
So here I'm saying that the data slice belongs to exactly one data set.

35:55  
I don't want it belonging to multiple because then it confuses people a little bit.

35:59  
Like our solar panels, the solar panels only belong to one solar array.

36:05  
We don't want them to be strapped on multiple people's houses at the same time.

36:09  
That doesn't make any sense.

36:11  
So it's physically in one place.

36:13  
So when we're describing this, we've got a few multiplicity things we can say, which you'll see actually in database design as well.

36:22  
Same kind of syntax.

36:23  
So 0 dot dot star is 0 more one is exactly 1.

36:30  
So not 0, but just exactly 1.

36:33  
One dot dot star is one or more and one dot dot sorry, zero dot .1 is 0 or 1.

36:40  
So you can put that next to the end to and it refers as if it were the other way around.

36:47  
So in this case, one data slice is attached to a data set, whereas the data set may have zero or more data slices.

36:55  
OK, so instead of having this explicit notation like that, you can use the aggregation or composition arrows.

37:06  
So this is equivalent to saying that you may have one or more in this case that way or zero more that way.

37:15  
All right, so you can choose if you want to use this operator or you can just go ahead and put lines in with explicit definition of how many map to the other ones.

37:30  
Now to draw these diagrams, you can draw them with any editor you like.

37:34  
I'm giving you a few.

37:36  
The one that we often use is this online editor diagrams.net, which allows you to create UML class diagrams and other UML diagrams which we will use in 993 next semester.

37:51  
There are other tools you could use.

37:53  
There's UML Designer, Visual Studio, sorry, Microsoft Visio Pro, which is a commercial piece of software you have to pay for, which is why we're sticking with this one.

38:05  
The other nice thing you can do is you can actually automatically generate them from the code.

38:11  
Now if you do this, you're not putting any effort into producing those diagrams.

38:16  
You're just taking the code and producing them from the code.

38:20  
And you might do that as a documentation aid for another developer.

38:24  
So you use go back, you use this kind of top level brainstorming tool, but then when you have finished, you then can generate documentation to give to somebody else.

38:38  
So you're not saying, Oh no, no, I'm not going to do any design, I'm just going to extract the design from the end.

38:43  
You do do some design, but then when you're finished and you want to provide somebody with better documentation, you can extract it from the code.

38:50  
Now in Python we can use PY reverse, which we will have on the lab computer, so you can try it out.

38:56  
In other languages you often use something called Doxygen that will do the same.

39:02  
It will generate UML diagrams.

39:05  
I can't remember if Java does it.

39:07  
It probably does and I've forgotten.

39:10  
OK, so that's all I have for today.

39:13  
I've got plenty of time for comments and questions.

39:17  
Does anybody want to ask me any questions?

39:19  
Yes, these ones correct, Yes, because the symbol implies what the numbers were implying on the other slide.

39:37  
So you can draw here an aggregational composition.

39:40  
If you want to make it super readable, you could put them in, but you don't have to because that symbol implies what we had up here.

39:51  
OK.

39:52  
So I don't mind if you use one or the other.

39:55  
The point is we're just trying to describe the one object may contain one or more or zero more of the other object, rather like a solar panel and a solar array relationship.

40:07  
It helps because you can then say, yeah, that object is going to be inside this other one, right?

40:14  
That's what we're saying.

40:21  
This only has 1 arrow from data slice.

40:23  
Yes, yes, right.

40:26  
So if you've not said anything, you assume it's one.

40:32  
So one data slice.

40:33  
So I've not said anything, it's one.

40:35  
Whereas this way around, this is one or more and this is 0 more.

40:41  
If you look up UML class diagram, there's just like a bunch of rules.

40:44  
How do you draw them?

40:45  
A bit like flow charts.

40:47  
You remember we were using boxes and whatever.

40:49  
So you just follow the syntax.

40:51  
Do you want to say this is going to be 0 or more?

40:53  
Do you want to say it's one or more?

40:56  
And you can choose, I mean, if you were describing to a client and you thought the client's got no idea what UML class diagrams looked like and you showed them this, that would be correct.

