

Object Oriented Programming

Classes and Objects

Why Use Object Oriented Programming (OOP)

- OOP is based on two principles:
 - Duplicate code is bad
 - Code will always be changed
- Can modify and maintain existing code much more easily
- Reuse of code in other programs through libraries
- Has better structure and design so is suited to big projects
- Clear modular structure with a defined interface and which can abstract and hide away details

Classes

- Classes are the integral component of object oriented programs.
- A class is the blueprint or template for the structure of an object.
- A class contains methods and property/attribute fields that describe the behaviours and characteristics of objects.

Create a class

Prototype	Example
<code>class NameOfClass (object)</code>	<code>class Employee (object) :</code>

By convention class names start with a capital letter.

Objects and instantiation

- If a class is the template of an object then an object is the realisation or instance of a class. Multiple objects can be created from a class.
- Instantiation refers the creation of objects. Remember the class is just a template, we can then create multiple objects from the template and this is what we refer to as instantiation.

Object instantiation

Prototype	Example
<code>Object = NameOfClass (object)</code>	<code>fred = Employee (object) :</code>

Attributes

Attributes are characteristics of an object and can be defined and initialised in the constructor method (more on this later)

```
class Employee():
    #constructor
    def __init__(self, name, dob, salary):
        self.name=name
        self.dob=dob
        self.salary=salary

# Object Instantiation
Fred=Employee("Fred","26th April 2000","£40,000")
Lucy=Employee("Lucy","5th May 2003","£23,000")
```

Methods

- Classes define behaviours using methods. Methods are subroutines defined inside a class. Classes can contain multiple methods.
- In Python, all methods need to have the object parameter (by convention called self) and they can have additional parameters. A class method can be called by:

Prototype	Example
<code>Object = NameOfClass.nameOfMethod()</code>	<code>Scooby = Dog.name ("Scooby")</code>

```
class Greeting():

    # method
    # self: object parameter,
    # name: an additional parameter
    def salutation(self,name):
        print("Hello",name)

# Object Instantiation
g=Greeting()
g.salutation("Fred")
```

Bounded verses unbounded methods calls

The need for an object parameter (self) becomes evident when we consider an unbounded method call where the object itself is explicitly passed as a parameter into the method call. By convention this object parameter is called self

```
class Greeting():
    def salutation(self,name):
        print("Hello",name)

g=Greeting()
g.salutation("Fred") # bounded
Greeting.salutation(g,"Fred") # unbounded
```

Calling a method from a method

A method can call another method within the same class using: `self.methodName()`

```
class Greeting:

    def salutation(self):
        self.formal()
        self.informal() # calling a method

    def formal(self):
        print ("Good afternoon sir")

    def informal(self):
        print ("Wagwan")

g = Greeting()
g.salutation()
```

Special methods: Constructor

The constructor is a special method that is called automatically whenever an object is created. In Python, it is given as:

```
def __init__(self):

class Greeting():

    def __init__(self): # Constructor
        print("Hello World")

g=Greeting() # Object Instantiation
```

Constructor with parameter example

```
class Greeting():

    def __init__(self,name): # Constructor
        print("Hello World", name)

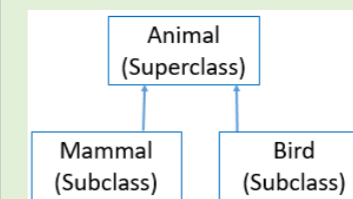
g=Greeting("Fred") # Object Instantiation
```

Inheritance

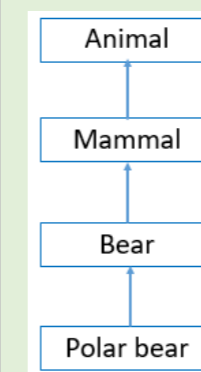
- Inheritance allows us to use methods and attributes in one class from another class.
- The inherited class is called the parent or super class. At the top of the inheritance chain is the base class, so parents and super classes are not always a base class.
- The subclass will have the methods and attributes from the superclass in addition to its own method and attributes
- Many programming languages only allow single inheritance. In Python multiple inheritance is allowed, but we will consider only single inheritance and it is generally deprecated.
- Inheritance defines an **IS-A** relation between classes. For instance:
 - A house **is a** building
 - A car **is a** vehicle
 - A dog **is an** animal

Invoking Inheritance from a superclass class into a subclass

Prototype	Example
<code>class Subclass(Baseclass):</code>	<code>class Mammal(Animal):</code>



Animal is the super / base / parent class. The mammal and bird classes are the derived / child / sub classes



A chain or hierarchy of classes can be established. At the top of the chain of classes is the base class. In this example it is Animal. As far as possible inheritance chains should be avoided.

```
# Example inheritance
# superclass/parent/base
import math
```

```
class Calculator():

    def square(self, x):
        return x*x

    def square_root(self, x):
        return math.sqrt(x)

# sub class/child
class Trigonometry(Calculator):

    def pythagorous(self, x, y):
        y2=self.square(y)
        x2=self.square(x)
        return self.square_root(x2+y2)

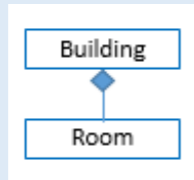
t=Trigonometry()
print(t.pythagorous(3,4))
print(t.square(3))
```

Composition and Aggregation Association

- A class is used by creating an instance of it with another class.
- Association links multiple objects and defines a **HAS-A** relation between classes. There are two forms of association:
 - ✓ Composition association: The child class cannot exist independently of the parent class
 - ✓ Aggregation association: The child class can exist independently of the parent class.

