UNIT – I

Oops Concepts- class, object, constructors, types of variables, types of methods. **Inheritance:** single, multiple, multi-level, hierarchical, hybrid, **Polymorphism:** with functions and objects, with class methods, with inheritance,**Abstraction:** abstract classes.

OOPs in Python

OOPs in Python is a programming approach that focuses on using objects and classes as same as other general programming languages. The objects can be any real-world entities. Python allows developers to develop applications using the OOPs approach with the major focus on code reusability.

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, a parrot is an object.

The example for class of parrot can be :

```
class Parrot:
    pass
```

Here, we use the class keyword to define an empty class Parrot. From class, we construct instances. An instance is a specific object created from a particular class.

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.

The example for object of parrot class can be:

```
obj = Parrot()
```

Here, obj is an object of class Parrot.

Suppose we have details of parrots. Now, we are going to show how to build the class and objects of parrots.

Example:

```
class Parrot:

    # class attribute
    species = "bird"

    # instance attribute
    def_init_(self, name, age):
        self.name = name
        self.age = age

# instantiate the Parrot class
blu = Parrot("Blu", 10)
woo = Parrot("Woo", 15)

# access the class attributes
print("Blu is a {}".format(blu._class_.species))
print("Woo is also a {}".format(woo._class_.species))

# access the instance attributes
print("{} is {} years old".format( blu.name, blu.age))
print("{} is {} years old".format( woo.name, woo.age))
```

Output

```
Blu is a bird
Woo is also a bird
Blu is 10 years old
Woo is 15 years old
```

In the above program, we created a class with the name Parrot. Then, we define attributes. The attributes are a characteristic of an object.

These attributes are defined inside the __init__ method of the class. It is the initializer method that is first run as soon as the object is created.

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

We can access the class attribute using_class_.species. Class attributes are the same for all instances of a class. Similarly, we access the instance attributes using blu.name and blu.age. However, instance attributes are different for every instance of a class.

constructor

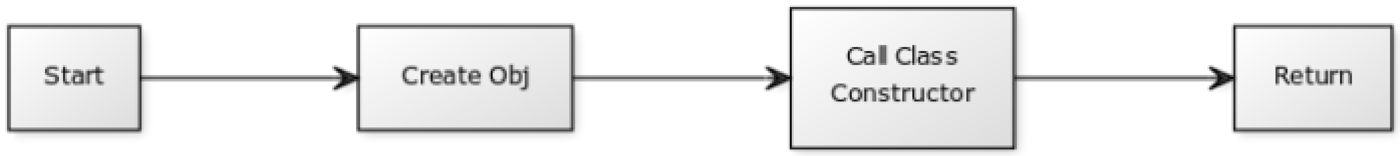
The constructor is a method that is called when an object is created. This method is defined in the class and can be used to initialize basic variables.

If you create four objects, the class constructor is called four times. Every class has a constructor, but its not required to explicitly define it.

Example:

Each time an object is created a method is called. That methods is named the **constructor**.

The constructor is created with the function **init**. As parameter we write the self keyword, which refers to itself (the object). The process visually is:



Inside the constructor we initialize two variables: legs and arms. Sometimes variables are named properties in the context of object oriented programming. We create one object (bob) and just by creating it, its variables are initialized.

```
classHuman:
def_init_(self):
      self.legs = 2
      self.arms = 2

bob = Human()
print(bob.legs)
```

The newly created object now has the variables set, without you having to define them manually. You could create tens or hundreds of objects without having to set the values each time.

python__init__

The function **init**(self) builds your object. Its not just variables you can set here, you can call class methods too. Everything you need to initialize the object(s).

Lets say you have a class Plane, which upon creation should start flying. There are many steps involved in taking off: accelerating, changing flaps, closing the wheels and so on.

The default actions can be defined in methods. These methods can be called in the constructor.

```
classPlane:
def_init_(self):
      self.wings = 2

# fly
      self.drive()
      self.flaps()
      self.wheels()

defdrive(self):
         print('Accelerating')

defflaps(self):
         print('Changing flaps')

defwheels(self):
         print('Closing wheels')

ba = Plane()
```

To summarize: A constructor is called if you create an object. In the constructor you can set variables and call methods.

Default value

The constructor of a class is unique: initiating objects from different classes will call different constructors.

Default values of newly created objects can be set in the constructor.

The example belwo shows two classes with constructors. Then two objects are created but different constructors are called.

```
classBug:
def_init_(self):
     self.wings = 4

classHuman:
def_init_(self):
     self.legs = 2
     self.arms = 2

bob = Human()
tom = Bug()

print(tom.wings)
print(bob.arms)
```

