unacademy

Practice Questions Part 1

Course on Data Structure and Algorithms Using Python

Ankush Saklecha • Lesson 9 • Dec 16, 2023

Python as OOPs

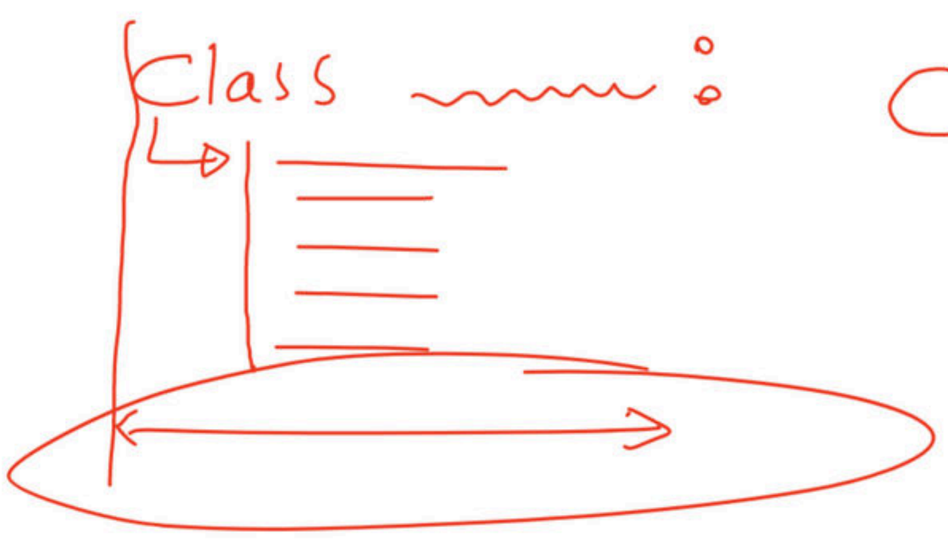
- Python is an object-oriented programming language.
- Major principles of object-oriented programming system are given below.
  - Class
  - Object
  - Method
  - Inheritance
  - Polymorphism
  - Data Abstraction
  - Encapsulation

Python Classes/Objects

- Classes provide a means of bundling data and functionality together.
- Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by its class) for modifying its state.
- The class can be defined as a collection of objects.
- It is a logical entity that has some specific attributes and methods.
- 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

```
class ClassName:
    <statement-1>
    .
    .
    <statement-N>
```



Class => Attributes (State)

method

Object

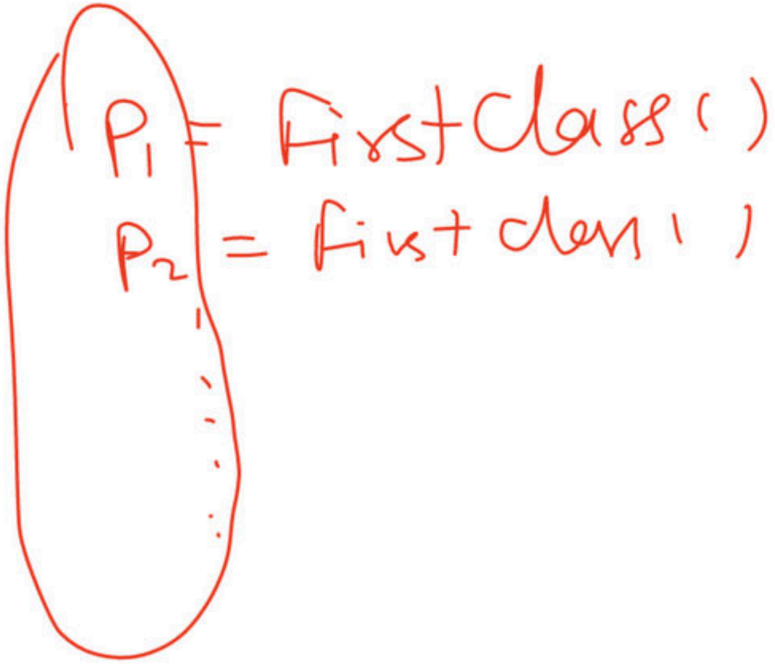
The object is an entity that has state and behavior. It may be any real-world object like the mouse, keyboard, chair, table, pen, e

Create Object

Now we can use the class named FirstClass to create objects:

```
class FirstClass:
  x = 50
p1 = FirstClass()
print(p1.x)
```

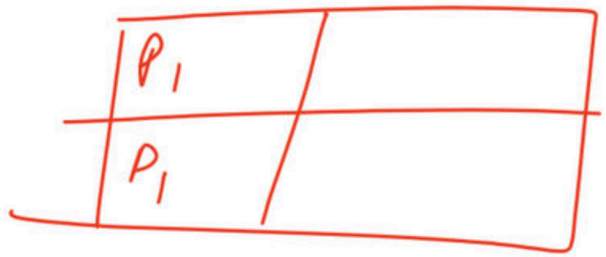
P2.x

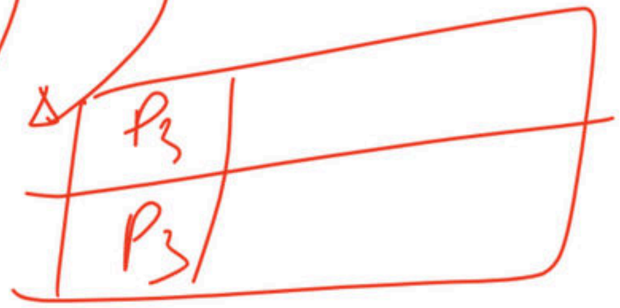


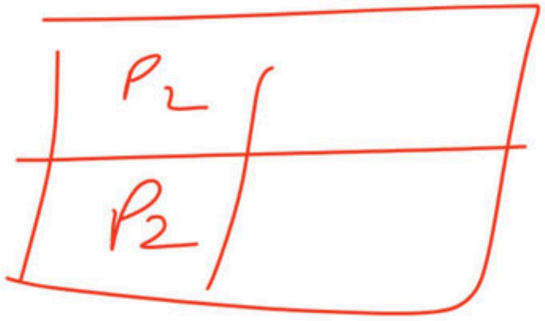
The __init__() Function

- It is built-in 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:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age
p1 = Person("John", 36)
print(p1.name)
print(p1.age)
```







The __str__() Function

The __str__() function controls what should be returned when the class object is represented as a string.

If the __str__() function is not set, the string representation of the object is returned:

The string representation of an object WITHOUT the __str__() function:

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

p1 = Person("Kush", 5)

print(p1)
```

```
<__main__.Person object at 0x2b892851a100>
```

The string representation of an object WITH the __str__() function:

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

  def __str__(self):
    return f"{self.name}({self.age})"

p1 = Person("John", 36)
p2 = Person("Raj", 40)
print(p1)
print(p2)
```

Hello my name is John

John (36)

Raj (40)

Object Methods

Objects can also contain methods. Methods in objects are functions that belong to the object.

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

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

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

Hello my name is John

```
class car:
    def __init__(self,modelname, year):
        self.modelname = modelname
        self.year = year
    def display(self):
        print(self.modelname,self.year)

c1 = car("Toyota", 2016)
c1.display()
```

The self Parameter

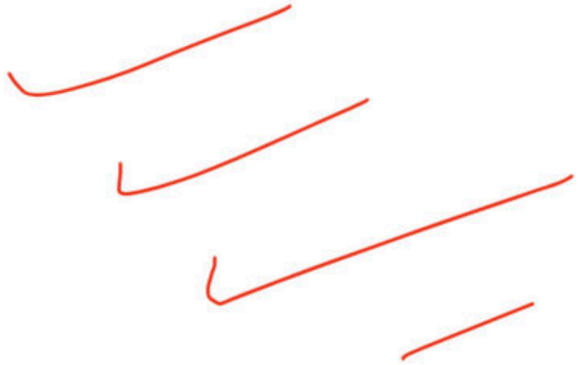
The `self` parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class.
It does not have to be named `self` , you can call it whatever you like, but it has to be the first parameter of any function in the class:

#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()
```

Modify Object Properties :

```
p1.age = 40 #update
del p1.age  #delete age property from object p1
del p1      # delete object p1
```



The pass Statement

`class` definitions cannot be empty, but if you for some reason have a `class` definition with no content, put in the `pass` statement to avoid getting an error.

```
class Person:
  pass
```

class Person :
Pass

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))
```

Every Python class keeps the following built-in attributes and they can be accessed using dot operator like any other attribute –

- **__dict__** – Dictionary containing the class's namespace.
- **__doc__** – Class documentation string or none, if undefined.
- **__name__** – Class name.
- **__module__** – Module name in which the class is defined. This attribute is "__main__" in interactive mode.
- **__bases__** – A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.

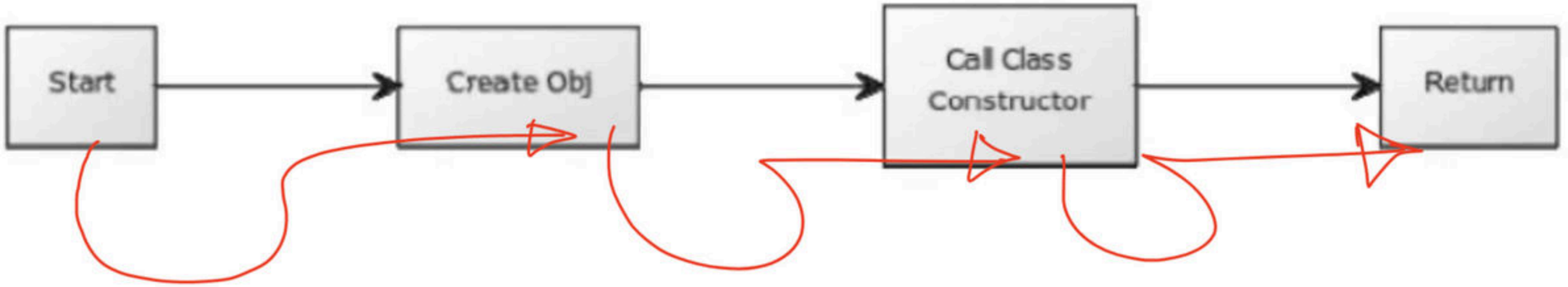
```
class Employee:

   def __init__(self, name="Bhavana", age=24):
      self.name = name
      self.age = age

   def displayEmployee(self):
      print ("Name : ", self.name, ", age: ", self.age)

