ICE NO.	TITLE	DATE	TEACHER'S SIGN / REMARKS

Inheritance :

- Whenever Entities are having is-A-relationship between them, then we will perform Inheritance
- Inheritance is the process of Acquiring the properties and functionalities of one class into Another class.
- The Class from which we Acquire properties is known as parent / Super / Base class.
- The Class into which we Acquire properties is known as Child / sub / Derived class

Classification of Inheritance :

Single level Inheritance (one parent, one child)
Multiple Inheritance (multiple parents, one child)
Hierarchial Inheritance (one parent, multiple child)
Multilevel Inheritance (it is a level of Inheritance)
Hybrid Inheritance (it is Combination of more than one type of Inheritance)

Single Level Inheritance : It is the process of Inheriting the properties of one class into single Another class

Syntax :

```
Class parent :
    pass
Class child (Parent) :
    pass
```

Note : Direction of Inheritance of properties
Right to Left
Direction of Execution of properties
Left to Right

Internal procedure which is happening while Inheritance

For Parent Class :

1. Parent Class Dictionary will be Created.
2. And all properties are defined into that dictionary.

For Child Class :

- Child class Dictionary will be Created.
- first properties of Parent will be defined in to child dictionary after that child class properties will be Derived.

Note : Direction of Inheritance happens Right to Left
child can Access properties of parent but parent cannot accepts child class properties.

Example :

```
Class A :
    x = 90
    y = 100
Class B(A) :
    x = 341
    z = 112
oa = A( )
ob = B( )
```

Modifying Generic Property of parent class with parent class.

```
A.y = 2000
print (A.y) # 2000
print (oa.y) # 2000
print (B.y) # 2000
print (ob.y) # 2000
```

if modify Generic properties of parent class with parent class then it will affect in parent class and its object, child class and its objects

```
A.x = 2222
print (A.x) # 2222
print (oa.x) # 2222
print (B.x) # 341
print (ob.x) # 341
```

A

	Keys	Values
A-1	x	90
A-2	y	100

x_{111}

B(A)

	Keys	Values
B-1	x	A-1
B-2	y	A-2
B-3	z	112

x_{222}

Modifying Generic Property of parent class with parent class object.

oa.y = 2000

print (A.y) # 100

print (oa.y) # 2000

print (B.y) # 100

Print (ob.y) # 100

OA

	Keys	Values
OA-1	x	A-1
OA-2	y	A-2

y_{111}

O(B)

Keys	Values
x	B-1
y	B-2
z	B-3

y_{222}

Note : If modify Generic property of parent class with parent class object then it will Effect only parent class object.

Modifying Generic Property of parent class with child class.

B.y = 2000

print (A.y) # 100

print (oa.y) # 100

print (B.y) # 2000

print (ob.y) # 2000

⇒ If modify Generic Property of parent class with child class and its child class then it will effect only in child objects

Modifying Generic Property of parent class with Child class object.

ob.y = 2000

print (A.y) 100

print (oa.y) 100

print (B.y) 100

print (ob.y) 2000

⇒ If modify G.P of parent Class with child class object it will affect only in that child class object

```
class Bank_V1 :

    bank_name = 'SBI'

    bank_roi  = 7

    bank_branch = 'HYDERABAD'

    def __init__(self, n, a, ac, b) :

        self.name    = n
        self.age     = a
        self.account = ac
        self.balance = b

    @classmethod
    def display_generic (cls) :

        print(f' bank name is {cls.bank_name}')

        print (f' bank roi is {cls.bank_roi}' )

        print ( f' bank branch is {cls. bank_ branch}')

    @ staticmethod

    def get_int ( ) :

        Value = int ( input ('Enter Value'))

        return Value

    def withdraw (self) :

        amount = self. get_int ( )

        if self. balance > amount :

            self. balance - = amount

            print ('withdraw is successful')

    else Else :

        ↔print ('low Balance')


    def deposite (self) :

        amount = self. get_int ( )

        self. balance + = amount

        print ('deposite is successful')

Class Bank_V2 ( Bank_V1 ) : Inheriting

    bank_ branch = ' Bangalore'

    bank_ ifsc = 1234

    def customer_ details (self) :

        print (f' name of customer is {self.name}')

        print (f' age of customer is {self. age}')

        print (f' account of customer is {self. account}')

        print (f' balance of customer is {self. balance}')

    @ classmethod

    def change_roi (cls) :

        newroi = cls. get_int ( ) [

        cls. bank_roi = newroi

        print ('roi is modified' )

madhu = Bank_V2 ('chaitra', 18, 56342, 12345)

madhu. bank_ details ( )

madhu. customer_ details ( )
```

Bank - V1

	Keys	Values
B-1	bank_name	'SBI'
B-2	bank_roi	7
B-3	bank_branch	'HYDERABAD'
B-4	__init__	
B-5	display_generic	
B-6	get_int	
B-7	withdraw	
B-8	deposite	

Bank - V2

Keys	Values	
bank_name	B-1	
bank_roi	B-2	
bank_branch	B/3 Bangalore	
bank_ifsc	1234 (created)	
__init__	B-4	
Display generic	B-5	
get_init	B-6	
withdraw	B-7	
deposite	B-8	
customer_details		Created in Version 2
change_roi		Created in Version 2

Polymorphism

Polymorphism is a process of a method behaving in different ways (forms / functionalities)

Polymorphism can be achieved in two ways

Method Overriding

Method Overloading

Method Overriding :

- In case of Inheritance we perform method overriding
- When we perform Exactly ~~Inherit~~ method overriding (is) whenever we need to change implementation of parent class (Super class) method in child class (sub class)
- How we perform → we define a method in child class which is same as that of parent class.
- We can override __init__ (constructor), static methods, Class methods as well as object method.

Bank_V2 :

bank_name = 'SBI'

bank_roi = 7

bank_branch = 'Bangalore'

bank_ifsc = 1234

Methods

- __init__ (name, age, account, balance, pin, adhar)
  (constructor over riding) ↳ [constructor implementation with six properties]
- Bank_details → 4 properties (class method overiding)
- get_int
- withdraw → implementation → first Validate pin then allow to withdraw]
  ↳ (object method overriding)
- Deposite
- Change_roi

```
class Bank_V2 (Bank_V1):

    bank_branch = 'Bangalore'
    bank_ifsc = 1234

    def __init__ (self, n, a, ac, b, pin, ad):
        self.name = n
        self.age = a
        self.account = ac
        self.balance = b
        self.pin = pin
        self.adhar = ad

    @classmethod
    def bank_details(cls):
        print (f'bank name is {cls.bank_name}')
        print (f'bank roi is {cls.bank_roi}')
        print (f'bank branch is {cls.bank_branch}')
        print (f'bank ifsc is {cls.bank_ifsc}')

    def withdraw (self):
        pi = self.get_int()
        if self.pin == pi:
            amount = self.get_int()
            if self.balance >= amount
                self.balance -= amount
                print ('withdrawal is successful')
            else:
                print ('low balance')
```

```
Else:
    print('incorrect pin')

@classmethod
def change_roi(cls):
    newroi = cls.get_int()
    cls.bank_roi = newroi
    print('roi is updated')

geetha = Bank_V2('GM', 24, 567876, 34156898, 12344, 98765432)
geetha.withdraw()
Bank_V2(Bank_V1)
```

Keys	Values
bank_name	(B-1)
bank_roi	(B-2)
bank_branch	~~(B-3)~~ Bangalore (overriding)
bank_ifsc	12344
__init__	~~(B-4)~~ Block with 6 args (MOR)
display_generic	~~(B-5)~~ printing 4 statements (MOR)
get_int	(B-6)
withdraw	~~(B-7)~~ block with new implementation (MOR)
deposite	(B-8)
Change_roi	Block

Note : If we use method overriding, code redundancy will be more

- So we have to use chaining along with method overiding.

Chaining : It is the process of calling constructor (or) method of another class in current class constructor/method.

We have two types of chaining

1. Constructor chaining
2. Method chaining

Constructor Chaining : It is the process of calling constructor of another class constructor in current class Constructor

We can perform in two ways

1. Super method
2. Class Name

Super() method : Super() method is used only in case of inheritance

→ super().__init__(arguments) ←

ClassName : Calling of constructor by using class Name can be used in case of inheritance as well as Non Inheritance

ClassName.__init__(self, arguments)

Method Chaining : Process of calling the method of another class in current class method.

We can perform in 2 ways

1. Super method
2. By class Name

Super() method : It is used only in case of Inheritance

Syntax : Super().Methodname (Arguments)

ClassName : Calling of method by using class Name can be used in case of inheritance and as well as Non-Inheritance.

Syntax : (calling classmethods chaining by class)

ClassName.MethodName (self, arguments)

ClassName.MethodName (Arguments)

```
Class Bank_v2 ( Bank_v1 ) :
    bank_branch = 'Bangalore'
    bank_ifsc = 1234
    def __init__ (self, n, a, ac, b, p, ad) :
        super().__init__(n, a, ac, b)
        # Bank_v1.__init__ (self, n, a, ac, b)
        self.pin = p
        self.adhar = ad
    def customer_details (self) :
        super().customer_details()
        # Bank_v1.customer_details (self)
        print (f'adhar of customer is {self.adhar}')
    @classmethod
    def bank_details (cls) :
        super().bank_details()
        Bank_v1.bank_details (self)
        print (f'ifsc of bank is {cls.bank_ifsc}')
    def withdraw (self) :
        pin = self.get_int()
        if pin == self.pin :
            super().withdraw()
            Bank_v1.withdraw (self)
        Else :
            print ('incorrect pin')
```

Seetha = Bank_V2('SD', 23, 415678, 3415678, 1234, 9876541112)

Seetha.customer_details()

Seetha.bank_details()

Seetha.withdraw()

Method overloading : It is a process of defining a method with same name multiple times in a same class but with different arguments

- In python method overloading is not possible
- partially we can achieve it by providing default Arguments.

