24-4-23

## Polymorphism:

->Poly means many, morphs means forms (types).

->Polymorphism is one of the features of OOPS.

-> It is an ability of a programming language to present the same interface for several different functionalities / operations.

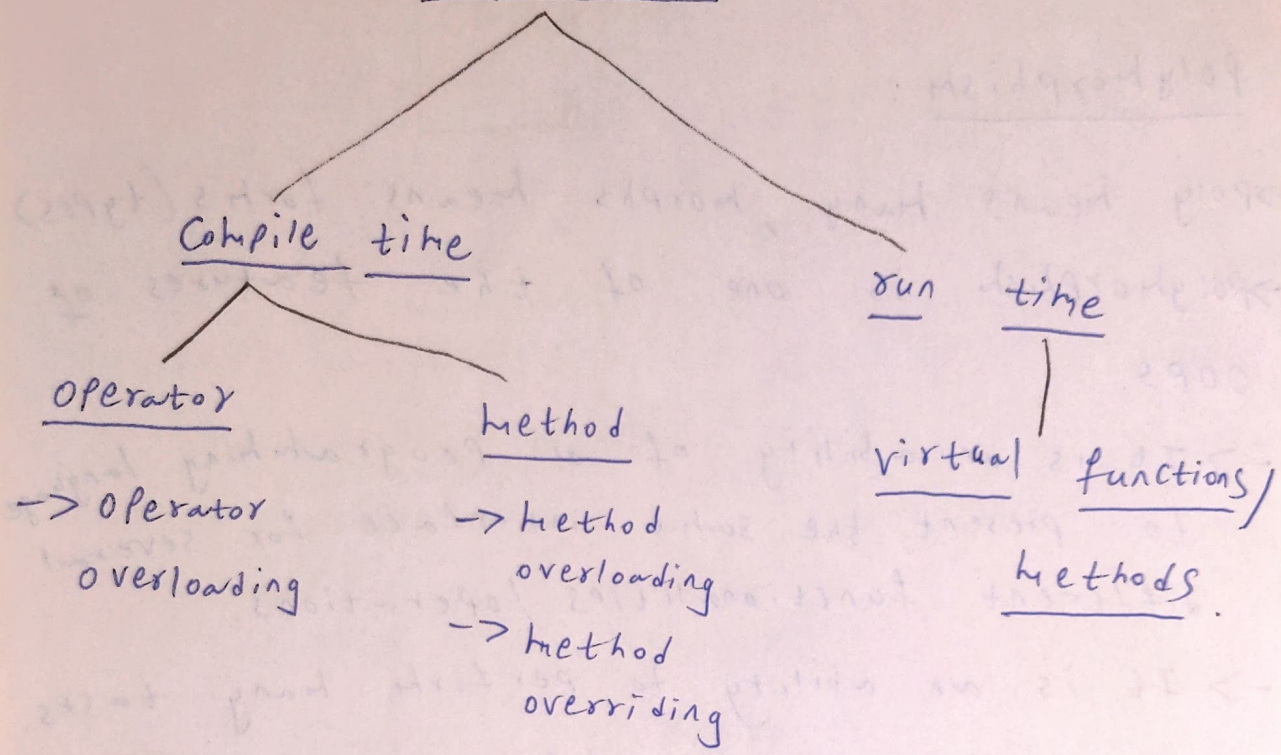
-> It is an ability to perform many tasks.

-> In terms of programming, it means reusing a single person it will work in multiple ways and multiple times.

-> It is the ability of an object or instance to take many forms in different instances.

-> It implements the concept of method overloading, method overriding, operator overloading and virtual functions.

# Polymorphism




## Method overloading :-

-> If 2 methods have the same name but different types of arguments or different no of arguments, then those methods are said to be overload methods.

Ex :- Sum (int a)

Sum (float a)

Sum (double float a)

note : Other languages.

-> In python method overloading is not possible. But operator overloading is possible.

-> If we are trying to declare multiple methods / functions with the same names and different number of arguments, then python will always consider only the last method/function. (because python will accept unique names for variable name, functions name and class name)

How we can handle overloaded method requirements in python:

->Most items, if method with variable number of arguments required then we can handle the default values (arguments) or with length arguments (packing) method.