print ("Employee.__doc__:", Employee.__doc__)
print ("Employee.__name__:", Employee.__name__)
print ("Employee.__module__:", Employee.__module__)
print ("Employee.__bases__:", Employee.__bases__)
print ("Employee.__dict__:", Employee.__dict__ )
```

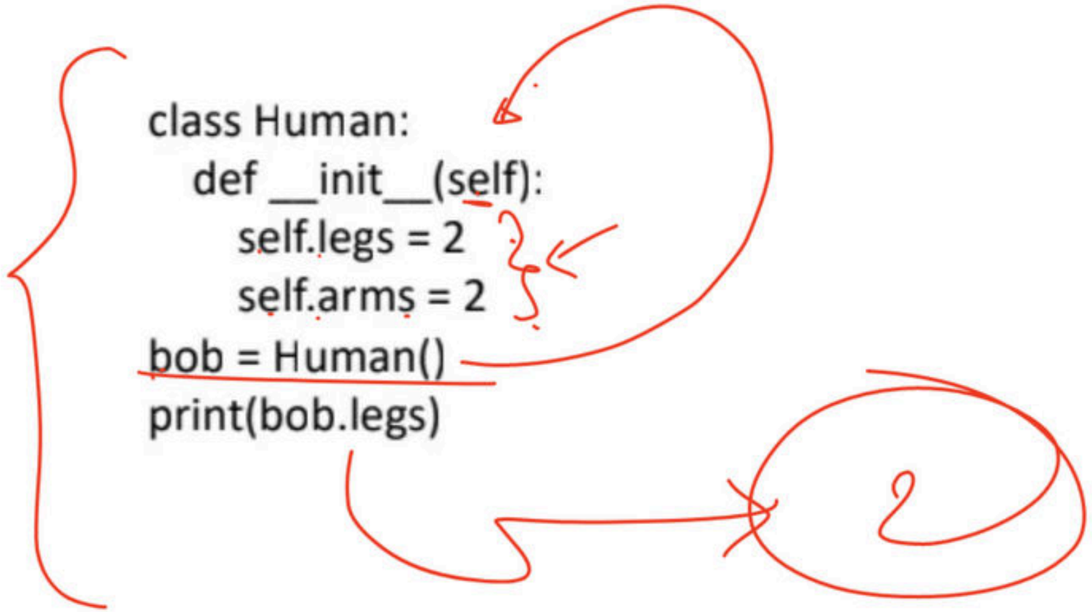
```
Employee.__doc__: None
Employee.__name__: Employee
Employee.__module__: __main__
Employee.__bases__: (<class 'object'>,)
Employee.__dict__: {'__module__': '__main__',
'__init__': <function Employee.__init__ at
0x0000022F866B8B80>, 'displayEmployee':
<function Employee.displayEmployee at
0x0000022F866B9760>, '__dict__': <attribute
'__dict__' of 'Employee' objects>, '__weakref__':
<attribute '__weakref__' of 'Employee' objects>,
'__doc__': None}
```

constructor

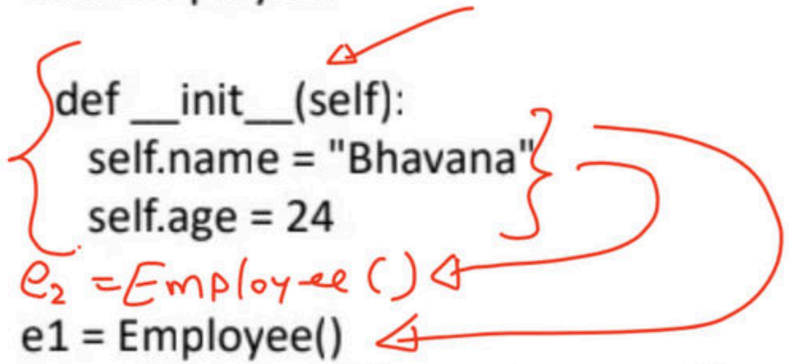
- The constructor is a method that is called when an object is created.
- This method is defined in the class and can be used to initialize basic variables.
- If you create four objects, the class constructor is called four times. Every class has a constructor, but its not required to explicitly define 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




class Human:
def __init__(self):
self.legs = 2
self.arms = 2
bob = Human()
print(bob.legs)

```
class Employee:
    def __init__(self):
        self.name = "Bhavana"
        self.age = 24
e2 = Employee()
e1 = Employee()
print ("Name: {}".format(e1.name))
print ("age: {}".format(e1.age))
```



Python Variable Types: Local & Global

There are two types of variables in Python, Global variable and Local variable.

global

Python5.2.py

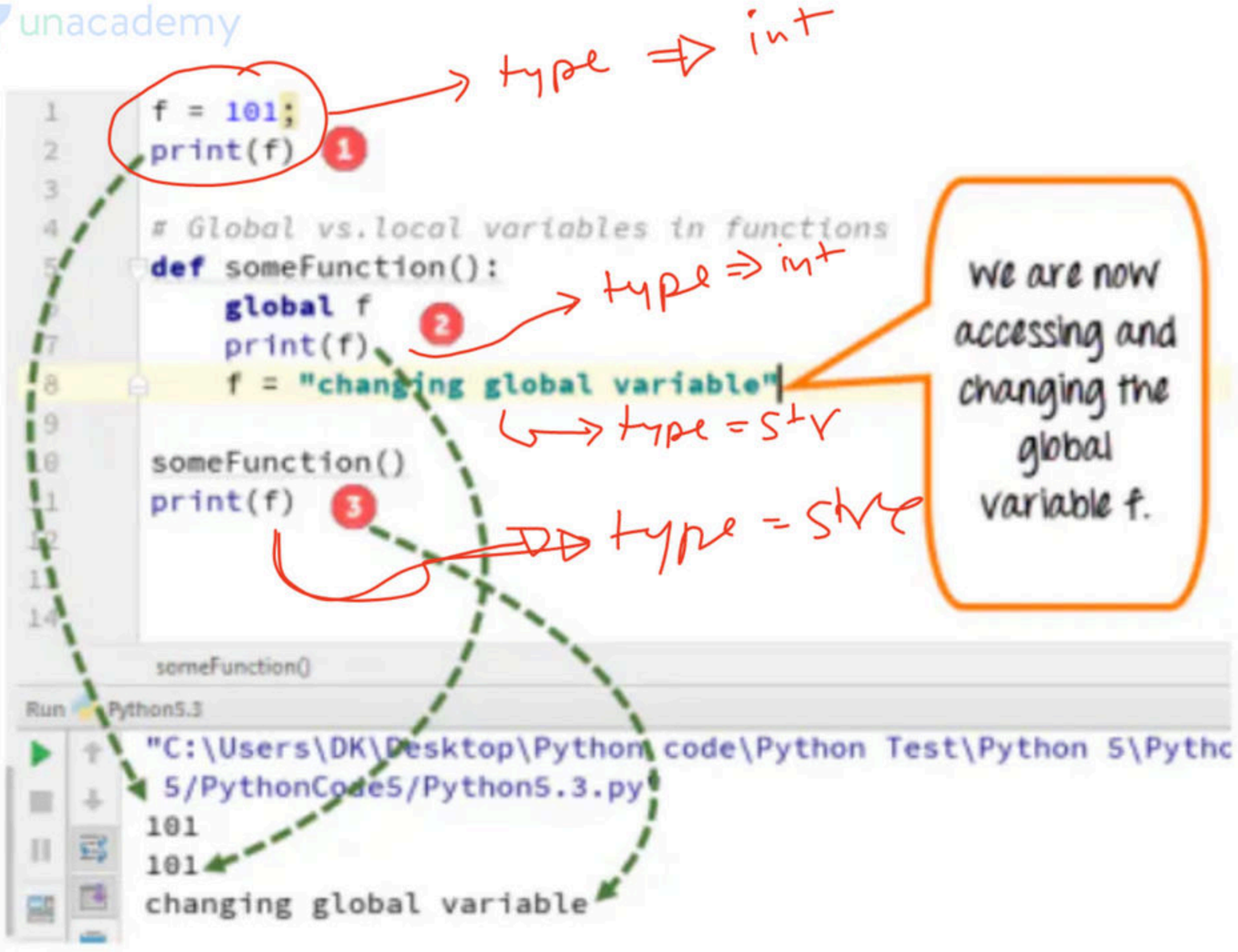
```
1  # Declare a variable and initialize it
2  f = 101
3  print(f)
4
5  # Global vs. local variables in functions
6  def someFunction():
7  # global f
8      f = 'I am learning Python'
9      print(f)
10
11 someFunction()
12 print(f)
13
```

101
I am learning Pyt.
101

f is a local variable declared inside the function.

Run Python5.2

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

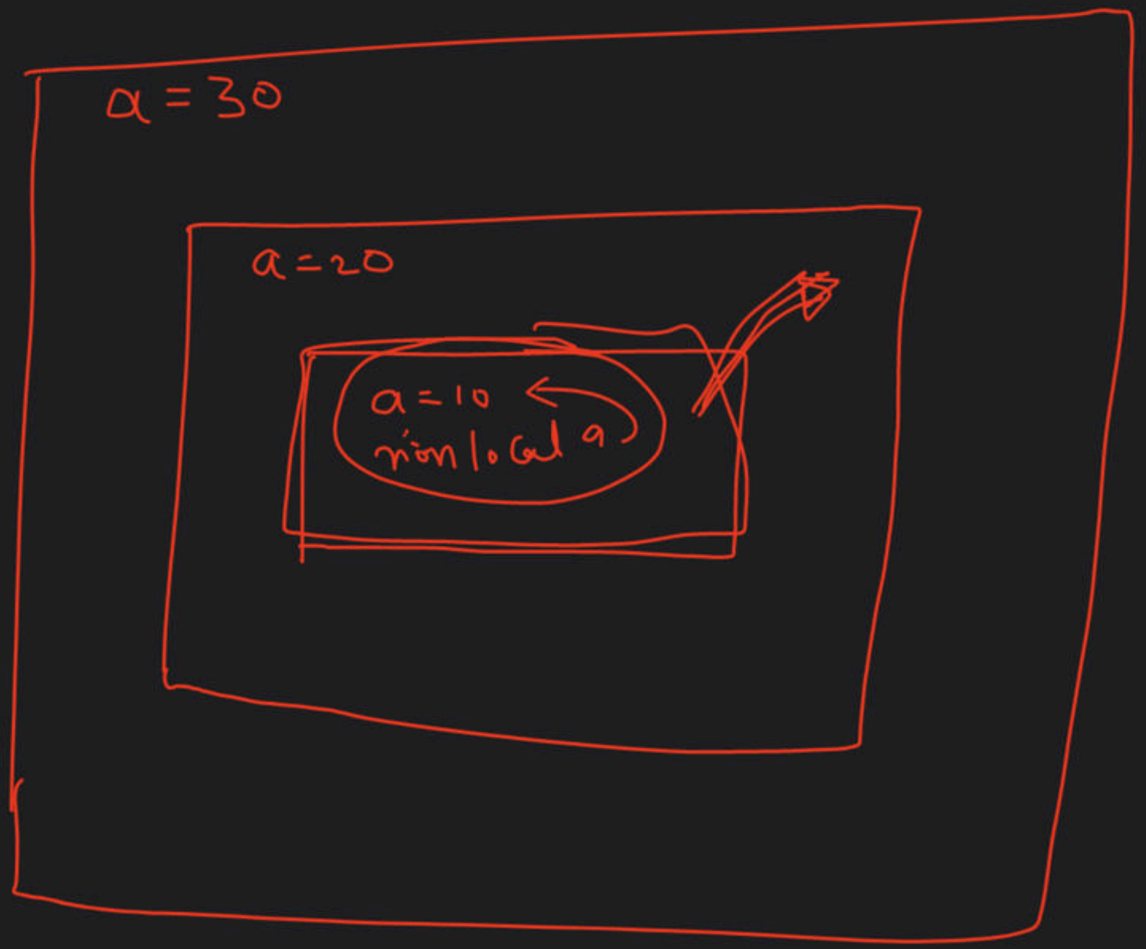

f = 101;
print(f)
1
type ⇒ int
# Global vs.local variables in functions
def someFunction():
global f
print(f)
2
type ⇒ int
f = "changing global variable"
type = str
someFunction()
print(f)
3
type = str
We are now accessing and changing the global variable f.
someFunction()
Run
Python5.3
"C:\Users\DK\Desktop\Python code\Python Test\Python 5\Pythc
5/PythonCode5/Python5.3.py
101
101
changing global variable

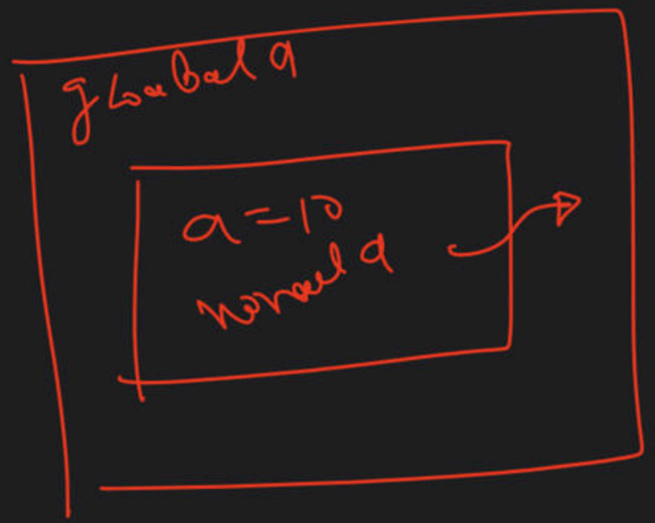
```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

