CAP776

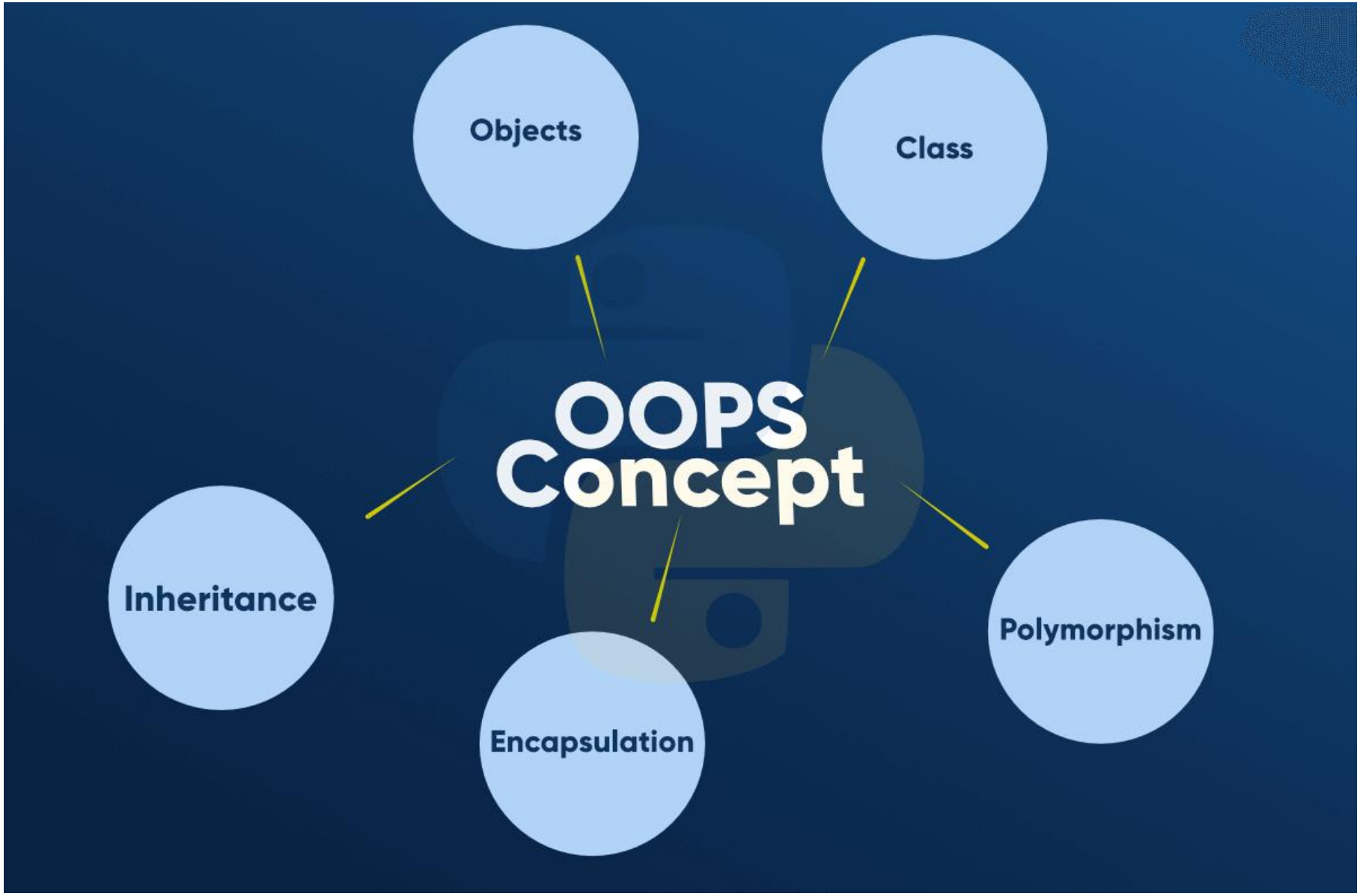
PROGRAMMING IN PYTHON

Created By:
Kumar Vishal
(SCA), LPU

OOP concepts:

OOP concepts :

- ✓ OOP features,
- ✓ encapsulation,
- ✓ inheritance,
- ✓ function overloading,
- ✓ operator overloading and method overriding,
- ✓ Exception handling,
- ✓ catching exceptions,
- ✓ catching multiple exceptions,
- ✓ raising exceptions,
- ✓ custom exception


Objects
Class
OOPS
Concept
Inheritance
Polymorphism
Encapsulation

Why class is important?


BP-6M
NOKIA
M2
8GB
16GB

Class

Class is a collection of similar types of objects.

For example: Fruits is class of mango, apple , orange etc.

Fruits

class in python

A class is a collection of attributes and methods.

Classes are created by keyword class.

Attributes are the variables that belong to a class.

Attributes are always public and can be accessed using the dot (.) operator.

Syntax:

```
class  class_name:
    data_members
    methods
```

e.g:

```
class  emp:
    id=1234
    name=“kumar”
     def disp(self):
          print(“id”,self.id)
    print(“name”,self.name)
```

self

1. Class methods must have an extra first parameter in the method definition. We do not give a value for this parameter when we call the method, Python provides it
2. If we have a method that takes no arguments, then we still have to have one argument.
3. This is similar to this pointer in C++ and this reference in Java.

Create instance :

instance_name=class_name()

e.g:

e=emp()

To call data member and methods:

print(e.id)

print(e.name)

e.disp()

Constructors and Destructors

Constructor in python

- By defining _ _ init_ _
- Paramerized/non parameterized

Eg:

```
class classA:
    def _ _init_ _(self):
        print(“constructor”)

    def __init__(self,name,id):
        self.id = id;
        self.name = name;
```

Note:

- ✓ Python does not support the concept of Constructor overloading.
- ✓ If we try to define the default constructor and the next parameterized constructor within the same class then only the parameterized constructor will remain active but the default constructor will not work.
- ✓ Solution of Constructor overloading using Default argument

```python
# creating class
class employee:
    def __init__(self):# default constructor
        self.eid=11002
        self.ename='Mandeep Singh'
    def __init__(self,empid,name):# parameterized constructor
        self.eid=empid
        self.ename=name
    def getDetails(self):
        print("Employee Id:{}".format(self.eid))
        print("Employee Name:{}".format(self.ename))

# Creating refrence for a class
emp1=employee()
emp1.getDetails()
emp2=employee(111,'Rahul')
emp2.getDetails()
```

Out put: Error

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [3], in <cell line: 14>()
     11         print("Employee Name:{}".format(self.ename))
     13 # Creating refrence for a class
---> 14 emp1=employee()
     15 emp1.getDetails()
     16 emp2=employee(111,'Rahul')
```

```
ss
:
__(self,empid,name='kkk'):# parameterized constructor with def
id=empid
name=name
ails(self):
"Employee Id:{}".format(self.eid))
"Employee Name:{}".format(self.ename))

rence for a class

111)
s()
222,'Rahul')
s()
```

```
Employee Id:111
Employee Name:kkk
Employee Id:222
Employee Name:Rahul
```

Destructors

Destructors are called when an object gets destroyed. In Python, destructors are not needed as much as in C++ because Python has a garbage collector that handles memory management automatically.

The **__del__()** method is a known as a destructor method in Python.

It is called when all references to the object have been deleted.