Solution 1 : (demo program with default arguments)

```
def sample(a = None, b=None, c=None):
    Print (a,b,c)
```

Sample (10, 20, 30)

Sample (10, 20)

Sample (10)

Solutions 2 : (demo program with variable length arguments)

```
def Sample(*a):
    print(a)
    print('solutions')

Sample()
Sample(10, 20, 30, 40)
Sample(10, 20, 30)
Sample(10, 20)
Sample(10)
```

Variable / constructor / method overriding :

-> Method overriding is a feature in object-oriented programming (OOP) that allows a child class to provide its own implementation of a method that is already defined in its parent class.

-> When a child class overrides a method, it provides a new implementation for the method that is executed instead of the original implementation defined in the parent class.

-> to override a method, the child class must have a method with the same signature (name and parameters) as the method in the parent class that it wants to override.

-> when the method is called on an object of the child class, the new method implementation in the child class is executed instead of the original method implementation in the parent class. This allows the child class to modify the behavior of the inherited method to suit its specific needs.

-> method overriding is an important feature of inheritance and polymorphism in OOPs, and it allows

for greater code resuability and flexibility.

-> In the child class, based on our own requirements the method names are the same. In this case the parent class method address is overridden to the child class new method address inside the child class memory.

-> This concept is called method overriding.

Example :- ①

```
Class Parent :
    x = 10  # static state
    def __init__(self, a, b): # Constructor
        print('this is parent class
              init method')
        self.a = a
        self.b = b
    def display(self): # Object method.
        print("Parent class method",
              self.a, self.b)
```

```
obj = Parent (10, 20)   # Parent Class object.

Class Child (Parent):
    x = 100   # Static State
    def __init__ (self, a, b):   # new Constructor
        print("this one is child Class init method")
        self.a = a
        self.b = b
    def display (self):   # object method.
        print("child class method"
              self.a, self.b)
obj1 = Child (500, 600)   # Child Class object.
```

Example :- (2)

```
Class sample ( ):
    def __init__ (self):
        print ("first init method")
    def __init__ (self):
        print ("second init method")
    @ staticmethod
    def display ( ):
        print("first display method")
    @ staticmethod
    def display ( ):
        print ("second display method")
```

25-4-23.

## Operator overloading:

-> Operator overloading is nothing but the ability to redefine (reimplementation) the behavior of an operator for user-defined types or classes. This allows you to use operators such as +, -, *, /, ==, !=, etc., with your own custom objects.

-> To overload an operator in python, you need to define a special method with a specific name that corresponds to the operator you want to overload. These special methods are also called magic methods or dunder methods (short for "double underscore methods")

### example:

We can use the same operator for multiple purposes, that is nothing but operator overloading. Python supports operator overloading

Ex: + operator can be used for arithmetic addition and in collection for concatenation.

```
print (10 + 20)          addition
print ([10] + [20, 30])  concatenation.
```

Ex: * operator can be used for arithmetic multiplication and string repetation process.

```
print (10 * 20)          multiplication
print ('-' * 30)         repetation.
```

ex: addition operation for our own class objects:

```
Class Sample:
    def __init__(self, x):
        self.x = x
a = Sample(10)
b = Sample(20)
print (a+b)
```

o/p:

```
print (a+b)
```

typeerror: unsupported operand type (s) for +:

->above program we are facing the problem with the addition operator in between two userdefine Class objects.

-> we came up with the concept of a magic method. we can overload + operator to work with Class objects also by using magic methods.

-> for each and every operator have their own magic methods are available in python. to overload any operator, we have to override that method in our class.

-> Internally + operator is implemented by using __add__() method. this method is called the magic method for + operator, we have to overload this method in our Class.

Class sample:

```
def __init__(self, x):
    self.x = x
def __add__(self, other):
    return self.x + other.x
a = sample(10)
```

```
b = sample(20)
print(a+b)
```

O/P :-

30.

-> In the above program def __add__(self, other): in this function definition we are using the two args one is self and another one is other, self args is considered the first object address. other args are considering the second object address.

The following is the list of operators and corresponding magic methods.

