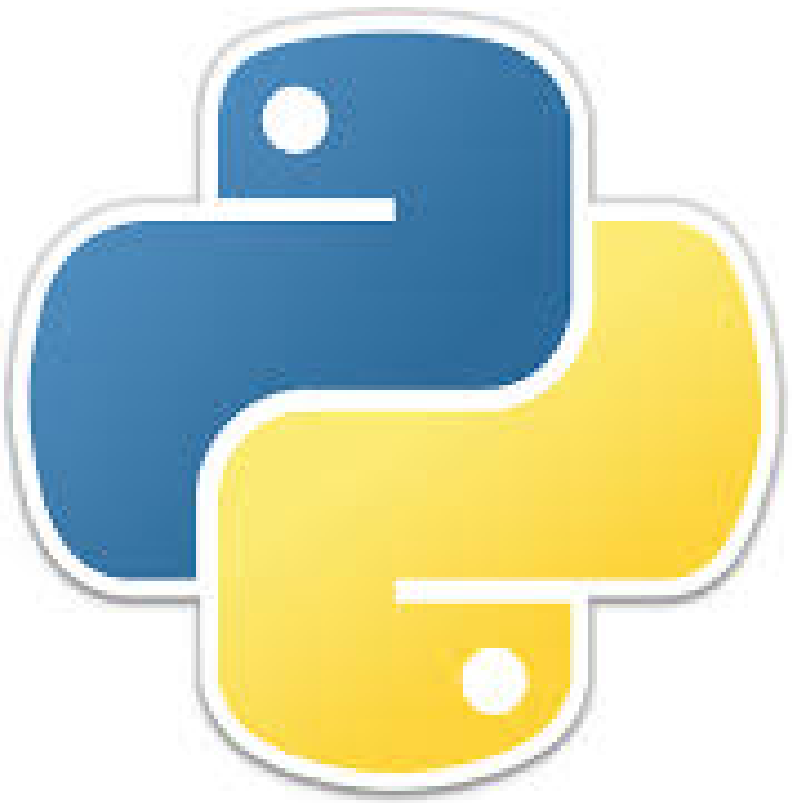

Python Object Oriented Programming

23.09.2019

Python Notes

CLASSES AND OBJECTS

Intro

- Python is an object oriented programming language.
- Almost everything in Python is an object, with its properties and methods.
- A Class is like an object constructor, or a "blueprint" for creating objects.

Create a Class and Object

- To create a class, use the keyword class. Then we can use the class named myClass to create objects:

Example:

```
class MyClass: # define a class
  x = 5

p1 = MyClass() # instantiate an object
```

```
print(p1.x)
```

Attributes and Methods

- To create a class, use the keyword class. Then we can use the class named myClass to create objects:

Example:

```
class Square:

    #here is an attribute
    edge = 5

    #here is a method
    def area(self): # if we don't give self as a parameter, edge will be undefined inside the area function.
        return "Area: "+str((self.edge)**2)  # we can reach the attribute edge only by self.edge

s1 = Square() # create a square object
print(s1.edge) # we can access attributes.
print(s1.area()) # we can access methods.

s1.edge = 7 # change edge
```

Constructor or Initializer (__init__( ) Function)

- The examples above are classes and objects in their simplest form, and are not really useful in real life applications.
- To understand the meaning of classes we have to understand the built-in __init__() function.
- All classes have a function called __init__(), which is always executed when the class is being initiated.
- Use the __init__() function to assign values to object properties, or other operations that are necessary to do when the object is being created:

Example : Create a class named Person, use the __init__() function to assign values for name and age:

```
class Person:
  def __init__(self, name, age):  # Don't need to define name and age globally they will be defined.
    self.name = name
    self.age = age

p1 = Person("John", 36)
```

```
print(p1.name) # access properties
print(p1.age) # access properties
```

The *self* Parameter

- The **`self`** parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class.
- It does not have to be named **`self`** , you can call it whatever you like, but it has to be the first parameter of any function in the class:

Example : Use the words *mysillyobject* and *abc* instead of *self*:

```
class Person:
  def __init__(mysillyobject, name, age):
    mysillyobject.name = name
    mysillyobject.age = age

  def myfunc(abc):
    print("Hello my name is " + abc.name)

p1 = Person("John", 36)
p1.myfunc()
```

Delete Object Parameters or Objects

- You can delete properties on objects or objects itself by using the **`del`** keyword:

Example

```
del p1.age
del p1
```

Encapsulation

- Restricting the access to attributes or methods.
- To make an attribute private we add "__" at the beginning of the related attribute.