```
# Python program to illustrate destructor
class Employee:
        # Initializing
        def __init__(self):
                print('Employee created.')
        # Deleting (Calling destructor)
        def __del__(self):
                print('Destructor called, Employee deleted.')
obj = Employee()
del obj
```

Python built-in class functions

1	getattr(obj,name,default)	It is used to access the attribute of the object.
2	setattr(obj, name,value)	It is used to set a particular value to the specific attribute of an object.
3	delattr(obj, name)	It is used to delete a specific attribute.
4	hasattr(obj, name)	It returns true if the object contains some specific attribute.

A Class in Python has three types of access modifiers:

- **Public Access Modifier**
- **Protected Access Modifier**
- **Private Access Modifier**

Public Access Modifier:

The members of a class that are declared public are easily accessible from any part of the program. All data members and member functions of a class are public by default.

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

Protected Access Modifier:

The members of a class that are declared protected are only accessible to a class derived from it. Data members of a class are declared protected by adding a single underscore '_' symbol before the data member of that class.

```
class Person:
    def __init__(self, name, age):
        self._name = name # protected
        self._age  = age  # protected
p1 = Person("kumar", 20)
p1._name
```

Private Access Modifier:

The members of a class that are declared private are accessible within the class only, private access modifier is the most secure access modifier.

Data members of a class are declared private by adding a double underscore ‘__’ symbol before the data member of that class.

```
class Person:
    def __init__(self, name, age):
        self.__name = name # private
        self.__age  = age  # private
p1 = Person("kumar", 20)
p1.__name
```

```
class Person:
    def __init__(self, name, age, sal):
        self.name     = name   # public
        self._age     = age    # protected
        self.__Salary = sal # private

p1 = Person("Rahul", 20, 25000)

print(p1.name)        # public: can be accessed
print(p1._age)        # protected: can be accessed but not advised
# print(p1.__Salary)  # private: will give AttributeError
```

Encapsulation

we can restrict access to methods and variables. This prevents data from direct modification which is called encapsulation. To achieve this we denote private attributes and define setter method to access this only.

```
class computer:
    __modelname="HP T15"
    __price=35000
    __newprice=40000
    def sellPrice(self):
        print("Model:{} Price: {}".format(self.__modelname,self.__newprice))
    def setNewPrice(self,newP):
        self.__newprice=newP

c=computer()
c.sellPrice()

c.__newprice=50000
c.sellPrice()