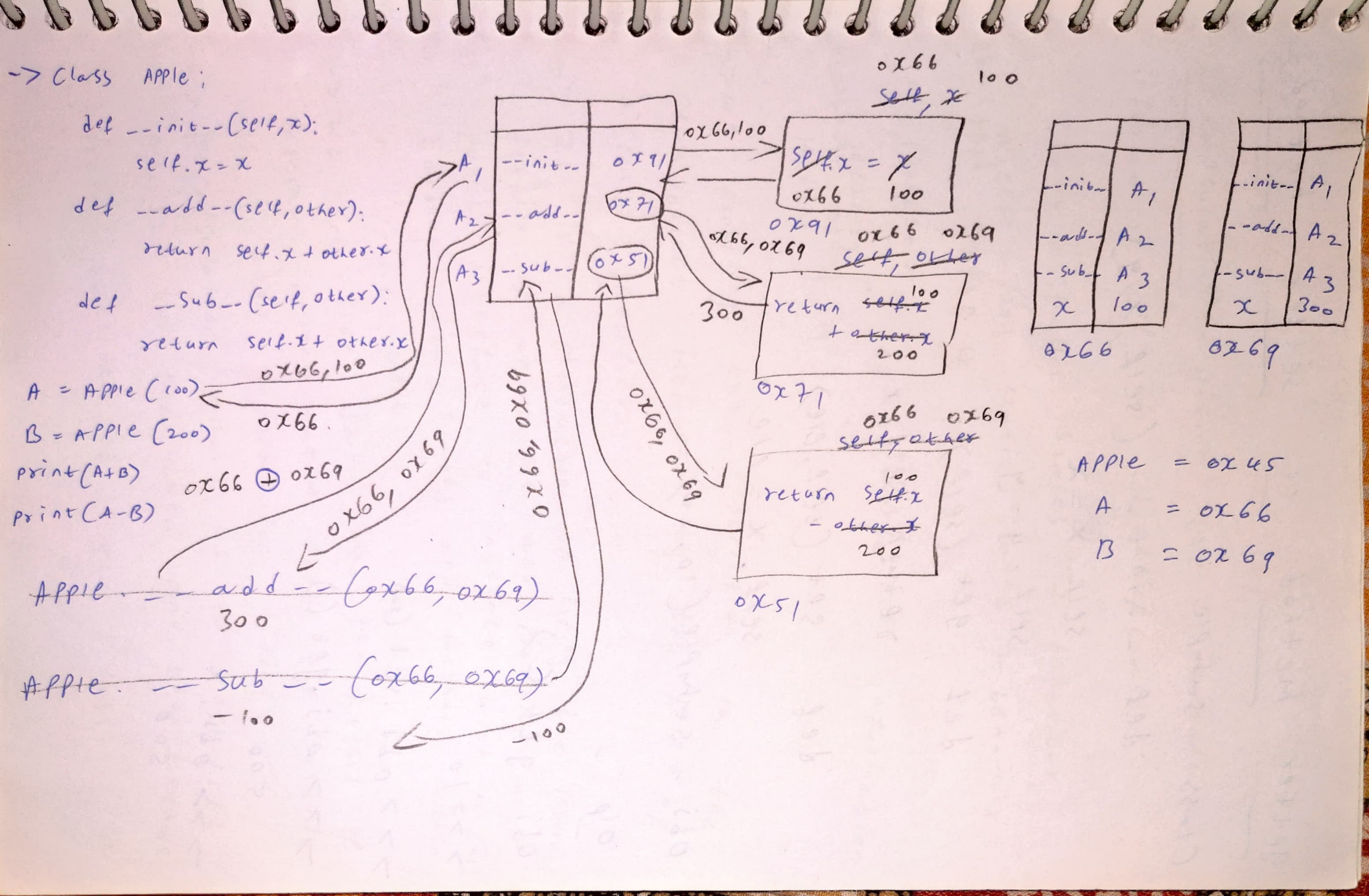
| operator | magic method |
|---|---|
| + | __add__(self, other) |
| - | __sub__(self, other) |
| * | __mul__(self, other) |
| / | __truediv__(self, other) |
| // | __floordiv__(self, other) |
| % | __mod__(self, other) |

| | |
|---|---|
| ** | __pow__(self, other) |
| < | __lt__(self, other) |
| > | __gt__(self, other) |
| <= | __le__(self, other) |
| >= | __ge__(self, other) |
| == | __eq__(self, other) |
| != | __ne__(self, other) |
| += | __iadd__(self, other) |
| -= | __isub__(self, other) |
| *= | __imul__(self, other) |
| /= | __itruediv__(self, other) |
| //= | __ifloordiv__(self, other) |
| %= | __imod__(self, other) |
| **= | __ipow__(self, other) |

-> Class Apple :

```
def __init__(self, x):
    self.x = x
def __add__(self, other):
    return self.x + other.x
def __sub__(self, other):
    return self.x + other.x
```

```
A = Apple(100)
B = Apple(200)
print(A+B)
print(A-B)
```




Apple = 0x45

A = 0x66

B = 0x69

## getter method or setter method :-

```
Class sample :
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def get(self):  # getter methods
        return self.x
    def set(self, ele)  # setter methods
        self.x = ele
obj = sample(10, 20)
```

o/p :-

```
obj.get()
>>> 10
>>> obj.set(500)
>>> obj.get()
    500
>>> obj.x
    500
```

26-4-23.