Composition uses an instance of a class. One object is contained in another object. A child object does not exist independently of its parent class. In other words, if a parent object does not exist, then the child object cannot exist. Composition is said to have a *strong HAS-A* relation.

For instance a building can have rooms, but if the building does not exist then the rooms cannot exist.



Composition uses an instance of a class. Subclass will not work without the superclass.

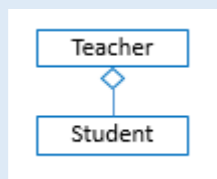
```
# Example composition
class ParentClass():
    def method(self):
        print("Hello")

class ChildClass():
    def method2(self):
        self.b=ParentClass() # instance of class

object=ChildClass()
object.b.method2() # invoke method from class
```

In **aggregation** one object is the owner of another object. The classes can exist independently of one another. Aggregation is said to have a *weak HAS-A* relation.

For instance, a student can have a teacher. But if teacher is removed, the student object will still exist



The object of a class is passed as a parameter into a method Subclass.

```
# Example aggregation
class ClassA():
    def salutation(self):
        print("Hello World")

class ClassB():
    def __init__(self,object):
        self.b=object

obj1=ClassA() # pass object as parameter
obj2=ClassB(obj1)
obj2.b.salutation()
```

Encapsulation, Overriding and Polymorphism

Encapsulation is used to hide data and methods, thereby preventing them from being accessed and changed. Public, private and protected methods and attributes have different accessibility.

Encapsulation allows us to restrict access to methods and attributes so we can prevent the data from being inadvertently modified

In Python private methods and attributes start with two underscores

Public methods	Accessible from outside the class
Private methods	Accessible in their own class
Public attributes	Accessible from anywhere
Private attributes	Accessible in their own class or via a defined method
Protected methods	Only current class can access methods (Accessibility depends on language)
Protected attributes	Only current class can access attributes (Accessibility depends on language)

Private and public methods

```
class Square:

    def __init__(self):
        self.__defineSquare()

    def __defineSquare(self): # private method
        print("square has 4 equal sides")

    def defineSquare(self):
        print("square has 4 equal sides")
```

```
blueSquare=Square()
blueSquare.defineSquare()
# not accessible, will not work
blueSquare.__defineSquare()
```

Class attributes are defined at the top of a class and not in methods. For each instance of an object the values of the class attributes is the same.

```
class Shape(object):
    # class attributes
    __colour = ""
    name = ""
    def __init__(self):
        self.__colour="blue"
        self.name="square"
    def getColour(self):
        print("Colour is: ", self.__colour)
    def getName(self):
        print("name is: ", self.name)

s=Shape()
print(s.getName())
s.name="triangle"
print(s.getName())
print(s.getColour())
# This is protected so changing colour will not work
s.__colour="red"
print(s.getColour())
```

Getter and Setter methods

Private class attributes cannot be accessed directly. Instead getter and setter methods are used to provide access to private class attributes. A get method returns the value of an attribute. A set method allows the value of an attribute to be changed.

```
class GetSet:
    __x=0

    def __init__(self,x):
        self.__x = x

    def get_x(self):
        return self.__x

    def set_x(self, x):
        self.__x = x

y=GetSet(12)
print(y.get_x())
```

Overriding

A method in child class will override a method and attributes in a parent class. This is achieved by using the same name for a method in the child class as in the parent class. In this situation the method in the child class will be used. What will the following code output?

```
class Parent(object):
    def salutation(self):
        print ("Hello")

class Child(Parent):
    def salutation (self):
        print ("Goodbye")

p=Parent()
p.salutation()
c=Child()
c.salutation()
```

Polymorphism allows us to have a function that can take on many forms for its parameters Polymorphism uses overriding. We can use a method of the same name in each class but works differently.