c.setNewPrice(50000)
c.sellPrice()
```

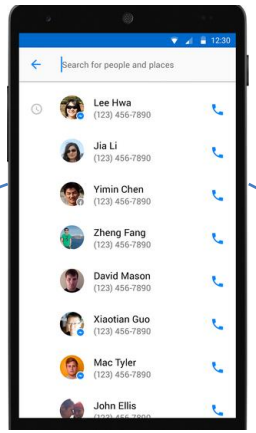
```
Model:HP T15 Price: 40000
Model:HP T15 Price: 40000
Model:HP T15 Price: 50000
```

Inheritance: reusability

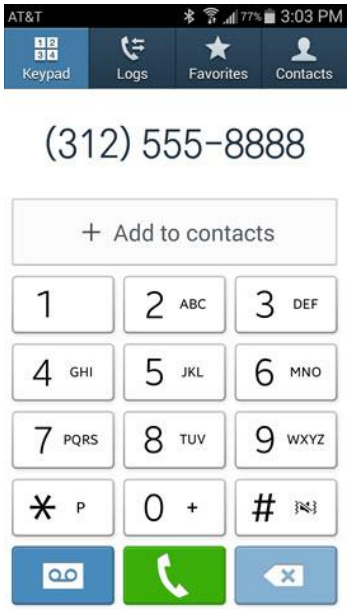
father

mother

child



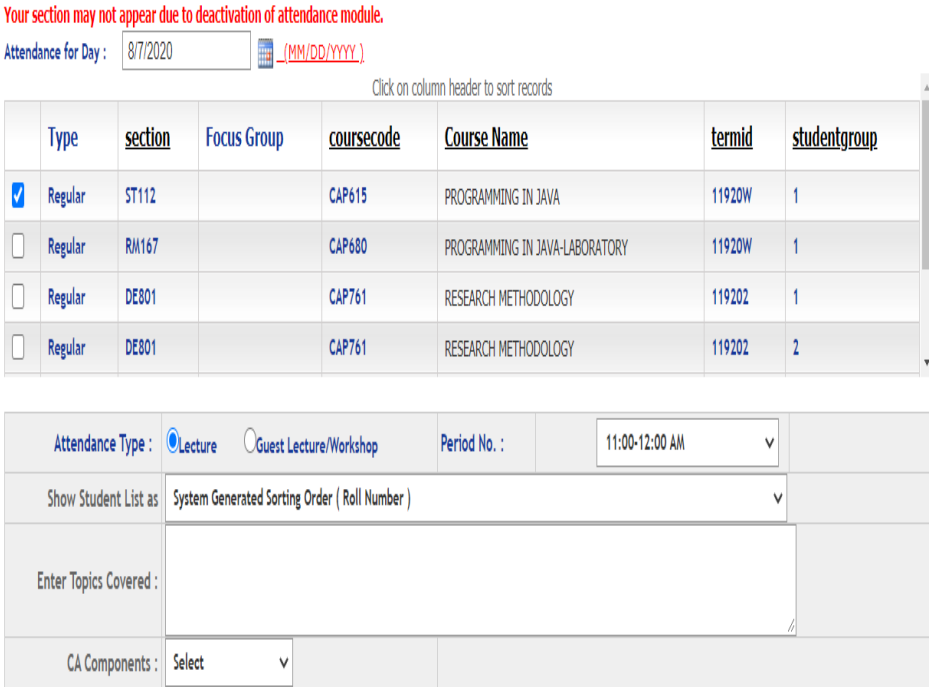
Contact List



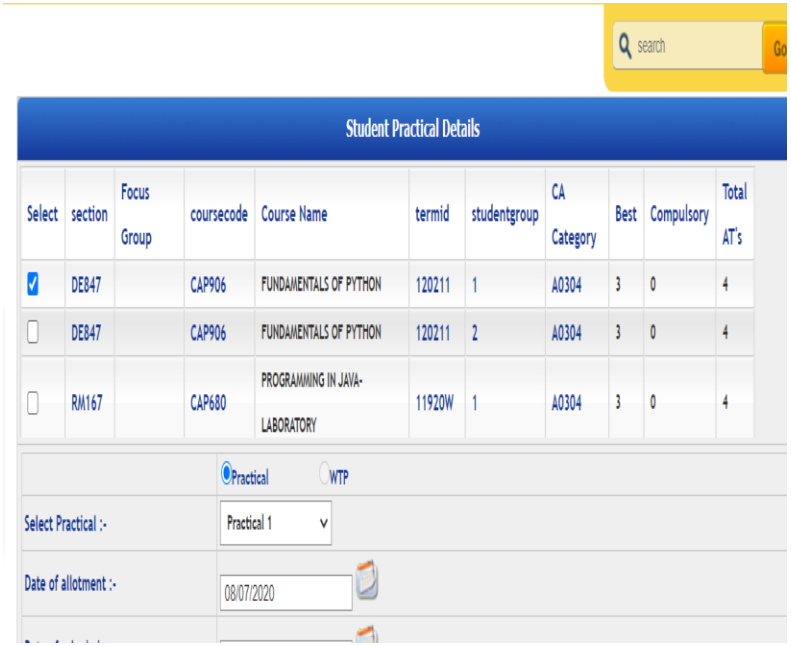
Students

RegistrationNumber	Stream
11601434	CAP - MCA
11604082	CAP - MCA
11616750	CAP - MCA
11907267	CAP - MCA
11908953	CAP - MCA
11909448	CAP - MCA
11909861	CAP - MCA
11910431	CAP - MCA
11910924	CAP - MCA
11911831	CAP - MCA
11814205	CAP - MCA
11915821	CAP - MCA
11915858	CAP - MCA
11916426	CAP - MCA
11900044	CAP - MCA
11901413	CAP - MCA
11901220	CAP - MCA

Students in Attendance Module



Same Students in CA Module



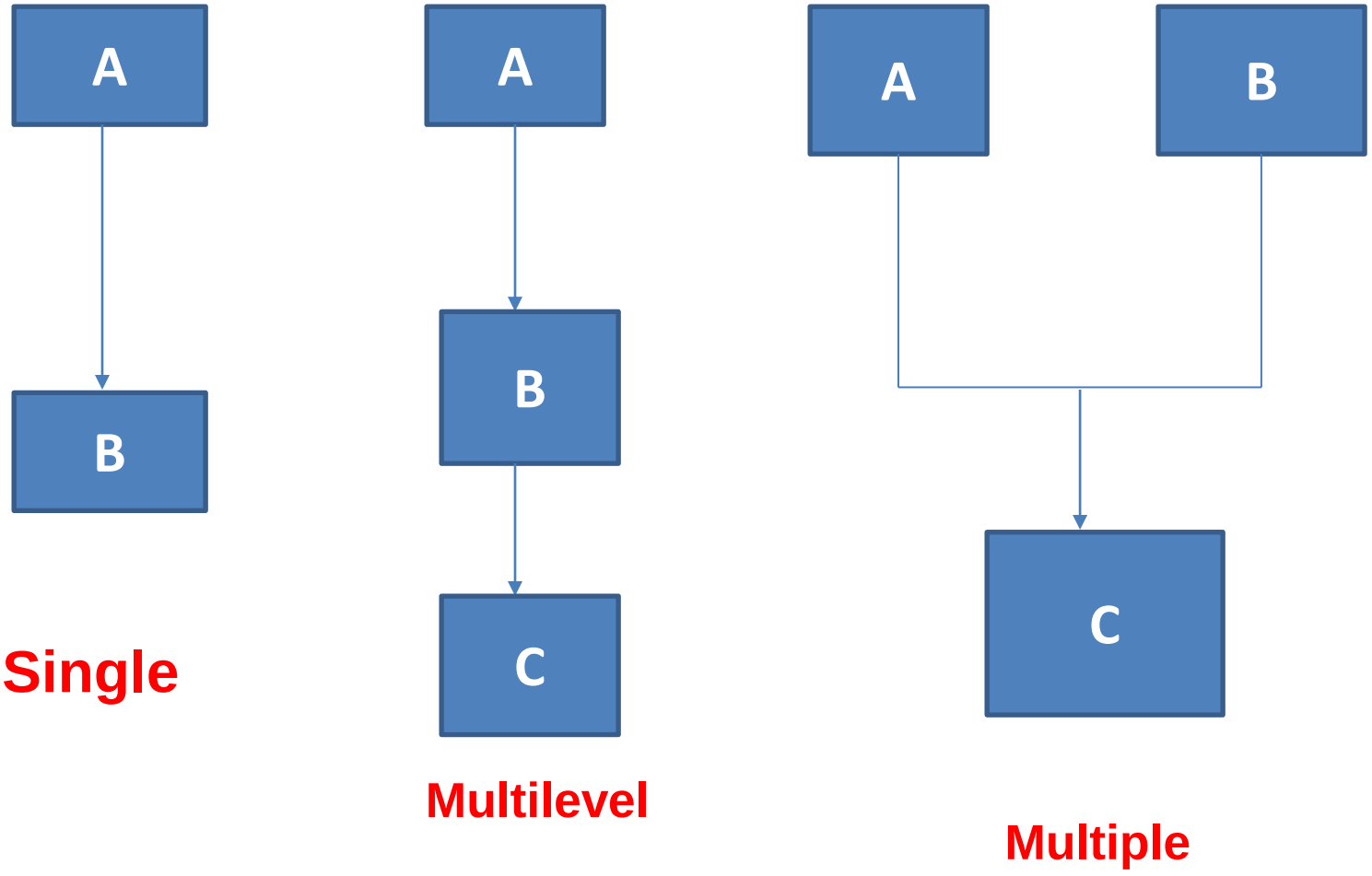
Inheritance

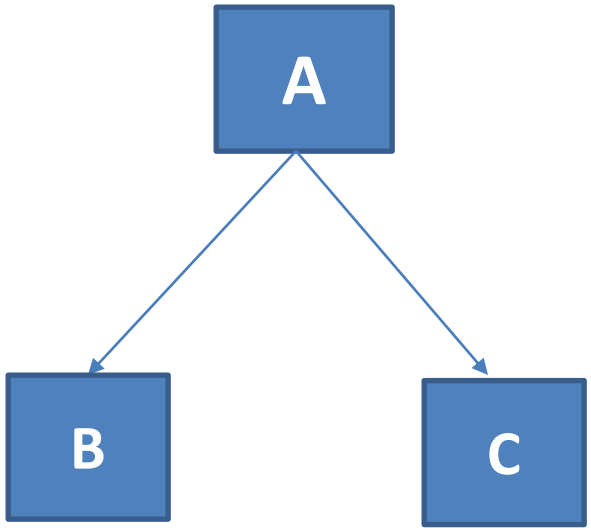
Syntax