Encapsulation :- Hiding the internal information of an object / class.

Class - static states + behaviour

object - non static states + behaviour

Protect
security
Hiding
} to Class/object information.

-> we providing the restriction to the Class/object members by the help of access specifiers.

-> It is a phenomenon of hiding the internal details of an object and providing a public interface to interact with it (or) bundling data states and methods within a single unit.

-> we can hide data and the method of internal representation from the outside is called information hiding.

-> Encapsulation allows us to restrict access to the data and the methods from outside of the class.

-> Whenever we are working with Class and dealing with sensitive data, then we are providing access to all variables and methods in the Class.

-> we are providing restrict to data and methods by using access specifiers or modifiers.

-> Encapsulation allows objects to maintain their internal state and behaviour without interference from external entities.

-> It helps to reduce complexity, increase security, and promote modularity and abstraction.

-> In practice, encapsulation involves defining private data members and methods that can only be accessed within the objects class. Public methods are then provided to interact with the object's state and behaviour, which can be accessed by external entities. this way, the internal implementation details are hidden and the objects Public interface provides a well-defined, consistent

way for other code to interact with it.

## Access modifiers :

-> Access modifiers are encapsulation that can be achieved by declaring the data members or method of the class. Python provides three types of access modifiers,

-> Public

-> private.

-> protect.

-> In other languages we have directly access modifiers like private, protect, public.

```
Public    a = 10
protect   b = 20
private   C = 30
```

-> But in python we don't have direct access modifiers, we achieve this by using single underscore for protection and double underscore for private and public directly we return the data or method.

```
a = 10
-b = 20
--C = 30.
```

Access modifiers limit access to the data and method of the Class.

-> public members : accessible anywhere from the Class or outside of the Class.

Note : all member variables / functions of the class by default public modifier.

Syntax :-

```
variable name = value
def methodname ( ) :        function definition.
        |
        |
methodname( )               function call.
```

-> protect members : accessible anywhere from the inside or outside of the Class.

-> to define a protected members, by prefix the member name with a single underscore

Syntax:

```
_variable_name = value

def _methodname( ):        function definition

_methodname( )             function call.
```

-> Protected data members are used when you implement inheritance and want to allow data member access to only one class.

Note: In python public and protected members are no different.

private members: accessible within the class.

-> to define a private member, by prefix the member name with a double underscore.

Syntax:

```
__variable_name = value

def __methodname( ):       -> function definition
        |
        |
__methodname( )            -> function call.
```

-> we can't access them directly from the class objects.

Note: we can directly access ~~private~~ public and protected members from the outside of the class through name.

->If we want access the private members outside of the class, then we are using the below syntax.

Syntax:

```
Classname__variablename
```

-> where Classname is the current class, and variable member is the private variable name.

-> This is all data hiding using access modifiers.

-> If we want to manipulate the private members static states or non static states, we have to use setter and getter method.

Setter method :- By using setter method we can assign the new value to private members either static or non static members.

Syntax :-

```
def setter(self, ele):          # object setter method
    self.__c = ele

@classmethod
def csetter(cls, ele)           # Class setter method.
    cls.__c = ele
```

getter method :- By using getter method we can ~~assign~~ get the value from private members.

```
def getter(self):               # object getter method
    return self.__c

@classmethod
def cgetter(cls):               # Class getter method
    return cls.__x
```

Program :

```
class sample :
    __x = 100
    def __init__(self, a, b, c):
        self.a = a
        self._b = b
        self.__c = c
    def setter(self, ele):          # object setter method
        self.__c = ele
    def getter(self):               # object getter method
        return self.__c
    @classmethod
    def csetter(cls, ele):          # class setter method
        cls.__x = ele
    @classmethod
    def cgetter(cls):               # class getter method
        return cls.__x
```

```
obj = sample(1,2,3)
print(sample.cgetter())
Sample.Csetter(400)
Print(sample.cgetter())

print(obj.getter())
obj.setter(300)
print(obj.getter())
```

27-4-23

## Exception handling

-> Exception: It is an interruption that stops the execution of the program based on the user inputs.