Output :

```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```





Types of methods:

Generally, there are three types of methods in Python:

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

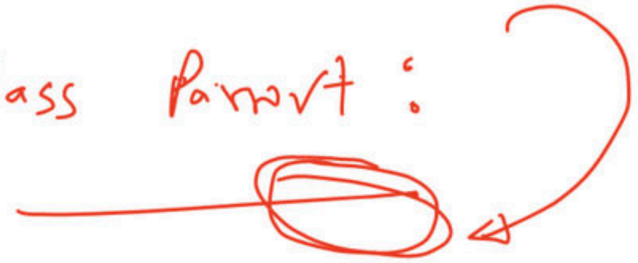
Before moving on with the topic, we have to know some key concepts.

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.

Class Parent :



1. Instance Method

```
# Instance Method Example in Python
class Student:

    def __init__(self, a, b):
        self.a = a
        self.b = b

    def avg(self):                    # instant method
        return(self.a + self.b)/2

s1 = Student(10,20)
print( s1.avg())
```

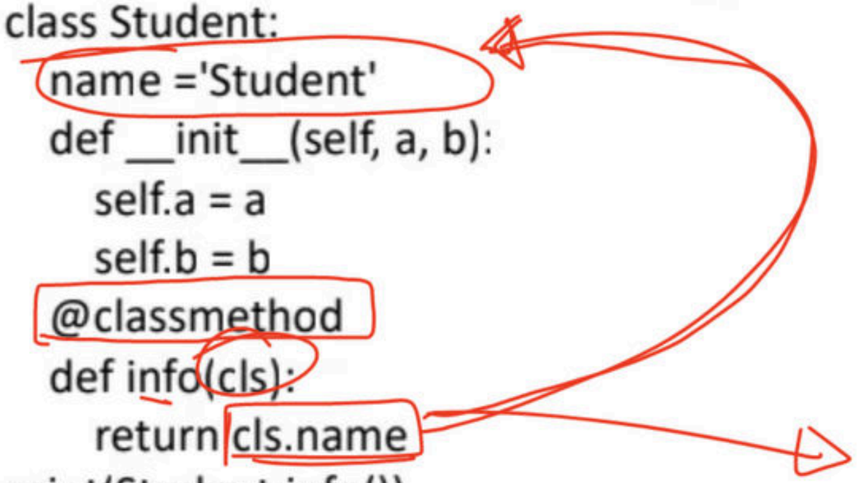
2. Class Method

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

classmethod(info)