Eg :

```
class A :
    def __init__(self, n):
        self.name = n
        print('init with one argument')

    def __init__(self, n, a=20):
        self.name = n
        self.age = a
        print('init with two arguments')

oa1 = A('abc')
```

↳ default argument

A

Keys	Values
__init__	1 arg (2 args)

Note : By default Every class is Inherited from Object class.

⇒ __dict__ : It is a special variable of object class

- If we use __dict__ with class it will return class related properties
- If we use __dict__ with object it will display specific properties
- Output format of __dict__ is a dictionary

Example :

```
print(Bank_V2.mro())
print('#' * 30)
print(Bank_V2.__dict__)
print('#' * 30)
print(geetha.__dict__)
```

Method Resolution Order :

- MRO stands for Method Resolution Order
- Used for Specifying the order in which we have to Execute the methods in case of Inheritance.
- Direction of representation is Left to right.

Multiple Inheritance : It is the process of acquiring properties of multiple parent class into Single child class

Example :

```
class Parent1 :
    pass
class Parent 2():
    pass
class Child (parent1, Parent 2 ) :
    pass
```

Ex:

```
Class Father :
    car = 'Nano Car'
    bike = 'Chekat bike'
    cycle = 'Chandrababu Cycle'
Class Mother :
    bike = 'Scooty'
    cycle = 'ladies cycle'
Class SON1 (Mother, father) :
    bike = 'Himalayan'
Class SON2 ( Father, Mother) :
    bike = 'Himalayan'
print (SON1.mro()) # SON1 --> Mother --> Father --> object class
S1 = SON1( )
print (S1. bike) # Himalayan
print (S1. cycle) # Ladies cycle
print (S1. car ) # Nano Car
print ( SON2. mro( )) # SON2 --> Father --> Mother --> object class
S2 = SON2 ( )
print (S2. bike) # Himalayan
print (S2. cycle) # Chandrababu cycle
print (S2. car) # Nano Car
```

Hierarchical Inheritance : Process of acquiring the properties of Single parent class into multiple child classes

Syntax :

```
class Parent:
    pass
class SON1 ( Parent):
    pass
Class SON2 (Parent ):
    pass
```

```
Class Father :
    car = 'Nano Car'
    bike = 'chetak bike'
    cycle = 'ChandraBabu Cycle'
Class SON1 (Father) :
    bike = 'Himalayan'
Class SON2 (Father) :
    bike = 'Duke'
print (SON1.mro()) # SON1 --> Father --> object class
S1 = SON1()
print (s1.bike) # Himalayan
print (s1.cycle) # Chandrababu cycle
print (s1.car) # Nano Car
print (SON2.mro()) # SON2 --> Father --> object class.
S2 = SON2()
print (s2.bike) # Duke
print (s2.cycle) # ChandraBabu Cycle
print (s2.car) # Nano Car
```

Multi level Inheritance :

It is the process of Inheriting properties of one parent class to one child class and again that Inherited properties getting inheriting to another child class.

Syntax :

```
Class Parent1 :
    pass
Class Child1 (Parent1) :
    pass
Class Child2 (Child1) :
    pass.
```

```
Class Bank_V1 :
    bank_name = 'SBI'
    bank_roi = 7
    bank_branch = 'Hyderabad'
    def __init__ (self, n, a, ac, bal) :
        self.name = n
        self.age = a
        self.account = ac
        self.balance = bal
    def customer_details (self) :
        print (f'name of customer is {self.name}')
        print (f'age of customer is {self.age}')
```

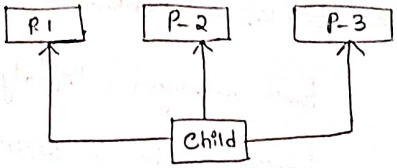
```
        print(f'account of customer is {self.account}')
        print(f'balance of customer is {self.balance}')

Class Bank_V2(Bank_V1):
    bank_branch = 'Bangalore'
    bank_ifsc = 12341
    def __init__(self, n, a, ac, bal, pin):
        super().__init__(n, a, ac, bal)
        self.pin = pin

Class Bank_V3(Bank_V2):
    def __init__(self, n, a, ac, bal, pin, ad):
        super().__init__(n, a, ac, bal, pin)
        self.adhar = ad
    def customer_details(self):
        super().customer_details()
        print(f'adhar of customer is {self.adhar}')

seetha = Bank_V3('SD', 23, 1234567, 9876543, 12341, 9432456)
seetha.customer_details()
```

Multiple Inheritance Example :




```
Class Bank_V1:
    bank_name = 'SBI'
    bank_branch = 'Hyderabad'
    bank_roi = 7
    def __init__(self, n, a, ac, b):
        self.name = n
        self.age = a
        self.account = ac
        self.balance = b
    def customer_details(self):
        print(f'name of customer {self.name}')
        print(f'age of customer {self.age}')
        print(f'customer account {self.account}')
        print(f'customer balance {self.balance}')
    @classmethod
    def bank_details(cls):
        print(f'bank name {cls.bank_name}')
        print(f'bank Rate of Interest {cls.bank_roi}')
        print(f'bank branch {cls.bank_branch}')
```

```
Class Bank_v2 (Bank_v1):
    bank_manager = 'Satya'
    bank_number = 1231156789
    def __init__(self, n, a, ac, b, pin):
        super().__init__(n, a, ac, b)
        self.pin = pin
    @classmethod
    def bank_details(cls):
        super().bank_details()
        print(f'bank manager is {cls.bank_manager}')
        print(f'bank number is {cls.bank_number}')

Class Bank_v3(Bank_v2, Bank_v1):
    bank_branch = 'Vizag'
    bank_ifsc = 12345
    def __init__(self, n, a, ac, b, pin, ad):
        super().__init__(n, a, ac, b, pin)
        self.adhar = ad
    def bank_details(cls):
        super().bank_details()
        print(f'bank ifsc is {cls.bank_ifsc}')
    def customer_details(self):
        super().customer_details()
        print(f'Customer adhar is {self.adhar}')

Seetha = Bank_v3('Seethadevi', 23, 23415, 12341, 23456, 6308)
Seetha.customer_details()
Seetha.bank_details()
```

o/p :
```
Name of customer is Seethadevi
age of customer is 23
account of customer is 23415
balance of customer is 12341
Aadhar of customer is 6308
Branch Manager is Satya
Branch Number is 1231156789
Bank ifsc is 123415
```
→ bank name is SBI
bank rate of interest 7
bank branch vizag

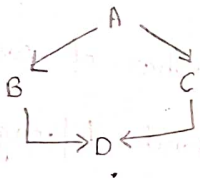
Hybrid Inheritance: It is a process of using more than one type of heritance

```
Class A:
    pass
Class B:
    pass
Class C(A):
    pass
Class D(B):
    pass
Class E(C,D):
    pass
```

```
print ( E. mro( )) # E --> C --> A --> D --> B --> object
```

Ex: 2 If parent classes are inherited from same parent then first preference will be given to immediate parent classes in case of MRO

```
Class A :
    pass
Class B(A):
    pass
Class C(A):
    pass
Class D(B, C):
    pass
print ( D. mro( )) # D --> B --> C --> A --> object
```



Special Methods / : Dunder Methods / Magic Methods Python :

- Python Implicitly will give a call to Magic method based on their functionalities
- Syntax for defining magic Methods

```
def __magicmethodname__(self):
    Statements
```

- __init__ , __del__ , __str__ etc are Examples of Magic methods

__init__

- It is Constructor which is used for Constructing an object
- __init__ called implicitly whenever object is Created

__str__

- __str__ called implicitly whenever an object is printed.
- __str__ will give only string type data as an output
- It will get called only when we print object but not when we perform operations on objects.

```
Class Book :
    def __init__(self, n, p, a):
        print ('__init__ is called')
        Self.name = n
        Self.price = p
        Self.author = a
    def __str__(self):
        return self.name → (display single o/p) # python
        return self.name + ' ' + self.author
                                    ( Concatenation to
                                    display both )
                                    # python Van Guido Rossum
python = Book ('PYTHON', 12345, 'Van Guido Rossum')
Django = Book ('Django', 87654, 'ABD')

print (python)
```

__del__ : It is a destructor which is used for deleting objects

__del__ is called implicitly whenever an object is created.

Example :

```
Class Book:
    def __init__(self, n, p, a):
        self.name = n
        self.price = p
        self.author = a
    def __del__(self):
        print('__del__ is called')

python = Book('Python', 12345, 'Van Guido Rossum')
Django = Book('Django', 87654, 'ABD')

del python # __del__ is called.
```

```
Class Employee:
    Company_name = 'JSpiders'
    Company_ceo = 'Satya'
    Emp_count = 0
    def __init__(self, n, i, s, e):
        self.Ename = n
        self.Eid = i
        self.Esalary = s
        self.Eexperience = e
        Employee.emp_count += 1
    def __del__(self):
        print(self, 'is deleted')
        Employee.emp_count -= 1
    def __str__(self):
        return self.Ename

Santhosh = Employee('Santhosh', 12344, 98765, 3)
Koti = Employee('Koti', 3456, 798765, 4)
print(Employee.emp_count)   2
print(Santhosh.emp_count)   2
print(Koti.emp_count)       2
del Koti # Koti deleted [__del__ is called]
print(Employee.emp_count)   1
print(Santhosh.emp_count)   1
```

