Object Oriented Programming

Course Contents

- Classes
- The Self
- Methods
- The 'init'
- Polymorphism
- Data Encapsulation / Abstraction
- Inheritance

Object Oriented Programming

- In all the programs we wrote till now, we have designed our program around functions i.e. blocks of statements which manipulate data. This is called the procedure-oriented way of programming.

- There is another way of organizing program which is to combine data and functionality and wrap it inside something called an object.

- This is called the object oriented programming paradigm.

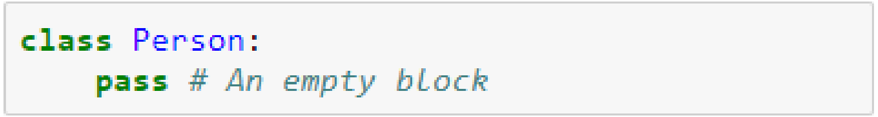
- Most of the time one can use procedural programming, but when writing large programs or have a problem that is better suited to this method, you can use object oriented programming techniques.

Object Oriented Programming

- Classes and objects are the two main aspects of object oriented programming.
- A class creates a new type where objects are instances of the class.
- An analogy is that you can have variables of type int which translates to saying that variables that store integers are variables which are instances (objects) of the int class.
- Fields are of two types - they can belong to each instance/object of the class or they can belong to the class itself. They are called instance variables and class variables respectively.
- A class is created using the class keyword.
- The fields and methods of the class are listed in an indented block.

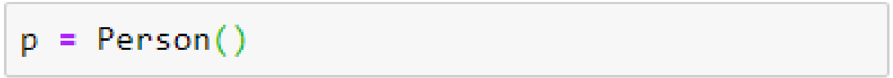
Classes

- We create a new class using the class statement and the name of the class. This is followed by an indented block of statements which form the body of the class. In this case, we have an empty block which is indicated using the pass statement.



```
class Person:
    pass # An empty block
```

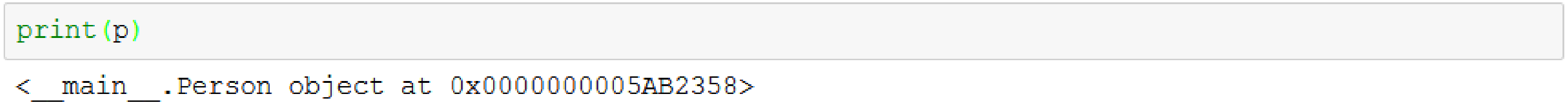
- Next, we create an object/instance of this class using the name of the class followed by a pair of parentheses.



```
p = Person()
```

Classes

- For our verification, we confirm the type of the variable by simply printing it.



```
print(p)
```

```
<__main__.Person object at 0x0000000005AB2358>
```

- It tells us that we have an instance of the Person class in the main module.

- Notice that the address of the computer memory where your object is stored is also printed.

The self

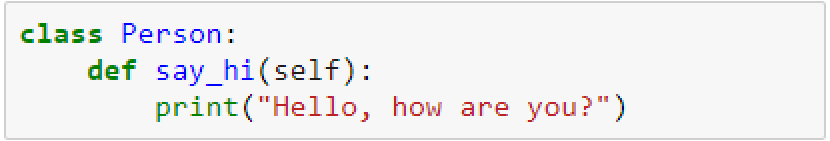
- Class methods have only one specific difference from ordinary functions - they must have an extra first name that has to be added to the beginning of the parameter list, but you do not give a value for this parameter when you call the method, Python will provide it.
- This particular variable refers to the object itself, and by convention, it is given the name self .
- Although, you can give any name for this parameter, it is strongly recommended that you use the name self - any other name is definitely frowned upon.
- There are many advantages to using a standard name - any reader of your program will immediately recognize it and even specialized IDEs (Integrated Development Environments) can help you if you use self .