```
# Class Method Implementation in python
class Student:
    name ='Student'
    def __init__(self, a, b):
        self.a = a
        self.b = b
    @classmethod
    def info(cls):
        return cls.name
print(Student.info())
```



- 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.

3. Static Method

A static method can be called without an object for that class, using the class name directly

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.

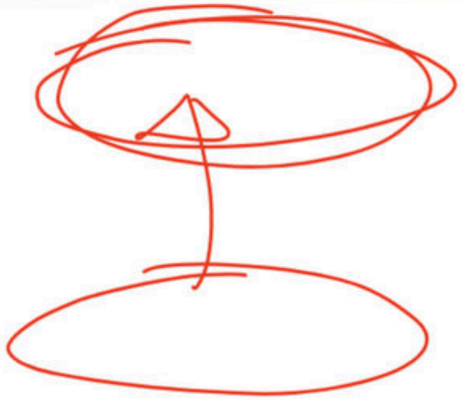
The inheritance is the process of acquiring the properties of one class to another class.

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**.

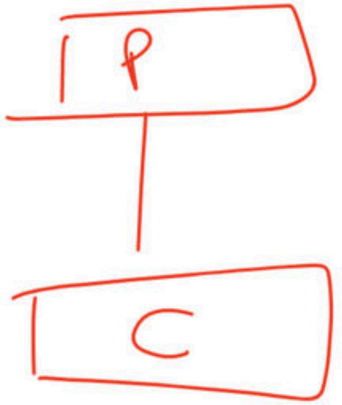
The **Child class** is the class which receives features from another class. The child class is also known as the **Derived Class** or **Subclass**.

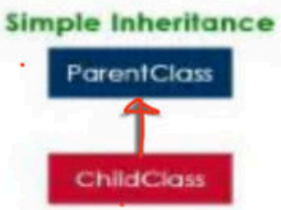
In the inheritance, the child class acquires the features from its parent class. But the parent class never acquires the features from its child class.

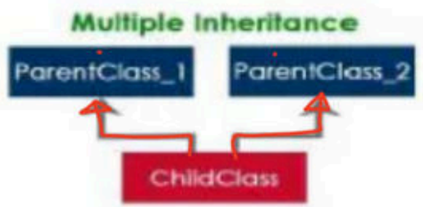


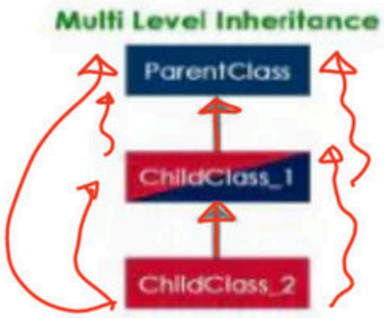
There are five types of inheritances, and they are as follows.

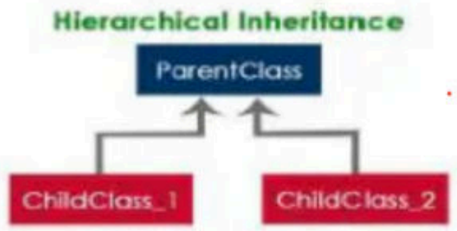
- **Simple Inheritance (or) Single Inheritance**
- **Multiple Inheritance**
- **Multi-Level Inheritance**
- **Hierarchical Inheritance**
- **Hybrid Inheritance**

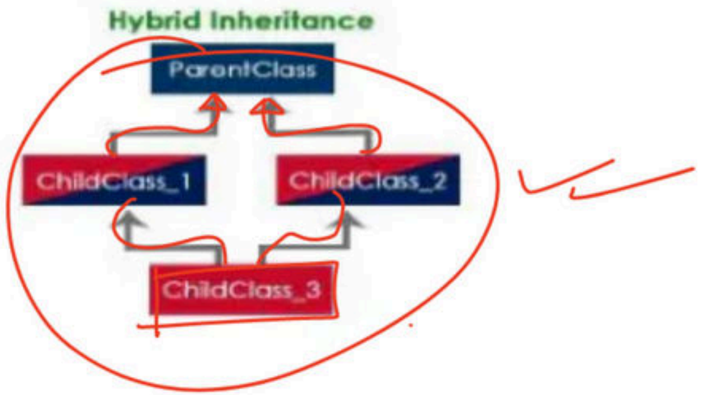



Simple Inheritance
ParentClass
ChildClass


Multiple Inheritance
ParentClass_1
ParentClass_2
ChildClass


Multi Level Inheritance
ParentClass
ChildClass_1
ChildClass_2


Hierarchical Inheritance
ParentClass
ChildClass_1
ChildClass_2


Hybrid Inheritance
ParentClass
ChildClass_1
ChildClass_2
ChildClass_3

Simple Inheritance (or) Single Inheritance

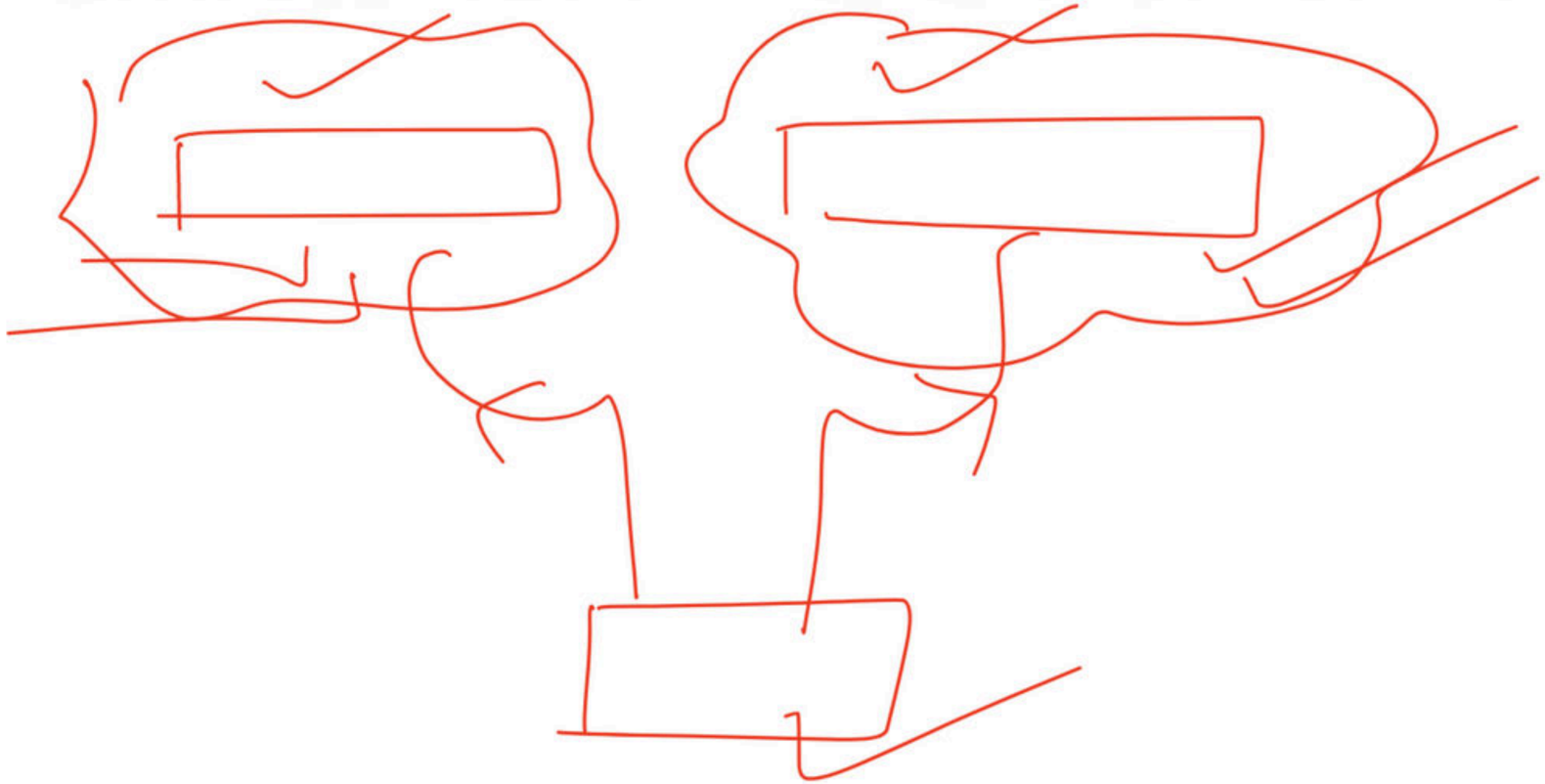
In this type of inheritance, one child class derives from one parent class. Look at the following example code.

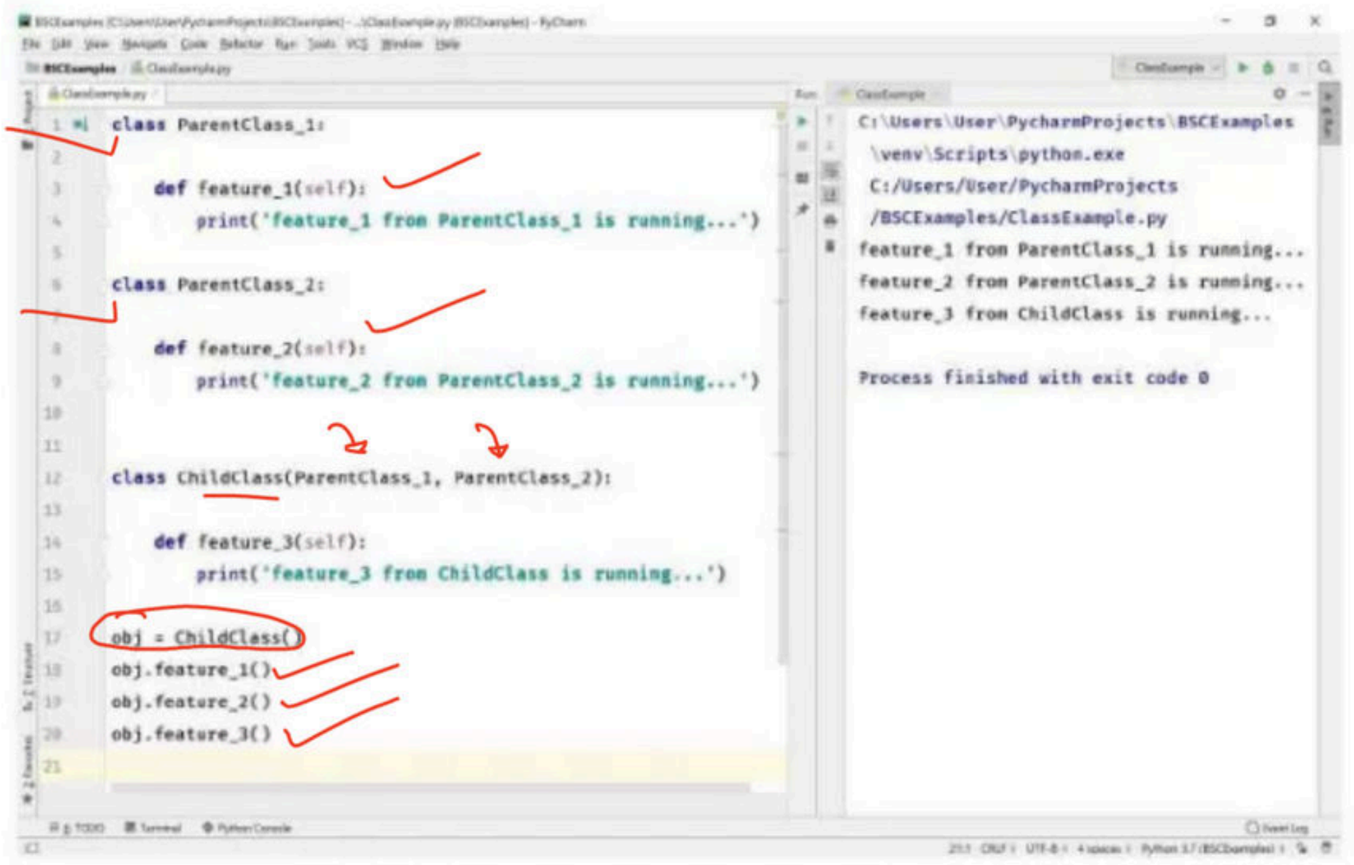
```python
class ParentClass:

    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.



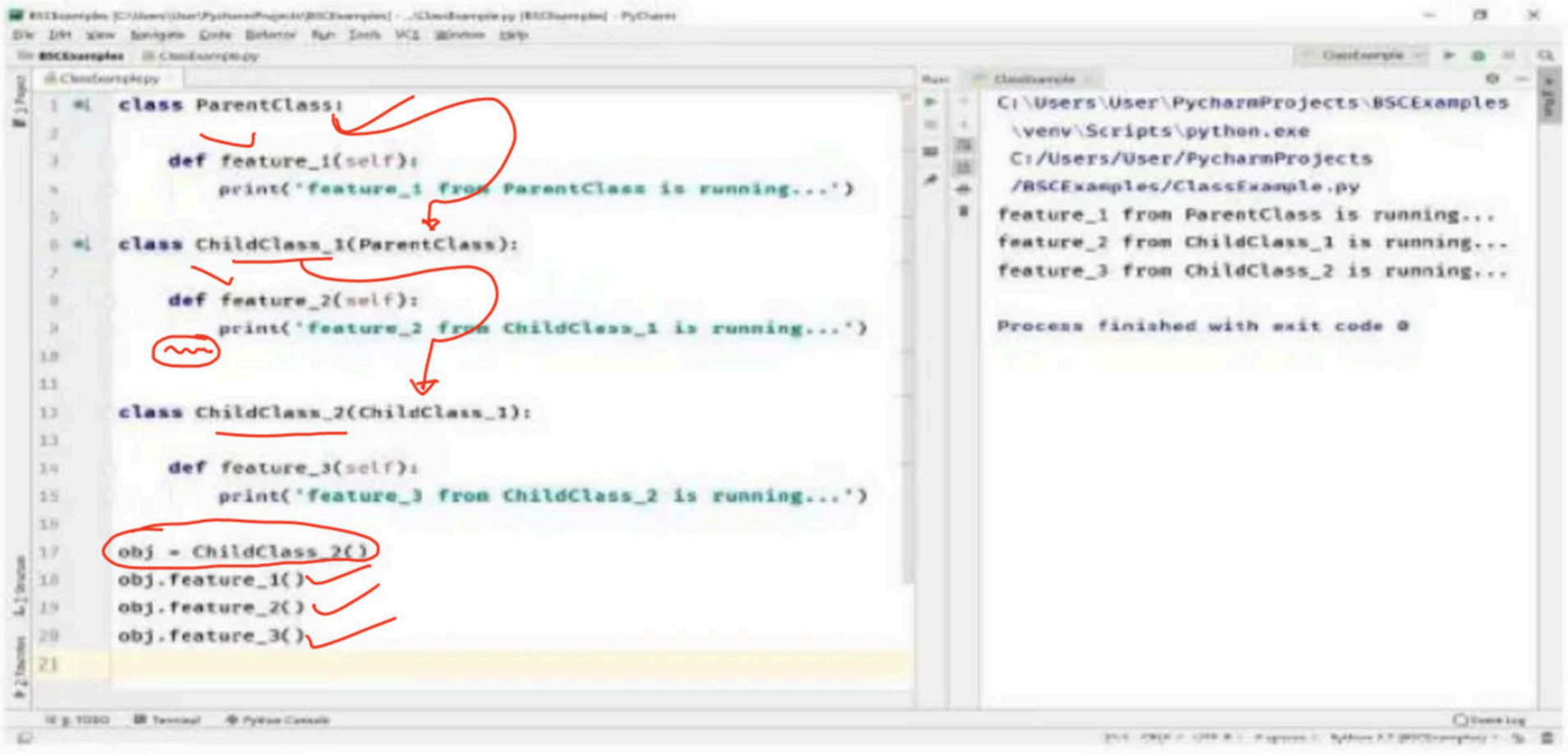


```
class ParentClass_1:

    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
```