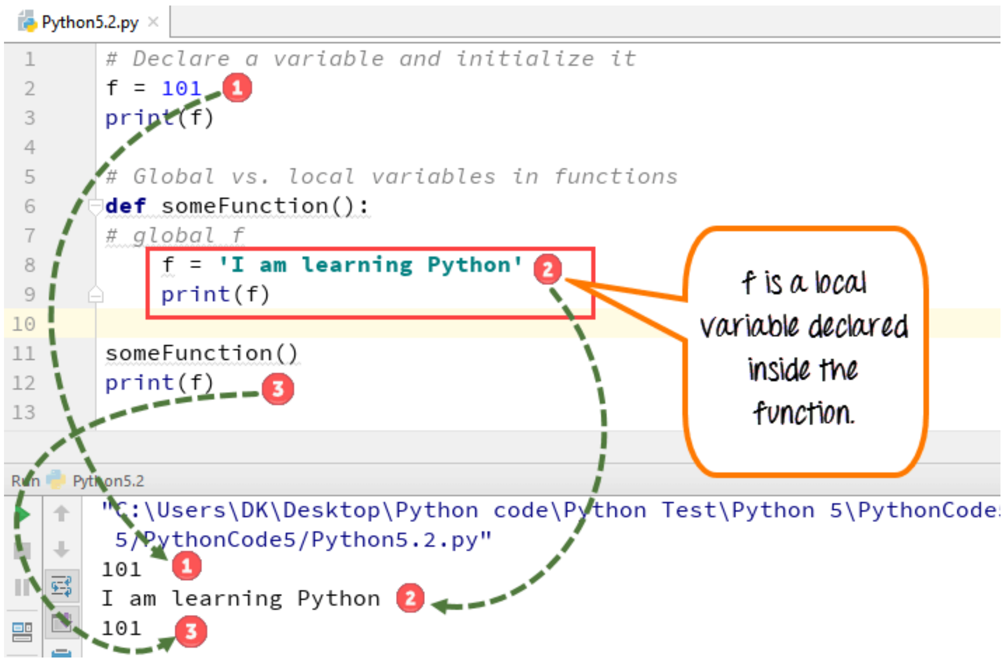
But creating multiple objects from one class, will call the same constructor.

Python Variable Types: Local & Global

There are two types of variables in Python, Global variable and Local variable. When you want to use the same variable for rest of your program or module you declare it as a global variable, while if you want to use the variable in a specific function or method, you use a local variable while Python variable declaration.

Let's understand this Python variable types with the difference between local and global variables in the below program.

1. Let us define variable in Python where the variable "f" is **global** in scope and is assigned value 101 which is printed in output
2. Variable f is again declared in function and assumes **local** scope. It is assigned value "I am learning Python." which is printed out as an output. This Python declare variable is different from the global variable "f" defined earlier
3. Once the function call is over, the local variable f is destroyed. At line 12, when we again, print the value of "f" is it displays the value of global variable f=101



```
1  # Declare a variable and initialize it
2  f = 101
3  print(f)
4
5  # Global vs. local variables in functions
6  def someFunction():
7  # global f
8      f = 'I am learning Python'
9      print(f)
10
11 someFunction()
12 print(f)
13
```

```
"C:\Users\DK\Desktop\Python code\Python Test\Python 5\PythonCode
 5/PythonCode5/Python5.2.py"
101
I am learning Python
101
```

Python 2 Example

```
# Declare a variable and initialize it
f = 101
print f
# Global vs. local variables in functions
def someFunction():
# global f
   f = 'I am learning Python'
   print f
```

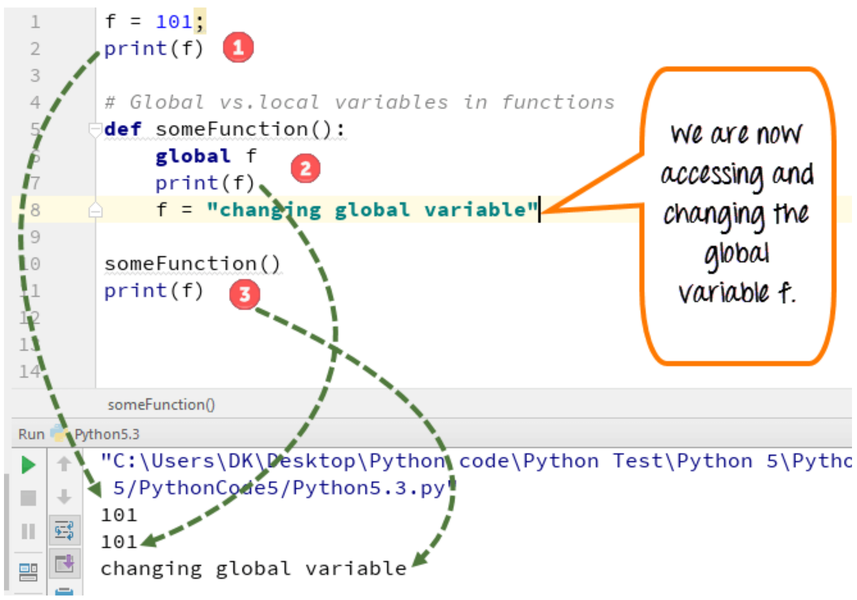
```
someFunction()
print f
```

Python 3 Example

```
# Declare a variable and initialize it
f = 101
print(f)
# Global vs. local variables in functions
def someFunction():
# global f
    f = 'I am learning Python'
    print(f)
someFunction()
print(f)
```

While Python variable declaration using the keyword **global,** you can reference the global variable inside a function.