-> An exception is a type of error which is going to occur due to the user inputs at the time of either Compile time or run time.

-> The main reason for the exception is inputs (users inputs or inputs from any other operations).

-> Exceptions are divided into two types.

1. Compile time exception
2. run time exception

## Compile time exception:

-> It is one type of exception that will occur at compile time. that exception we will call it as syntax error.

-> these errors are purely done by users.

-> Because the Python interpreter executes line by line. If a user doesn't follow the syntax rules, then syntax exceptions are raised.

## run time exception:

-> It is also one type of exception that will occur in run time that exceptions we will call it as run time error.

-> these errors are done by programs or other operations.

-> Run time errors are divided by two types

1. Software exception
2. System exception (Hardware exception)

-> Because python executes the program. it the program raised the exceptions based on inputs.

-> Example: string out of range, i/o error / keyboard exception, value error, type error.

-> If an exception arrives in the program, execution will be stopped to avoid such abrupt terminations. we came up with the concept of handling the exception (error) as exception handling.

Exception Handling:

-> Exception is a blockage that has been created for the flow of executions due to the input provided.

-> The phenomenon of handling the exception to avoid the abrupt termination of the program is called exception handling.

-> In this exception handling, we will be handling by using three blocks there are.

1. Try block
2. Except block
3. Finally block

-> first we are identifying the location where an exception might arise. And second, put all those statements in a program based on our scenarios.

-> we handle it using some set of blocks:

try : It is a block of statements where an exception might arise.

except : when an exception is arrived, what is the solution or alternative method of exception.

finally : this block contains that set of statements that is needed to get executed whether an exception is raised or not.

Note : Finally block is not a mandatory statement block.

try :

-> try is a block that contains the set of statements which causes the abrupt termination of the program or the statement where an exception might arise.

-> Whatever statement is arrived at error those statements we written inside the try block.

-> The try block handles / finds the exception and it is intimate to the exception (except) block then the exception block is active.

-> When an exception is raised to handle it smoothly we write the remedial statements inside a block called except.

Except :

-> It contains the remedial statements that state what to do when an exception arrives in the try block.

-> The except blocks are divided into three types :

1. Specific except block
2. Generic except block
3. Default except block.

Specific except block :

-> It can handle only the specific type of exception.

Syntax :

except specific-exception-name :

|

| statements

|

example :

```
print('start')
a = 'hello'
try:
    for i in range(0,100):
        print(a[i])
except indexError:
    print('here small error is there')
print('end')
```

Note : It handles only indexerror.

Specific except blocks are written before generic or default except blocks.

## Generic except block:

-> It can handle the software/application program type of exception.

-> this block contains exception as the name of the exception which is the parent class for all the application/software/program related exceptions.

### Syntax:

```
except exception :
        |
        | statements
        |
```

### example:

```
print ('start')
a = 'hello'
try:
    for i in range (0, 1000):
        print (a[i])
except exception:
    print ("here small error is there")
Print ('end')
```

Note : It handles only application/software/program exception, it will not accept hardware/system exceptions.

-> Generic except block is written after all the specific except blocks.

Default except block :

-> It can handle any type of exception.

-> this block contains all types of exceptions.

-> we are not going to specify the type of exception but this can handle every type of exception.

Syntax :

```
except :
    |
    | statements
    |
```

example :

```
print ("start")
```

```
try:
    for i in range(0,1000):
        print(i)
except:
    print("here shall error is there")
print('end')
```

-> in above program is execute perfectly but when execution time we press (ctrl + c) then it not raise the key interrupt exception.

Note: It handles any type of exception.

->Default except block is written after all the except blocks.

finally:

-> It is one type of exception block.

-> This finally block executes whether the exception is arised or not but the finally block is executed.

Syntax:

```
Finally:
    |
    | statements
    |
```

example:

```
Print ('start')
try:
    for i in range (0, 1000):
        Print (i)
except:
    print ('here small error is
        there')
finally:
    Print ('end')
```

example: for all except blocks.

```
print ('start')
try:
    a = 'hello'
    for i in range (0, 150):
        print (i)
except indexerror:
    print ('specific exception')
except exception:
    print ('generic exception')
except:
    print ('default exception')
```

```
finally:
    print('Stop')
```

Note: If in case we are using all the types of except blocks in our programs. We should follow the precedence of the except block.