* modify Generic property directly with ^ className in constructor

Hierarchical Inheritance

```
Class Mobile :

    mobile_sim = 'airtel'
    mobile_IMEI = 123456
    mobile_battery = 4500

    def __init__(self, n, c, num):
        self.name = n
        self.color = c
        self.number = num

    @classmethod
    def features(cls):
        print('Dial is available')
        print('Receiving calls')
        print('Sending and receiving messages')

    def look(self):
        print(f'mobile name is {self.name}')
        print(f'mobile color is {self.color}')
        print(f'mobile number is {self.number}')

Class Samsung (Mobile):

    def __init__(self, n, c, num, ds, cam):
        super().__init__(n, c, num)
        self.dualsim = ds
        self.camera = cam

    def samsung_look(self):
        super().look()
        print(f'mobile dualsim {self.dualsim}')
        print(f'mobile camera {self.camera}')

Class Iphone (Mobile):

    def __init__(self, n, c, num, wp, siri):
        super().__init__(n, c, num)
        self.waterproof = wp
        self.siri = siri

    def iphone_look(self):
        super().look()
        print(f'mobile is waterproof {self.waterproof}')
        print(f'mobile is having siri {self.siri}')

object1 = Iphone('Apple','black', 9848022338, 'yes', 'siri')
object2 = Samsung('Samsung','black', 9848022338, 'yes', 'yes')
object2.features()
object2.look()
object2.samsung_look()
```

o/p:
```
Dial is Available
Receiving Calls
Sending and Receiving messages
mobile name is Samsung
mobile color is black
mobile number is 9848022338
mobile dualsim yes
mobile camera yes.
```

Operator Overloading :

1. It is the process of changing the functionalities of operators when we use with user defined objects
2. for Every operator we have specific method.

Table with Operators and their respective magic method.

Operator	Magic Methods
+	__add__ (self, second)
-	__sub__ (self, second)
*	__mul__ (self, intValue)
/	__truediv__ (self, intvalue)
//	__floordiv__ (self, intvalue)
**	__pow__ (self, intvalue)
>	__gt__ (self, intvalue) (second)
<	__lt__ (self, intvalue)
>=	__ge__ (self, intvalue arguments)
<=	__lfe__ (self, intvalue)

Examples :

```
Class Book :

    def __init__(self, n, p, a):
        self.name = n
        self.price = p
        self.author = a

    def __str__(self):
        return self.name

    def __add__(self, second):
        return self.price + second.price

    def __sub__(self, second):
        return self.price - second.price

    def __mul__(self, intvalue):
        return self.price * intvalue

    def __truediv__(self, intvalue):
        return self.price / intvalue

python = Book('python', 12345, 'Van Guido Rossum')
Django = Book('Django', 87654, 'ABD')
print(python + Django) # python.__add__(django)
print(python - Django) # python.__sub__(django)
print(python * 3)
print(django / 2)
```

=>
```
Class Rectangle :

    def __init__ ( self , l , b ) :
        self.length = l
        self.breadth = b

    def __gt__ (self , second ) :
        a1 = self.length * self.breadth
        a2 = self.length * self.breadth
        return True if a1 > a2 else False

R1 = Rectangle (20,10) # 200
R2 = Rectange (30,10) # 300
print ( R1 > R2 )
```

O/p : False

→ ternary

31/March

Encapsulation :

- Encapsulation is a process of binding the properties and behaviours of an object under one roof.
- Encapsulation can be achieved by Creating Class
- Main Advantage of Encapsulation is restricting the scope of Accessibility of states and behaviours
- We can implement restrictions using Access Specifiers (Modifiers)

Access Specifiers :

Access Specifiers are used for creating the Scope of accessibility of Variables / Methods.

3 Types of Access Specifiers :

Public
Protected
Private

→ Public Access Specifiers :

- Syntax for representing public Variables / Methods

  Variablename
  Methodname

- Public Variables or methods can be accessed through different packages
- Syntax for accessing public Access Specifiers

  Objectname. Variable / methodname

→ Protected Access Specifiers :

- Protected access Specifiers are declared with Underscore before their names
- Syntax for representing protected Variables (or) Methods

  _variablename
  _methodname

- Protected Access Specifiers can accessed only with in that package
- Syntax for accessing Protected Access Specifiers

  Objectname._Variable / Methodname

→ Private Access Specifiers :

- Private Access Specifiers are declared with double underscore before their names
- Syntax for representing private Variables (or) Methods

  __variablename
  __methodname

- Private Access Specifiers can be accessed only with in that class
- Syntax for accessing private variables outside the class.

  Objectname._classname__Variable/methodname

```
Class Ipl :
    Semis = 'This time Our game'
    -finals = 'One More step away to Trophy'
    --trophy = 'finally we made it'
    def heart(self):
        print('RCB has won heart but not Trophy')
    def _display(self):
        print('Protected display')
    def __happiness(self):
        print('No more heart winning only Trophy')

RCB = IPL()
print(RCB.semis)
RCB.heart()
print(RCB._finals)
RCB.__display()
print(RCB._Ipl__trophy)
RCB._Ipl__happiness()
```

Ipl

Keys	Values
Semis	'This time Our Game'
finals	'One more step away
_Ipl__trophy	'finally we made it'
heart	
-display	
_Ipl__happiness	

Abstraction :

- Abstraction is used to hide internal functionality of method from users
- The users only interact with basic implementation of method, but inner working is hidden
- User is familiar with 'what function does' but they don't know 'how it does'
- python doesn't provide abstract class itself. we need to import the abc module, which provides the base for defining Abstract Base classes (ABC)
- Abstract class is a class with one (or) more abstract methods (abstract class is a prototype class)
- Abstract method is a method with no implementation
- We cannot create object for Abstract class,

Main purpose of it is to define how other classes should look like i.e., what methods and properties they are Expected to have.

Syntax : Defining Abstract Class

```
from abc import ABC, abstractmethod (importing)
        ↑ module
class AbstractclassName (ABC):
                        ↑ Abstract Base Class
    def abstractmethodname (self):
        pass                          → Non mandatory abstract method

    @abstractmethod
    def abstractmethodname (self):
        pass                          → Mandatory Abstract method.
```

Example :

```
from abc import ABC, abstractmethod

class TimeTable (ABC):

    @abstractmethod
    def breakfast (self):
        pass
    @ abstractmethod
    def lunch (self):
        pass
    @ abstractmethod
    def dinner (self):
        pass
Class Harshad (TimeTable):
    def breakfast (self):
        Print ('Dosa')
```

```
    def lunch(self):
        print('Biryani')
    def dinner(self):
        print('Biryani')

H1 = Harshad()
H1.breakfast()  # Dosa
H.lunch()       # Biryani
H.dinner()      # Biryani   //
```

(2)

```
from abc import ABC, abstractmethod
class Animal(ABC):
    @abstractmethod
    def move(self):
        pass
    @abstractmethod
    def speak(self):
        pass
class Dog(Animal):
    def move(self):
        print('I can walk')
    def Speak(self):
        print('Bow Bow')
class Snake(Animal):
    def move(self):
        print('I can crawl')
    def speak(self):
        print('Buss Buss')

D = Dog()
D.speak()
```

Decoration of functions :

→ Function Decorators : without changing functionality of our divide function — function we define a smart function

```
def smart_divide(arg):        # arg ↑ address
    def swap(a,b):
        if a<b:
            a,b = b,a
        arg(a,b)  # divide function gets a call
    return swap
@smart_divide  # divide = smart_divide(divide)  # divide = swap
def divide(a,b):
    print(a/b)

divide(10,2)  # swap(10,2)
divide(5,15)  # swap(5,15)
```

based on our requirement

J, divide at last it will hold swap function address

→ Single Tone Class : The class which we can Create only one object → SingleTone Class functionality is Used in all the booking applications

- We can achieve single Tone functionality in python by Using decorators.