The self

- How Python gives the value for self and why we don’t need to give a value for it.
- Let us have a class called **List** and an instance of this class called **shoplist**.
- When we call a method of this object as ***shoplist.append(‘milk’)***, this is automatically converted by Python into ***List.append(shoplist, ‘milk’)*** - this is all the special self is about.
- This also means that if you have a method which takes no arguments, then you still have to have one argument - the self .
- The self in Python is equivalent to the this pointer in C++ and the this reference in Java and C#.

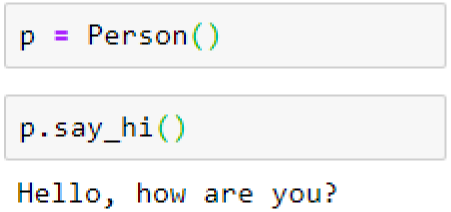
Methods

- Classes/objects can have methods just like functions except that we have an extra self variable.
- We see the self in action in given example.



```
class Person:
    def say_hi(self):
        print("Hello, how are you?")
```

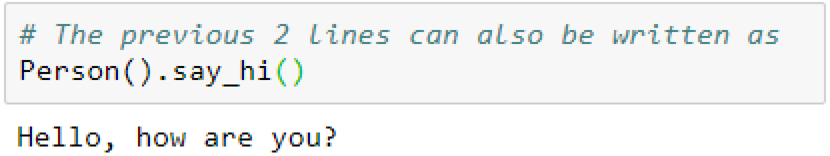
- Notice that the ***say_hi*** method takes no parameters but still has the self in the function definition.



```
p = Person()
```

```
p.say_hi()
```

```
Hello, how are you?
```

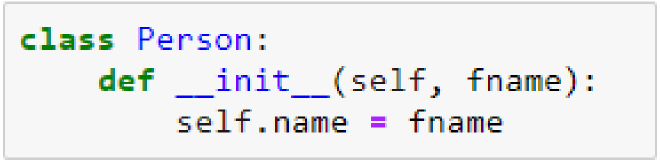


```
# The previous 2 lines can also be written as
Person().say_hi()
```

```
Hello, how are you?
```

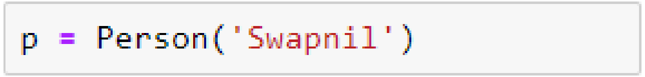
The init method

- There are many method names which have special significance in Python classes.
- The init method is run as soon as an object of a class is instantiated.
- The method is useful to do any initialization you want to do with your object.
- Notice the double underscores both at the beginning and at the end of the name.

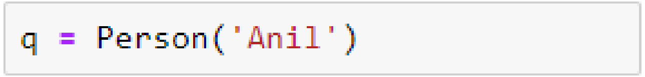


```python
class Person:
    def __init__(self, fname):
        self.name = fname
```

- We do not explicitly call the init method but pass the arguments in the parentheses following the class name when creating a new instance of the class.

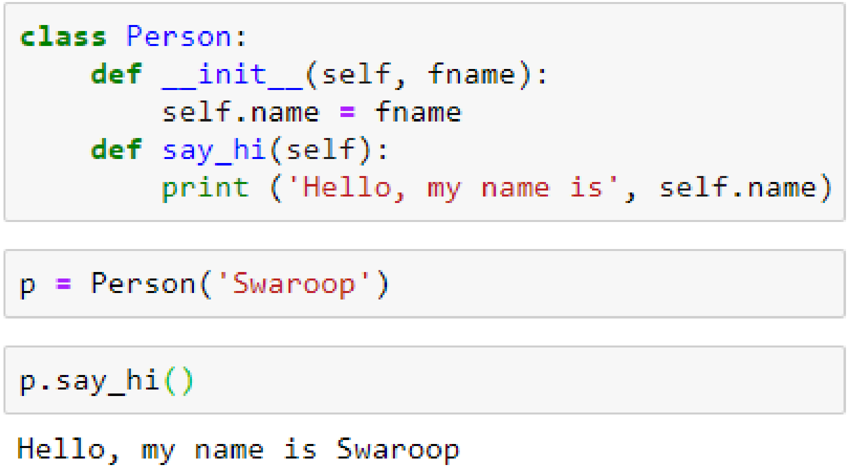


```python
p = Person('Swapnil')
```



```python
q = Person('Anil')
```