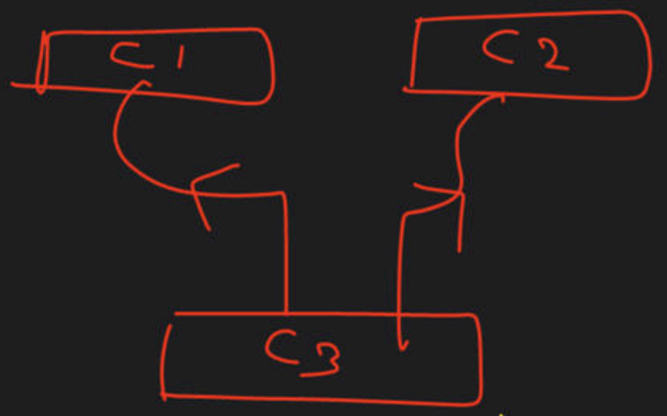
Multi-Level Inheritance

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

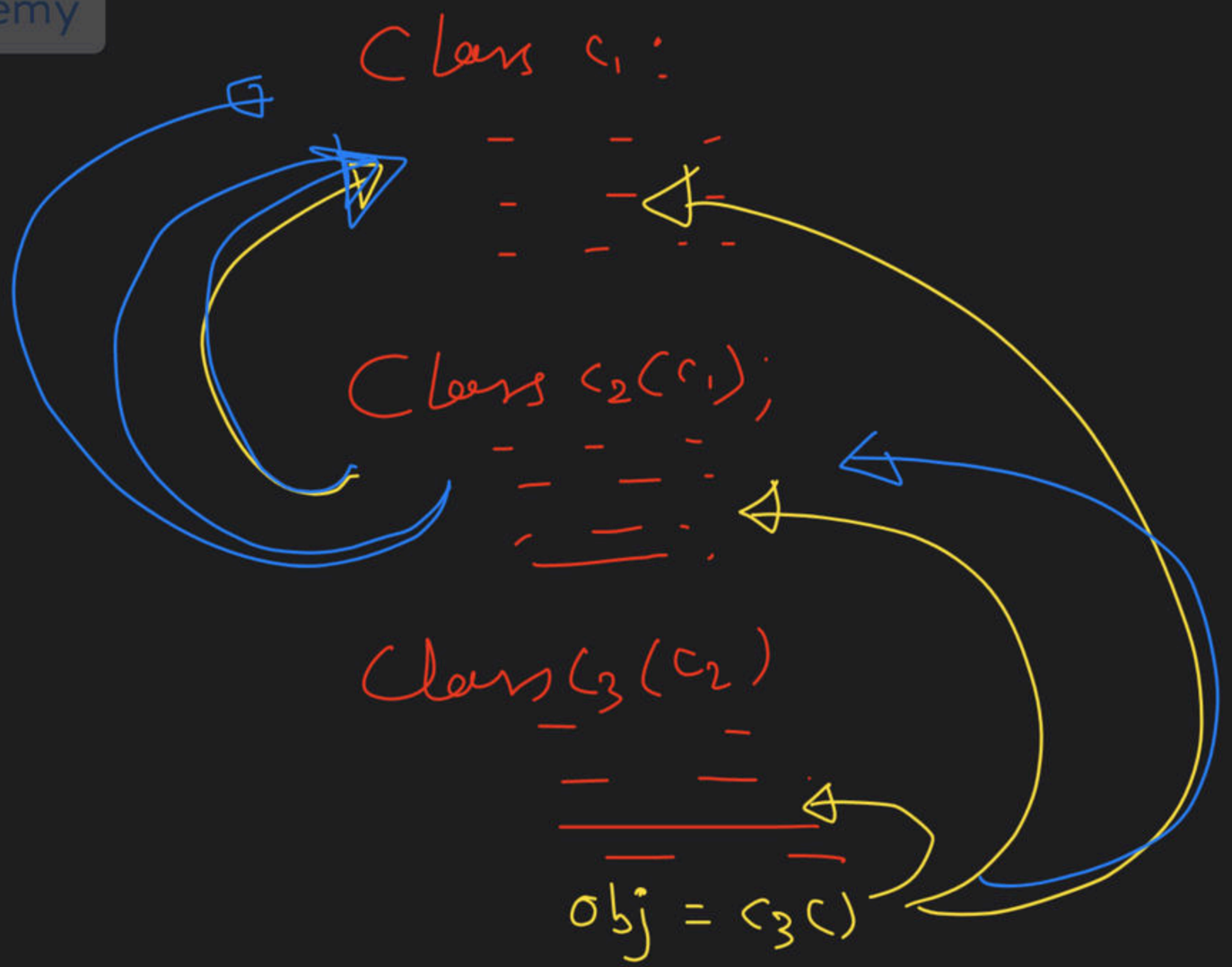


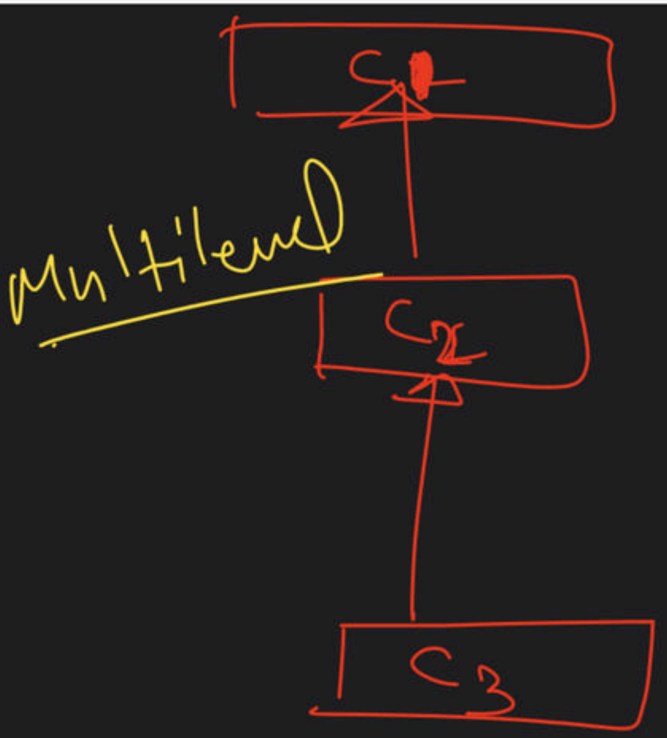
unacademy

```
Class C1 :
    def M1(self) :
        - - - - -

Class C2 :
    def M2(self) :
        - - - - - -

Class C3(C1, C2) :
    - - - -
    - - - - -
Obj = C3();
Obj.M1();
Obj.M2();
Obj.M3();
```