Example : def singletone(func): # func = Multiplex class address

decorator function

```
    d = { }
    def inner( ):
        if func not in d:
            d[func] = func( )
        return d[func]
    return inner [ inner function address ]
```

To check whether multiplex class is present (or) not if not it will Create and store in dictionary as a key value pair

Multiplexclass = Multiplex class object address

return Multiplex class object

@ singletone # Multiplex = singletone(Multiplex)

decorated class

```
Class Multiplex :
    def __init__(self):
        self.tickets = 300
    def booking(self, n):
        if self.tickets >= n:
            self.tickets -= n
            print('tickets are booked')
        Else:
            print('sold out')
```

```
Harshad = Multiplex( )
print(Harshad)
Harshad.booking(15)
print(Harshad.tickets)
Santhosh = Multiplex( )
print(Santhosh)
Santhosh.Booking(100)
print(Santhosh.tickets)
```

Example :

```
def time_Calculator(func):        → fibo function Address
    def inner(n, a, b):
        import time [To import time from time module]
        ST = time.time( ) [Starting Time which indicates Current Time]
        func(n, a, b) (fibo function Call)
        ET = time.time( ) [Ending Time
        print(ET - ST)
    return inner # inner function Address
```

@ time_ Calculator : Variable fibo = time_calculator (fibo) # fibo = Inner function Address.
function Address

```
def fibo (n, a, b) :
    if n == 1 :
        print (a)  for 1 Element
    Elif n == 2 :
        print (a, b) for 2 Elements
    Else :
        print (a, b, end = ' ')
        for i in range (n-2) : as we know already 2 Elements
            c = a+b
            print (c, end = ' ')
            a, b = b, c
```

fibo (10 2 3) (Inner function will get a call)

Iterators : Iteration is a process of traversing through Sequence and Extracting a Value one after another

- The Operators which do the process of Iteration is Known as Iterators
- For loop works on Principle of Iterators
- Strings, List, Tuples, dictionaries set are all Iterable objects

[Iteration is process of Extracting Each and Every Element in a Sequence

→ on which → we perform Iterables (COT)

→ By which we perform Iterators (for loop)]

Iterators works with 2 sets of operation

1. Initialization (Creates and Returns Iterable Object)
2. Traversing (fetches one Value from Iterable object)

Initialization :

- Create Iterable Object
- Initialize one Value for Iteration
- Return Iterable Object for Traversing.

Traversing :

- Fetches Each and Every Element in given Sequence (or) Range
- If there is no Elements to fetch at last then it will raise stop Iteration Error.

Classification :

① Built in Iterators

② Custom / User defined Iterators

① Built in Iterators :

In case of Built in Iterators

- Iter Perform Initialisation → It creates and returns a Iterable Object

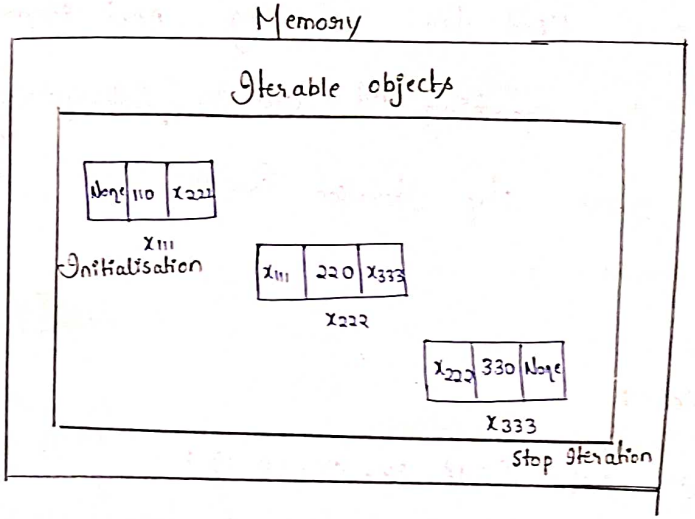
  Syntax : Varname = iter(COT)

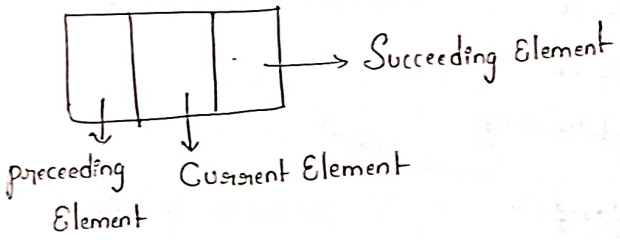
- Next function performs Traversing : fetches one Value from Iterable Object

  Syntax : next(Iterable Object)

Ex :

```
I0 = iter([110, 220, 330])
print(next(I0))  # 110
print(next(I0))  # 220
print(next(I0))  # 330
print(next(I0))  # Stop Iteration
```





Note : Till Now we have seen Built in Iterators we can Create Custom Iterators by Using Oops

Custom Iterator :

① We have to Create a class with __iter__() and __next__() methods to create custom Iterator

② __iter__() Used for initializing Temporary Variable with Starting Value and used for returning Iterable Object.

→ `__next__()` Used for returning next item in sequence. On reaching end and in subsequent calls, it will raise StopIteration Error

Example :

```
Class Series :

    def __init__(self, sv, ev, up=1) :
        self.sv = sv
        self.ev = ev
        self.updation = up

    def __iter__(self) :
        print('__iter__')
        self.i = self.sv
        return self

    def __next__(self) :
        print('__next__')
        if self.i < self.ev :
            dummy = self.i
            self.i += self.updation
            return dummy
        Else :
            raise StopIteration

S = Series(1, 5)

for a in s :
    print(a)
```

(1) Some value

for { Traversing returning }

[first Value for Traversing]

Iterable object

Initialise a Value

to traversing purpose for __next__ method

[dummy = i and increement i Value]

to (1) based on updation

If we didn't provide this it will keep on printing None

→ Print Cubes :

```
class Cubes :

    def __init__(self, sv, ev, up=1) :
        self.sv = sv
        self.ev = ev
        self.updation = up

    def __iter__(self) :
        self.i = self.sv
        return self

    def __next__(self) :
        if self.i < self.ev :
            dummy = self.i**3
            self.i += self.updation
            return dummy
        else :
            raise StopIteration

s = Series(1, 5)
for a in s :
    print(a)
```

Cubes number

Fibo Series

```
Class FiboSeries( ):

    def __init__(self, FE, SE, N):
        self.FE = FE      First Element after traversing
        self.SE = SE
        self.N = N        Number of Elements we need in Series

    def __iter__(self):
        self.i = 1 [i Value for traversing]
        return self
            ↳ return Iterable Object

    def __next__(self):
        if self.i <= self.N:
            dummy = self.FE
            self.FE = self.SE
            self.SE = self.FE + dummy
            self.i += 1
            return dummy
        raise StopIteration

FS = FiboSeries(2, 3, 10)
for a in FS:
    print(a)
```

```
Class Fibo:
```

after Traversing in Series it is first Element

Disadvantages :

- We have to Create a class with __iter__() and __next__() to Create a Custom Iterator and track all the states and we have to raise StopIteration Error if no Values are there to return.

Definition of Generators :

- Python Generators are simple way of creating iterators
- All work we mentioned above are automatically handled by generators in Python
- Generator is a function with yield statement instead of return
- python Generator is a function which is used for returning an Iterable generator object which can be Iterated to Extract Values one by one
- Generators follows pause and Execute approach.

Syntax :

```
def generatorFunction( ):
    ...
    yield Something
    ...
    yield Something
```

Example :

① defined
```
def hello( ):
    print('hello started')
    yield ['returned by first yield of hello', 90]
    print('execution hello after first yield')
    yield 20
```
③ after function call it will create an Iterable object and store in he

② function call
```
he = hello( )  [he holding Generator Iterable object]
print(he)  (prints Generator Object) ④
for i in he :  [how many times yield statement is there over for loop will Execute that many times]
    print(i)
print('*' * 20) //
```

Ex : print Series using Generator.

```
def Series(s, e, u=1):
    while s<e :
        yield s
        s+=u

S = Series(16, 34, 2)
for i in S :
    print(i)
```

② Print fibo using Generators

```
def fibo(FE, SE, N):          (10)
    count = 1
    while count<=N:
        dummy = F.E
        F.E = S.E
        S.E = F.E + dummy
        yield dummy
        count+=1

f = fibo(2, 3, 10)
for a in F :
    print(a)
```

→

```
def fibo(a, b, n):
    dummy = 1
    while dummy<=n :
        yield a
        a, b = b, a+b
        dummy+=1

fb = fibo(2, 3, 5)
for i in fb :
    print(i)
```

o/p :
```
2
3
5
8
13
```

Differences b/w Return and Yield.

Return	Yield
Used for returning Some Values / data	Same
output format tuple	Same
we can return any type of data	Same
we can return any no. of data	Same
after writting a return Statement we cannot write any other statements inside that block	After writing yield Statement inside a generator function we can write N no. of statements inside that block
directly Values will be returned	It will return an Generator Object.

Difference b/w Generators / Iterators

Generators	Iterators
• In Creating a python Generator we use a function.	In Creating an Iterator in python, we use iter( ) and next( ) functions
• A Generator in python makes use of 'yield' keyword	A python Iterator doesn't use yield statement
• python generator saves states of local variables every time 'yield' pauses loop in python	• An Iterator does not make use of local variables, all it needs is iterable to iterate on
• a generator does not need a class in python	• You can Create Custom Iterator using a python class
• Generators let us write fast and Compact code	• Iterators are slow Compared to Generators
• Less Memory Efficient Compared to Iterators	• More Memory-Efficient.

Generators	Functions
• Functions with Yield Statement	• Functions contain Return Statement
• When Called, returns an object	• Function returns a Value based on user definition
• does not start Execution Immediately	• Will be Executed Immediately
• Once function Yields, the functions is paused & control is transferred to caller.	• No concept of yielding and pausing Execution.
• When function terminates, StopIteration is raised automatically on further Calls	• Does not raise any Exception after termination of function.
• We can Use Generators with for loops directly.	• We Cannot use functions with for loops directly

File Handling : File handling is the process of performing Operations on file by using python Language.

Process of file Handling :

- Open file
- Perform the operations (writing, adding, reading)
- Close the file

Opening the file : By Using Open function we can open a file for performing operations.