The init method

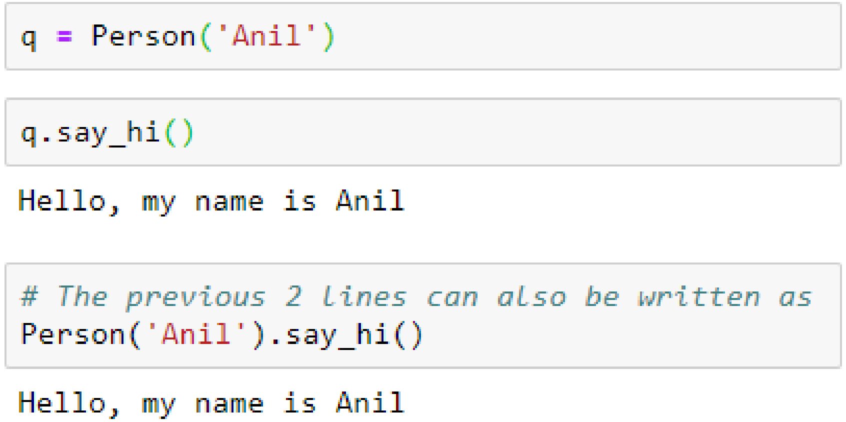


```
class Person:
    def __init__(self, fname):
        self.name = fname
    def say_hi(self):
        print ('Hello, my name is', self.name)
```

```
p = Person('Swaroop')
```

```
p.say_hi()
```

```
Hello, my name is Swaroop
```



```
q = Person('Anil')
```

```
q.say_hi()
```

```
Hello, my name is Anil
```

```
# The previous 2 lines can also be written as
Person('Anil').say_hi()
```

```
Hello, my name is Anil
```

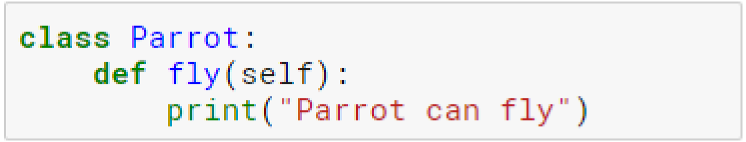
Polymorphism

- **Polymorphism** is achieved through method **overloading.** Method **overloading** means there are several methods present in a class having the same name but different types/number of parameters. Like other languages , python does not supports method overloading by default. But there are different ways to achieve polymorphism in Python.

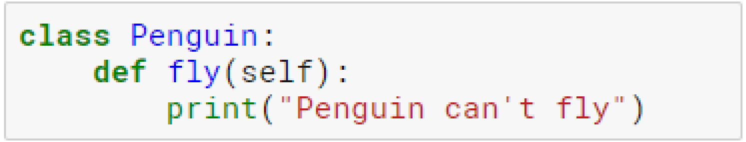
- Polymorphism is an ability (in OOP) to use common interface for multiple form (data types). Suppose, we need to color a shape, there are multiple shape option (rectangle, square, circle). However we could use same method to color any shape. This concept is called Polymorphism.

Polymorphism

- In the given program, we will define two classes **Parrot** and **Penguin**. Each of them have common method fly() method. However, their functions are different.