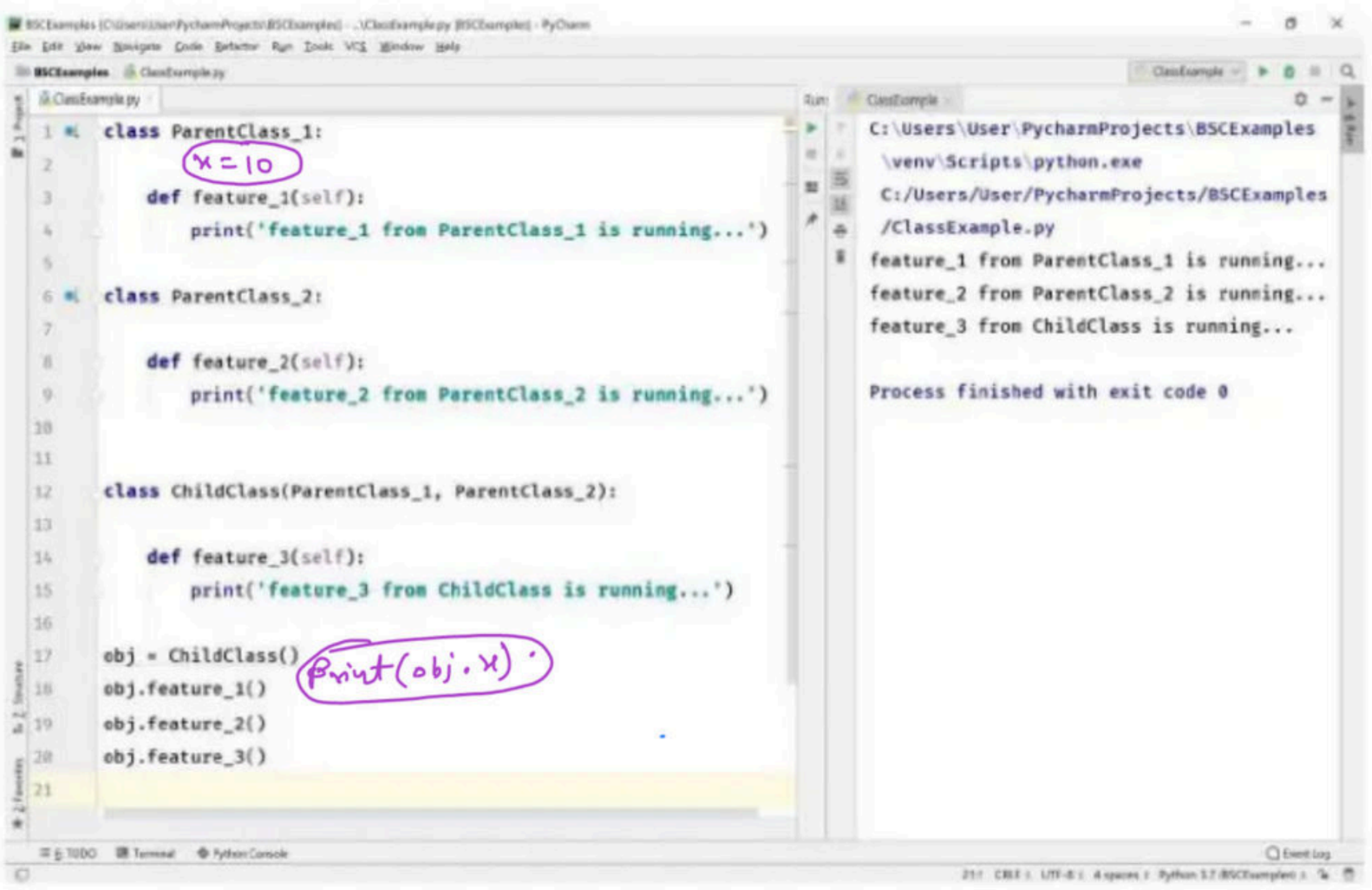
Multiple





Hierarchical Inheritance

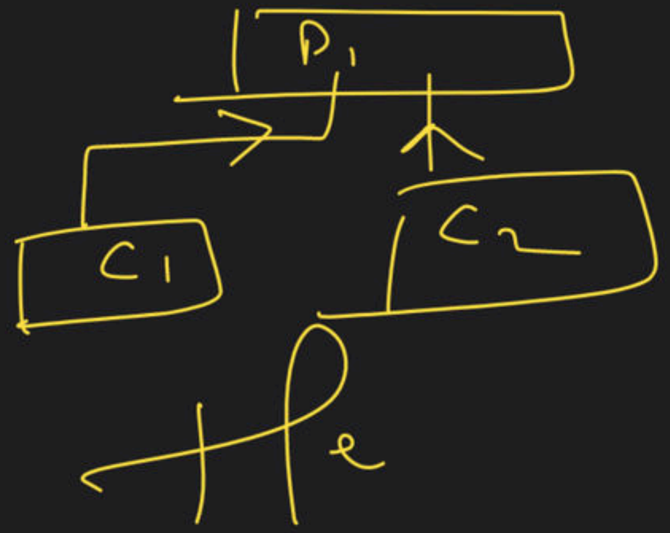
In this type of inheritance, two or more child classes derive from one parent class. Look at the following example code.



Class P_1 :

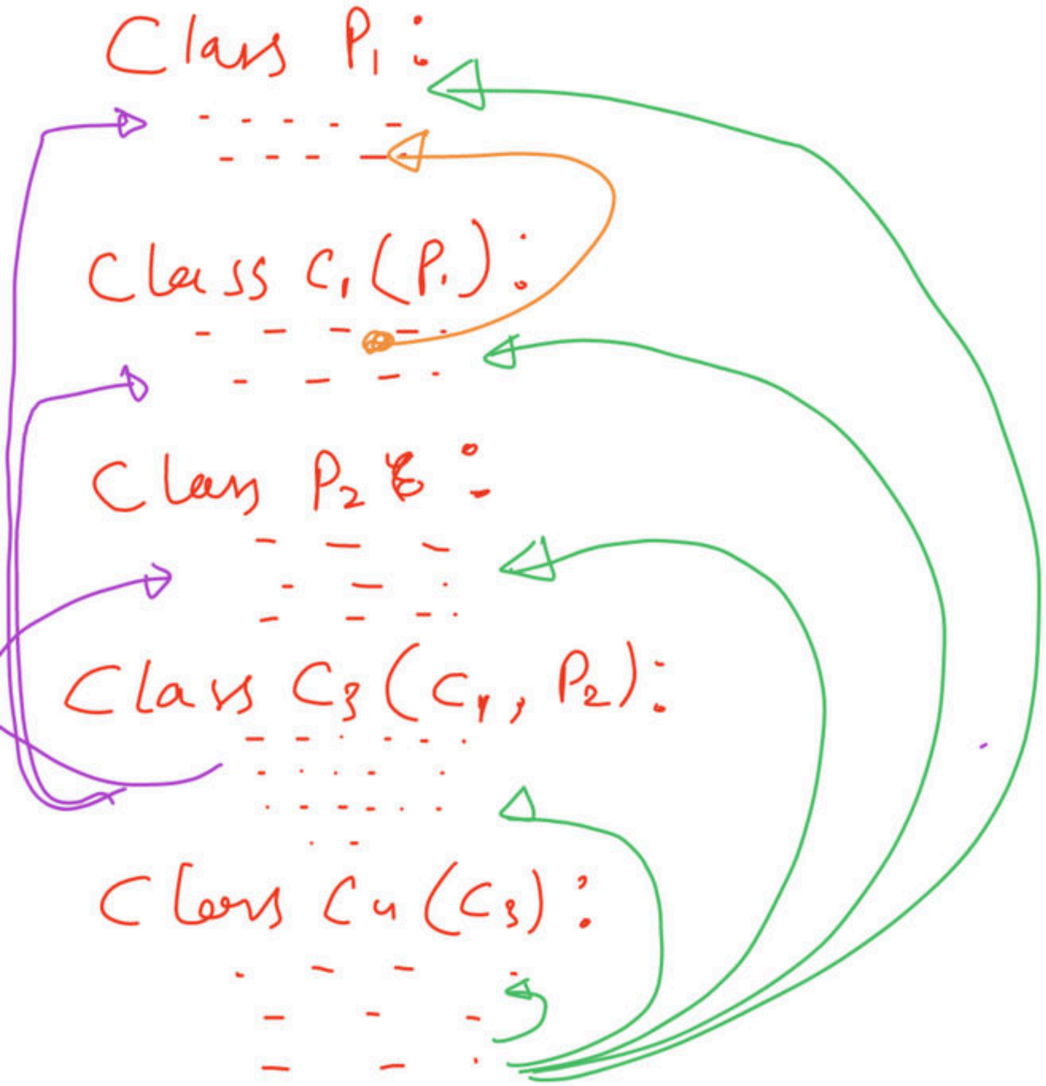
Class $C_1(P_1)$:

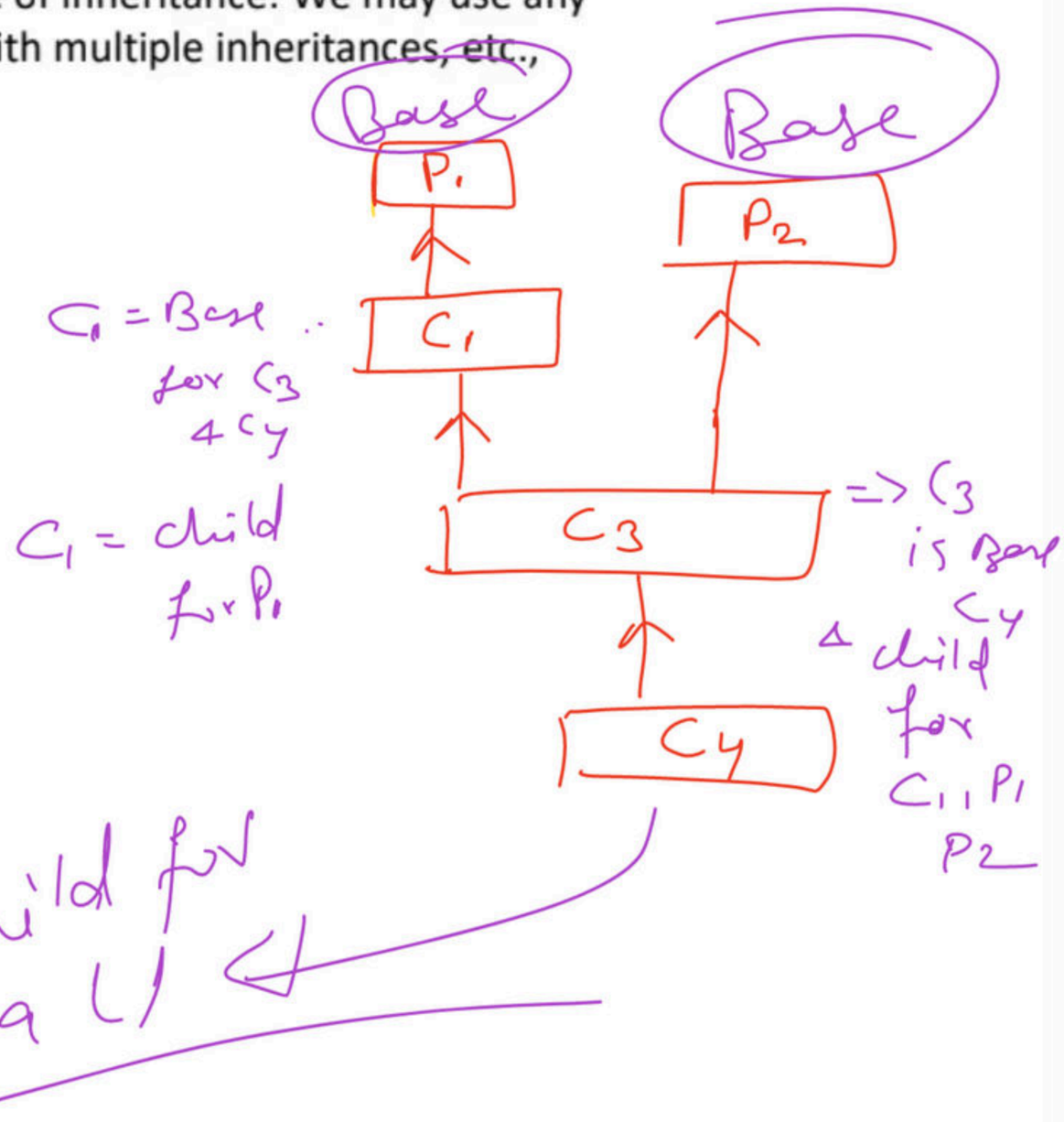
Class $C_2(P_1)$:



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.,





len ( )

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 resul
```

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.

unacademy

Circle

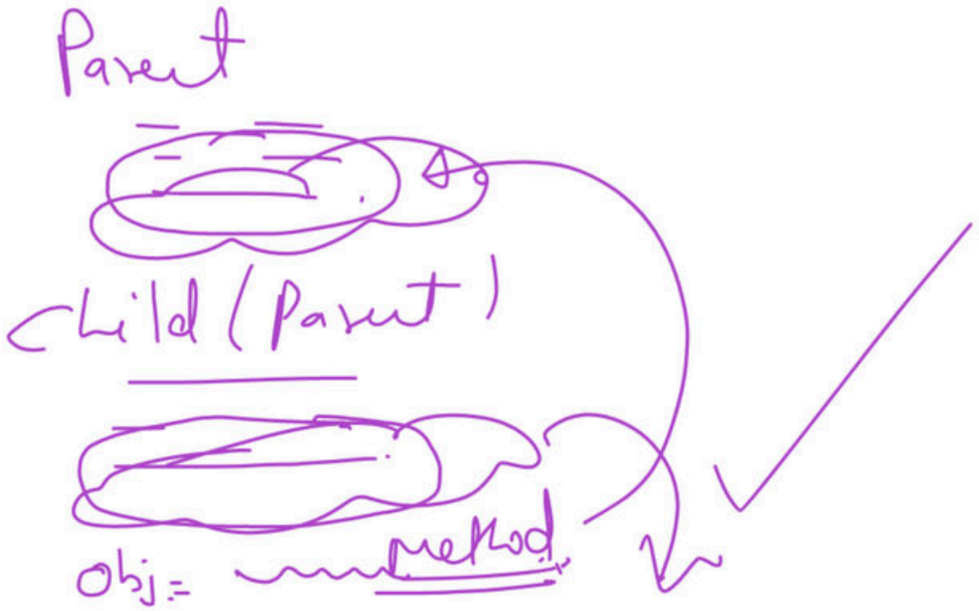
related.

area (A)

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

- **Method overriding : 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 few 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.

Note : Python doesn't support method overloading based on different number of parameters in functions.



Defining Polymorphic Classes