```
class BaseClass:
  Body of base class
class DerivedClass(BaseClass):
  Body of derived class
```

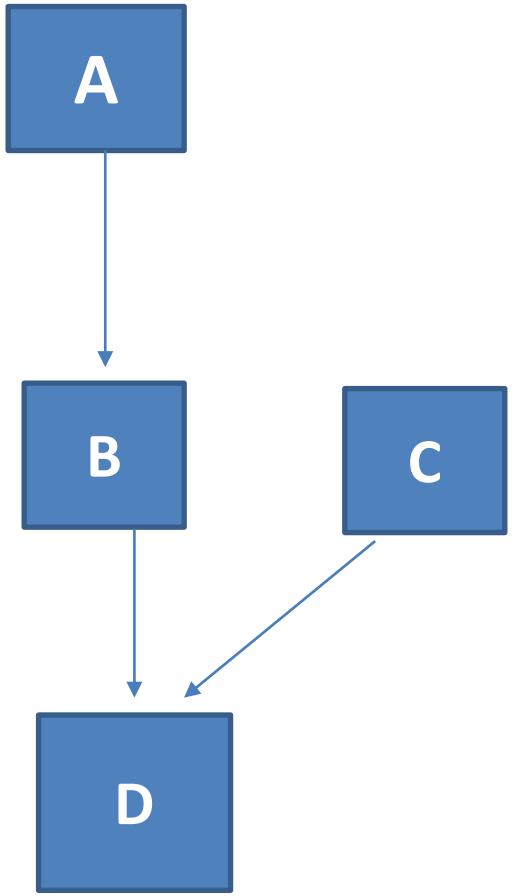
Inheritance: types of inheritance

- One class can hire properties from other class
- Advantages: Reusability





Hierarchical Inheritance



Hybrid Inheritance

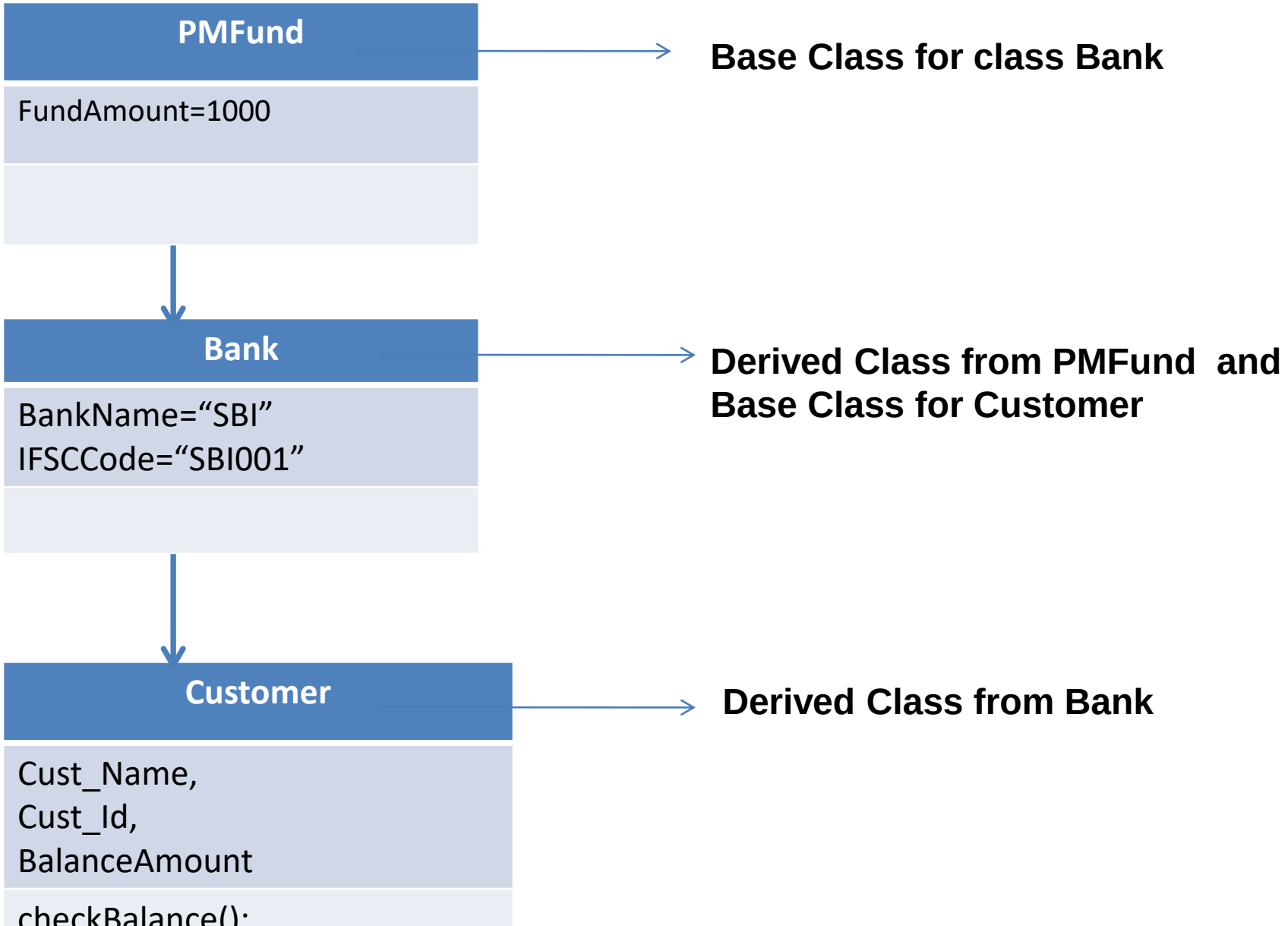
Single inheritance Example