Syntax : open(path, mode)

Note : Open function creates a file object on which we can perform operations

Path : path is used for Specifying location of file on which we have to perform the operations

1. If python and txt file are in same folder path will be directly file Name
2. If python and txt file are in different folders. path ⇒ Entire Path of file Name.

Mode : Used for Specifying Type of operation we need to perform on file.

Mode	Symbol
Read	'r'
write	'w'
Append	'a'

Write : we open a file in write mode when we have to write new data into files

Once we open a file in write mode then

1. If file is not present then it will create a new file
2. If file is already present it will Erase the Existing data and allows us to write new data

Methods to write data into files :

File objects is having 2 methods by using which we can write data into files

1. Write
2. Writelines

Write method :

1. It is used for writing only string type of data into files

Syntax : write(data)

Note : data must be only strings.

Ex : fo = open('data.txt','w')
(fo ← file object)

s = 'hai this is write mode'

fo.write(s)
(data →)

fo.close()
↳ close method to close and save the file

Writelines method :

1. Used for writing any type of data into files

Syntax :

writelines(data)

Note : Any Collection of data but it must have only strings as Elements

Example :

```
fo = open('data.txt','w')
L = ['hai\n','hello\n','bye\n','99\n']
fo.writelines(L)
fo.close()
```

close() method : To close and save the file.

Note :

It is Mandatory to close the file after writing some data into files, after closing the file only data will get saved//

-Append : We open a file in append mode when we have to add new data into files along with Existing data

- Once we open a file in append mode then.
- If file is not present then it will create a new file
- If file is present it will allows u to write new data after Existing data

Note : we can use write and writelines method to write the data into files Even in Case of append mode.

Ex :

Read : We open a file in read mode when we have to read data from files

- Once we open a file in read mode then
- If file is not present then it will throw an Error
- If file is present then it will allow you to read data from files in order to read the data from files we have 3 methods

1. Read
2. Readline
3. Readlines

Read : It is used for reading Entire data from file and represent the data in string

Syntax : Read( )

Example :

```
f = open('new.txt','r')
data = f.read( )
print(type(data))
print (data)
print (f.tell( ))
f.seek (20)
print (f.tell( ))
data1 = f.read( )
print (type (data1))
print (data1)
```

Readline : It is used for reading a single line of data (or) Specified number of characters from files and represent the data in string

R=Y

Syntax : Readline ([No. of characters])

Example :

```
f = open.txt ('new.txt','r')
data = f.readline (5)
print(type(data))
print (data)
data1 = f.readline( )
print(type(data1))
print( data1)
f.seek (50)
data1 = f.readline(10)
print (type (data1))
print (data1)
```

Readlines : It is used for reading a Entire data from file and represent the data in List.

Syntax : readlines( )

Example :

```
fo = open ('appenddata.txt','r')
data = fo.readlines( )
print (data)
print (type (data))
```

Tell( ) : Tell method is used for specifying then Current position of the cursor

Seek( ) : Taking Cursor to Specified Index position

Syntax : seek (index_position)

#

wap to print how many lines of data present in given file ?

```
f = open ('madhu.txt','r')
data = f.readlines( )
s = len(data)
print (s)
f.close( )
```

o/p : 6

```
print (len(data))
```

How many words of data present in given file

```
f = open('madhu.txt','r')
data = 'hai guys Good morning how are you'
s = data.split( )
l = len(s)
print (l)
f.close( )
```

o/p : 7

How many characters of data present in given file

```
f = open ('madhu.txt','r')
d = 'hai guys Good morning how are you'
c = 0
for i in d :
    if i!=' ' :
        c+=1
print (c)
```

o/p : 27

Important methods / attributes of file handling

Name : Name of the opened file

Mode : Mode in which we have opened file ('w', 'r', 'a')

Readable( ) : Returns True if file opened in read mode otherwise False

writable( ) : Returns True if file opened in write (or) append mode otherwise False

Closed : Returns True if file is closed otherwise False

Ex :

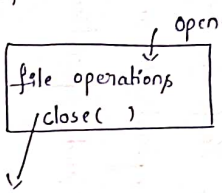
```
f = open('hai.txt', 'a')   opened
print(f.name)  # hai.txt
print(f.mode)  # append mode
print(f.readable( ))  # False
print(f.writable( ))  # True
f.close( )   closed
print(f.closed)  # True
```

Opening a file by using with Keyword :

① By Using with Keyword we can open a file in form of Block

② If we open a file by using with Keyword by default file will be closed before control is coming out of the block.

Syntax : with open(path, mode) as AliasingName :



Ex :

```
with open('hai.txt','a') as file :
    file.write('file is opened by using with keyword')
print(file.closed)  # False
# print(file.closed) # True.
```

w+ mode : In this mode we can

w+ mode	a+ mode	r+ mode
• In this mode the Cursor will be at starting Position • We Can perform both write and read Operations • We ~~Can't~~ shouldn't perform read then write operation as there will be no content ~~will be~~ is available	• In this mode the Cursor will be at Ending position (i.e, as the data is already Existed.) • We can Perform both read and then write operations and write and then read operation. • New data will be added at the End • To read data in a+ mode seek method is mandatory	• In this mode the Cursor will be at starting position Even though the data is present as it is in r+ mode • We can Perform both read and write operations and write and then read operations. • If you want to write data after read in r+ mode at particular position the data will be overridden [in Case of write] • After reading the data if you want to write it will be added at last

Json

Java Script Object Notation. [Passing Techniques]

- It is Generalised Text format
- Json data will be stored in form of Key Value Pairs
- We can Easily transfer json data from one source to another source
- inorder to deal with json data python has provided json module

Writing Json data into files

we can write json data into files by using 2 functions

dump

dumps

1. Dumps : Dumps function it can just convert python objects to Json object.

- If you want to write that data we can write by using write method.

Syntax : dumps (python objects_u_need_to_convert)

Ex : with open ('jsondata2.txt','w') as file :

```
    import json
    po = {'name':'lokesh','age':20,'mobile':(987,567),
          'male': True, 'female': False, 'GF':None}
```

```
print (po)
jo = json.dumps(po)
print (jo)
file.write(jo)
```

dump : dump function is capable of converting python data into json data. and it can write converted data into files as well

Syntax : dump(arg1, arg2)

arg1 = data which you need to convert

arg2 = file object into which you need to write converted data

```
with open('jsondata3.txt', 'w') as file :
    import json
    po = {'name': 'Satya', 'age': 20, 'mobile': (12,34)
          'male': True, 'female': False, 'GF': None}
    json.dump(po, file)
```

Reading of Json data from files

We can read json data by using 2 methods

1. loads
2. load

1. loads : loads function is capable just to converting read json data into python objects

- we have to read
- after reading loads functions will convert

Syntax : loads (jsondata)

Ex :-
```
with open('jd.txt', 'r') as jsf :
    import json (import json module)
    Jso = jsf.read() (reading file)
    print (Jso) (print java Script Object)
    po = json.loads(Jso) (Reconverting Jso to po)
    print (po) (print python Object)
```

2. Load : Capable of reading and converting the json data into normal data

Syntax : load (arg1)

arg1 = file object from which u need to read

Example :
```
with open('jd2.txt', 'r') as jsf :
    import json
    po = json.load(jsf)
    print (po)
```

Pickling Process (or) Encryption Process :

- It is the process of Converting user defined data into binary format of data

Unpickling (or) Decryption Process :

- It is the process of re-converting the binary data into User defined data

- We can achieve this pickling and unpickling by Using pickle module

diff b/w json and pickle

	json data	binary data
file Extension	.txt, .json	.pkl
module	json	pickle
file mode	r, w, a	rb, wb, ab

Writing binary data into files

By Using dumps and dump function of pickle module we can write binary data into files

writing binary data by using dumps

```
with open('pickling1.pkl','wb') as f:
    D = {'name':'Ashu','age':2}
    import pickle
    BO = pickle.dumps(D)
    f.write(BO)
```

Binary data by Using dump

```
with open('pickling1.pkl','wb') as f:
    D = {'name':'Ashu','age':2}
    import pickle
    BO = pickle.dump(D,f)
```

Reading of binary data from files by using load of pickle module.

```
with open('pickling1.pkl','rb') as f:
    import pickle
    po = pickle.load(f)
    print(Po)    //
```

Comprehensions

1. Comprehensions in python are used for Creating new Sequences from iterables / another Sequence
2. Comprehensions are single-line statements
3. Based on Expressions if u want to define sequences then we go for Comprehensions.

Classification of Comprehensions

In python we have 3 types of Comprehensions

1. List
2. Set
3. Dictionary

List Comprehension syntax with one Value and one for loop

VariableName = [Value for Loop]

Ex: adding numbers into list by using Normal Approach

```
L = [ ]
for i in range(1, 11):
    L.append(i)

print(L)
```

adding numbers into list by using List Comprehension Approach.

```
L = [i for i in range(1, 11)]
```

adding Squares of numbers into List by Using List Comprehension Approach.

```
L = [i**2 for i in range(1, 11)]
```

List Comprehension syntax with one Value and one for loop and one if condition.

```
VarName = [Value for Loop if condition]
```

Example:

adding Even numbers into List by Using Normal Approach.

```
L = [ ]
for i in range(1, 11):
    if i%2==0:
        L.append(i)

print(L)
```

adding Even numbers into List by using List Co

```
L = [ j for j in range(1, 11) if j%2 ==0]
```

Ex

```
T = (11, 22, 33, 100, 200, 20)
L = [i for i in T if i%2==0]
L
[22, 100, 200, 20]   //
```

② `L = [ i for i in T if i%2==0 and i>100]`

```
>> L
[200]
```

Ex: for 2 if conditions (multiple)

List Comprehension Syntax with one Value and one for loop and multiple if condition.