Example

```
class BankAccount:
    def __init__(self,name,money,address):
        self.name = name
        self.__money = money # money attribute is private. Accessable only inside the class.
           self.__address = address # address attribute is private. Accessable only inside the class.

    #define get and set methods to access private attributes
    def getMoney(self):
        return self.__money

    def setMoney(self,amount):
        self.__money = amount

    def __increase(self): # a method can be private as well
        self.__money = self.__money + 50000

p1 = BankAccount("Walter",1000000,"Albuquerque")
p2 = BankAccount("Jessi",50000,"Albuquerque")

print("Walter's money:", p1.getMoney())
p1.setMoney(5000000)
print("Walter's money after set:", p1.getMoney())
```

INHERETANCE

Intro

- Inheritance allows us to define a class that inherits all the methods and properties from another class.
- **Parent class** is the class being inherited from, also called base class.
- **Child class** is the class that inherits from another class, also called derived class.

Example

```
class Person: #Parent class
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname
```

```
  def printname(self):
    print(self.firstname, self.lastname)


class Student(Person): #Child class inherits the properties and methods from the Person class:
  pass #Use the pass keyword when you do not want to add any other properties or methods to the class.

x = Student("Mike", "Olsen") # child inherits the constructor of the parent.
x.printname() # child inherits all other attributes and methods as well.
```

Add the __init__( ) Function

- So far we have created a child class that inherits the properties and methods from its parent.
- When you add the `__init__()` function, the child class will no longer inherit the parent's `__init__()` function.
- The child's `__init__()` function **overrides** the inheritance of the parent's `__init__()` function.

Example

```
class Person: #Parent class
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname


class Student(Person): #Child class
  def __init__(self, fname, lname): # overriding the parent's init function
    #add properties etc.
```

- To keep the inheritance of the parent's __init__() function, add a call to the parent's __init__() function:

Example

```
class Student(Person):
  def __init__(self, fname, lname):
    Person.__init__(self, fname, lname) # calls parent's constructor.
#or super().__init__(fname, lname)

# super() function  will make the child class inherit all the methods and properties from its parent:
# By using the super() function, you do not have to use the name of the parent element.
```

Add Properties and Methods to the Child Class

Example

```
# define Parent class
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)


# define Child class
class Student(Person):
  def __init__(self, fname, lname, year): # I want extra attributes for the child like year
    super().__init__(fname, lname) # call parent's constructor
    self.graduationyear = year # add the extra attribute

  def welcome(self): # add the extra method
    print("Welcome", self.firstname, self.lastname, "to the class of", self.graduationyear)

x = Student("Mike", "Olsen", 2019))
```

- If you add a method in the child class with the same name as a function in the parent class, the inheritance of the parent method will be overridden.

Overriding and Polymorphism

- In literal sense, **Polymorphism** means the ability to take various forms.
- Polymorphism allows us to define methods in the child class with the same name as defined in their parent class.
- You will encounter situations where the method inherited from the parent class doesn't quite fit into the child class.
- In such cases, you will have to re-implement method in the child class. This process is known as **Method Overriding**.
- There is **no overloading for python**

Example

```
class Animal: # parent
    def toString(self):
        print("Animal")


class Monkey(Animal): # child
```

```
    def toString(self): # toString method is overrided. Same thing is done before for the __init__() method.
        print("Monkey")


a1 = Animal()
a1.toString() #prints Animal

m1 = Monkey()
m1.toString() #prints Monkey
```

Abstract Classes

- An abstract class can be considered as a blueprint for other classes, allows you to create a set of methods that must be created within any child classes built from your abstract class.
- Abstract classes cannot be instantiated.
- Subclass should override all the abstract methods specified in abstract class

Example

```
from abc import ABC, abstractmethod

class Animal(ABC): #super class

    @abstractmethod
    def walk(self):pass # must be overrided

    @abstractmethod
    def run(self):pass #must be overrided


    def scream(self):pass  #don't have to override


class Bird(Animal): #child class

    def __init__(self):
        print("bird")

    def walk(self): # abstract methods are overrided
        print("walk")
    def run(self): # abstract methods are overrided
        print("run")
```

More blogs