```
class organization:
    _orgName=""
    def __init__(self):
        self._orgName="XYZ Pvt Ltd"
    def getOrg(self):
        print("Organization Name:{}".format(self._orgName))
class login(organization):
    __userName=12345
    __password='ABCD'
    def getDetails(self):
        print("Org:{} User Name:{}".format(self._orgName,self.__userName))
log=login()
log.getDetails()
```

```
Org:XYZ Pvt Ltd User Name:12345
```

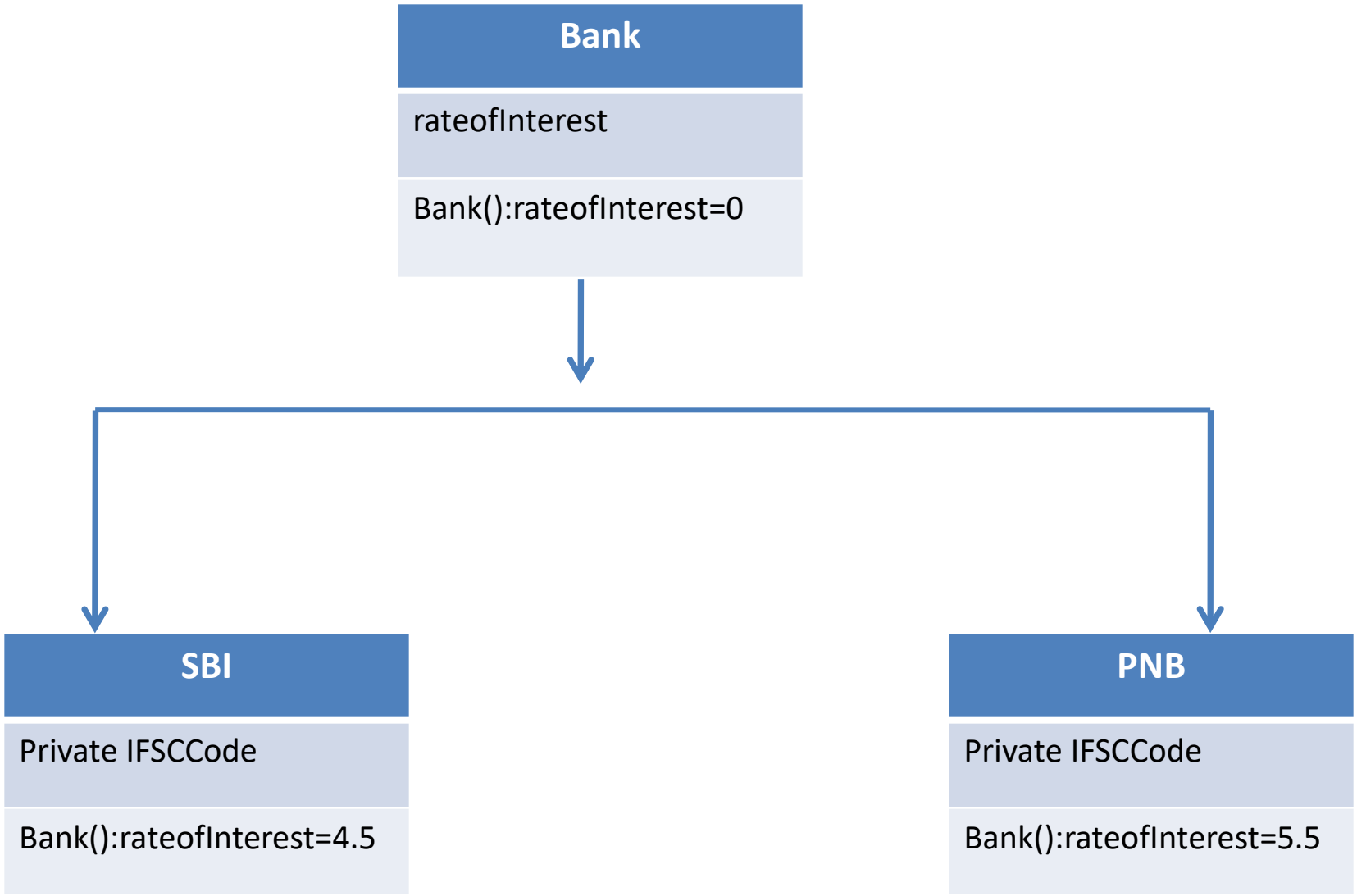
Multilevel inheritance:

One class is going to become base class for other derived class



Hierarchical inheritance:

One base class and multiple derived class, one –to-many relationship



Multiple Inheritance

class Base1:

class Base2:

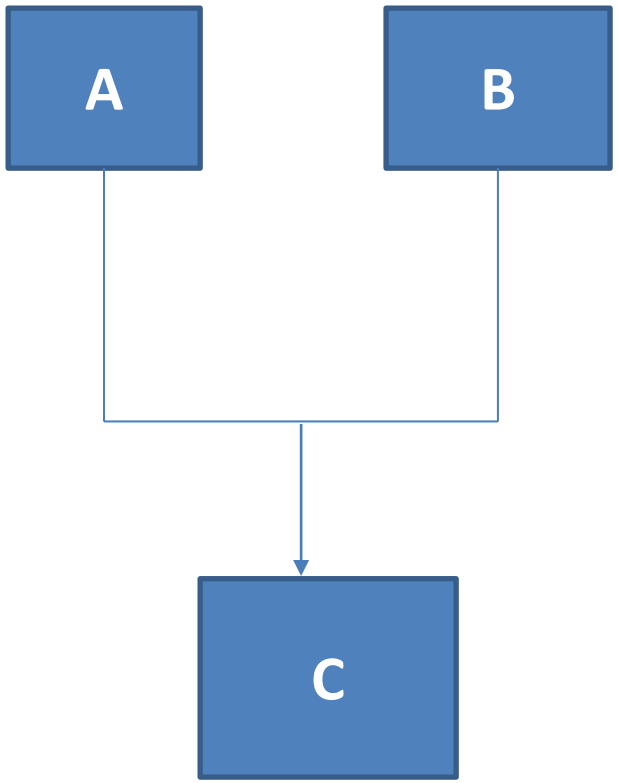
.

.

.

class BaseN:

class Derived(Base1, Base 2, ... BaseN):



Academic

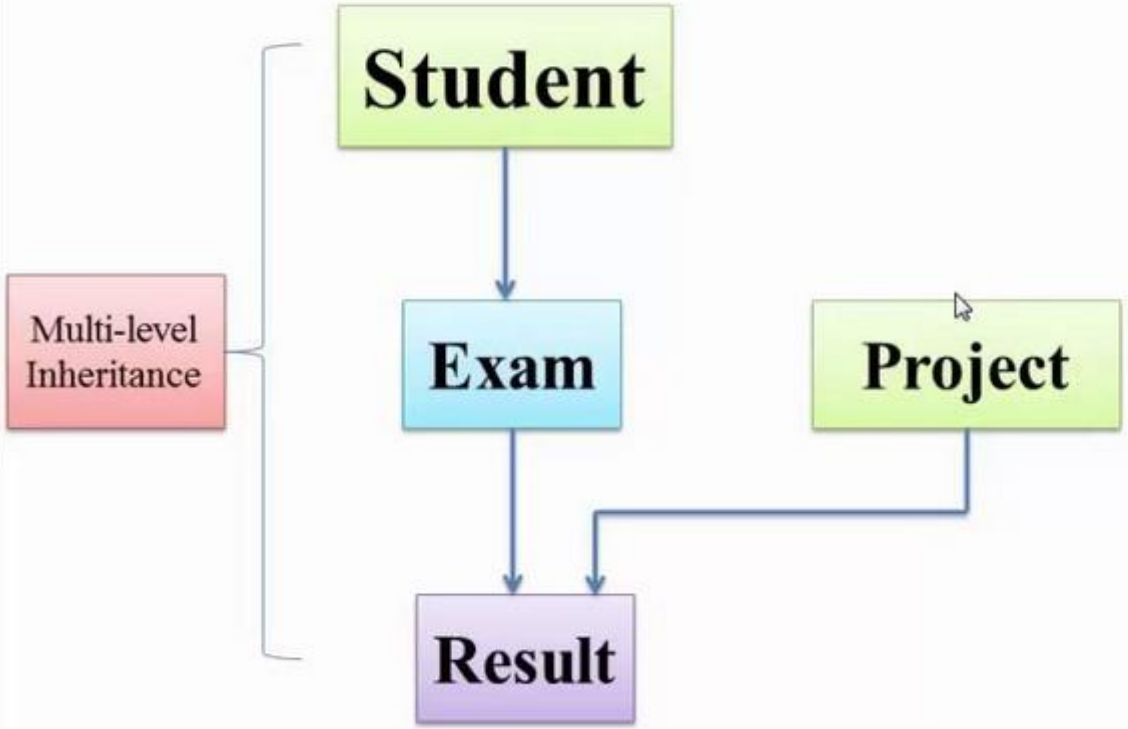
student