1. Variable "f" is **global** in scope and is assigned value 101 which is printed in output
2. Variable f is declared using the keyword **global**. This is **NOT** a **local variable**, but the same global variable declared earlier. Hence when we print its value, the output is 101
3. We changed the value of "f" inside the function. Once the function call is over, the changed value of the variable "f" persists. At line 12, when we again, print the value of "f" is it displays the value "changing global variable"



```
1   f = 101;
2   print(f)
3
4   # Global vs.local variables in functions
5   def someFunction():
6       global f
7       print(f)
8       f = "changing global variable"
9
10  someFunction()
11  print(f)
12
13
14
```

```
"C:\Users\DK\Desktop\Python code\Python Test\Python 5\Pythc
 5/PythonCode5/Python5.3.py"
101
101
changing global variable
```

Python 2 Example

```
f = 101;
print f
# Global vs.local variables in functions
def someFunction():
  global f
  print f
  f = "changing global variable"
someFunction()
print f
```

Python 3 Example

```
f = 101;
print(f)
# Global vs.local variables in functions
def someFunction():
  global f
  print(f)
  f = "changing global variable"
someFunction()
print(f)
```

Types of methods:
Generally, there are three types of methods in Python:

1. Instance Methods.
2. Class Methods
3. Static Methods

Before moving on with the topic, we have to know some key concepts.

Class Variable: A class variable is nothing but a variable that is defined outside the constructor. A class variable is also called as a **static variable**.

Accessor(Getters): If you want to fetch the value from an instance variable we call them accessors.

Mutator(Setters): If you want to modify the value we call them mutators.

1. Instance Method

This is a very basic and easy method that we use regularly when we create classes in python. If we want to print an instance variable or instance method we must create an object of that required class.

If we are using self as a function parameter or in front of a variable, that is nothing but the calling instance itself.

As we are working with instance variables we use self keyword.

Note: Instance variables are used with instance methods.

Look at the code below

```
# Instance Method Example in Python

classStudent:



def_init_(self, a, b):

       self.a = a

       self.b = b



defavg(self):

return(self.a + self.b)/2



s1 = Student(10,20)

print( s1.avg())
```

Copy

Output:

```
15.0
```

In the above program, a and b are instance variables and these get initialized when we create an object for the Student class. If we want to call avg() function which is an instance method, we must create an object for the class.

If we clearly look at the program, the self keyword is used so that we can easily say that those are instance variables and methods.

2. Class Method

classsmethod() function returns a class method as output for the given function.

Here is the syntax for it:

```
classmethod(function)
```

The classmethod() method takes only a function as an input parameter and converts that into a class method.

There are two ways to create class methods in python:

1. Using classmethod(function)
2. Using @classmethod annotation

A class method can be called either using the class (such as C.f()) or using an instance (such as C().f()). The instance is ignored except for its class. If a class method is called from a derived class, the derived class object is passed as the implied first argument.

As we are working with ClassMethod we use the cls keyword. Class variables are used with class methods.

Look at the code below.

```
# Class Method Implementation in python

classStudent:

   name ='Student'

def_init_(self, a, b):

      self.a = a

      self.b = b



@classmethod

definfo(cls):

return cls.name



print(Student.info())
```

Copy

Output:

```
Student
```

In the above example, name is a class variable. If we want to create a class method we must use @classmethod decorator and cls as a parameter for that function.

3. Static Method

A static method can be called without an object for that class, using the class name directly. If you want to do something extra with a class we use static methods.

For example, If you want to print factorial of a number then we don't need to use class variables or instance variables to print the factorial of a number. We just simply pass a number to the static method that we have created and it returns the factorial.

Look at the below code

```
# Static Method Implementation in python

classStudent:

    name ='Student'

def_init_(self, a, b):

        self.a = a

        self.b = b



@staticmethod

definfo():

return"This is a student class"



print(Student.info())
```

Copy
Output

```
This a student class
```

Types of inheritances:

The inheritance is a very useful and powerful concept of object-oriented programming. Using the inheritance concept, we can use the existing features of one class in another class.

The inheritance is the process of acquiring the properties of one class to another class.

In inheritance, we use the terms like parent class, child class, base class, derived class, superclass, and subclass.

The **Parent class** is the class which provides features to another class. The parent class is also known as **Base class** or **Superclass**.

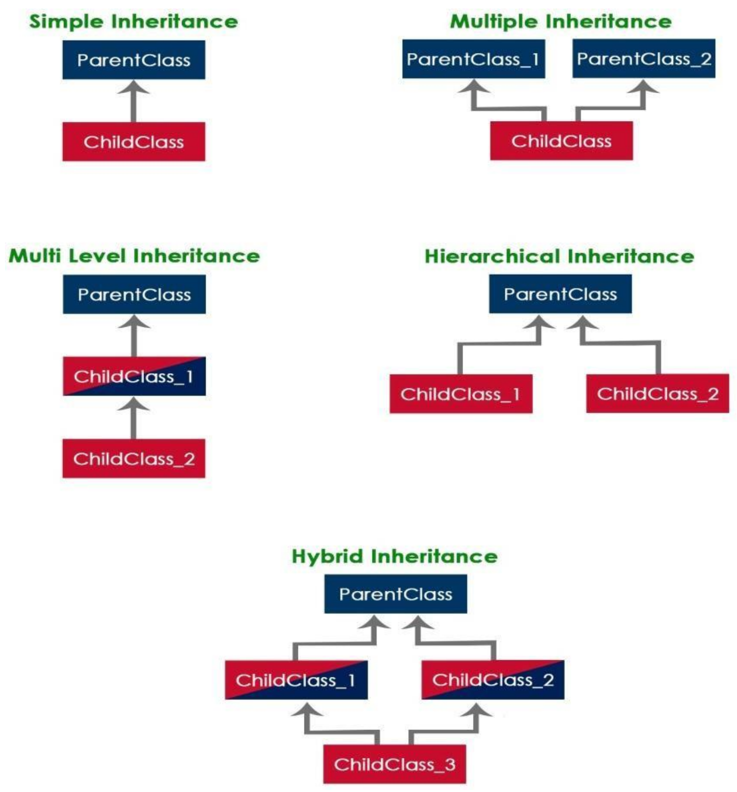
The **Child class** is the class which receives features from another class. The child class is also known as the **Derived Class** or **Subclass**.

