**Class and Concepts**

Python is an object oriented programming language. One of the popular approach to solve a programming problem is by creating objects. This is known as Object-Oriented Programming (OOP).

An object has two characteristics:

* attributes
* Behavior

Let's take an example: Parrot is an object,

* name, age, color are attributes

singing, dancing are behavior

**Class**

A class is a blueprint for the object. We can think of class as a sketch of a parrot with labels. It contains all the details about the name, colors, size etc. Based on these descriptions, we can study about the parrot. Here, parrot is an object.

Eg: class Parrot:

#code to pass

**Object**

An object (instance) is an instantiation of a class. When class is defined, only the description for the object is defined. Therefore, no memory or storage is allocated.

Eg: obj = Parrot()

**Methods**

Methods are functions defined inside the body of a class. They are used to define the behaviors of an object.

def func(self):

print('Hello')

#invocation : obj.func()

**Constructors in Python**

Class functions that begins with double underscore (\_\_) are called *special functions* as they have *special meaning*.

**\_\_init\_\_() function**

This special function gets called whenever a new object of that class is instantiated.

**Example:**

class Employee:

'Common base class for all employees'

empCount = 0

def \_\_init\_\_(self, name, salary):

self.name = name

self.salary = salary

Employee.empCount += 1

def displayCount(self):

print("Total employee",Employee.empCount)

def displayEmployee(self):

print("Name: ",self.name,"\nSalary: ",self.salary)

print(Employee.\_\_doc\_\_)

emp1 = Employee("Zara", 2000) #This would create first object of Employee class

emp2 = Employee("Manni", 5000) #This would create second object of Employee class

emp1.displayEmployee()

emp2.displayEmployee()

emp1.displayCount()

emp2.displayCount()

print("Total employee",Employee.empCount)

**RESULT:**

Common base class for all employees

Name: Zara

Salary: 2000

Name: Manni

Salary: 5000

Total employee 2

Total employee 2

Total employee 2

///////////////////////////////////////////////////////////

* This type of function is also called constructors in Object Oriented Programming (OOP).

We normally use it to initialize all the variables.

* You may have noticed the *self* parameter in function definition inside the class but,

we called the method simply as *obj.func()* without any arguments. It still worked.

* This is because, whenever an object calls its method, the object itself is passed as the

first argument. So, *obj.func()* translates into  *Parrot.func(obj)*.

* In general, calling a method with a list of *n* arguments is equivalent to calling the

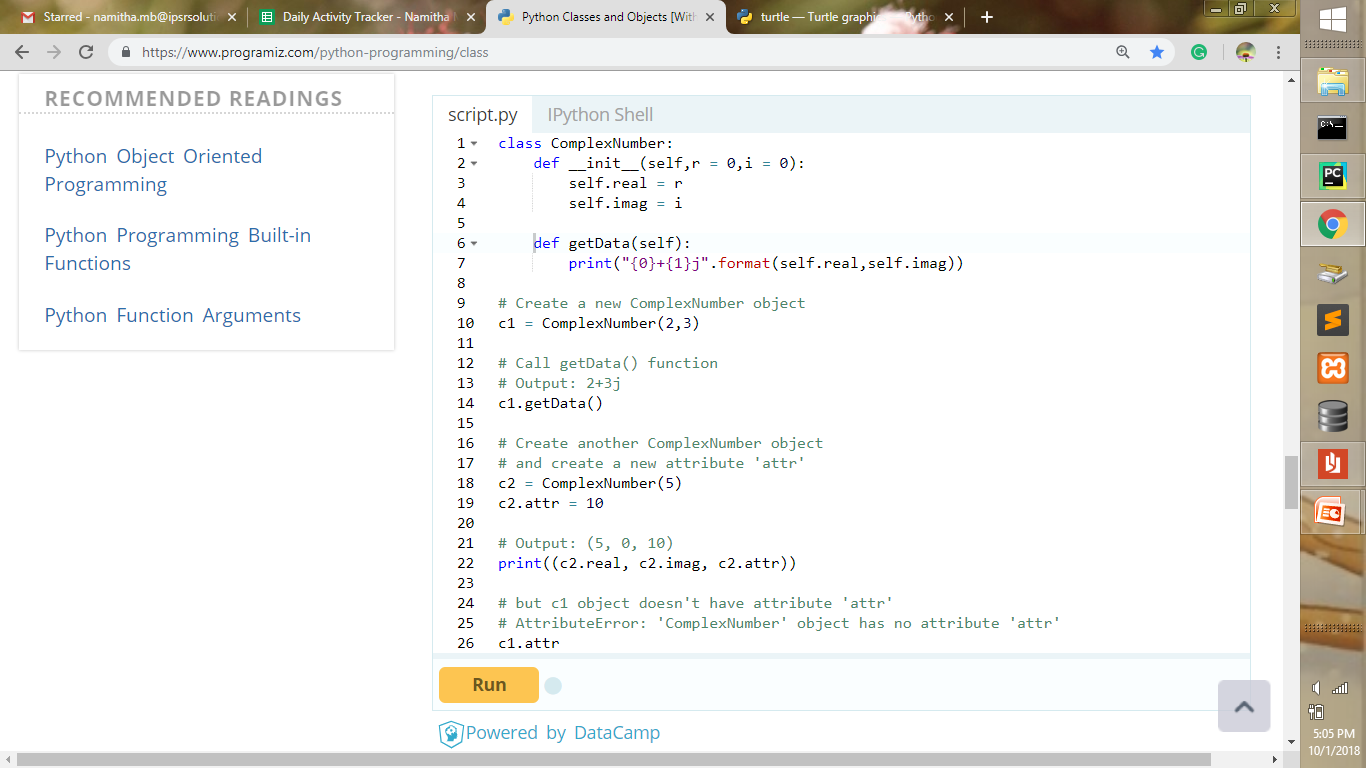
corresponding function with an argument list that is created by inserting the method's

