Python Classes and Objects

Classes and Objects

- An object is anything that has properties to describe it and can take certain actions to modify itself.
- A house, a car, a person and a dog are examples of objects in real life.
- A house has properties that describe it, such as:
  - Its colour, say, white
  - The number rooms it has, say five
- These properties distinguish it from another house that is blue with ten rooms.

Classes and Objects

- Operations on any house might include:
  - Building it
  - Cleaning it
  - Remodelling it
- Different houses all belong to the class of things known as House
- All houses can be built, cleaned or remodelled.
- In programming, operations are called methods
- Properties are called instance variables (or attributes or fields or data!)
- Programming with classes and objects is knows as object-oriented programming.

Python data are all objects

- Everything in Python is an object.
- We've seen hints of this already…
  - "hello".upper()
  - list3.append('a')
  - dict2.keys()
- These are method being called on objects.
- New object classes can easily be defined in addition to these built-in data-types.
- In fact, programming in Python is typically done in an object oriented fashion.

Defining a Class

- A class is a special data type which defines how to build a certain kind of object.
- The class also stores some data items that are shared by all the instances of this class.
- Instances are objects that are created which follow the prescription given inside of the class.

Methods in Classes

- Define a method in a class by including function definitions within the scope of the class block
- There must be a special first argument self in all of method definitions which gets bound to the calling instance
- There is usually a special method called __init__ in most classes
- We'll talk about both later…

A simple class definition

```
class Student:
    """A class representing a student"""

    def __init__(self, n, a):
        self._name = n
        self._age = a

    def getAge(self):
        return self._age
```

Creating an object (or instance)

- Use the class name with ( ) notation and assign the result to a variable
- __init__ serves as a constructor for the class.
- Usually does some initialization work
- The arguments passed to the class name are given to its __init__() method
- So, the __init__ method for student is passed “Bob” and 21 and the new class instance is bound to b:

```
b = Student("Bob", 21)
```

The constructor __init__

- The **__init__** method can take any number of arguments.
- Like other functions or methods, the parameters can be defined with default values, making them optional to the caller.
- However, the first argument **self** in the definition of __init__ is special…

The self parameter

- The first parameter of every method is a reference to the current instance of the class.
- We name this parameter self
- In __init__, self refers to the object currently being created; so, in other class methods, it refers to the instance whose method was called.

The self parameter

- Although you must specify self explicitly when defining the method, you don't include it when calling the method.
- Python passes it for you automatically

Defining a method:

(this code inside a class definition.)

```
def set_age(self, num):
  self.age = num
```

Calling a method:

```
>>> x.set_age(23)
```

Data attributes

- Data attributes are created and initialized by an __init__() method.
- Simply assigning to a name creates the attribute
- Inside the class, refer to data attributes using self

Class Attributes

- All instances of a class share one copy of a class attribute. When any instance changes it, the value is changed for all instances
- Class attributes are defined within a class definition and outside of any method
- Access class attributes using self.__class__.name notation
- You can use a class attribute to keep track of how many objects have been created.

Class Attributes

```
class Student:
    """A class representing a student"""

    count = 0

    def __init__(self, n, a):
        self._name = n
        self._age = a
        self.__class__.count += 1

    def getAge(self):
        return self._age

    def getCount(self):
        return self.__class__.count
```

```
>>> a.getCount()
1
>>> b = Student("Smart Kid", 15)
>>> a.getCount()
2
>>> b.getCount()
2
```

Inheritance

- Classes can extend the definition of other classes
- Allows use of methods and attributes already defined in the previous one
- To define a subclass, put the name of the superclass in brackets after the subclass's name.
- For example we can define a class Undergrad which inherits from the more general Student.
- We call Student the parent class or and Undergrad the child class. Other terms are superclass or base class for the parent class and subclass for the child class.

Inheritance

```
class Undergrad(Student):
    """A class representing an undergraduate student"""

    def __init__(self, n, a, m):
        Student.__init__(self, n, a) # Call parent constructor
        self._major = m

    def getMajor(self):
        return self._major
```

```
>>> a = Student("Mickey Mouse", 90)
>>> b = Undergrad("Smart Kit", 15, "Computer Science")
>>> a.getAge()
90
>>> b.getAge()
15
>>> b.getMajor()
'Computer Science'
```

Redefining Inherited Methods

- A child class can redefine inherited methods to suit its own purposes.
- For example, we might have the method getStudentInfo() in the parent which prints out the student’s name.
- An undergraduate also has a major, so we should redefine the method to print out the major as well.
- Doing so hides the inherited method. When a child object calls getInfo(), there will be no conflict. Its own version will be executed.

The class called **object**

- There is a class called **object**, which is the mother of all classes
- Whenever you define a class such as Student, it automatically inherits from object.
- You an write class Student(**object**) but this is not necessary.
- One of the methods that every class inherits from object is the __str()__ method.
- This method enables a string representation of an object to be created.
- Your class should redefine this method so that it behaves appropriately.

Redefining the __str__ method

```
class Student:
    """A class representing a student"""

    count = 0

    def __init__(self, n, a):
        self._name = n
        self._age = a
        self.__class__.count += 1

    def getAge(self):
        return self._age

    def getCount(self):
        return self.__class__.count

    def __str__(self):
        return "Name: " + self._name \
               + "\nAge: " + str(self._age)
```