Sometimes an object comes in many types or forms. We access them using the same method. For instance if we take the len function it can take many types such as strings and lists.

```
len("Hello")
len([1,2,3,4])

class English():
    def greeting(self):
        print("Hello")

class French():
    def greeting(self):
        print("Bonjour")

def greeting(language):
    language.greeting(language)

greeting(French)
greeting(English)
```

Polymorphism with abstract class and inheritance

Abstract class: The method is not supplied in the superclass. It must be implemented in the subclass.

```
class Language():
    def greeting(self):
        raise NotImplementedError("Subclass must implement abstract method")
```

```
class English(Language):
    def greeting(self):
        print("Hello")
```

```
class French(Language):
    def greeting(self):
        print("Bonjour")
```

```
e=English()
f=French()
e.greeting()
f.greeting()
```

Static, abstract and virtual methods

Static attributes are defined for the class. Notice that in Python they are not defined within methods and we can access them without creating an object instance.

```
class Greeting(object):
    salutation_english="Hello"
    salutation_french="Bonjour"
```

```
print(Greeting.salutation_english)
g=Greeting()
print(g.salutation_english)
print(g.salutation_french)
```

With **static methods** there is no need for object instantiation.

Using the @staticmethod decorator

```
class StaticClass(object)
    @staticmethod #decorator
    def static_method(x):
        print (x)
```

```
StaticClass.static_method(2)
```

Example

```
class Calculator(object):
    @staticmethod #decorator
    def add_nums(x,y):
        print (x+y)
```

```
Calculator.add_nums(2,3)
```

Without using the @staticmethod decorator

```
class Calculator(object):
    def add_nums(self,x,y):
        print (x*y)
Calculator.add_nums = staticmethod(Calculator.add_nums)
Calculator.add_nums(2,3)
```

Abstract method - The method is not supplied in the base class. It forces the method to be implemented in the subclass.

```
class Superclass(object):
```

```
def greeting(self):
    raise NotImplementedError("Please Implement this method")
```

```
class Subclass(Superclass):
    def greeting(self):
        print ("Hello World")
```

```
s=Subclass()
s.greeting()
```

Virtual methods defined in a superclass can be overridden by methods in a subclass. The difference between virtual and abstract methods is that virtual methods have implementation whereas abstract methods do not. All methods in Python are virtual.

```
class Parent:
    def greeting(self):
        print ("Hello from Parent")
        self.greeting_virtual()
```

```
def greeting_virtual(self,name):
    print "Hello"
```


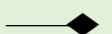
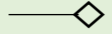
```
class Child(Parent):
    def greeting_virtual(self):
        print "Hello from virtual Child"
```

```
c = Child()
c.greeting()
```

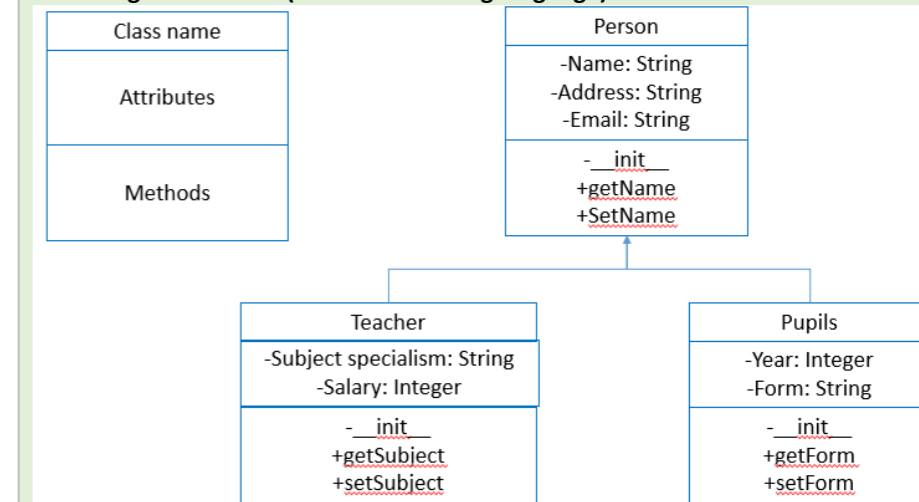
An **interface** is a class containing abstract methods. An interface enforces classes to have a fixed set of methods. The methods are run from the class and not the interface.

```
class A():
    def abstractMethod1(self):
        pass
    def abstractMethod1(self):
        pass
```

Drawing and interpreting class diagrams

Inheritance (arrow)	
Composition (black diamond)	
aggregation (white diamond)	
public	+
Private	-
Protected	#

Class diagrams in UML (Unified Modelling language)



OOP Design principles

Favour composition over inheritance

- Each class can be tested more easily using composition. It is not possible to test a child class independently of a parent class using inheritance.
- There can be side-effects for child classes if a method in the parent class is changed.
- Composition is more flexible. A new class can replace another class easily if composition is used.
- Inheritance can rely too much on long inheritance chains.

Encapsulate what varies

Encapsulation allows us to make future changes to the code more easily. By making attributes and methods private we are protecting other parts of the code from change because they are not accessing the encapsulated elements.

Program to interfaces, not implementation

An interface is a set of abstract methods. The methods are given by the subclass and not the superclass. This is a way of finding commonality between unrelated classes that need to have common methods. This allows us to have a framework for all our classes.