object before the first argument.

* For these reasons, the first argument of the function in class must be the object itself.

This is conventionally called *self*. It can be named otherwise but we highly recommend

to follow the convention.



**EXAMPLE:**

class C:

def \_\_init\_\_(self,r=0,i=0):

self.real = r

self.imag = i

def getData(self):

print(self.real,' : ',self.imag)

c1 = C(2,3)

c1.getData()

c2 = C(5)

print(c2.real)

print(c2.imag)

**RESULT:**

2 : 3

5

0

10

Traceback (most recent call last):

File "C:\Users\user\Desktop\pythn.py", line 14, in <module>

print(c1.attr)

AttributeError: 'C' object has no attribute 'attr'

//////////////////////////////////////////////////////////

In the given example, we define a new class to represent numbers.

It has two functions, *\_\_init\_\_()* to initialize the variables (defaults to zero) and *getData()* to display the number properly.

We created a new attribute *attr* for object *c2* and we read it as well.

**EXAMPLE:**

class Parrot:  
 *# class attribute* species = "bird"  
  
 *# instance attribute* def \_\_init\_\_(self, name, age):  
 self.name = name  
 self.age = age  
 def method1(self):  
 print(self.name, " is ", self.age, " years old")  
  
  
*# instantiate the Parrot class*blu = Parrot("Blue", 10)  
woo = Parrot("Wool", 15)  
egle=Parrot("meetu",12)  
egle.method1()  
  
*# access the class attributes*print("Blue is a ", blu.\_\_class\_\_.species)  
print("Wool is also a ", woo.species)  
  
*# access the instance attributes*print(blu.name, " is ", blu.age, " years old")  
print(woo.name, "is", woo.age, " years old")

