Introduction to programming

## Classes and Object Oriented Programming

Object oriented programming assumes that everything within your program is an object. These objects interact with each other and can be combined to create more complex systems.

### What are objects?

Objects can be thought of as real world things. For example, a bike, a toaster and a phone are all objects. The key principle behind object oriented design is to create **objects** in your programs that interact with each other rather than creating long blocks of code which, although perform the desired action well, may not be maintainable or re-usable. When following an object oriented design methodology everything in your program can be thought of as an object. If you are creating details for a user, the user is an object. If you are saving the user details to a database, the database (and possibly the connection to it) is treated as an object.

### Classes

When referring to object oriented design, a class is an outline of what data an object will contain and also the associated actions that the object can perform. A variable defined in a class is called a member and a function is called a method. A simple example would be a user object:

**User object**

|  |  |
| --- | --- |
| Data | Actions |
| Login name | **Get login name** |
| Password | **Authenticate, Reset password** |
| Full name | **Get name, Change name** |
| Department | **Get department, Change department, Calculate salary** |

From the above table you can see that a user object will hold information about the user’s login details, their name and which department they work in. The class also defines actions that the user object can perform. A lot of these actions are **getter** methods i.e. they exist purely to retrieve the value that is stored for the user. This is because most languages that follow an object oriented design pattern prevent direct access to the data stored in an object. This process is called **encapsulation**.

### Encapsulation

In languages such as Java where object oriented design is strongly encouraged there are special keywords available which specify the level of access that variables in the object have. The two most common are **public** and **private.** A public member can be accessed directly from an object that has been created from a class. For example, take an object that has been created called **user** that has a public member called **name**; to access it’s value the syntax would be **user.name**. If a member is declared as private, it cannot be accessed in this way and requires another way e.g. a public method that returns it value to retrieve the member’s value.

Python doesn’t employ the concept of public and private members or method however there are ways to indicate to programmers that a member or method should be treated as private. In Java a User class can be implemented like this:

**public** **class** **User{**

**private** String name**;**

**public** **User(**String name**){**

**this.**name **=** name**;**

**}**

**public** String **getName(){**

**return** **this.**name**;**

**}**

**}**

The syntax might look strange as you have been mainly looking at Python over the course however you should be able to see the private and public keywords at work. The class above defines a single member called name which is a string. It is also declared as **private** so it cannot be accessed by **user.name** for example. This is why there is the **public** method **getName** which simply returns the value of the **name** member. The same class in Python could look like:

**class** **user**():

**def** **\_\_init\_\_**(self, name):

self**.**name **=** name

**def** **get\_name**(self):

**return** self**.**name

### Purpose of object oriented programming

There are several reasons why programmers adopt an object oriented approach to designing programs. Some of the benefits of programming in this style are:

* Maintainable code: as components of a program are written in discrete units (classes) it is easier to track down errors when they occur.
* Re-usable code: most classes will be able to be extracted from a particular program and then re-used in new ones. The **User** class for example could easily be used in different programs.
* Encapsulation: as described above
* Abstraction: a programmer can hide all but the relevant data about a particular object. An example of this when a programmer ‘hides’ a member variable by using a **private** keyword.

### Writing Python classes

You can define a class in Python using the the **class** keyword.

**class** **user**():

*# Your members and methods*

As with most blocks of code in Python, everything you want including in the class definition should be indented underneath the opening line. To add new members (variables) to the class, you simply declare them as you would in a normal program.

**class** **user**():

*# Your members and methods*

login\_name **=** "jbubb"

password **=** "cnffjbeq"

full\_name **=** "James Bubb"

departement **=** "IT

As mentioned above, Python has no concept of **public** or **private** so all the members above will be public.

When you want to use a class you have defined you need to create an object based on the class’ blueprint. This process is known as instantiation or creating an instance of an object.

current\_user **=** user()

It can be thought of as if the class **user** were a function and it is **returning** a copy of a user object which you can then use in your program.

The above example, is not very flexible as every time you create a new **user** object it will have the same login name, password etc. What would be more useful if you could specify the user details when creating the object. This can be done and is achieved by implementing a class **constructor**.