```
# without polymorphism
class Square:
    side =5
    def calculate_area_sq(self):
        return self.side * self.side
class Triangle:
    base =5
    height =4
    def calculate_area_tri(self):
        return 0.5* self.base * self.height
sq = Square()
tri = Triangle()
print("Area of square: ", sq.calculate_area_sq())
print("Area of triangle: ", tri.calculate_area_tri())
```

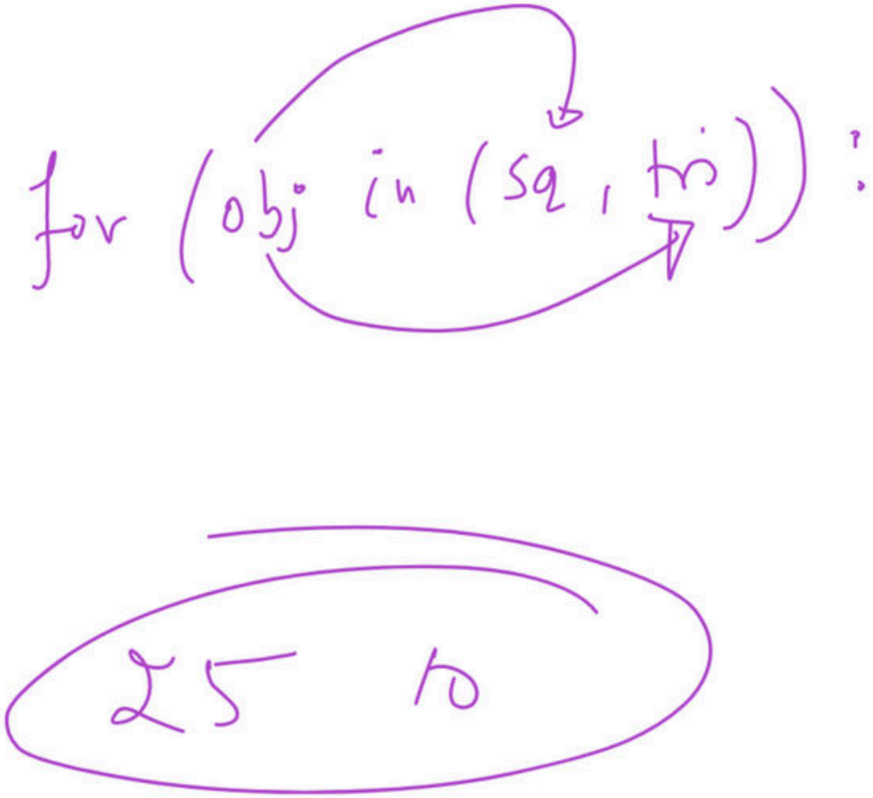
```
# polymorphic classes
class Square:
    side =5
    def calculate_area(self):
        return self.side * self.side
class Triangle:
    base =5
    height =4
    def calculate_area(self):
        return 0.5* self.base * self.height

sq = Square()
tri = Triangle()
print("Area of square: ", sq.calculate_area())
print("Area of triangle: ", tri.calculate_area())
```

25

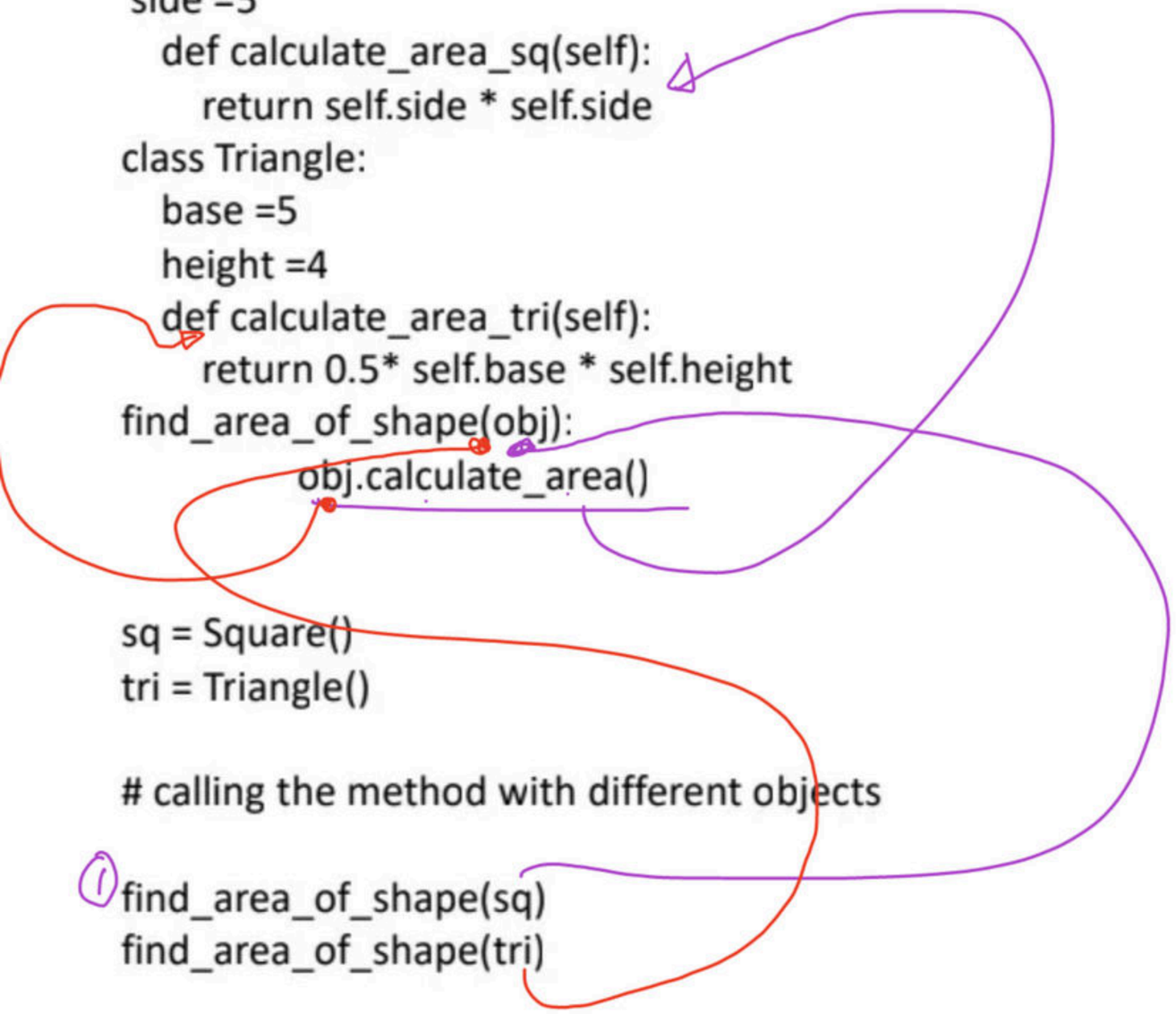
10

Polymorphism with Class Methods

```
class Square:
    side =5
    def calculate_area_sq(self):
        return self.side * self.side
class Triangle:
    base =5
    height =4
    def calculate_area_tri(self):
        return 0.5* self.base * self.height

sq = Square()
tri = Triangle()

for(obj in(sq, tri)):
 obj.calculate_area()
```

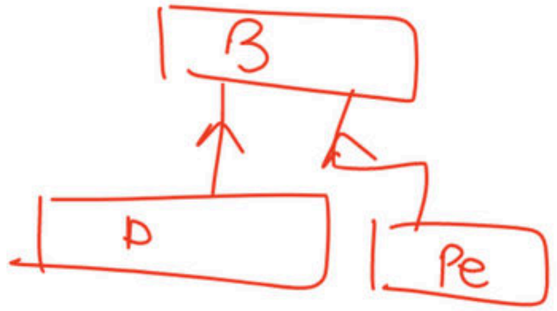


```
#Polymorphism with function
class Square :
    side =5
    def calculate_area_sq(self):
        return self.side * self.side
class Triangle:
    base =5
    height =4
    def calculate_area_tri(self):
        return 0.5* self.base * self.height
find_area_of_shape(obj):
        obj.calculate_area()


sq = Square()
tri = Triangle()

# calling the method with different objects

find_area_of_shape(sq)
find_area_of_shape(tri)
```



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()
```



Hierarchical

1
2
1
3
1
4

obj.super().flight()

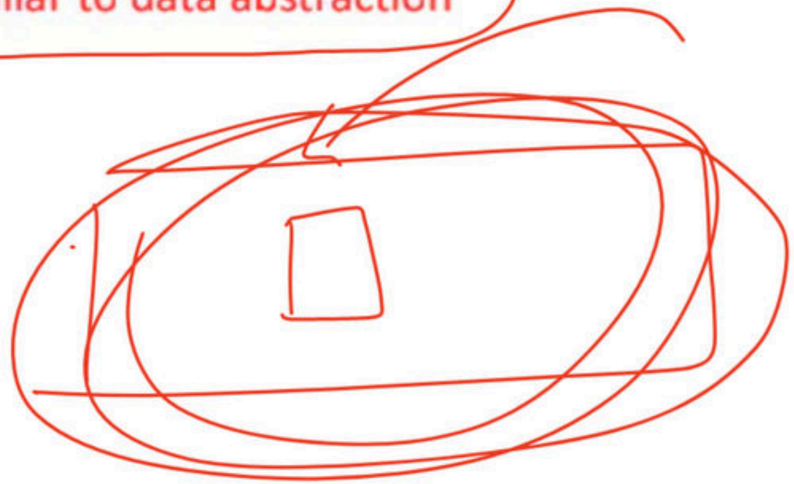
unacademy

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.