**RESULT:**

meetu is 12 years old

Blue is a bird

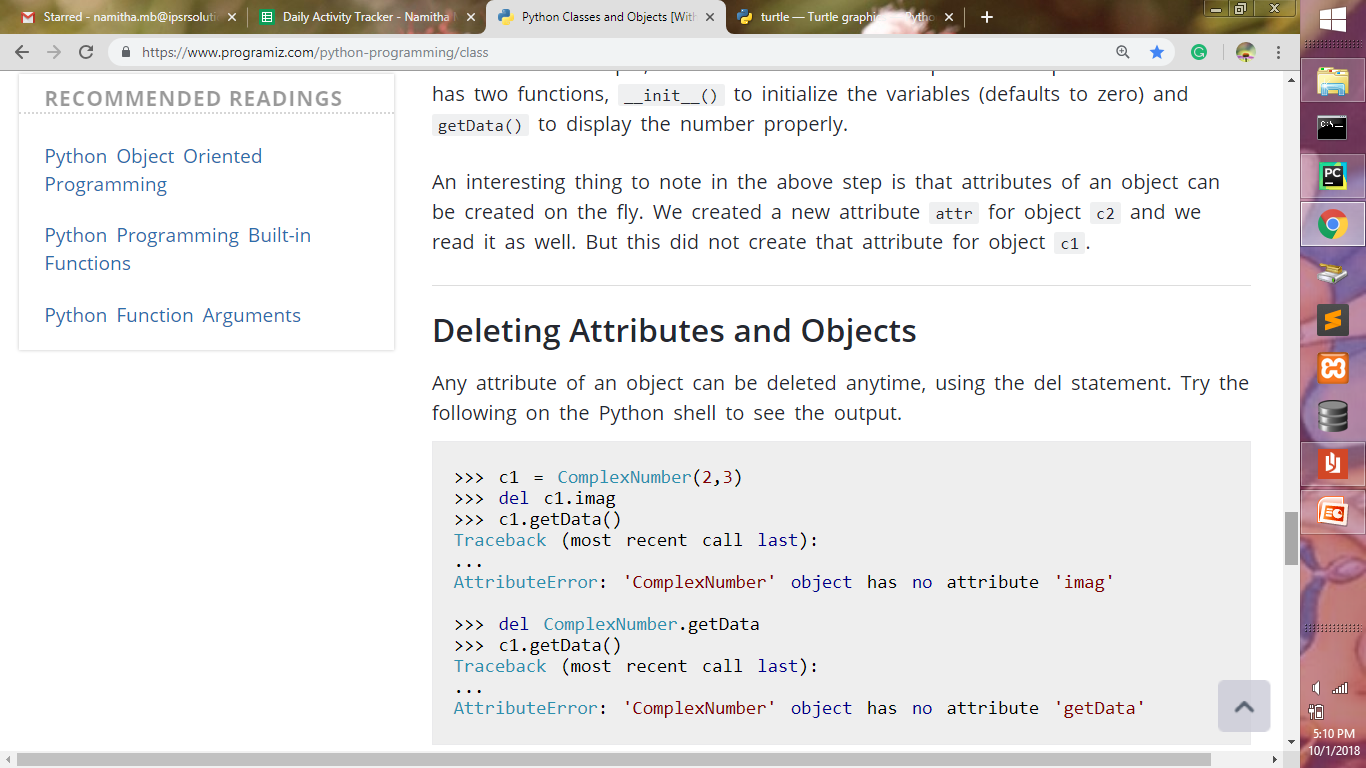
Wool is also a bird

Blue is 10 years old

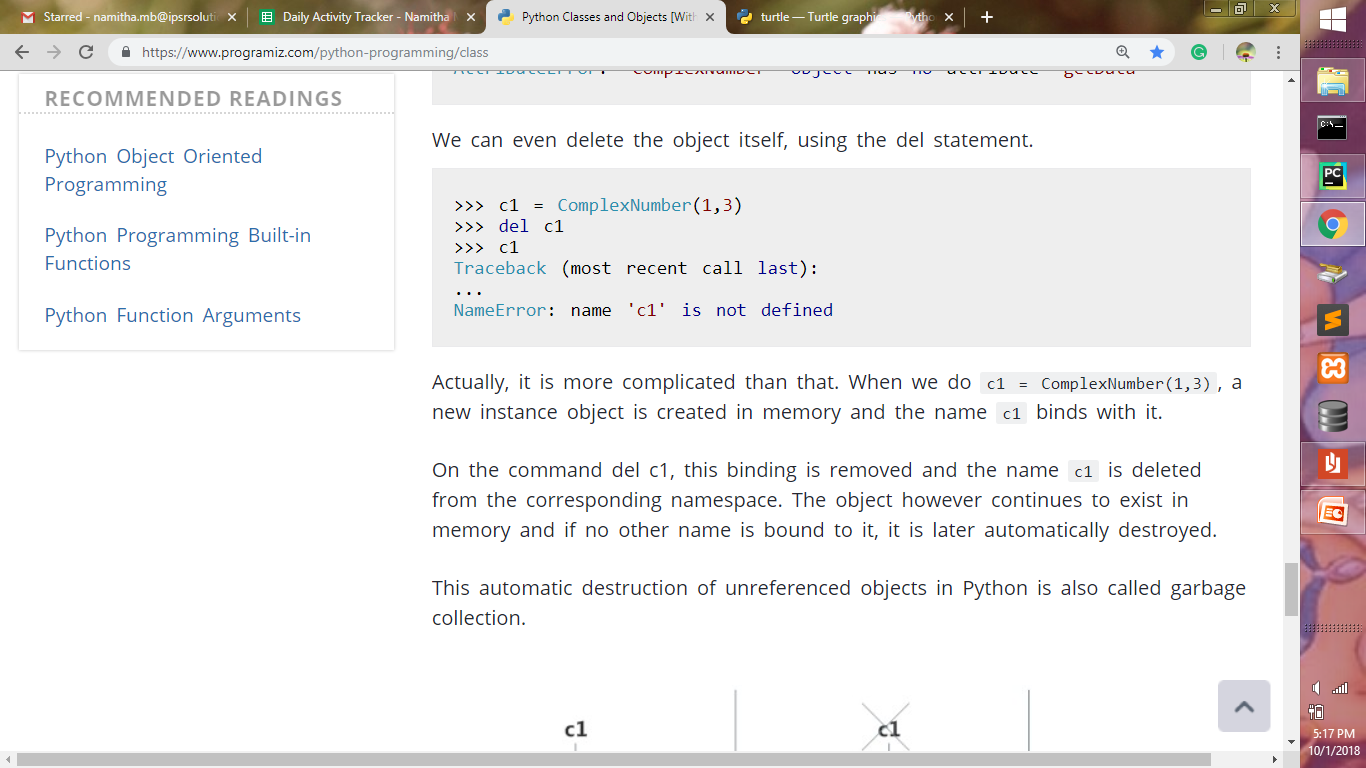
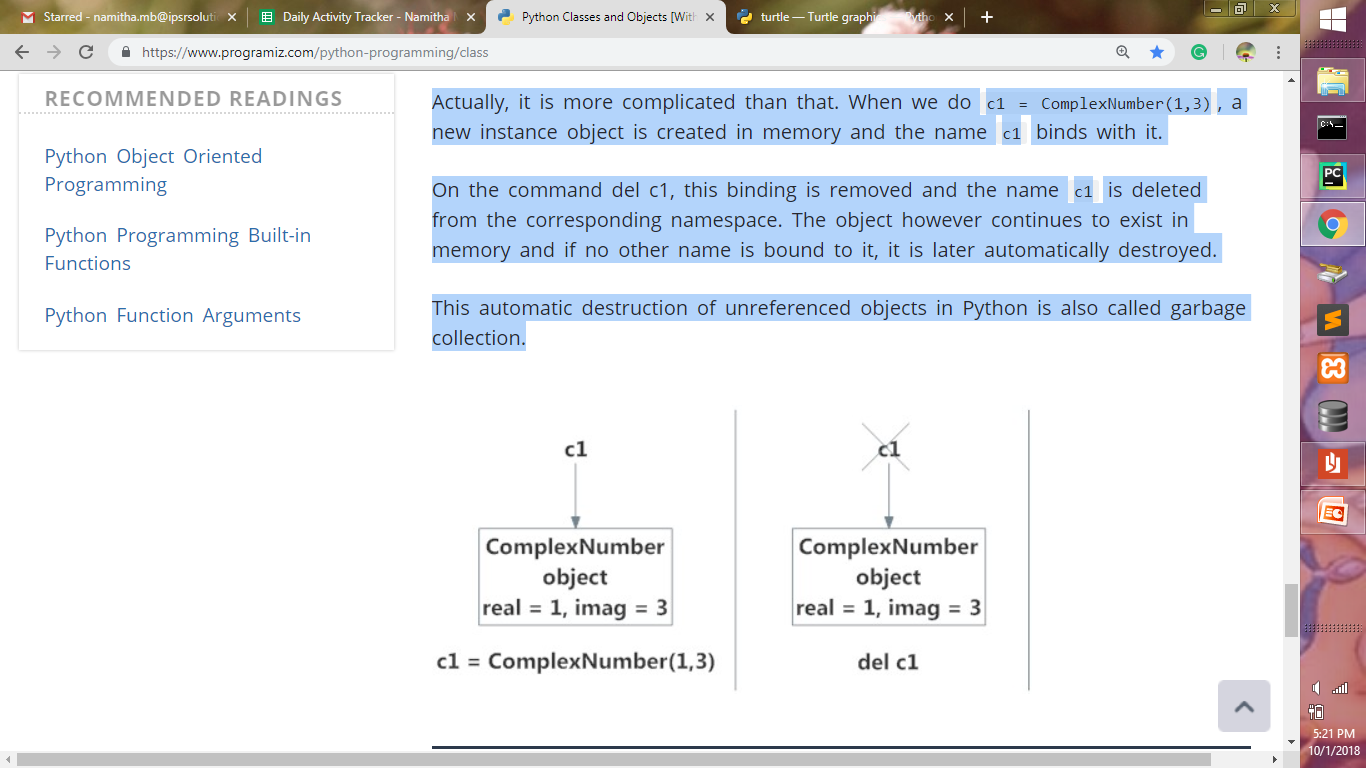
Wool is 15 years old

**Destructors in Python**

Any attribute of an object can be deleted anytime, using the *del* statement



We can even delete the object itself, using the *del* statement.

**EXAMPLE:**

class C:

def \_\_init\_\_(self,r=0,i=0):

self.real = r

self.imag = i

def getData(self):

print(self.real,' : ',self.imag)

c1 = C(2,3)

#del c1.imag

#del C.getData

#del c1

print(c1.imag)

print(c1.real)

c1.getData()

//////////////////////////////////

* Actually, it is more complicated than that. When we do *c1 = ComplexNumber(1,3),*

a new instance object is created in memory and the name *c1* binds with it.

* On the command *del c1*, this binding is removed and the name c1 is deleted from the

corresponding namespace.

* The object however continues to exist in memory and if no other name is bound to it,

It is later automatically destroyed.