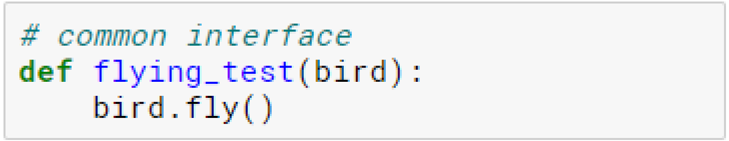


```python
class Parrot:
    def fly(self):
        print("Parrot can fly")
```

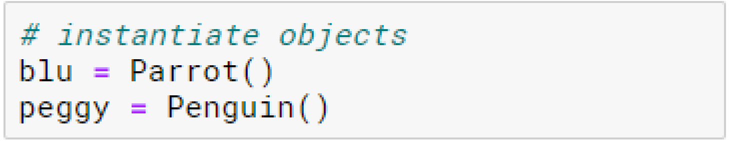


```python
class Penguin:
    def fly(self):
        print("Penguin can't fly")
```

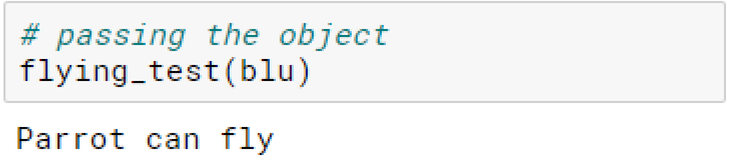
- To allow polymorphism, we create common interface **flying_test()** function that can take any object. If we pass the objects **blu** and **peggy** in the *flying_test()* function, it run effectively.



```python
# common interface
def flying_test(bird):
    bird.fly()
```

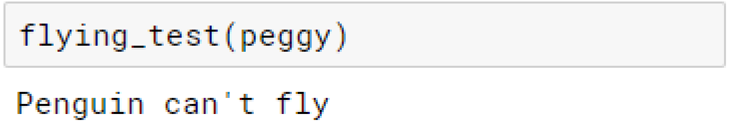


```python
# instantiate objects
blu = Parrot()
peggy = Penguin()
```



```python
# passing the object
flying_test(blu)
```

```
Parrot can fly
```



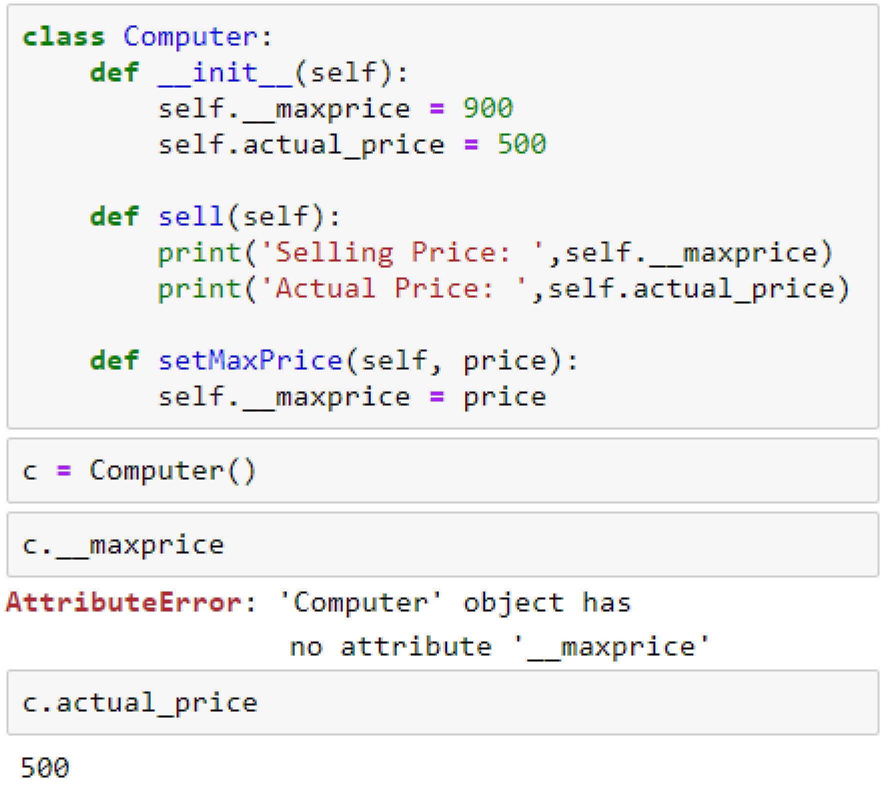
```python
flying_test(peggy)
```

```
Penguin can't fly
```