In the inheritance, the child class acquires the features from its parent class. But the parent class never acquires the features from its child class.

There are five types of inheritances, and they are as follows.

- **Simple Inheritance (or) Single Inheritance**
- **Multiple Inheritance**
- **Multi-Level Inheritance**
- **Hierarchical Inheritance**
- **Hybrid Inheritance**

The following picture illustrates how various inheritances are implemented.



Creating a Child Class

In Python, we use the following general structure to create a child class from a parent class.

Syntax

```
classChildClassName(ParentClassName):
   ChildClass implementation
.
.
```

Let's look at individual inheritance type with an example.

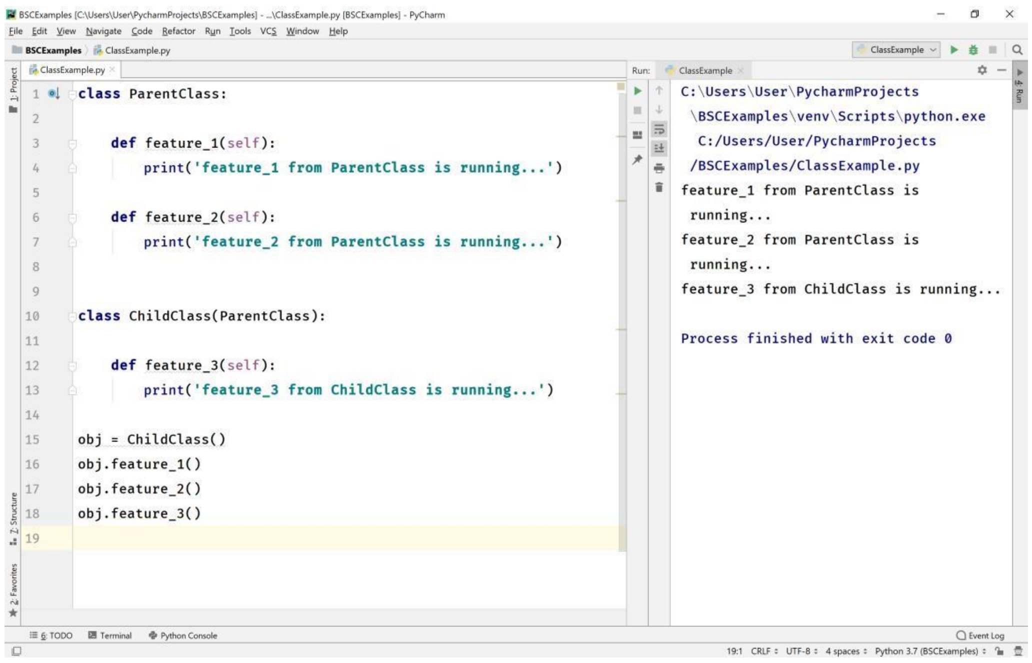
Simple Inheritance (or) Single Inheritance

In this type of inheritance, one child class derives from one parent class. Look at the following example code.

Example

```
classParentClass:

deffeature_1(self):
print('feature_1 from ParentClass is running...')

deffeature_2(self):
print('feature_2 from ParentClass is running...')


classChildClass(ParentClass):

deffeature_3(self):
print('feature_3 from ChildClass is running...')

obj = ChildClass()
obj.feature_1()
obj.feature_2()
obj.feature_3()
```

When we run the above example code, it produces the following output.



```
class ParentClass:

    def feature_1(self):
        print('feature_1 from ParentClass is running...')

    def feature_2(self):
        print('feature_2 from ParentClass is running...')


class ChildClass(ParentClass):

    def feature_3(self):
        print('feature_3 from ChildClass is running...')

obj = ChildClass()
obj.feature_1()
obj.feature_2()
obj.feature_3()
```

```
C:\Users\User\PycharmProjects
 \BSCExamples\venv\Scripts\python.exe
  C:/Users/User/PycharmProjects
 /BSCExamples/ClassExample.py
feature_1 from ParentClass is
 running...
feature_2 from ParentClass is
 running...
feature_3 from ChildClass is running...

Process finished with exit code 0
```