```
class academic:
    _course="CAP776"
class sports:
    _game="cricket"
class student(academic,sports):
    __regno=123456
    def getDetails(self):
        print("Reg No.:{} course:{} Game:{}".format(self.__regno,self._course,se
s1=student()
s1.getDetails()
```

```
Reg No.:123456 course:CAP776 Game:cricket
```

Hybrid Inheritance

Combination of two or more type of inheritance to design a program.



What if both parent classes have a method with the same name?

```
class P1:
  def m1(self):
    print("Parent1 Method")
class P2:
  def m1(self):
    print("Parent2 Method")
class C(P1, P2):
  def m2(self):
    print("Child Method")
c=C()
c.m2()
```

In the above scenario, which method will child class inherit?

then it depends on the order of inheritance of parent classes in the child class.

class C(P1, P2): ===>P1”s class method will be considered

class C(P2, P1): ===>P2”s class method will be considered

CONSTRUCTORS in INHERITANCE

By default, the super class's constructor will be available to the subclass.

If child class and super class both have constructors, then?

if you create an object to child class then child class constructor will be executed. While creating object for a class, that class's constructor is first priority.

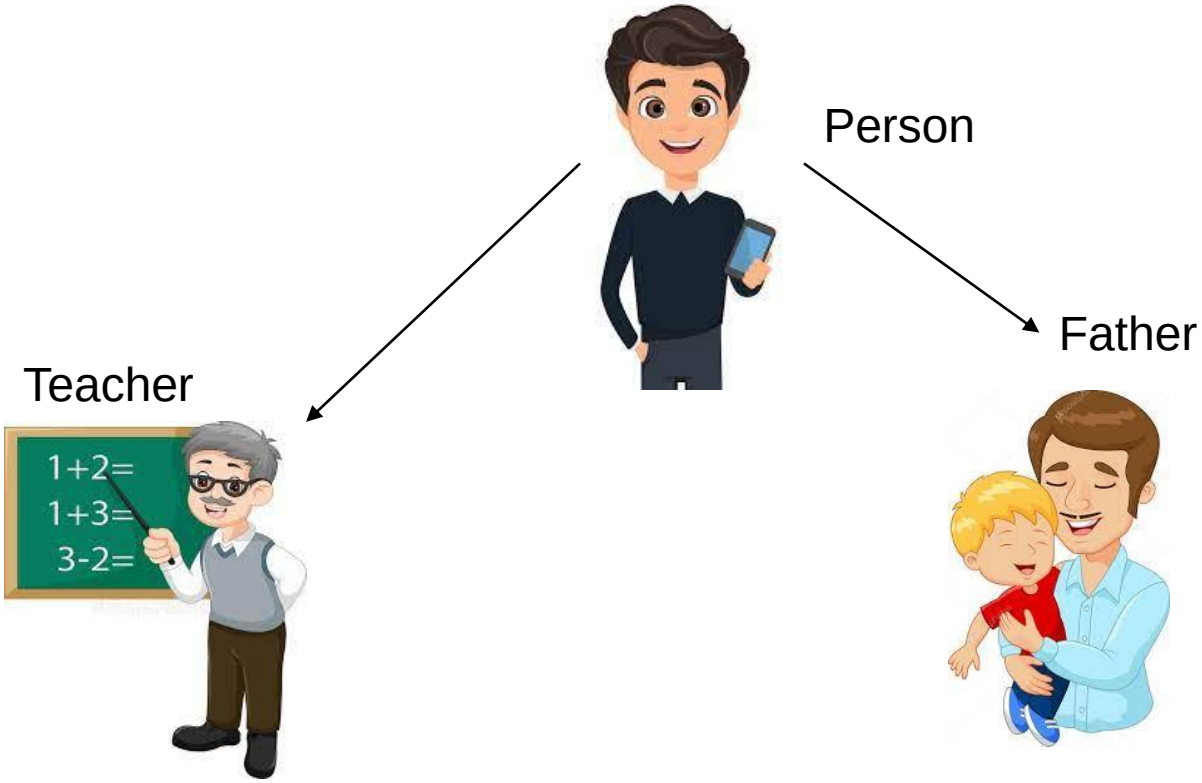
Can we call super class constructor from child class constructor?

Yes, we can call super class constructor from child class constructor by using super() function. super() is a predefined function which is useful to call the superclass constructors, variables and methods from the child class.