Nested exception blocks:

```
a = int(input('enter the a value'))
b = int(input('enter the b value'))
try:
    c = a/b
except Zerodivisionerror:
    print('here error is solved')
try:
    print(c)
except exception:
    print('c is nothing')
finally:
    try:
        for i in range(0,1050):
            print(i)
```

except:

Print(`now exception is arised ; solved that")

finally:

print("finally")

Throwing the exception based on the user requirement:

raise:

-> It is a predefined keyword
-> It is used to throw exceptions based on the user requirements or program requirements.

Syntax:

raise name of exception (or)
raise name of exception(`message to be displayed')

note: raise is mainly used to throw the custom exceptions.

-> It is mainly used in development.

Example 1:

```
print('start')
a = 10
if a == 10:
    raise indexerror
print ('stop')
```

Example 2:

```
print('start')
a = 10
if a == 10:
    raise indexerror("check out the condition")
print('stop')
```

Assert :-

-> It is a predefined keyword.

-> It is a phenomenon of comparing and throwing the exception is called assertion.

-> assertion can be done by using a keyword called assert.

-> It is used to compare and also throws the exception based on the user requirements or program requirements.

-> In the assert keyword we are written the condition.

-> If the condition is satisfied then an exception will not the thrown or else an exception is thrown.

Syntax :

assert condition (or)

assert condition, "exception message"

Note : assert is mainly used for checking and throws the exceptions.

-> It is mainly used in unit testing.

28-4-23

Specific except block :

①

```
try:
    a = int (input ("enter the value"))
    c = a+b
except NameError:
    b = int (input ("enter the value"))
    c = a+b
finally
    print(c).
```

(2)

```
s = 'helloworld'
i = 0
try:
    while i < len(s) + 100:
        print(i)
        i += 1
except IndexError:
    print('out of index')
except keyboardInterrupt:
    print("thank you")
finally:
    for i in "bye":
        print(i)
```

O/P :-

```
1
2
3
4
.
.
.
95
(keyboard Interrupt) thank you
b
y
e
```

generic except block :

(1)

```
print('start')
try:
    for i in range(0, 500):
        print(i)
except Exception:
    print('thank you')
print('stop')
```

O/P :-

```
1
2
3
.
.
.
137
keyboard Interrupt
```

②

```
print ('start')
Class  sample :
    x = 10
obj = sample()
try :
    print ('try block')
    print (sample.y)
except  Exception :
    Sample.y = input ("enter the value:")
finally :
    print (sample.y)
    print ('stop')
```

default except block :

```
Count = 100
ch = 'Go'
for i in range (0, 500):
    try:
        print (i/count, ch)
        Count - = 1
    except :
        count = 100
        ch = 'back'
```

raise :

```
def Sample(a,b):
    if type(a) == type(b):
        C = a+b
        print(c)
    else:
        raise ValueError("both the values
                          Should be same")
Sample(10,20).
```

Create the Custom errors :

```
Class PYSERROR(Exception):
    def __init__(self, x):
        return
def Sample(a,b):
    if type(a) == type(b):
        C=a+b
        print(c)
    else:
        raise PYSERROR("both the types
                        Should be same")
:Sample(10, 20.2)
```

## Create our own exception: (custom exception)

Custom exception is the exception classes that are created by the programmers based on our requirements.

Syntax:

```
Class Name_of_Exception(Exception):
    pass
```

where:

Exception this Parent Class exception of all the exceptions.

Example:

```
Class PySpiderException(Exception):
    pass
```

→ All the custom exceptions should be mandatory thrown by the programmers itself by using raise keyword.

Syntax:

```
class Name_of_Exception(Exception):
    def __init__(self, msg):
        self.message = msg
```

where :

Exception this Parent Class exception of all the exceptions.