41:07  
But they might say, really, which way round is it, 0 to one?

41:11  
And therefore you might just do that just to be absolutely, explicitly clear, like one has 2 numbers on it and the other one is just one.

41:21  
Yeah.

41:21  
It's not really indicating that.

41:25  
Yeah, you have to take it what it implicitly means.

41:28  
Yeah.

41:28  
So yeah, probably the numbers are clearer anyway.

41:33  
Anybody else?

41:35  
Yeah, this string conversion thing that, Yeah.

41:45  
Why do you reach the turn rather than?

41:53  
So let's see when you're just displayed this.

41:59  
OK, so this function repr has to return a text string.

42:05  
And now that's just what it's required to do.

42:08  
Yeah, it's like an agreement between.

42:12  
You have an agreement like I will buy your car for this amount of money, right?

42:16  
There's an agreement.

42:17  
So between the Python programme language and the developer, the agreement is if you make an repr function, it has to return a text string.

42:27  
That's it.

42:29  
You could put print above here.

42:30  
So you could have a colon print whatever you like, but you have to return the text string at the end.

42:35  
Now the point of this repr function is that somewhere you want to turn your object into a text string, you might want to do that.

42:46  
I don't know anywhere you want in your programme, right?

42:49  
So, yeah.

42:51  
So you don't normally have a printer here.

42:53  
If you go back to one of the other lectures where I was saying, you know, print and return are not the same thing here we are just trying to return the value because we are returning a text string.

43:05  
If we wrap this object with print, it will, as we saw, oops, sorry, it will just go ahead and call this function and then print the return on the screen.

43:16  
Yeah, yes.

43:30  
Yes, good question.

43:31  
So there are a few more of these where I've got multiple data members.

43:34  
So the standard is that you want this text string to create an object, which let's say you put, age in here, right?

43:45  
So then if you didn't have a default, you'd have to give it the name and the age.

43:51  
Therefore, this constructor should have name equals something, age equals something, right?

43:58  
So the repr should return the text string which has got all the things in to make an object.

44:05  
So if you put the age in up here, the age goes in here as well, right?

44:10  
So there's only one repr function.

44:12  
You can't have multiple, it won't let you do that.

44:17  
Anybody else?

44:24  
OK, let's have a look.

44:26  
Have I got any more code to look at?

44:29  
I think I've covered all of it.

44:31  
Oh, yes.

44:31  
Here's our UML example.

44:33  
So yeah, you can stare at that yourselves.

44:35  
Example five.

44:39  
There you have example 5.

44:41  
Certainly you can.

44:42  
Example 5.

44:45  
Yes.

44:50  
That might be wrong.

44:51  
We may not still plan to involve with or without the member functions and the, oh, those functions.

45:04  
So you're saying I should take these member functions out and run it and they should still print true or false?

45:11  
I don't think it will work, but let's let's go ahead and try it.

45:21  
Right.

45:23  
So in this case they the equals is false and not equals is true.

45:31  
Now it's the other way around.

45:32  
Why is it the other way around?

45:33  
Because they're two different things in memory.

45:36  
You've got like 1 is sitting in one space in memory, one is sitting in the other.

45:41  
So it's probably doing the comparison based on the memory.

45:45  
So if you imagine you you had an object you made and then you passed it by reference to another variable, and then you're asking are they the same?

45:55  
And then, yes, that would give you the right answer.

45:57  
But here, because they're two separate things in memory, it's.

46:00  
Yeah, it's giving you the other answer.

46:03  
Good question.

46:03  
Yeah.

46:05  
As ever, questions are very good and helpful.

46:08  
Let's see if I can edit this back.

46:13  
Anybody else?

46:17  
All right, well, I'm gonna stick around in case anybody's shy.

46:21  
Otherwise, please come and have a go in the lab.

46:24  
Ask plenty of questions where things aren't clear.

46:28  
And I hope you have a good day if you don't want to ask me any more questions.

46:32  
Thanks.