```
class A:
   def __init__(self):
      print("super class A constructor")
class B(A):
   def __init__(self):
      print("Child class B constructor")
      super().__init__()

b=B()
```

Polymorphism



Polymorphism is made from 2 words – '**poly**' and '**morphs**.' The word 'poly' means 'many' and 'morphs' means 'many forms.' To put it simply, polymorphism allows us to do the same activity in a variety of ways.

Polymorphism types:

Preparing food according to hotel menu

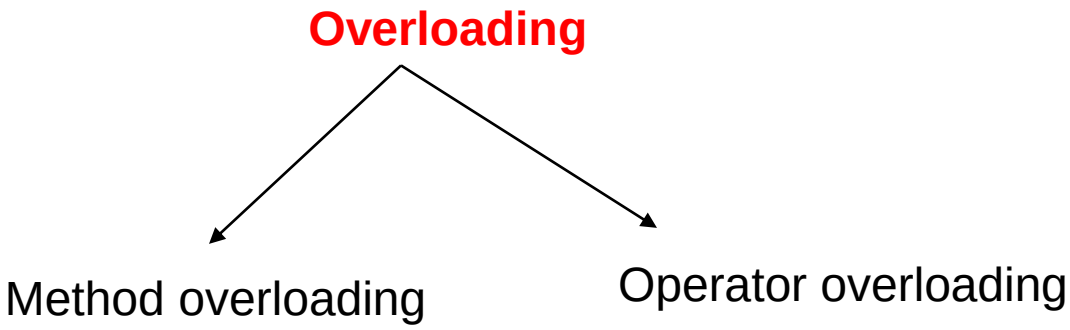
Compile time)

During competition preparing food

(Run-time)

Polymorphism is supported in Python via method overriding and operator overloading. However, Python does not support method overloading.

Note: If we are trying to declare multiple methods with the same name and different number of arguments, then Python will always consider only the method which was last declared.



operator overloading.

- ✓ If we use the same operator for multiple purposes, then it is nothing but operator overloading.
- ✓ ‘+’- addition operator can be used for Arithmetic addition and String concatenation as well
- ✓ * multiplication operator can be used for multiplication for numbers and repetition for strings, lists, tuples etc.

Operator overloading is another type of polymorphism in which the same operator performs various operations depending on the operands. Python allows for operator overloading.

Polymorphism in + operator:

```
a = 10
b = 20
print('Addition of 2 numbers:', a + b)

str1 = 'Hello '
str2 = 'Python'
print('Concatenation of 2 strings:', str1 + str2)

list1 = [1, 2, 3]
list2 = ['A', 'B']
print('Concatenation of 2 lists:', list1 + list2)
```

Polymorphism in * operator:

```
a = 10
b = 5
print('Multiplication of 2 numbers:', a * b)

num = 3
mystr = 'Python'
print('Multiplication of string:', num * mystr)
```

How to Overload the Operators in Python?

Suppose the user has two objects which are the physical representation of a user-defined data type class.

The user has to add two objects using the "+" operator, and it gives an error. This is because the compiler does not know how to add two objects.

So, the user has to define the function for using the operator, and that process is known as "operator overloading".

dir(int)

Operator	Expression	Internally
Addition	`p1 + p2`	`p1.__add__(p2)`
Subtraction	`p1 - p2`	`p1.__sub__(p2)`
Multiplication	`p1 * p2`	`p1.__mul__(p2)`
Power	`p1 ** p2`	`p1.__pow__(p2)`

Operator	Expression	Internally
Less than	`p1 < p2`	`p1.__lt__(p2)`
Less than or equal to	`p1 <= p2`	`p1.__le__(p2)`
Equal to	`p1 == p2`	`p1.__eq__(p2)`
Not equal to	`p1 != p2`	`p1.__ne__(p2)`
Greater than	`p1 > p2`	`p1.__gt__(p2)`
Greater than or equal to	`p1 >= p2`	`p1.__ge__(p2)`

Overload(+)

```
class PigyBank:
    def __init__(self,rupees):
        self.rupees=rupees
    def __add__(self,rs100):
        return self.rupees+rs100.rupees
obj1= PigyBank(100)
obj2= PigyBank(200)
print(obj1+obj2)
```

```
class Student:
   def __init__(self, name, marks):
      self.name=name
      self.marks=marks
s1=Student(“Manu", 100)
s2=Student(“Taran", 200)
print("s1>s2 =", s1>s2)
print("s1<s2 =", s1<s2)
print("s1<=s2 =", s1<=s2)
print("s1>=s2 =", s1>=s2)
```

```
class Student:
  def __init__(self, name, marks):
    self.name=name
    self.marks=marks
  def __gt__(self, other):
    return self.marks>other.marks
  def __lt__(self, other):
    return self.marks<=other.marks
s1=Student("Manu", 100)
s2=Student("Taran", 200)
print("s1>s2 =", s1>s2)
print("s1<s2 =", s1<s2)
```

Overload(>)

```
class A:
    def __init__(self,a):
        self.a=a
    def __gt__(self,b):
        if(self.a>b.a):
            print("first value greater")
        else:
            print("second value greater")
a1=A(10)
b1=A(18)
c1=a1>b1
```

What will be output?