Multiple Inheritance

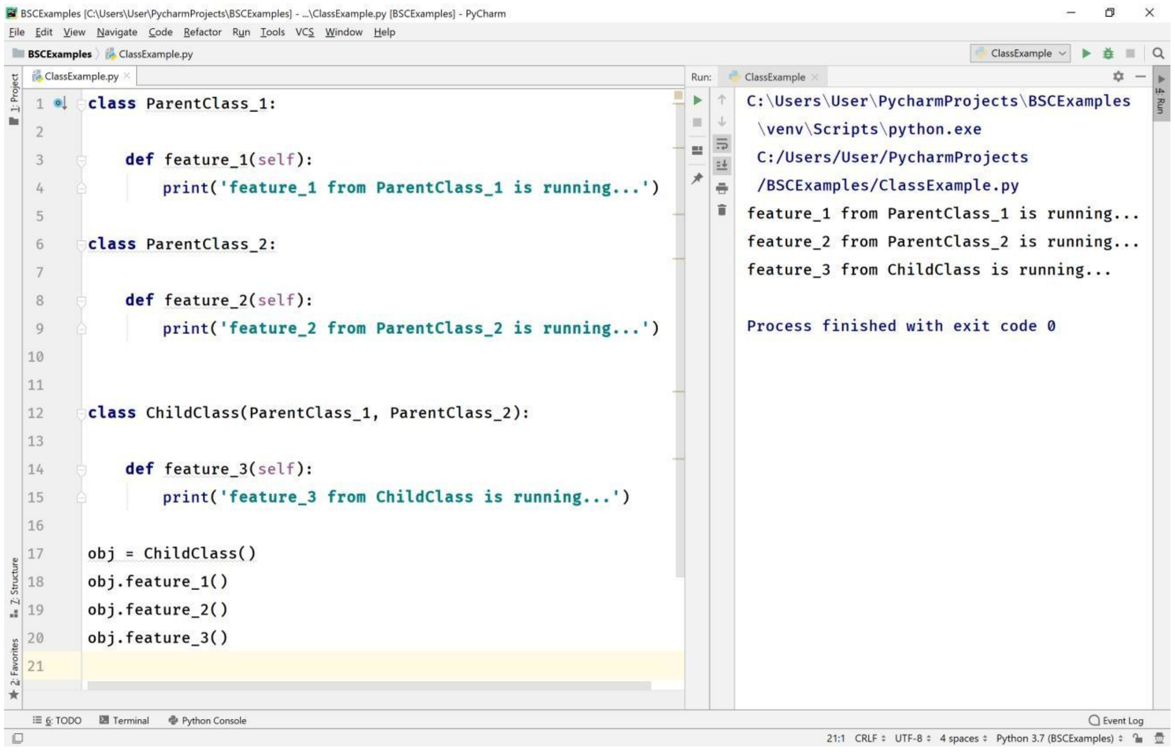
In this type of inheritance, one child class derives from two or more parent classes. Look at the following example code.

Example

```
classParentClass_1:


deffeature_1(self):
print('feature_1 from ParentClass_1 is running...')


classParentClass_2:


deffeature_2(self):
print('feature_2 from ParentClass_2 is running...')
```

```
classChildClass(ParentClass_1, ParentClass_2):

deffeature_3(self):
print('feature_3 from ChildClass is running...')

obj = ChildClass()
obj.feature_1()
obj.feature_2()
obj.feature_3()
```

When we run the above example code, it produces the following output.


BSCExamples [C:\Users\User\PycharmProjects\BSCExamples] - ...\ClassExample.py [BSCExamples] - PyCharm
File Edit View Navigate Code Refactor Run Tools VCS Window Help
BSCExamples
ClassExample.py
ClassExample
Run:
1: Project
Z: Structure
2: Favorites
6: TODO
Terminal
Python Console
Event Log
21:1 CRLF UTF-8 4 spaces Python 3.7 (BSCExamples)

```
class ParentClass_1:

    def feature_1(self):
        print('feature_1 from ParentClass_1 is running...')

class ParentClass_2:

    def feature_2(self):
        print('feature_2 from ParentClass_2 is running...')


class ChildClass(ParentClass_1, ParentClass_2):

    def feature_3(self):
        print('feature_3 from ChildClass is running...')

obj = ChildClass()
obj.feature_1()
obj.feature_2()
obj.feature_3()
```

```
C:\Users\User\PycharmProjects\BSCExamples
 \venv\Scripts\python.exe
 C:/Users/User/PycharmProjects
 /BSCExamples/ClassExample.py
feature_1 from ParentClass_1 is running...
feature_2 from ParentClass_2 is running...
feature_3 from ChildClass is running...

Process finished with exit code 0
```

Multi-Level Inheritance

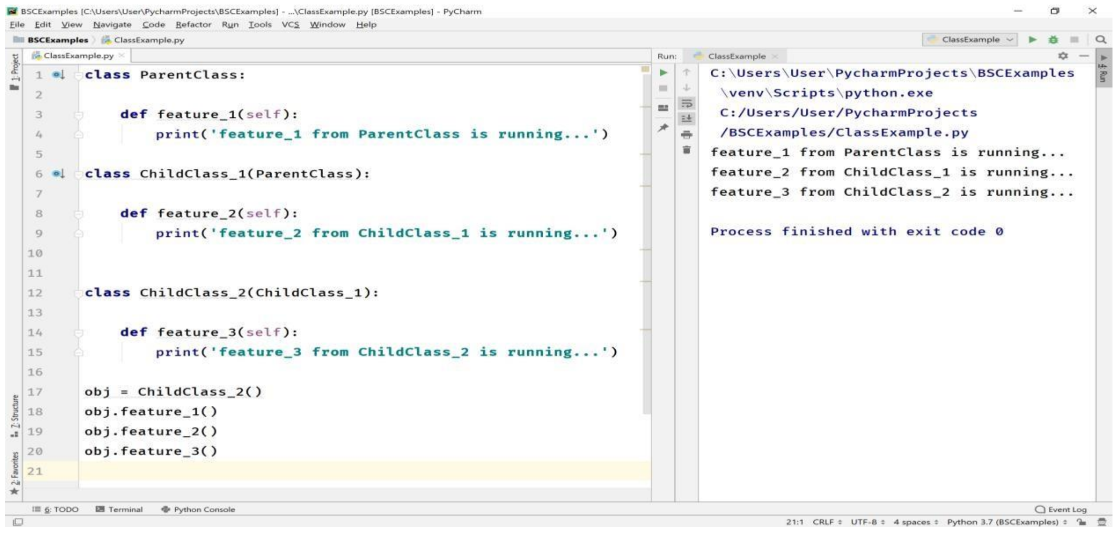
In this type of inheritance, the child class derives from a class which already derived from another class. Look at the following example code.