Example:

```
Class PySpiderException(Exception):
    def __init__(self, msg):
        self.message = msg
```

Example:

```
Class PySpiderException(Exception):
    def __init__(self, msg):
        self.msg = msg
print("start")
a = 10
if a == 10:
    raise PySpiderException("hai welcome")
Print('stop')
```

ex1:

```
class PySpiderException(Exception):
    def __init__(self, msg="error"):
        self.msg = msg
print('start')
a = 10
```

```
if a == 10:
    raise PySpiderException('hai!')
print('stop')
```

ex 2:

```
class PySpiderException(Exception):
    def __init__(self, msg = "error"):
        self.msg = msg

    def __str__(self) # this method used the format of the exception message.

        return str(self.msg) # return value of __str__ method must be string only.
print('start')
a = 10
if a == 10:
    raise PySpiderException('hai!')
print('stop')
```

1-5-23

## Assert :

ex :-

```
class pyserror(exception):
    def __init__(self, x):
        return
def sample(a,b):
    if type(a)==type(b):
        c = a+b
        return c
    else:
        raise pyserror("both the value should be same")
sample(10,30)
```

O/P :

```
>>> assert sample(10,30)==40
```

# o/p is given by user. This assert keyword checks whether the input is matched with o/p or not if matched doesn't display anything, else shows error.

```
>>> assert sample(10,30)==50
assertionerror:
```

# Abstraction

**Abstraction**: It is a phenomenon of hiding the internal functionality of a specific task or code.

-> abstraction is used to hide the internal functionality of the method from the users.

-> The user can access the basic implementation of the functionality, but the inner working process is hidden.

-> user is simplifying with what method we are doing but they don't know how it does.

-> In this concept we use abstract class and abstract methods.

**Abstract class**:

-> Abstract class is nothing but structure / format / pattern / blueprint of other classes. It acts as a guideline for other classes.

-> Abstract class is used to provide standard structure / interface / set of methods for different implementations using any other class.

-> Abstract class is one that contains one or more than one abstract methods.

-> the abstract class is defined by using "abc module":

-> first import the abc package to our program file by using below syntax.

Syntax:

```
from abc import ABC
```

where:

abc is package name for abstract
ABC is parent abstract class

Note: abc is a predefined package in Python

How to create our own abstract class:

-> create a class and give the name of the class and inherit the ABC parent class to our classes.

ex:-

```
from abc import ABC
class Sample(ABC):
    pass
```

Note: Sample is abstract class.

-> Python doesn't provide the abstract class itself. that's why we are doing the above procedure.

-> Abstract Class can be useful when we are developing medium scale and large scale applications.

-> Abstract Class can contain the normal method and abstract method.

-> In abstract Class we can't create object.

Abstract Method :

-> It is one with a declaration, but not implementation in abstract class.

-> the implementation happens in the child class method by using the same abstract method name. In child class defined as abstract name with user required implementation.

-> The ABC works by decorating methods of the parent class as abstract.

-> we are declaring the abstract method for our methods using the decorated abstractmethod.

Syntax:

@abstractmethod.

-> When we are using @abstractmethod decorator first we import the method into the abc package and then define an abstract method to our function.

Syntax:

from abc import ABC, abstract method

-> It dynamically becomes the abstract method inside the abstract class.

How to define abstract classes in our program

ex:- from abc import ABC, abstract method

@abstractmethod

def methodname():

pass

Note: above example we are defining the abstract method inside the abstract class.

Note : When we define the @abstract method in any method inside abstract class, we can't create an object for abstract class.

ex1:

```
from abc import ABC, abstractmethod
Class sample(ABC):
    @abstractmethod
    def display():
        pass
    @abstractmethod
    def show():
        pass
Class demo(sample):
    print("demo class")
obj = demo()
res = obj.display()
```

Note : In the above example we can't create objects for abstract class and inherited child class, because we are not creating a method with the same abstract method name in child class.

-> we are not following the Pattern for abstract so thats the reason it shows error. Regarding this problem, we have the below example.

ex:-

```
from abc import ABC, abstractmethod
class sample(ABC):
    @abstractmethod
    def display():
        pass
    @abstractmethod
    def show():
        pass
class demo(sample):
    print("demo class")
    def display(self):
        a = 10
        b = 20
        return a * 20
    def show(self):
        print(show)
obj = demo()
res = obj.display()
```

-> In this example we define the one abstract base class and inherited child class named as sample and we also define the two abstract methods called display and show.

-> the sample class is inherited by the demoClass.

-> we implement the abstract method in such child class.

-> we created the object of the subclass and invoked the display and show methods.

-> the hidden implementations for the display() and show() methods inside the child class.

-> the abstract method display, show method, defined in abstract class is never invoked.