Data Encapsulation / Abstraction

- Abstraction is an important aspect of object-oriented programming.
- In python, we can perform data hiding by adding the double underscore (_ _) as a prefix to the attribute which is to be hidden. After this, the attribute will not be visible outside of the class through the object.
- Using OOP in Python, we can restrict access to methods and variables. This prevent data from direct modification which is called encapsulation.
- We defined a class Computer. We use __init__() method to store the maximum selling price of computer. We try to modify the price. However, we can’t change it because Python treats the **__maxprice** as private attributes.
- To change the value, we used a setter function **setMaxPrice**() which takes price as parameter.

Data Encapsulation / Abstraction



```python
class Computer:
    def __init__(self):
        self.__maxprice = 900
        self.actual_price = 500

    def sell(self):
        print('Selling Price: ',self.__maxprice)
        print('Actual Price: ',self.actual_price)

    def setMaxPrice(self, price):
        self.__maxprice = price
```

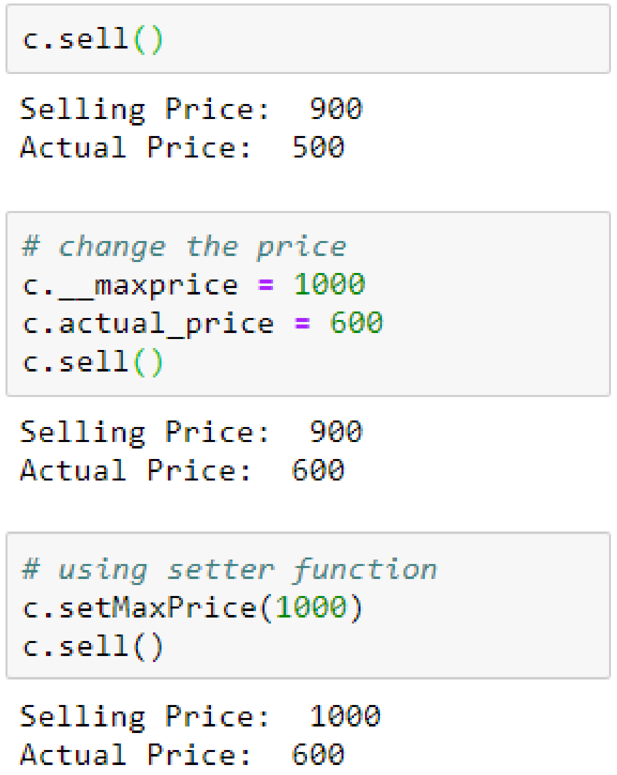
```python
c = Computer()
```

```python
c.__maxprice
```

```
AttributeError: 'Computer' object has
                 no attribute '__maxprice'
```

```python
c.actual_price
```

```
500
```



```python
c.sell()
```

```
Selling Price:  900
Actual Price:  500
```

```python
# change the price
c.__maxprice = 1000
c.actual_price = 600
c.sell()
```

```
Selling Price:  900
Actual Price:  600
```

```python
# using setter function
c.setMaxPrice(1000)
c.sell()
```

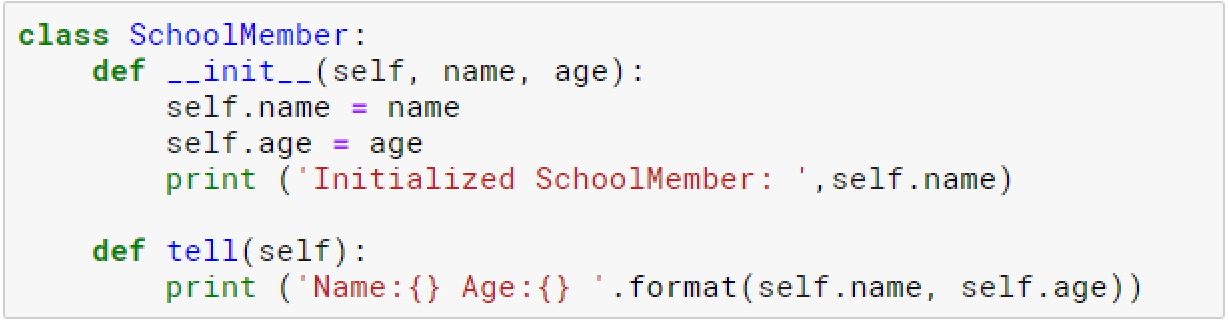
```
Selling Price:  1000
Actual Price:  600
```