Example

```
classParentClass:
```

```
deffeature_1(self):
print('feature_1 from ParentClass is running...')

classChildClass_1(ParentClass):

deffeature_2(self):
print('feature_2 from ChildClass_1 is running...')



classChildClass_2(ChildClass_1):

deffeature_3(self):
print('feature_3 from ChildClass_2 is running...')

obj = ChildClass_2()
obj.feature_1()
obj.feature_2()
obj.feature_3()
```

When we run the above example code, it produces the following output.


BSCExamples [C:\Users\User\PycharmProjects\BSCExamples] - ...\ClassExample.py [BSCExamples] - PyCharm
File Edit View Navigate Code Refactor Run Tools VCS Window Help
BSCExamples
ClassExample.py
ClassExample
Run:
TODO
Terminal
Python Console
Event Log
21:1 CRLF UTF-8 4 spaces Python 3.7 (BSCExamples)

```
class ParentClass:

    def feature_1(self):
        print('feature_1 from ParentClass is running...')

class ChildClass_1(ParentClass):

    def feature_2(self):
        print('feature_2 from ChildClass_1 is running...')


class ChildClass_2(ChildClass_1):

    def feature_3(self):
        print('feature_3 from ChildClass_2 is running...')

obj = ChildClass_2()
obj.feature_1()
obj.feature_2()
obj.feature_3()
```

```
C:\Users\User\PycharmProjects\BSCExamples
 \venv\Scripts\python.exe
 C:/Users/User/PycharmProjects
 /BSCExamples/ClassExample.py
feature_1 from ParentClass is running...
feature_2 from ChildClass_1 is running...
feature_3 from ChildClass_2 is running...

Process finished with exit code 0
```

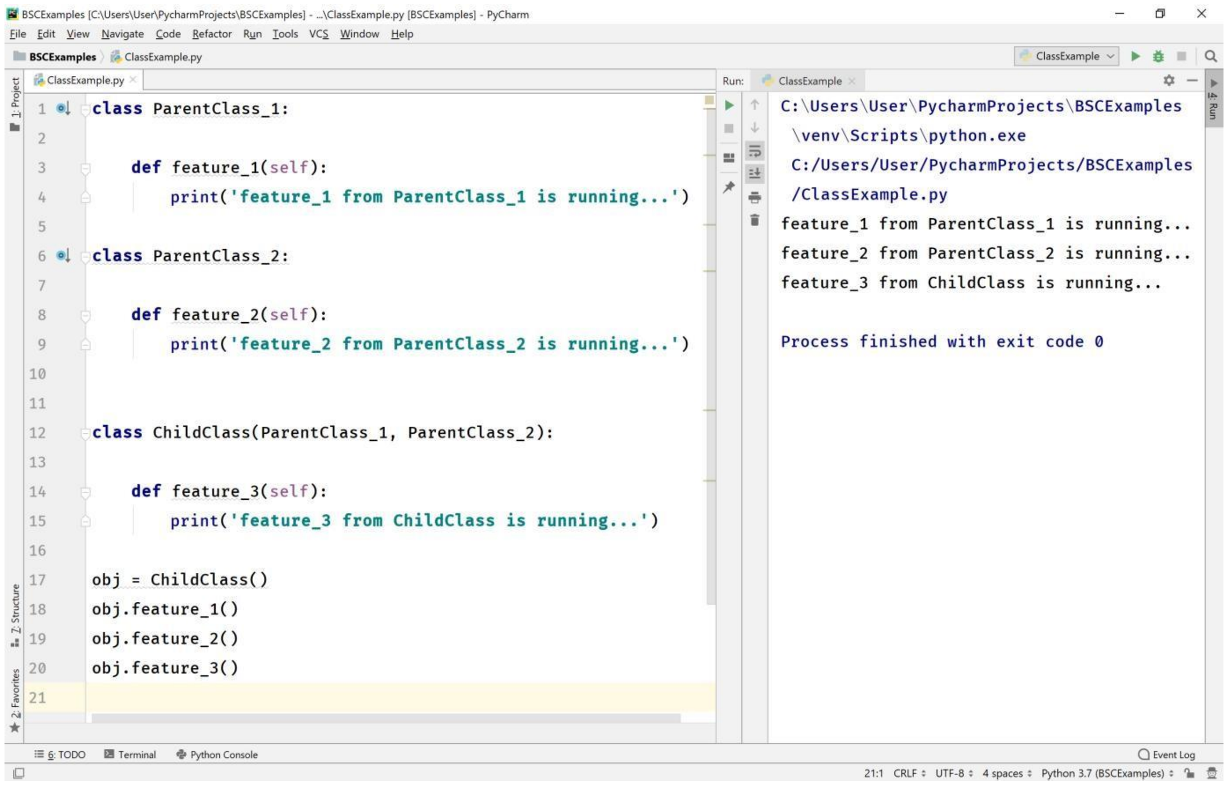
Hierarchical Inheritance

In this type of inheritance, two or more child classes derive from one parent class. Look at the following example code.