* This automatic destruction of unreferenced objects in Python is also called **garbage**

**collection**.

**Built-in class attributes**

According to *Python docs*, the following are the *built-In Class Attributes*

Every python class keeps following built-in attributes and they can be accessed using dot (.) operator like any other attribute:

Class name. \_\_dict\_\_ : Dictionary containing the class's namespace.

Class name. \_\_doc\_\_ : Class documentation string or None if undefined.

Class name.\_\_name\_\_: Class name.

Class name.\_\_module\_\_: Module name in which the class is defined.

– This attribute is "\_\_main\_\_" in interactive mode.

Classname.\_\_bases\_\_ : A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.

**EXAMPLE:**

class A:

print('Friday')

class C(A):

"hello"

def \_\_init\_\_(self,r=0,i=0):

self.real = r

self.imag = i

def getData(self):

print(self.real,' : ',self.imag)

print(C.\_\_dict\_\_)

print("\n")

print(C. \_\_doc\_\_ )

print("\n")

print(C.\_\_name\_\_)

print("\n")

print(C.\_\_module\_\_)

print("\n")

print(C.\_\_bases\_\_)

**RESULT:**

Friday

{'\_\_module\_\_': '\_\_main\_\_', '\_\_doc\_\_': 'hello', '\_\_init\_\_': <function C.\_\_init\_\_ at 0x0000023B0CA1C1E0>, 'getData': <function C.getData at 0x0000023B0F136400>}

hello

C

\_\_main\_\_

(<class '\_\_main\_\_.A'>,)

/////////////////////////////////////////

**Inheritance**

Inheritance enables us to define a class that takes all the functionality from parent class and allows us to add more.

Inheritance is a powerful feature in object oriented programming. It refers to defining a new [class](https://www.programiz.com/python-programming/class) with little or no modification to an existing class. The new class is called ***derived (or child)* class** and the one from which it inherits is called the ***base (or parent)* class**.

Python Inheritance Syntax

class BaseClass:

Body of base class

class DerivedClass(BaseClass):

Body of derived class

*Derived class* inherits features from the *base class*, adding new features to it. This results into re-usability of code.

**Example:**

class Person:

def \_\_init\_\_(self, first, last):

self.firstname = first

self.lastname = last

def Name(self):

return self.firstname + " " + self.lastname

class Employee(Person):

def \_\_init\_\_(self, first, last, staffnum):

Person.\_\_init\_\_(self,first, last)

self.staffnumber = staffnum

def GetEmployee(self):

return self.Name() + ", " + self.staffnumber

x = Person("Ganga", "Harikumar")

y = Employee("Jaison", "Simpson", "1007")

print(x.Name())

print(y.Name())

print(y.GetEmployee())

**Result:**

Ganga Harikumar

Jaison Simpson

Jaison Simpson, 1007

**Super()**

In Python, *super()* built-in has two major use cases:

* Allows us to avoid using base class explicitly
* Working with Multiple Inheritance

**Example:**

class A:

def \_\_init\_\_(self, zz):

print(zz, 'is an animal.');

class B(A):

def \_\_init\_\_(self, z):

print(z, 'is a warm-blooded animal.')

super().\_\_init\_\_(z)

class C(B):

def \_\_init\_\_(self, y):

print(y, "can't fly.")

super().\_\_init\_\_(y)

class D(C):

def \_\_init\_\_(self, x):

print(x, "can't swim.")

super().\_\_init\_\_(x)

class E(D):

def \_\_init\_\_(self):

print('Dog has 4 legs.');

super().\_\_init\_\_('Dog')

d = E()

**Result:**

Dog has 4 legs.

Dog can't swim.

Dog can't fly.

Dog is a warm-blooded animal.

Dog is an animal.

**Example:**

1. class Animal:
2. def \_\_init\_\_(self, animalName):
3. print(animalName, 'is an animal.');
4. class Mammal(Animal):
5. def \_\_init\_\_(self, mammalName):
6. print(mammalName, 'is a warm-blooded animal.')
7. super().\_\_init\_\_(mammalName)
9. class NonWingedMammal(Mammal):
10. def \_\_init\_\_(self, NonWingedMammalName):
11. print(NonWingedMammalName, "can't fly.")
12. super().\_\_init\_\_(NonWingedMammalName)
13. class NonMarineMammal(Mammal):
14. def \_\_init\_\_(self, NonMarineMammalName):
15. print(NonMarineMammalName, "can't swim.")
16. super().\_\_init\_\_(NonMarineMammalName)
17. class Dog(NonMarineMammal, NonWingedMammal):
18. def \_\_init\_\_(self):
19. print('Dog has 4 legs.');
20. super().\_\_init\_\_('Dog')
22. d = Dog()
23. print('')
24. bat = NonMarineMammal('Bat')

**Result:**

Dog has 4 legs.