Varname = [Value for loop if cond1 and/or cond2]

from given Tuple add only odd numbers which r greater than 50 into List

T = (11, 78, 34, 99, 23, 67, 90, 70)

L = [i for i in T if i%2==1 and i>50]

print(L) //

⇒ Add all Even numbers and odd numbers > 30

X L = [i for i in T if i%2==0 or (if i%2==1 ~~and~~ or i>30)] X

Not needed

X print(L) X //

L = [i for i in T if i%2==0 or i>30]

print(L)

List Comprehension Syntax with one Value for loop if condition Else

Varname = [Value if condition Else for loop]

data = [i if i>0 Else 0 for i in [1, 2, 3, -88, 100, -5

print(data)

Value, multiple for loops and if condition

Syntax : [Value for loop1 for loop2 ... for loop n if condition]

Ex: data = [[i, j, k] for i in range(1,10) for j in range(1,10)

for k in range(1,10) if i==j==k]

print(data)

Normal Approaches.

Ex: L=[ ] [1st way]

for i in range(1,11):

L.append([i] * 3) //

② [2nd way] → Nested for loops

L=[ ]

for i in range(1,11):

for j in range(1,11):

for k in range(1,11):

if i==j==k:

L.append([i, j, k]) //

Set Comprehension : Syntax for 1 Value one for loop

VariableName = { Value for loop } { Syntax with one Value and one for loop

Ex: L = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

S = { i * 3 for i in L }

print(S)

o/p : {3, 6, 9, 12, 15, 18, 21, 24, 27, 30}

Syntax : Set Comprehension Syntax with one Value and a for loop and if Condition

VarName = { Value for loop if Condition }

Ex : L = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

S = { i * 3 for i in L if i%2 == 0 }

print(S)

o/p : { 6, 12, 18, 24, 30 } //

Syntax for Set Comprehension with one Value and one for loop and multiple if Condition.

VarName = { Value for loop if Cond1 and/or Cond2 }

Ex : t = (11, 56, 67, 39, 78, 20, 50)

S = { i for i in t if i%2 == 1 and i > 50 }

print(S)

o/p : {67}

* Add all Even numbers and odd numbers > 30

t = (11, 56, 67, 39, 78, 20, 30, 90, 84)

(or) S = { i for i in t if i%2 == 0 or i > 30 }

print(S)

o/p : { 67, 39, 78, 20, 50, 84, 56, 90 } //

(and) S = { i for i in t if i%2 == 0 and i > 30 }

print(S)

o/p : { 78, 30, 50, 84, 56, 90 } //

Set Comprehension Syntax with one Value one for loop and if Else Condition.

VarName = { Value if Condition Else for loop }

Ex : t = (1, 2, 3, -88, 100, -99)

S = { 1 if i > 0 else 0 for i in t }

print(S)

o/p : { 0, 1 }

Set Comprehension Multiple for loops and if Condition

Syntax : VarName = { Value for loop1 for loop2 .. for loopN if Condition }

Ex :

```
S = {(i,j,k) for i in range(1,10) for j in range
    (1,10) for k in range(1,10) if i==j==k}

print(S)

o/p : {(6,6,6),(5,5,5),(9,9,9),(2,2,2)
       (7,7,7),(3,3,3),(4,4,4),(8,8,8)
       (1,1,1)}
```

//

Dictionary Comprehension :

It is a single line statement by using which we can create dictionary from given sequence

```
T = [1,3,4,6,5]                          T = [1,3,4,6,5]

o/p1 : d = {i:i**2 for i in T}           o/p1: {1:1, 3:9, 4:16, 6:36, 5:25}

o/p2 : d = {i:i**2 for i in T if i%2==1}
           i*i                           o/p2 : {1:1, 3:9, 5:25}

o/p3 : d = {i:T[i] for i in range(len(T))}   (i -> index positions, T[i] -> elements)
       o/p: {0:1, 1:3, 2:4, 3:6, 4:5}
```

} one approach

Enumerate function : Used for returning the Enumerate object of index positions their respective values of specified collection data type

```
Ex :- d = {k:v for k,v in enumerate([11,22,33,44])}

>> d
{0:11, 1:22, 2:33, 3:44}

T = [1,3,4,6,5]
     0 1 2 3 4

o/p : {'A':1, 'B':3, 'C':4, 'D':6, 'E':5}

S = 'ABCDE'
     0 1 2 3 4

d = {S[i]:L[i] for i in len(T)}
```

↳ Comparing Index Positions //

Zip function : Used for Combining Elements of given Collection Data Types Based on their respective index positions

Ex : `zip('ABCDE', [1, 3, 4, 6, 5])`

// It will create an Zip Object //

```
list(zip('ABCDE', [1, 3, 4, 6, 5]))
[('A', 1), ('B', 3), ('C', 4), ('D', 6), ('E', 5)]

d = {k:v for k,v in zip('ABCDE', [1,3,4,6,5])}

» d
{'A': 1, 'B': 3, 'C': 4, 'D': 6, 'E': 5}
```

I/P `S = 'aBcd'`

O/P `{'A': 'aaa', 'B': 'bbb', 'C': 'ccc', 'D': 'ddd'}`

⇒ `d = {i.upper(): i.lower()*3 for i in S}` //

Keys → Upper

Values → lower

SQLite is a C library that provides a light-weight disk-based database that doesn't require a separate server for processing.

* SQL Connections *

Create Connection Object :

① To use sq SQLite3 in Python, first of all, you have to import sqlite3 module

② Create a Connection Object which will connect us to the database and will let us Execute the sql statements

③ We can Create Connection Object by using connect class of sqlite3

Syntax for Creating Connection Object

```
sqlite3.connect('DataBasename.db')
```

SQLite3 Cursor Object :

① To Execute SQLite Statements in python, you need a Cursor object

② The SQLite3 Cursor is method of Connection object

```
Con = sqlite3.connect('mydatabase.db')
CursorObj = Con.cursor()
```

→ After Creating Cursor Object we have to utilize Execute to perform CRUD operations

→ The Commit ( ) method save all the changes we make

```
def Create_co( ):
    import sqlite3
    co = sqlite3.connect('harshad.db')
    return co
co = Create_co( )

def Create_table(co):
    curobject = co.cursor( )
    curobject.execute('Create table Employee(id integer
                       primary key, name text)')
    co.commit( )
    print('table is Created')
# Create_table(co)  1 time

def insert_data(co):
    curobject = co.cursor( )
    curobject.execute('insert into Employee(id, name)
                       values(4,"sql")')
    co.commit( )
    print('data is inserted')
# insert_data(co)

def retrieve_data(co):
    curobject = co.cursor( )
    queryset = curobject.execute('Select * from Employee')
    for i in queryset:
        print(i)
# retrieve_data(co)

def update_data(co):
    curobject = co.cursor( )
    curobject.execute("update Employee set name = "Ashu"
                       where id = 3')
    co.commit( )
    print('update is done')
# update_data(co)
retrieve_data(co)

def delete_data(co):
    curobject = co.cursor( )
    curobject.execute('delete from Employee where id=4')
    co.commit( )
    print('delete is done')
delete_data(co)
retrieve_data(co).
```

Multi Threading.

Threading : On Computing, a process is an instance of a Computer Program that is being Executed

Any Process has 3 Basic Components

- An Executable Program
- The Associated data needed by program (Variables, work space, buffers, etc.
- The Execution context of the program (State of process)

Thread : A thread is an Entity within a process that can be scheduled for Execution

(or)

A thread is a Sequence of instructions within a program that can be Executed independently of other code

Thread Control Block (TCB) will control all thread Operations.

By default Each and Every Programming language work on Uni-Threading principle.

If you want to do multi-threading then we can do it Externally

Necessity of Achieving Multi threading:

1. Whenever there is a wait time in program then at that time CPU will not take any other programs for Execution then due to idle time of CPU Efficiency of program reduces
2. We can reduce idleness of CPU by Using Multi Threading.

Multi Threading : Multi-threading is defined as ability of a processor to Execute multiple threads concurrently

Whenever first started thread is waiting then at that particular Time Second thread will be under Execution.

In a Simple single-core CPU it is achieved using frequent switching between threads. This is termed as Context switching. In Context switching the state of a thread is saved and state of another thread is loaded whenever any interuppt (due to I/O (or) manually set) takes place. Context switching takes place so frequently that all the threads appear to be running paralelly (this is termed as Multitasking

C

Classification of Threads

Threads are classified into two types

1. Kernel threads
2. User defined threads

1. Kernel Threads :

1) These are threads which are created by OS inorder to Execution purpose
2) Os will not allow to user create Kernel Threads

2. User_defined Threads :

1) These are threads which are created by USER inorder to Execution purpose
2) we can create thread by using threading module of python import threading
3) By Using thread class of threading module we can create user_defined Threads

Syntax for Creating User Created Threads :

ThreadVariablename = threading.Thread(target, args, name)

Target : the function to be Executed by thread

Args : the Arguments to be passed to target function.

name : name of created thread

Example :

```
t1 = threading.Thread(target = fun1, args = (5,),
                      name = 'thread1')
```

Some of functions of Threading module :

Methods of thread Class

1. Start( ) helps us to start Execution of Thread
2. Join( ) helps us to wait untill Termination of Thread Execution.
3. is_alive( ) The is_alive method checks whether a thread is still Executing.
4. name( ) helps you to print name of thread.