**class** **user**():

**def** **\_\_init\_\_**(self):

**print** "User created"

new\_user **=** user()

A constructor is a special function that is defined inside the body of a class definition. It has the special name **\_\_init\_\_** and it must take at least one argument. The mandatory argument (**self** in the example above but could be called anything else) is a reference to the newly created object. This will be useful to know in the next examples.

When the new user object is created in the example above the constructor (\_\_init\_\_ function) is called automatically and therefore the **print “User created”** statement is executed. Try creating the above class and then create a new user to check that the message appears OK.

The class you have created still does not make users with different member variables which is why you need to access the **self** variable. The **self** variable is a reference to the newly created object. It can be used to assign arguments that are specified in the class’ constructor and assign them to the instance of the object. For example, if you update the \_\_init\_\_ constructor for the user class to accept arguments for login name, password, full name and department:

**class** **user**():

**def** **\_\_init\_\_**(self, login\_name, password, full\_name, department):

self**.**login\_name **=** login\_name

self**.**password **=** password

self**.**full\_name **=** full\_name

self**.**department **=** department

**print** "User created"

new\_user **=** user("jbubb", "cnffjbeg", "James Bubb", "IT")

The arguments passed to the \_\_init\_\_ constructor can then be assigned to the newly created object. This is done by assigning the name of the argument passed to the **self** variable. It is usually good practice to name the new **self** member variable the same as the argument variable passed in to aid clarity.

Once the constructor has been updated, when a new user is created the correct number of arguments need to be provided to ensure the new object can be created correctly. Try updating your user class to take arguments in the constructor. You could also provide default values in the \_\_init\_\_ function.

The user class can then be further expanded to include the methods that were outlined in the user table initially.

**class** **user**():

...

**def** **authenticate**(self):

**return** self**.**password **==** "cnffjbeg"

**def** **reset\_password**(self):

self**.**password **=** "cnffjbeg"

**print** "Password updated"

**def** **get\_name**(self):

**return** self.full\_name

**def** **change\_name**(self, new\_name):

self**.**full\_name **=** new\_name

...

new\_user **=** user("jpbubb", "cnffjbeg", "James Bubb", "IT")*#*

The above example shows the user class with some of the methods defined. Some of the methods, such as **authenticate** and **get\_name** simply return a value; these are the **getter** style methods. The other methods, **reset\_password** and **change\_name** are referred to as **setter** methods as they are actually updating the values of object. Try implementing the rest of the methods that were outlined in the initial user object table.

The methods for a new user object can be accessed like this:

new\_user **=** user("jpbubb", "cnffjbeg", "James Bubb", "IT")*#*

**print** new\_user**.**get\_name()

**if** new\_user**.**authenticate():

**print** "You are logged in"

new\_user**.**change\_name("Dave Jones")

**print** new\_user**.**get\_name()

### Inheritance

One of the features of object oriented programming is the ability for classes to inherit members and methods from existing class definitions. This creates a way for **base** classes to be extendedso that additional functionality and data can be added. For example, the **user** class we have been looking at is fairly generic and could apply to any type of user. We can extend the generic base **user** class to make a specific **IT\_User** class which contains members and methods which are only relevant to someone who works in the IT department.

**class** **IT\_User**(user):

**def** **\_\_init\_\_**(self, login\_name, password, full\_name, department, skills):

self**.**login\_name **=** login\_name

self**.**password **=** password

self**.**full\_name **=** full\_name

self**.**department **=** department

self**.**skills **=** skills

**def** **list\_skills**(self):

**print** "Skills:"

**for** s **in** self**.**skills:

**print** s

**def** **fix\_things**(self):

**print** "I am here because you broke something!"

The constructor for this class looks very similar to the **user** class except it adds an additional member variable **skills**. It also extends the **user** class by adding two new methods; **list\_skills** and **fix\_things**. The **IT\_User** class will also have **inherited** all of the previous methods from the **user** class. This means that the methods such as **get\_name, change\_name** etc that were defined in the user class can still be used. If you create a new IT\_User object you will see that they are still accessible, along with the new methods:

new\_it\_user **=** IT\_User("jbubb", "cnffjbeg", "James Bubb", "IT", ["Windows", "Linux", "MS Exchange", "FTP"])

**print** new\_it\_user**.**get\_name()

new\_it\_user**.**list\_skills()

### Object oriented programming in Java

As a comparison to how the **user** class has been implemented in Python, here is the same class written in Java.