Example

```
classParentClass_1:

deffeature_1(self):
print('feature_1 from ParentClass_1 is running...')

classParentClass_2:

deffeature_2(self):
print('feature_2 from ParentClass_2 is running...')



classChildClass(ParentClass_1, ParentClass_2):

deffeature_3(self):
print('feature_3 from ChildClass is running...')

obj = ChildClass()
obj.feature_1()
obj.feature_2()
obj.feature_3()
```

When we run the above example code, it produces the following output.



```
class ParentClass_1:

    def feature_1(self):
        print('feature_1 from ParentClass_1 is running...')

class ParentClass_2:

    def feature_2(self):
        print('feature_2 from ParentClass_2 is running...')


class ChildClass(ParentClass_1, ParentClass_2):

    def feature_3(self):
        print('feature_3 from ChildClass is running...')

obj = ChildClass()
obj.feature_1()
obj.feature_2()
obj.feature_3()
```

```
C:\Users\User\PycharmProjects\BSCExamples
 \venv\Scripts\python.exe
 C:/Users/User/PycharmProjects/BSCExamples
 /ClassExample.py
feature_1 from ParentClass_1 is running...
feature_2 from ParentClass_2 is running...
feature_3 from ChildClass is running...

Process finished with exit code 0
```

Hybrid Inheritance

The hybrid inheritance is the combination of more than one type of inheritance. We may use any combination as a single with multiple inheritances, multi-level with multiple inheritances, etc.,

Polymorphism:

Polymorphism is a concept of object oriented programming, which means multiple forms or more than one form. Polymorphism enables using a single interface with input of different datatypes, different class or may be for different number of inputs.

In python as everything is an object hence by default a function can take anything as an argument but the execution of the function might fail as every function has some logic that it follows.

For example,

```
len("hello")# returns 5 as result

len([1,2,3,4,45,345,23,42])# returns 8 as result
```

In this case the function len is polymorphic as it is taking **string** as input in the first case and is taking **list** as input in the second case.

In python, polymorphism is a way of making a function accept objects of different classes if they behave similarly.

Method overriding is a type of polymorphism in which a child class which is extending the parent class can provide different definition to any function defined in the parent class as per its own requirements.

Method Overloading

Method overriding or function overloading is a type of polymorphism in which we can define a number of methods with the same name but with a different number of parameters as well as parameters can be of different types. These methods can perform a similar or different function.

Python doesn't support method overloading on the basis of different number of parameters in functions.

Defining Polymorphic Classes

Imagine a situation in which we have a different class for shapes like Square, Triangle etc which serves as a resource to calculate the area of that shape. Each shape has a different number of dimensions which are used to calculate the area of the respective shape.

Now one approach is to define different functions with different names to calculate the area of the given shapes. The program depicting this approach is shown below:

```
classSquare:

   side =5

defcalculate_area_sq(self):

return self.side * self.side



classTriangle:

   base =5

   height =4

defcalculate_area_tri(self):

return0.5* self.base * self.height
```

```
sq = Square()

tri = Triangle()

print("Area of square: ", sq.calculate_area_sq())

print("Area of triangle: ", tri.calculate_area_tri())
```

```
Area of square: 25

Area of triangle: 10.0
```

The problem with this approach is that the developer has to remember the name of each function separately. In a much larger program, it is very difficult to memorize the name of the functions for every small operation. Here comes the role of method overloading.

Now let's change the name of functions to calculate the area and give them both same name calculate_area() while keeping the function separately in both the classes with different definitions. In this case the type of object will help in resolving the call to the function. The program below shows the implementation of this type of polymorphism with class methods:

```
classSquare:

    side =5

defcalculate_area(self):

return self.side * self.side



classTriangle:

    base =5

    height =4

defcalculate_area(self):

return0.5* self.base * self.height



sq = Square()
```

```
tri = Triangle()

print("Area of square: ", sq.calculate_area())

print("Area of triangle: ", tri.calculate_area())
```

```
Area of square: 25

Area of triangle: 10.0
```

As you can see in the implementation of both the classes i.e. Square as well as Triangle has the function with same name calculate_area(), but due to different objects its call get resolved correctly, that is when the function is called using the object sq then the function of class Square is called and when it is called using the object tri then the function of class Triangle is called.

Polymorphism with Class Methods

What we saw in the example above is again obvious behaviour. Let's use a loop which iterates over a tuple of objects of various shapes and call the area function to calculate area for each shape object.

```
sq = Square()

tri = Triangle()



for(obj in(sq, tri)):

    obj.calculate_area()
```

Now this is a better example of polymorphism because now we are treating objects of different classes as an object on which same function gets called.

Here python doesn't care about the type of object which is calling the function hence making the class method polymorphic in nature.