Functions of Threading module :

1. threading.active_count( ) : Returns number of thread Object that are active.
2. threading.Enumerate( ) : Returns a list of all thread object that are all active

Example :

```
import time
import threading
def gf(n):
    for i in range(5):
        print('I am Talking to GF')
        time.sleep(2)
```

```
def exgf(n):
    for i in range(5):
        print('ExGf is Speaking to me')
        time.sleep(2)

T1 = time.time()
# I am Creating gf Thread
gf = threading.Thread(target=gf, args=(5,), name='Priya')
# I am Creating exgf Thread
exgf = threading.Thread(target=exgf, args=(5,), name='Sweety')
# Starting Execution of gf Thread
gf.start()
# Starting Execution of exgf Thread
exgf.start()
print('no of active threads are,' threading.active_count())
print(threading.enumerate())
gf.join()
exgf.join()
print('gf is active or not', gf.is_alive())
T2 = time.time()
print('Time taken is', T2-T1) //
```

→ Regular Expressions ←

- Regular Expressions is a Sequence of characters that forms a search pattern
- Regular Expression can be used to check if a string contains specified search pattern inorder to achieve
- Regular Expression we have to import re module by below Syntax : import re

Examples of Match Search and findall (first occurrence)
↳ it will create a match object if the pattern is present in main string at starting position only

```
import re
s = 'hai python'
re.match('h', s)
< re.matchobject ; span=(0,1), match='h'>
re.match('i w{3}', s)
< re.mo ; span=(0, 3), match='hai'>
re.match('p', s)
re.search('h', s)
< re.MO ; span=(0, 1), match='h' >
re.search('p', s)
< re.MO ; span=(4, 5), match='p' >
re.search('w', s)
```

- Eventhough the pattern is present in string but not at starting position it will give none (as well as not present)

(None as it is not present at starting)

→ it will search for Entire string and create a match object (first occurrence)

— None (Nothing)

re.findall('h', s) → (List of matched contents throughout the string)

['h', 'h'] [two occurrence]

re.findall('p', s)

['p']

re.findall('pk', s) [Not present]

[] Empty List

. - Any Character (Except newline character /n)

^ \A Startswith (given pattern is starting with (or) not)

$ \z Endswith (given pattern is Ending with (or) not)

* Zero (or) More Occurences (must be mentioned with pattern (or) character)

+ One (or) More Occurences "

? 0 to one occurences "

{ } Exactly Specified number of occurences (or) "
User defined objects range

s = 'hai python'

re.search('^p', s)

re.search('p$', s)

s2 = 'hai help heep hp heeep hep heeeeep'

re.findall('he*p', s2)

['heep', 'hp', 'heep', 'hep', 'heeeeep']

re.findall('he+p', s2)

['heep', 'heeep', 'hep', 'heeeeep']

re.findall('he?p', s2)

['hp', 'hep']

re.findall('he{2}p', s2)

['heep']

re.findall('he{2,3}p', s2)

['heep', 'heeep']

re.findall('he{2,7}p', s2)

['heep', 'heeep', 'heeeeep']

```
S= 'hai heep hp hello'
re.findall('he*p|hai', s)
['hai', 'heep', 'hp']
re.findall('|w', s)
['h', 'a', 'i', 'h', 'e', 'e', 'p', 'h', 'p', 'h', 'e', 'l', 'l', 'o']
>> s1 = 'Hlai 12-@bye'
re.findall('|w', s1)
['H', 'a', 'i', 'l', '2', '_', 'b', 'y', 'e']
re.findall('|w{2}', s1)
['Ha', '12', 'by']
re.findall('|w{3}', s1)
['Hlai', '12-', 'bye']
re.findall('|w{4}', s1]
[]
re.findall('|W', s1)
[' ', '@']
re.findall('|d', s1)
['1', '2']
re.findall('\d{2}', s1)
```

```
import re
>> s = 'abcdef'
>> re.findall('abc', s)
['abc']
>> re.findall('[abc]', s)
['a', 'b', 'c']
>> re.findall('[efghijk|mnop]', s)
['e', 'f']
>> s1 = 'abfuqgjlop'
>> re.findall('[e-p]', s1)
['f', 'g', 'j', 'l', 'o', 'p']
>> re.findall('[a-k]', 'Abfuqgjlop')
['b', 'f', 'g', 'j']
>> re.findall('[a-zA-z]', 'Abfuqgjlop')
['A', 'b', 'f', 'u', 'q', 'g', 'j', 'l', 'o', 'p']
>> re.findall('[a-zA-z0-9]', 'Aj784gjK@FT')
['A', 'j', '7', '8', '4', 'g', 'j', 'K', 'F', 'T']
>> re.findall('[a-zA-z0-9]{2}', 'Aj784gjK@FT')
['Aj', '78', '4g', 'jK', 'FT']
```

```
re.findall('[a-z A-z 0-9]'{3}, 'Ajt84gjK@FT')

['Ajt', '84g']

>>re.findall('[4-9][0-9]', 'Ajt84gjK@FT89 1764')

['78', '89']

>> re.findall('[4-9][0-9]', 'Ajt84gjK@FT891764')

['78', '89', '76']

>> re.findall('[AjK]', 'Ajt84gjK@FT891764')

['A', 'j', 'j', 'K']

>> re.findall('[^Ajk]', 'Ajt84gjK@FT891764')

['t', '8', '4', 'g', '@', 'F', 'T', '8', '9',
 '1', 't', '6', '4']

>> re.findall('@',

['@']

>> re.findall('[@]', 'Ajt8+4gjK@FT891764')

['+']

>> re.findall('\+',

['+']
```

Note: [abc] → Either 'a' or 'b' or 'c'

[^abc] → Neither 'a' nor 'b' nor 'c'

Special functionality characters should not be given directly. it should be represented in [ ]

Valid Email Pattern.

① First character can be alpha, digits 1-9, - [a-z A-z 1-9]

② Between Fc to . we have 0 to n occurences [\w *]

③ . present for 0 to 1 time (?)

④ B/w . to @ we have 1 to n occurences (+)

⑤ @ we have

⑥ gmail

⑦ .

⑧ com

g/p : Mail

'[a-z A-z 1-9] \w * [.] ? \w + @gmail [.] com'

```
import re
S= input('Enter Email id')
re.findall('[a-z A-z 1-9] \w * [.] ? \w + @ gmail[.] com/in', S)
                                                  /hotmail
```

* Valid Indian Number

```
import re
s = input ('Enter number')
re.findall ('[+]91-[6-9]\d{9}', s)
                    ↓
                  [0-9]
```

( ) parenthesis: It is used for creating Groups

- It is used whenever we wants to check the Content which is matched by sub patterns

Ex:

```
s = '22.98'
>> re.match ('\d+[.]\d+', s)
<re.Match object; span=(0,5) match='22.98'>
>> MO = re.match ('(\d+)[.](\d+)', s)
>> MO.groups()
('22','98')
>> MO = re.match ('(\d+)([.])(\d+)', s)
>> MO.groups()
('22','.','98')
```

```
s = 'hai Python'
re.search ('a', s)
< re.Match object; span = (1, 2), match = 'a'>
re.search ('\Aa', s)        ⟶ starts with.
```

\b and \B : These are used for Searching words while we are using \b and \B we have to represent raw pattern

\b → starting (or) Ending
- starting → r'\b pattern'
- Ending → r'pattern\b'

\B → Not at starting, Not at Ending
- Not at starting → r'\B pattern'
- Not at Ending → r'pattern\B'

Ex:

```
s = 'hai python how are you'
>> re.findall ('\bh', s)   [ should be represented in raw pattern ]
[ ]
>> re.findall (r'\bh', s)
['h','h']
```

S = 'hai python how are you is'

re.findall(r'i\b', S) # (hai)

['i']

re.findall(r'\Bh', S)

['h']

> re.findall(r'i\b', S) # is

['i'] //

Exceptional Handling.

Error : Error is an unexpected Event which Terminates Execution of remaining Statements

Types of Errors

→ Syntax Error

→ Operational Error

Note : We cannot handle Syntax Error

Exception Handling : An Unexpected Event occured during Exception Execution of programmes which Terminates the Execution of other Statements is known as Exception

- When an Un-Expected Event is occured during Execution, an Exception Object is Created and raised to represent that unexpected Event
- The raised object should be Handled otherwise it will Terminates the Execution of program
- This Un-Expected Event is handled By Using the Try Except Blocks in python

Blocks of Exception Handling :

try :

Code which might cause an Error

Except :

Code which is responsible to handle the Error

} → we have to Error mention respective class.

Else :

Code which you to have to Execute if there is No Error in Try

finally :

Code which you have to Execute Irrespective of Error Occurence

Try with one Except and Else block

```
print ('start')
a = int (input ('Enter a number'))
b = int (input ('Enter a number'))
try:
    print ('try is started')
    res = a/b
    print ('try is Ended')
except ZeroDivisionError:
    print ('is have Handled')
else:
    print (res)
print ('end')
```

o/p : start
Enter a number 5
Enter a number 0
try is started
is have handled
End.

* Types of Errors in Python *

① NameError: Name Error in python is raised when the interpreter Encounters a Variable (or) function name that is yet to be defined / Variable is not defined.