Dog can't swim.

Dog can't fly.

Dog is a warm-blooded animal.

Dog is an animal.

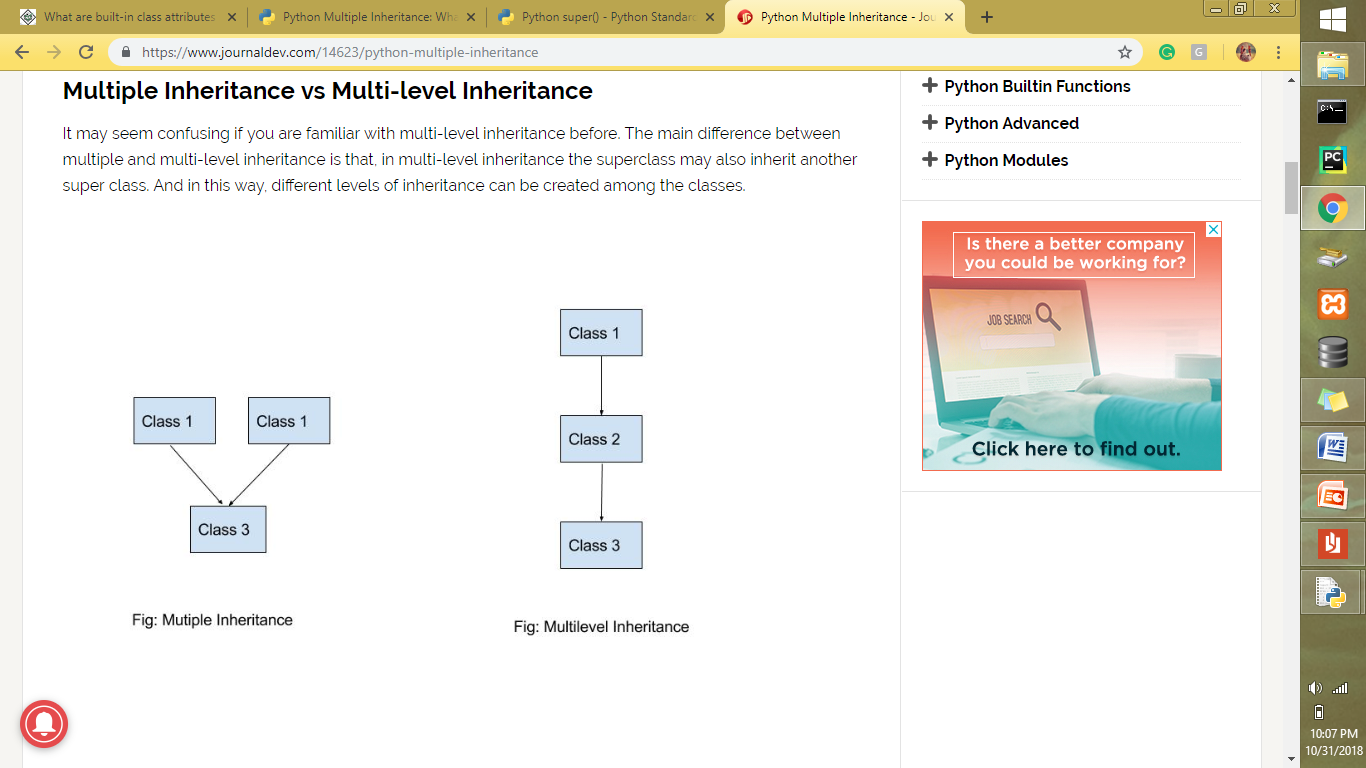
Bat can't swim.

Bat is a warm-blooded animal.

Bat is an animal.

**Multiple Inheritance**

One class extending more than one class is called *multiple inheritance*. This is one of the cool specialties of python which makes it more convenient than java in some cases (Java doesn’t support multiple inheritance).



**Example:**

class A:

def \_\_init\_\_(self):

self.name = 'John'

self.age = 23

def getName(self):

return self.name

class B:

def \_\_init\_\_(self):

self.name = 'Richard'

self.id = '32'

def getName(self):

return self.name

class C(A, B):

def \_\_init\_\_(self):

A.\_\_init\_\_(self)

B.\_\_init\_\_(self)

def getName(self):

return self.name

C1 = C()

print(C1.getName())

**RESULT**

Richard

/////////////////////////////////

Class C inherits both A and B. Both of them has an attribute ‘name’.