Inheritance

- One of the major benefits of object oriented programming is reuse of code and one of the ways this is achieved is through the inheritance mechanism.
- Inheritance can be best imagined as implementing a type and subtype relationship between classes.
- Suppose you want to write a program which has to keep track of the teachers and students in a college. They have some common characteristics such as name, age and address. They also have specific characteristics such as salary, courses and leaves for teachers and, marks and fees for students.
- You can create two independent classes for each type and process them but adding a new common characteristic would mean adding to both of these independent classes. This quickly becomes unwieldy.

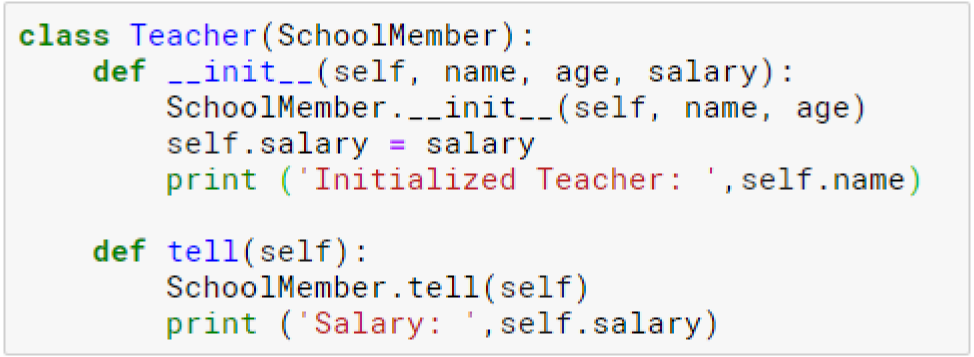
Inheritance

- A better way would be to create a common class called **SchoolMember** and then have the **teacher** and **student** classes inherit from this class i.e. they will become sub-types of this type (class) and then we can add specific characteristics to these sub-types.
- There are many advantages to this approach.
- If we add/change any functionality in **SchoolMember** , this is automatically reflected in the subtypes as well.

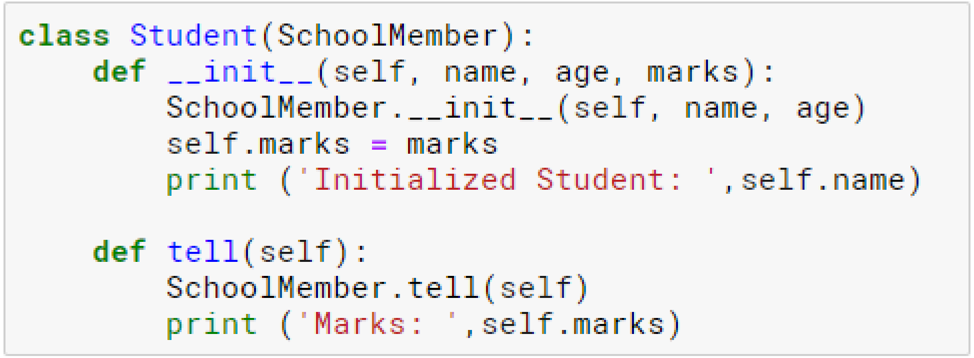


```
class SchoolMember:
    def __init__(self, name, age):
        self.name = name
        self.age = age
        print ('Initialized SchoolMember: ',self.name)

    def tell(self):
        print ('Name:{} Age:{} '.format(self.name, self.age))
```