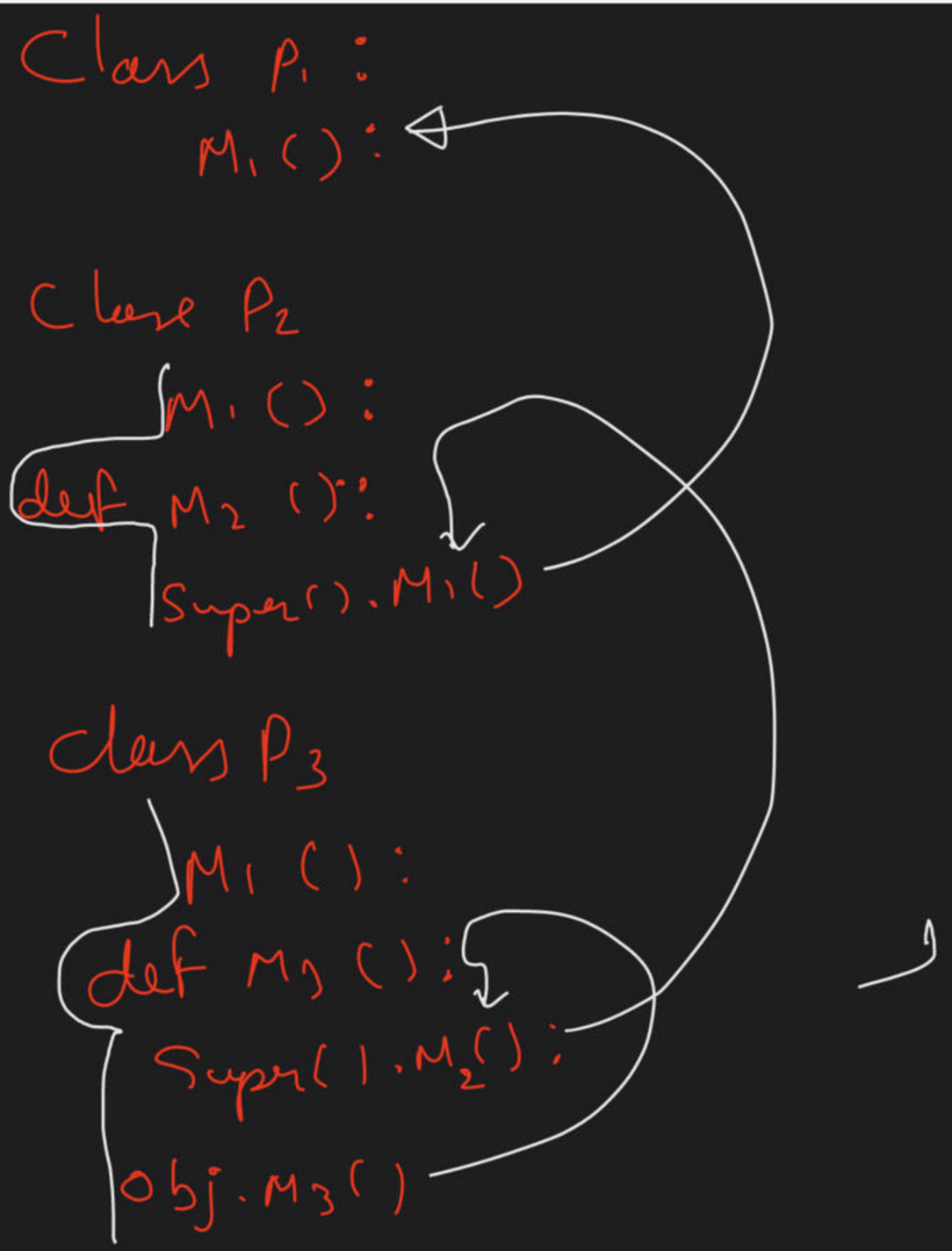
Pre-defined functions are similar to data abstraction

len ( )



```
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.
```


Class P1 :
M1 ( ) :
Class P2
M1 ( ) :
def M2 ( ) :
super( ) . M1 ( )
class P3
M1 ( ) :
def M3 ( ) :
super( ) . M2 ( ) :
obj . M3 ( )

Area of a rectangle: 24
Area of a circle: 153.86
Area of a square: 16
Area of a triangle: 10.0

Python File Open

File handling is an important part of any web application.

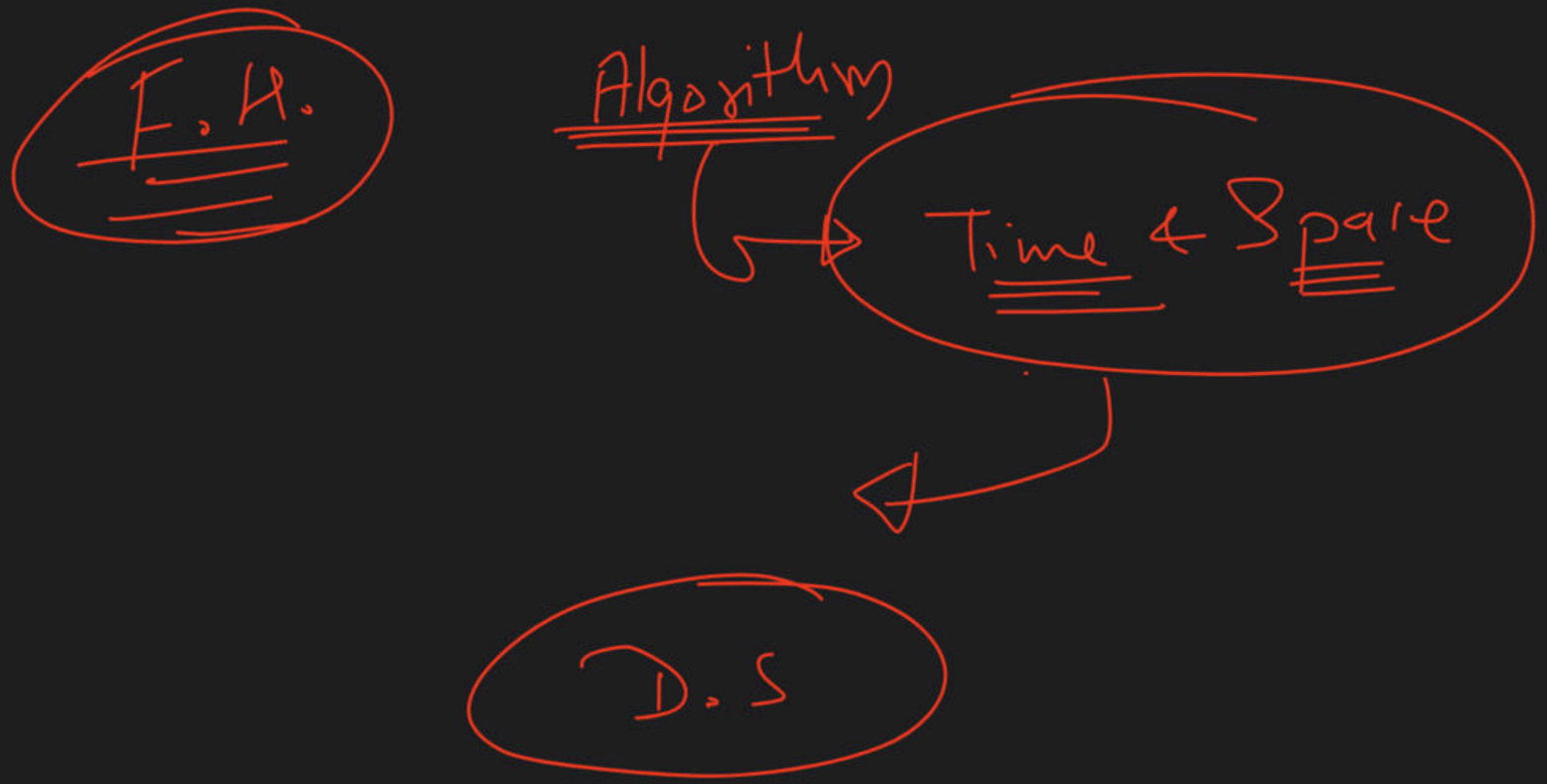
Python has several functions for creating, reading, updating, and deleting files.

File Handling

The key function for working with files in Python is the open() function.

The open() function takes two parameters; filename, and mode.

There are four different methods (modes) for opening a file:



"r" - Read - Default value. Opens a file for reading, error if the file does not exist
"a" - Append - Opens a file for appending, creates the file if it does not exist
"w" - Write - Opens a file for writing, creates the file if it does not exist
"x" - Create - Creates the specified file, returns an error if the file exists
"t" - Text - Default value. Text mode
"b" - Binary - Binary mode (e.g. images)

Syntax

To open a file for reading it is enough to specify the name of the file:

```
f = open("demofile.txt")
```

Or

```
f = open("demofile.txt", "rt")
```

Open a File on the Server

Assume we have the following file, located in the same folder as Python:

newfile.txt

```
Hello! Welcome to newfile.txt
This file is for testing purposes.
Good Luck!
```

To open the file, use the built-in `open()` function.

The `open()` function returns a file object, which has a `read()` method for reading the content of the file:

```
f = open("newfile.txt", "r")
print(f.read())
```

If the file is located in a different location

```
f = open("D:\\myfiles\welcome.txt", "r")
print(f.read())
```

By default the `read()` method returns the whole text, but you can also specify how many characters you want to return:

```
f = open("demofile.txt", "r")
print(f.read(5))
```

Read Lines

You can return one line by using the `readline()` method:

```
f = open("demofile.txt", "r")
print(f.readline())
```

```
f = open("demofile.txt", "r")
print(f.readline())
print(f.readline())
```

```
f = open("demofile.txt", "r")
for x in f:
  print(x)
```

Close Files

It is a good practice to always close the file when you are done with it.

```
f = open("demofile.txt", "r")
print(f.readline())
f.close()
```

Write to an Existing File

To write to an existing file, you must add a parameter to the `open()` function:
`"a"` - Append - will append to the end of the file
`"w"` - Write - will overwrite any existing content

```
f = open("newfile2.txt", "a")
f.write("Extra content added to  the file!")
f.close()

#open and read the file after the appending:
f = open("newfile2.txt", "r")
print(f.read())
```

Python Delete File

Delete a File

To delete a file, you must import the OS module, and run its `os.remove()` function:

```
import os
os.remove("newfile.txt")
```

Check if File exist:

To avoid getting an error, you might want to check if the file exists before you try to delete it:

```
import os
if os.path.exists("demofile.txt"):
  os.remove("demofile.txt")
else:
  print("The file does not exist")
```

Delete Folder

To delete an entire folder, use the `os.rmdir()` method:

```
import os
os.rmdir("myfolder")
```