The ‘name’ when printed is ‘Richard’ instead of ‘John’. In the constructor of C, the first constructor called is the one of A. So, the value of name in C becomes same as the value of name in A. But after that, when the constructor of B is called, the value of name in C is overwritten by the value of name in B. So, the name attribute of C retains the value ‘Richard’ when printed. The result would be same even if we declared class C as:

Class C(B, A):

The hierarchy becomes completely depended on the order of \_\_init\_\_() calls inside the subclass. To deal with it perfectly, there is a protocol named MRO (Method Resolution Order).

### Method Resolution Order (MRO)

Let’s replace the previous code with a slightly modified version.

**EXAMPLE:**

class A:

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.name = 'John'

self.age = 23

def getName(self):

return self.name

class B:

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.name = 'Richard'

self.id = '32'

def getName(self):

return self.name

class C(A,B):

def \_\_init\_\_(self):

super().\_\_init\_\_()

def getName(self):

return self.name

C1 = C()

print(C1.getName())

**RESULT:**

John

**EXAMPLE:**

class A:

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.name = 'John'

self.age = 23

def getName(self):

return self.name

class B:

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.name = 'Richard'

self.id = '32'

def getName(self):

return self.name

class C(B,A):

def \_\_init\_\_(self):

super().\_\_init\_\_()

def getName(self):

return self.name

C1 = C()

print(C1.getName())

print(C.\_\_mro\_\_)

**RESULT:**

Richard

(<class '\_\_main\_\_.C'>, <class '\_\_main\_\_.B'>, <class '\_\_main\_\_.A'>, <class 'object'>)

///////////////////////////

MRO works left to right way. *super()* in the *\_\_init\_\_* method indicates the class that is in the next hierarchy. At first, the the *super()* of C indicates A. Then *super()* in the constructor of A searches for its superclass. If it doesn’t find any, it executes the rest of the code and returns. So the order in which constructors are called here is:  
C -> B -> A  
If we call *print(C.\_\_mro\_\_)*, then we can see the MRO trace route.

Once the constructor of A is called and attribute ‘name’ is accessed, it doesn’t access the attribute ‘name’ in B.

Last\_class\_name.\_\_mro\_\_

**EXAMPLE**

class a:

print("hello")

class b:

print("hai")

class c:

print("good")

**RESULT**

hello

hai

good

**EXAMPLE OF MULTIPLE INHERITANCE**

#definition of the class starts here

class Person:

#defining constructor

def \_\_init\_\_(self, personName, personAge):

self.name = personName

self.age = personAge

#defining class methods

def showName(self):

print(self.name)

def showAge(self):

print(self.age)

#end of class definition

# defining another class

class Student:

def \_\_init\_\_(self, studentId):

self.studentId = studentId

def getId(self):

return self.studentId

class Resident(Person, Student): # extends both Person and Student class

def \_\_init\_\_(self, name, age, id):

Person.\_\_init\_\_(self, name, age)

Student.\_\_init\_\_(self, id)

# Create an object of the subclass

resident1 = Resident('John', 30, '102')

resident1.showName()

print(resident1.getId())

**RESULT**

John

102

**EXAMPLE OF MULTILEVEL INHERITANCE**

# Python code to demonstrate example of

# multilevel inheritance

class Details1:

def \_\_init\_\_(self):

self.\_\_id=0

print('one')

def setId(self):

self.\_\_id=int(input("Enter Id: "))

def showId(self):

print("Id: ",self.\_\_id)

class Details2(Details1):

def \_\_init\_\_(self):

self.\_\_name=""

print('two')

def setName(self):

self.setId()

self.\_\_name=input("Enter Name: ")

def showName(self):

self.showId()

print("Name: ",self.\_\_name)

class Details3(Details2):

def \_\_init\_\_(self):

self.\_\_gender=""

print('three')

def setGender(self):

self.setName()

self.\_\_gender=input("Enter Gender: ")

def showGender(self):

self.showName()

print("Gender: ",self.\_\_gender)

class Employee(Details3):

def \_\_init\_\_(self):

self.\_\_desig=""

self.\_\_dept=""

print('four')

def setEmployee(self):

self.setGender()

self.\_\_desig=input("Enter Designation: ")

self.\_\_dept= input("Enter Department: ")

def showEmployee(self):

self.showGender()

print("Designation: ",self.\_\_desig)

print("Department: ",self.\_\_dept)

e = Employee()

e.setEmployee()

e.showEmployee()

**RESULT**

four

Enter Id: 2

Enter Name: Arun

Enter Gender: M

Enter Designation: Python

Enter Department: Wozti

Id: 2

Name: Arun

Gender: M

Designation: Python

Department: Wozti

**EXAMPLE OF MULTIPLE INHERITANCE**

class a:

print("hello")

class b:

print("hai")

class c(a,b):

print("good")

d = c()

**RESULT**

hello

hai

good

**EXAMPLE**

class A:

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.name = 'Job'

self.age = 23

def getName(self):

return self.name

class B:

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.name = 'kohli'

self.age = 24

def getName(self):

return self.name

class C:

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.name = 'Sehwag'

self.age = 25

def getName(self):

return self.name

class D:

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.name = 'Rohit'

self.age = 26

def getName(self):

return self.name

class E:

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.name = 'Bretlee'

self.age = 27

def getName(self):

return self.name

class F:

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.name = 'Richard'

self.id = '32'

def getName(self):

return self.name

class G(A, B, C, D, E, F):

def \_\_init\_\_(self):

super().\_\_init\_\_()

def getName(self):

return self.name

C1 = G()

print(C1.getName())

C1 = F()

print(C1.getName())

C1 = E()

print(C1.getName())

C1 = D()

print(C1.getName())

C1 = C()

print(C1.getName())

C1 = B()

print(C1.getName())

C1 = A()

print(C1.getName())

**RESULT**

Job

Richard

Bretlee

Rohit

Sehwag

kohli

Job

**EXAMPLE**

class List:

def \_\_init\_\_(self, L1, L2):

self.L1 = L1

self.L2 = L2

def add(self):

L3=self.L1+self.L2

return L3

def substract(self):

L3 = self.L1-self.L2

return L3

def multiply(self):

L3 = self.L1 \* self.L2

return L3

nm=List(20,11)

print(nm.add())

print(nm.substract())

print(nm .multiply())

**RESULT**

31

9

220

**EXAMPLE**

class List:

def \_\_init\_\_(self, L1):

self.L1=L1

def append(self, r):

self.L1.append(r)

return self.L1

def delete(self, s):

del self.L1[s]

return self.L1

def display(self):

return self.L1

L1=[1, 2, 3]

mm=List(L1)

print(mm.display())

print(mm.append(15))

print(mm.delete(0))

**RESULT**

[1, 2, 3]

[1, 2, 3, 15]

[2, 3, 15]

**EXAMPLE**

class Area\_rectangle:

def \_\_init\_\_(self, length, breadth):

self.length=length

self.breadth=breadth

def ART(self):

self.Area = self.length \* self.breadth

return self.Area

Are= Area\_rectangle(4,5)

print (Are.ART())

**RESULT**

20

**EXAMPLE**

class Animal:

def \_\_init\_\_(self, Animal):

print(Animal, 'is an animal.');

class Mammal(Animal):

def \_\_init\_\_(self, mammalName):

print(mammalName, 'is a warm-blooded animal.')

super().\_\_init\_\_(mammalName)

class NonWingedMammal(Mammal):

def \_\_init\_\_(self, NonWingedMammal):

print(NonWingedMammal, "can't fly.")

super().\_\_init\_\_(NonWingedMammal)

class NonMarineMammal(Mammal):

def \_\_init\_\_(self, NonMarineMammal):

print(NonMarineMammal, "can't swim.")

super().\_\_init\_\_(NonMarineMammal)

class Dog(NonMarineMammal, NonWingedMammal):

def \_\_init\_\_(self):

print('Dog has 4 legs.');

super().\_\_init\_\_('Dog')

d = Dog()

print('')

bat = NonMarineMammal('Bat')

print(Dog.\_\_mro\_\_)

**RESULT**

Dog has 4 legs.

Dog can't swim.

Dog can't fly.

Dog is a warm-blooded animal.

Dog is an animal.

Bat can't swim.

Bat is a warm-blooded animal.

Bat is an animal.

(<class '\_\_main\_\_.Dog'>, <class '\_\_main\_\_.NonMarineMammal'>, <class '\_\_main\_\_.NonWingedMammal'>, <class '\_\_main\_\_.Mammal'>, <class '\_\_main\_\_.Animal'>, <class 'object'>)

//////////////////////////////////

Every class in Python is derived from the class *object*. It is the most base type in Python.

So technically, all other class, either built-in or user-defines, are derived classes and all objects are instances of object class.

////////////////////////////////

**EXAMPLE**

print(issubclass(list,object))

print(isinstance(5.5,object))

print(isinstance("Hello",object))

**RESULT**

True

True

True

//////////////////////////////////

Method resolution order (MRO) of a class can be viewed as the *\_\_mro\_\_* attribute or *mro()* method. The former returns a tuple while latter returns a list.

////////////////////////////////

**EXAMPLE**

class a:

print("hello")

class b(a):

print("hai")

class c(b):

print("good")

print(c.\_\_mro\_\_)

print(c.mro())

**RESULT**

hello

hai

good

(<class '\_\_main\_\_.c'>, <class '\_\_main\_\_.b'>, <class '\_\_main\_\_.a'>, <class 'object'>)

[<class '\_\_main\_\_.c'>, <class '\_\_main\_\_.b'>, <class '\_\_main\_\_.a'>, <class 'object'>]