Inheritance



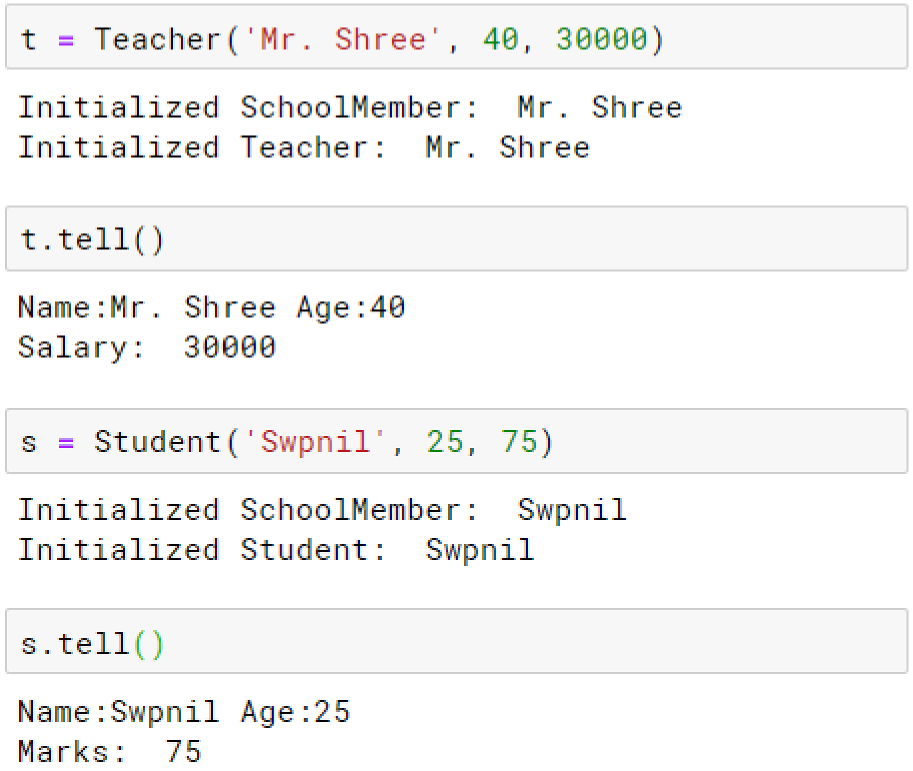
```python
class Teacher(SchoolMember):
    def __init__(self, name, age, salary):
        SchoolMember.__init__(self, name, age)
        self.salary = salary
        print ('Initialized Teacher: ',self.name)

    def tell(self):
        SchoolMember.tell(self)
        print ('Salary: ',self.salary)
```



```python
class Student(SchoolMember):
    def __init__(self, name, age, marks):
        SchoolMember.__init__(self, name, age)
        self.marks = marks
        print ('Initialized Student: ',self.name)

    def tell(self):
        SchoolMember.tell(self)
        print ('Marks: ',self.marks)
```

- Also observe that we reuse the code of the parent class and we do not need to repeat it in the different classes as we would have had to in case we had used independent classes.
- The **SchoolMember** class in this situation is known as the *base class* or the *superclass*.
- The **Teacher** and **Student** classes are called the *derived classes* or *subclasses*.

Inheritance



```
t = Teacher('Mr. Shree', 40, 30000)
```

```
Initialized SchoolMember:  Mr. Shree
Initialized Teacher:  Mr. Shree
```

```
t.tell()
```

```
Name:Mr. Shree Age:40
Salary:  30000
```

```
s = Student('Swpnil', 25, 75)
```

```
Initialized SchoolMember:  Swpnil
Initialized Student:  Swpnil
```

```
s.tell()
```

```
Name:Swpnil Age:25
Marks:  75
```