```
class A:
    def show(self):
        pass
class B(A):
    def show(self):
        print("hello")
b1=B()
b1.show()
```

A hello

B hellohello

C nothing will display

D Error

Method overriding

- ✓ Overriding refers to the process of implementing something again, which already exists in parent class, in child class.
- ✓ When a method in a subclass has the same name, same parameters or signature and same return type(or sub-type) as a method in its super-class, then the method in the subclass is said to **override** the method in the super-class.

```
class person:
    def role(self):
        print("I am kumar")
class Teacher(A):
    def role(self):
        print("I am a Teacher")

t1=Teacher()
t1.role()
```

Abstract Class in Python

- ✓ An abstract class is a collection of abstract and non-abstract methods.
- ✓ Abstract methods means A method without definition means without body part.
- ✓ Non-abstract means A method with definition means with body part.
- ✓ Abstract class can contain Constructors, Variables, abstract methods, non-abstract methods, and Subclass.
- ✓ Abstract methods should be implemented in the subclass or child class
- ✓ if in subclass the implementation of the abstract method is not provided, then that subclass, automatically, will become an abstract class.
- ✓ Object creation is not possible for abstract class

How to declare an abstract method in Python:

- Every abstract class in Python should be derived from the ABC class which is present in the abc module.
- Abstract methods, in python, are declared by using **@abstractmethod** decorator.

```
from abc import ABC, abstractmethod
class Bank(ABC):
    @abstractmethod
    def rate(self):
        pass
    def details(self):
        banklist=['SBI','PNB','OBC']
        print(banklist)
class SBI(Bank):
    def rate(self):
        return 4.5
s1=SBI()
print(s1.rate())
```

Exception handling in Python

- An exception can be defined as an unusual condition in a program
- Python provides a way to handle the exception so that the code can be executed without any interruption.

The problem without handling exceptions

```
a=int(input("enter a"))
b=int(input("enter b"))
c=a/b
print(c)
```

Use the try-except statement

```
try:
    #block of code

except Exception1:
    #block of code

except Exception2:
    #block of code

#other code
```

Example:

```
try:
    a=int(input("enter a"))
    b=int(input("enter b"))
    c=a/b
    print(c)
except(ZeroDivisionError):
    print("Invalid Input")
```

We can also use the else statement with the try-except statement

```
try:
    #block of code

except Exception1:
    #block of code

else:
    #this code executes if no except block is executed
```

```python
# use the else statement with the try-except statement
try:
    a=int(input("enter a"))
    b=int(input("enter b"))
    c=a/b
    print(c)
except:
    print("Invalid input")
else:
    print("Thank you")
```

```
enter a6
enter b3
2.0
Thank you
```

A list of common exceptions:

1. **ZeroDivisionError:** Occurs when a number is divided by zero.
2. **NameError:** It occurs when a name is not found. It may be local or global.
3. **IndentationError:** If incorrect indentation is given.
4. **IOError:** It occurs when Input Output operation fails.
5. **EOFError:** It occurs when the end of the file is reached, and yet operations are being performed.

IOError

```
# example of IOError
try:
#this will throw an exception if the file doesn't exist.
    fileptr = open("file.txt","r")
except IOError:
    print("File not found")
else:
    print("The file opened successfully")
    fileptr.close()
```

```
File not found
```

NameError

```
# handling NameError
try:
    num=input()
    print(nu)
except NameError:
    print("NameError occured. Some variable isn't defined.")
```

```
k
NameError occured. Some variable isn't defined.
```

try...finally block

```
try:
    # block of code
    # this may throw an exception
finally:
    # block of code

    # this will always be executed
```

```python
# example of finally block
try:
#this will throw an exception if the file doesn't exist.
    fileptr = open("file.txt","r")
except IOError:
    print("File not found")
finally:
    fileptr.close()
    print("finally block")
```

```
File not found
finally block
```

Declaring Multiple Exceptions

```
try:
    #block of code
  except (<Exception 1>,<Exception 2>,<Exception 3>,...
<Exception n>)
    #block of code
  else:
    #block of code
```

```python
# multiple exceptions
try:
    a=int(input("enter a"))
    b=int(input("enter b"))
    c=a/b
    print(c)
except(ZeroDivisionError,ValueError)as e:
    print("Invalid input error of:",e)
```

```
enter a6
enter bm
Invalid input error of: invalid literal for int() with base 10: 'm'
```

To throw (or raise) an exception, use the raise keyword.

Syntax

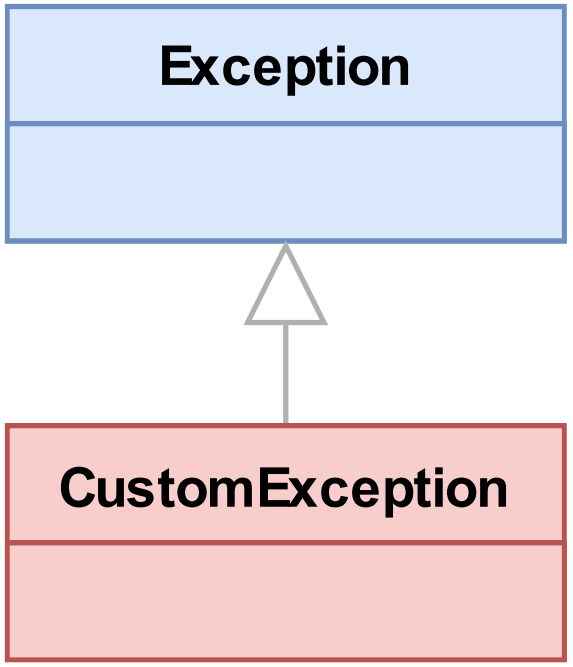
```
raise Exception_class,<value>
```

custom exception

users can define custom exceptions by creating a new class. This exception class has to be derived from the built-in Exception class.

```
class CustomException(Exception):
    def __init__(self,msg='Exception occurs'):
        self.msg = msg
        super().__init__(self.msg)
try:
    raise CustomException()
except CustomException as ex:
    print(ex)
```

```
Exception occurs
```



```
class ageNotValid(Exception):
    def __init__(self,age,msg="age not valid"):
        self.age=age
        self.msg=msg
        super().__init__(self.msg)
class checkAge:
    def allow(self,age):
        try:
            if age>=18:
                print("valid")
            else:
                raise ageNotValid(age)
        except ageNotValid as ex:
            print(ex)
c1=checkAge()
c1.allow(16)
```

```
age not valid
```

THANKS