Main Concepts of Object-Oriented Programming (OOPs)
Class
Objects
Polymorphism
Encapsulation
Inheritance
Data Abstraction
Python is a versatile programming language that supports various programming styles, including
object-oriented programming (OOP) through the use of objects and classes.
An object is any entity that has attributes and behaviors. For example, a parrot is an object. It has
attributes - name, age, color, etc.
behavior - dancing, singing, etc.
Similarly, a class is a blueprint for that object.
PYTHON CLASS AND OBJECT:
class Parrot:
class attribute
name = ""
age = 0

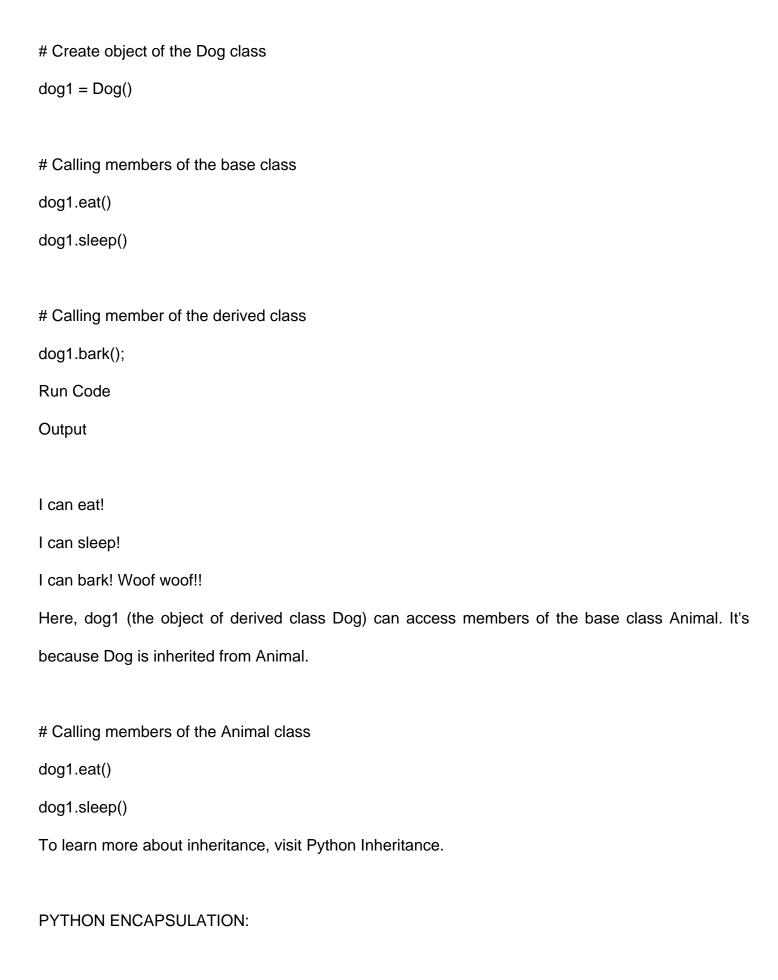
Python Object Oriented Programming

```
# create parrot1 object
parrot1 = Parrot()
parrot1.name = "Blu"
parrot1.age = 10
# create another object parrot2
parrot2 = Parrot()
parrot2.name = "Woo"
parrot2.age = 15
# access attributes
print(f"{parrot1.name} is {parrot1.age} years old")
print(f"{parrot2.name} is {parrot2.age} years old")
Run Code
Output
Blu is 10 years old
Woo is 15 years old
In the above example, we created a class with the name Parrot with two attributes: name and age.
```

Then, we create instances of the Parrot class. Here, parrot1 and parrot2 are references (value) to our new objects.

We then accessed and assigned different values to the instance attributes using the objects name





Encapsulation is one of the key features of object-oriented programming. Encapsulation refers to the bundling of attributes and methods inside a single class.

It prevents outer classes from accessing and changing attributes and methods of a class. This also helps to achieve data hiding.

In Python, we denote private attributes using underscore as the prefix i.e single _ or double ___. For example,

class Computer:

c.___maxprice = 1000

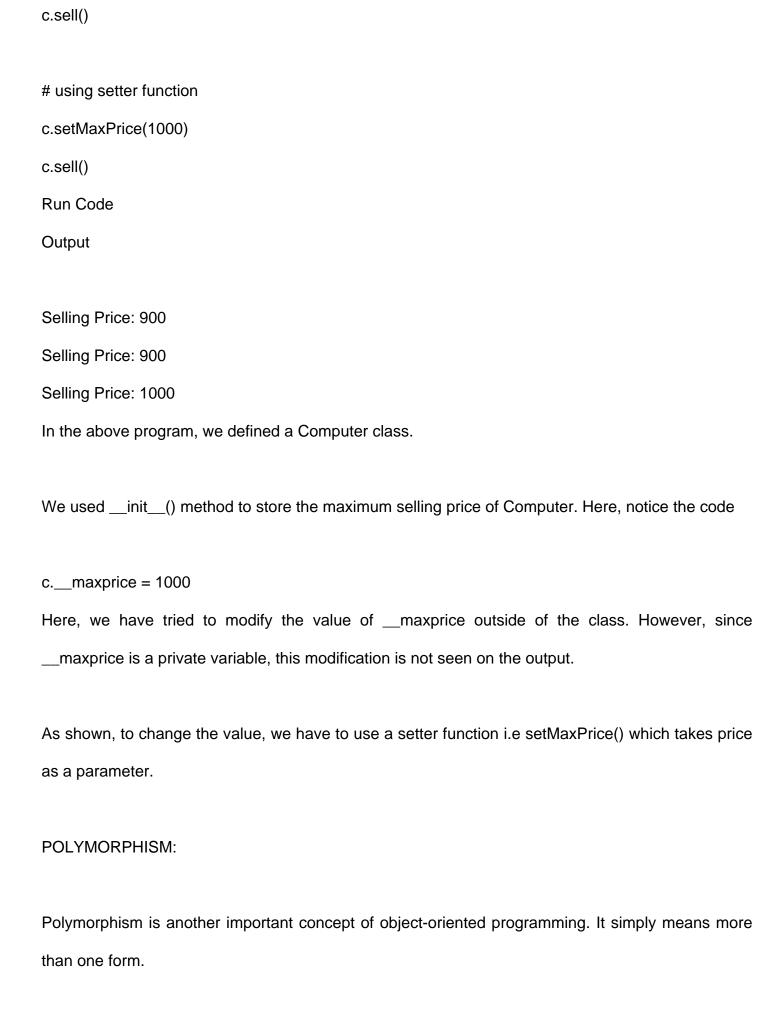
```
def __init__(self):
    self.__maxprice = 900

def sell(self):
    print("Selling Price: {}".format(self.__maxprice))

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

c = Computer()
c.sell()

# change the price
```



That is, the same entity (method or operator or object) can perform different operations in different scenarios.

```
Let's see an example,
class Polygon:
  # method to render a shape
  def render(self):
     print("Rendering Polygon...")
class Square(Polygon):
  # renders Square
  def render(self):
     print("Rendering Square...")
class Circle(Polygon):
  # renders circle
  def render(self):
     print("Rendering Circle...")
# create an object of Square
s1 = Square()
s1.render()
# create an object of Circle
```

c1 = Circle()
c1.render()
Run Code
Output
Rendering Square
Rendering Circle