Polymorphism with Functions

Just like we used a loop in the above example, we can also create a function which takes an object of some shape class as input and then calls the function to calculate area for it. For example,

```
find_area_of_shape(obj):

    obj.calculate_area()
```

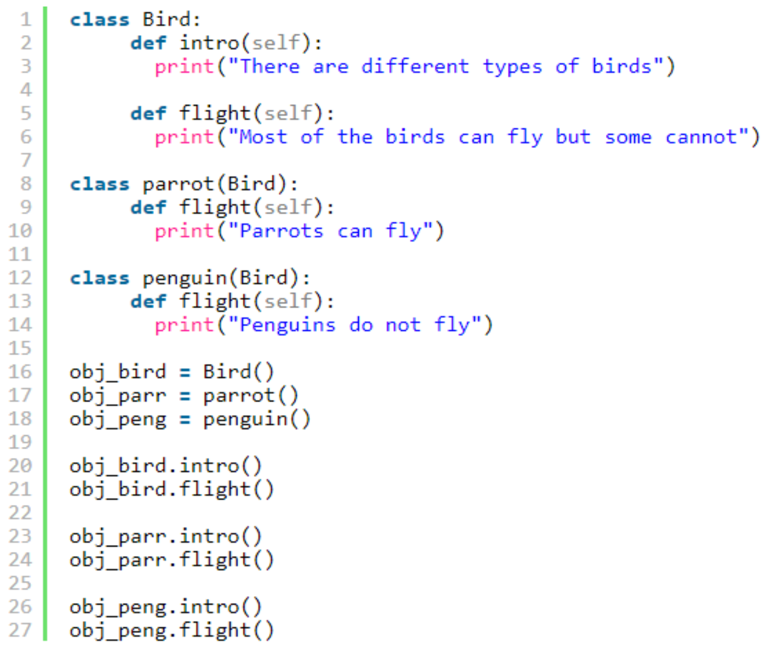
```
sq = Square()

tri = Triangle()



# calling the method with different objects

find_area_of_shape(sq)

find_area_of_shape(tri)
```

In the example above we have used the same function find_area_of_shape to calculate area of two different shape classes. The same function takes different class objects as arguments and executes perfectly to return the result. This is polymorphism.

Polymorphism with Inheritance

Polymorphism in python defines methods in the child class that have the same name as the methods in the parent class. In inheritance, the child class inherits the methods from the parent class. Also, it is possible to modify a method in a child class that it has inherited from the parent class.

This is mostly used in cases where the method inherited from the parent class doesn"t fit the child class. This process of re-implementing a method in the child class is known as Method Overriding. Here is an example that shows polymorphism with inheritance:



```
class Bird:
     def intro(self):
       print("There are different types of birds")

     def flight(self):
       print("Most of the birds can fly but some cannot")

class parrot(Bird):
     def flight(self):
       print("Parrots can fly")

class penguin(Bird):
     def flight(self):
       print("Penguins do not fly")

obj_bird = Bird()
obj_parr = parrot()
obj_peng = penguin()

obj_bird.intro()
obj_bird.flight()

obj_parr.intro()
obj_parr.flight()

obj_peng.intro()
obj_peng.flight()
```

Output:

```
There are different types of birds
Most of the birds can fly but some cannot
There are different types of bird
Parrots can fly
There are many types of birds
Penguins do not fly
```

These are different ways to define polymorphism in Python. With this, we have come to the end of our article. I hope you understood what is polymorphism and how it is used in Python.

Abstraction

Abstraction is one of the most important features of object-oriented programming. It is used to hide the background details or any unnecessary implementation.

Pre-defined functions are similar to data abstraction.

For example, when you use a washing machine for laundry purposes. What you do is you put your laundry and detergent inside the machine and wait for the machine to perform its task. How does it perform it? What mechanism does it use? A user is not required to know the engineering behind its work. This process is typically known as *data abstraction*, when all the unnecessary information is kept hidden from the users.

Code

In Python, we can achieve abstraction by incorporating abstract classes and methods.

Any class that contains abstract method(s) is called an **abstract class**. Abstract methods do not include any implementations – they are always defined and implemented as part of the methods of the sub-classes inherited from the abstract class. Look at the sample syntax below for an abstract class:

```
from abc import ABC
// abc is a library from where a class ABC is being imported. However, a separate class can also be created.
The importing from the library has nothing to do with abstraction.

Class type_shape(ABC):
```

The class type_shape is inherited from the ABC class. Let‟s define an abstract method area inside the class type_shape:

```
from abc import ABC
class type_shape(ABC):
  // abstract method area
  def area(self):
    pass
```

The implementation of an abstract class is done in the sub-classes, which will inherit the class type_shape. We have defined four classes that inherit the abstract class type_shape in the code below

Example:

```
from abc import ABC
class type_shape(ABC):
  def area(self):
    #abstract method
    pass

class Rectangle(type_shape):
  length = 6
  breadth = 4
  def area(self):
    return self.length * self.breadth

class Circle(type_shape):
  radius = 7
  def area(self):
    return 3.14 * self.radius * self.radius

class Square(type_shape):
  length = 4
  def area(self):
    return self.length*self.length

class triangle:
  length = 5
  width = 4
  def area(self):
    return 0.5 * self.length * self.width


r = Rectangle() # object created for the class 'Rectangle'
c = Circle() # object created for the class 'Circle'
s = Square() # object created for the class 'Square'
t = triangle() # object created for the class 'triangle'
print("Area of a rectangle:", r.area()) # call to 'area' method defined inside the class.
print("Area of a circle:", c.area()) # call to 'area' method defined inside the class.
print("Area of a square:", s.area()) # call to 'area' method defined inside the class.
print("Area of a triangle:", t.area()) # call to 'area' method defined inside the class.
```

Output
1.14s

```
Area of a rectangle: 24

Area of a circle: 153.86

Area of a square: 16

Area of a triangle: 10.0
```