② Import Error : Raised when the imported module is not found

③ Memory Error : Raised when an operation runs out of memory

④ Syntax Error : Raised by when function and operation are applied in an incorrect type / Syntax is wrong

⑤ Indentation Error : Raised when there is an incorrect indentation

⑥ Type Error : Raised when a function (or) operation is applied to an object of wrong type, such as adding a string to an integer.

⑦ Value Error : Raised when operation / function receives an argument with right type but wrong Value.

⑧ Key Error : Key Error is raised when Key is not found.

Ex : dkay = { 'a' : 1 , 'b' : 2 }
print (dkay ['c'])
Key : 'c'
Error

⑨ Zero Division Error : Raised when you divide a value or Variable with Zero.

⑩ Index Error : Occurs when the index of a Sequence is out of range.

x Memory Error : Raised when programs run out of memory x

⑪ Runtime Error : occurs when an Error does not fall into any Category.

⑫ System Error : Raised when Interpreter detects an Internal Error.

⑬ Attribute Error : Raised when attribute assignment (or) Reference fails.

⑭ KeyboardInterrupt : Raised when user inputs interrupt keys (Ctrl + C (or) Delete)

⑮ EOF Error "End-of-Line Error" : Occurs when python has reached End of user input without receiving any input. [The reason Eof Error occurs is that python attempts to print out your input in Variable string when no data is given.

(Python 2)

15) UnboundLocalError : When we are trying to access/Modify Global Variable in Local Space.

16) Stop Iteration Error : Exception that is raised when an Iteration is stopped. It is generally raised by Built-in function that have reached End of their Sequence

17) File Not Found Error : Will arise when working with files that are missing. python may fail to retrieve a file, if you have written wrong spelling of filename (or) file does not Exist

18) Assertion Error : Raised by assert Keyword if the given Condition is false.

Single Try with multiple Except Blocks :

Based on number of Errors raised we define that many multiple Except Blocks

```
print ('start')
a = int (input ('Enter number'))
b = int (input ('Enter number'))
try :
    print ('try is started')
    res = a/b
    print (x)
    print ('try is Ended')
Except ZeroDivisionError as z :
    print(z)
Except NameError as n :
    print (n)
else :
    print (res)
print ('end')
```

o/p :

```
start
Enter number 4
Enter number 2
try is started
name 'x' is not defined
end.
```

Default Except Block : Is responsible for handling Any type of Error But it can't Specify which Error it has handled.

We can define default Except Block by Using except Keyword.

```
print ('Start')
a = int (input ('Enter a Value'))
b = int (input ('Enter b Value'))
try :
    print (' start of try ')
    res = a/b
    print (x)
    print (' End of try ')
Except ZeroDivisionError as z :
    print (z)
Except NameError as n :
    print(n)
Except :
    print (' Unknown Error is Handled')
Else :
    print (res) //
```

Generic Exception Class : It is Responsible for handling Any type of Error and it can Specify which Error it has handled.

- We can Create Generic Exception Block By Using Exception Class

```
a = Eval (input ('Enter a '))
b = Eval (input ('Enter b'))
try :
    print ('start of Try')
    res = a/b
    print (res)
    print (x)
    print (' End of Try')
Except Exception as e :
    print (e) //
```

Finally Block : Finally Block will be Executed Irrespective of Error Occurence

- Finally Block is Responsible for performing Operations like closing of opened file and Revoking DataBase Connections
-

Ex:

```
print ('start')
try:
    print ('try is started')
    fo = open('result.txt','w')
    data = int (input ('Enter data'))
    fo.write (data)
    print ('try is Ended')
except Exception as e:
    print (e)
finally:
    print ('i am inside finally')
    fo.close ()
print (fo.closed)
print ('end') //
```

// Handlers in Different functions / Handling Errors.

```
def f1():
    print ('f1 is started')
    a = 10/0
    print ('f1 is Ended')
def f2():
    print ('f2 is started')
    f1()
    print ('f2 is Ended')
print ('main starts')
f2()
print ('main Ends')
```

Execution : As Error occured, First it will check for Handler in Current function. if not it will check in Caller function even if is not present there, then it will look for Handler from where you have called caller function

o As we have not defined handler any where Exception causes Termination.

Note : If the Exception is raised it must be handled in Current function (or) in Caller function (or) from where the caller function is Called.

(1) // when there is no handler in any function / Main Space.

Handler in Suspect function :

```
def f1( )
    print ('first line of f1')
try: try:
    result = 10/0
Except Exception as e :
    print (e)
print ('Last line of f1')

def f2( ):
    print ('first Line of f2')
    f1( )
    print ('Last line of f2')

print ('Main Started')
f2( )
print ('Main Ended')
```

Handler in Caller function :

```
def f1( ) :
    print ('f1 is started')
    a= 10/0
    print ('f1 is Ended')

def f2( ) :
    print ('f2 is Started')
    try:
        f1( )
    Except Exception as e :
        print (e)
    print ('f2 is Ended')

print ('main starts')
f2( )
print ('main Ends')
```

Handler in place from where we have called function.

```
def f1( ) :
    print ('f1 started')
    a= 10/0
    print ('f1 Ended')
def f2( ) :
    print ('f2 is started')
    f1( )
    print ('f2 is Ended')
print ('main starts')
try:
    f2( )
```

```
Except Exception as e :
    print(e)
print('main Ends')
```

Raise Keyword : Raise Keyword is used for Creating and raising the Exception based on User Requirement.

Note : By Using Raise Keyword we can create and raise both Built in and User defined Exception Object.

Ex :-

```
print('start')
try:
    print('first line of try')
    L=[11, 22, 334, 67]
    ip = int(input('Enter ip'))
    if ip > len(L):          → user defined { Built-in Exception class }
        raise IndexError('index out of Range')
    print('Last line of Try')
except Exception as e :
        print(e)
else :
    print(L[ip])
print('end')
```

User Defined Exception Classes :

1. User Defined Exception Class can be created by a class which is inherited from Base Exception Class

```
Class ClassName(Base Exception):
    Statements
```

2. Use Raise Keyword to Call User defined Exception class as shown below.

```
raise ClassName(Arguments)
```

3. After raising an Error handle it by using Generic Exception ~~Class~~ Handler

Note : We Cannot handle user defined Exception classes by using Generic Exception class.

Ex :

```
class DemoError(Base Exception):
    def __init__(self, msg):
        self.msg = msg
print('main started')
try:
    raise DemoError('User has raised this Error')
except DemoError as d :
    print(d.msg)
print('main Ended')
```

Nested Try and Except Block :-

It is the process of defining Try Except Block inside another Try Block

```
print ('start')
try:
    print ('first line of outer try')
    L = [11, 22, 33, 44]
    ip = int (input ('Enter ip'))
    print (L[ip])
    try:
        print ('inner try is started')
        x = int (input ('enter x'))
        y = int (input ('Enter y'))
        res = x/y
        print ('inner try is Ended')
    Except Exception as e :
        print(e)
    else :
        print (res)
    print ('Last line of outer try')
Except Exception as e :
    print (e)
print ('end')
```

```
class PrimeError(BaseException):
    def __init__(self, msg):
        self.msg = msg
print ('start')
try:
    print ('first line of Try')
    n = int (input ('Enter n value'))
    for i in range (2, n//2 + 1):
        if n%i == 0:
            raise PrimeError ('Not Prime')
Except PrimeError as p :
    print(p)
else :
    print ('n is prime')
print ('end') . //
```

```
Class CredentialsError(BaseException):
    def __init__(self, msg):
        self.msg = msg
Class InsufficientError(BaseException):
    def __init__(self, msg):
        self.msg = msg

Class Bank:
    bank_name = 'SBI'
    bank_roi = 7
    bank_branch = 'Bangalore'
    def __init__(self, n, a, ac, bal, pw):
        self.name = n
        self.age = a
        self.account = ac
        self.balance = bal
        self.password = pwd
    def withdraw(self):
        amount = int(input('Enter amount'))
        if self.balance > amount:
            self.balance -= amount
            print('Withdrawal Successful')
        Else:
            raise InsufficientError('Low Balance')

Seetha = Bank('Seetha', 23, 12345, 10000, 1342)
UN = input('Enter UserName')
pw = int(input('Enter password'))
try:
    if UN != Seetha.name or pw != Seetha.password:
        raise CredentialError('Log in Details are Invalid')
    Else:
        try:
            Seetha.withdrawal()
        Except InsufficientError as g:
            print(g)
Except CredentialError as c:
    print(c)
```

// Assert : Assert Keyword is used whenever we are de bugging the code (Testing)

→ Syntax : assert Condition, 'Message'

→ Execution If the given Condition is True, it will allow Interpreter to Execute Remaining Statements

If the Condition is false it will Create and Raise AssertionError which will terminate Remaining Statements.

Example :

```
a = -23
try:
    assert a>0, 'given a value is less than 0'
Except AssertionError as As :
    print (As)
Else:
    print (a)
```

o/p: given Value is less than 0

IsInstance : Used to check given Value is Instance of Specified class if it is it will give True (or) Else false

Ex :

```
a = 23
b = 0
try:
    assert isinstance (a, int) and isinstance (b, int),
                                'Not integers'
    assert b != 0, 'given divisors is zero'
    res = a/b
Except AssertionError as ae :
    print (ae)
Else :
    print (res)
```

o/p : Given Divisor is Zero