**import** java.util.HashMap**;**

**import** java.util.Map**;**

**public** **class** **User{**

**private** String loginName**,** password**,** fullName**,** department**;**

**public** **User(**String loginName**,** String password**,** String fullName**,** String department**){**

**this.**loginName **=** loginName**;**

**this.**password **=** password**;**

**this.**fullName **=** fullName**;**

**this.**department **=** department**;**

**}**

**public** String **getLoginName(){**

**return** **this.**loginName**;**

**}**

**public** **boolean** **authenticate(){**

**return** **this.**password**.**equals**(**"cnffjbeg"**);**

**}**

**public** **void** **resetPassword(){**

**this.**password **=** "cnffjbeg"**;**

System**.**out**.**println**(**"Password updated"**);**

**}**

**public** String **getName(){**

**return** **this.**fullName**;**

**}**

**public** **void** **changeName(**String newName**){**

**this.**fullName **=** newName**;**

**}**

**public** **void** **changeDepartment(**String newDepartment**){**

**this.**department **=** newDepartment**;**

**}**

**public** **int** **calculateSalary(){**

Map**<**String**,** Integer**>** salaries **=** **new** HashMap**<**String**,** Integer**>();**

salaries**.**put**(**"HR"**,** 25000**);**

salaries**.**put**(**"Support"**,** 18000**);**

salaries**.**put**(**"IT"**,** 40000**);**

**return** salaries**.**get**(this.**department**);**

**}**

**}**

There are few things to notice:

* The member variables in the class are declared as **private**. This means they cannot be directly accessed or re-assigned.
* The methods in the class are declared as **public**. This means that these methods **can** be accessed directly.
* The **self** variable is missing in the method definitions. This is because Java doesn’t require this variable to be explicitly included but is provided in the method body implicitly however it’s name is set as **this**.

Here is the IT user class implemented in Java too:

**public** **class** **ITUser** **extends** User**{**

**private** String**[]** skills**;**

**public** **ITUser(**String loginName**,** String password**,** String fullName**,** String department**,** String**[]** skills**){**

**super(**loginName**,** password**,** fullName**,** department**);**

**this.**skills **=** skills**;**

**}**

**public** **void** **listSkills(){**

System**.**out**.**println**(**"Skills:"**);**

**for(int** i**=**0**;** i **<** skills**.**length**;** i**++){**

System**.**out**.**println**(this.**skills**[**i**]);**

**}**

**}**

**public** **void** **fixThings(){**

System**.**out**.**println**(**"I'm here because you broke something!"**);**

**}**

**}**

The constructor for the IT User class is slightly different in Java. There is another keyword **super()** which has been included. This special keyword calls the constructor of the **superclass** i.e. the constructor for the **User** class that **ITUser** inherits from. This saves having to assign all of the arguments provided to the constructor to the newly created object in the **ITUser** constructor.

In order to test the classes in Java, another program needs to be written to test it.

**public** **class** **ObjectExample{**

**public** **static** **void** **main(**String**[]** args**){**

User employee1**;**

String**[]** itSkills **=** **{**"Windows"**,** "Linux"**,** "MS Exchange"**,** "FTP"**};**

ITUser it\_user1**;**

employee1 **=** **new** User**(**"jbubb"**,** "cnffjbeg"**,** "James Bubb"**,** "IT"**);**

it\_user1 **=** **new** ITUser**(**"djones"**,** "cnffjbeg"**,** "Dave Jones"**,** "Support"**,** itSkills**);**

System**.**out**.**println**(**employee1**.**calculateSalary**());**

System**.**out**.**println**(**it\_user1**.**getName**());**

it\_user1**.**listSkills**();**

**}**

**}**