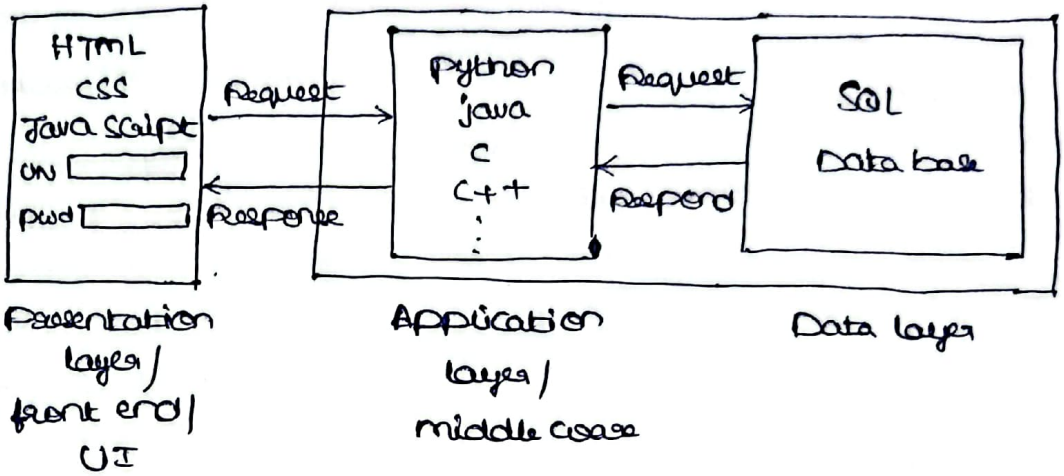
Python

→ Python was introduced in 1991 by "Guido van Rossum"
→ In 2001 it got downloaded.
→ He inspired by a comedy series called "monty python flying class circus" later he named the programming language as python.

Web application



Note: 1) Open source: In open source when the source code is exposed to the users.
2) Interpreter: The process of compiling the code line by line is called interpreter
3) Dynamically typed: it is a phenomenal of making the type and size of the memory location as variable.

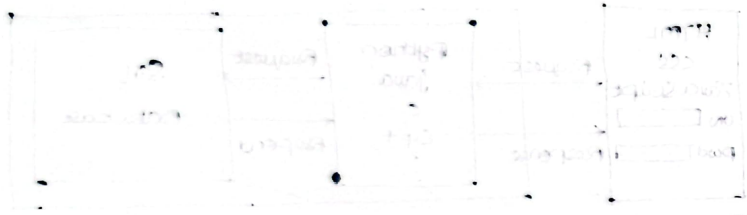
Definition of Python: Python is a open source, high level programming language.

Features of Python:

→ High level programming language
→ Dynamically typed programming language
→ Easy to learn and analyse
→ It supports OOPs & Procedural language
→ Interpreted and compiled programming language
→ Huge numbers of libraries (around 7+ lakhs)
→ It is platform independent
→ Open source
→ Secure

Field of application:

1) Soft ware development, testing and data analysing.
2) AI and ML
3) Complex mathematical equation.
4) Data science (scientist), Researchers.
5) IOT's, cyber security.
6) Build applications such as mobile & web applications.



continue: This statement can be used when we need to skip the current iteration of a loop and move on to the next iteration based on the condition.

Ex: for i in rang(1,6):
    if i%3==0:
        continue
    print(i)

O/P: 1 2 4 5

1 in (1,6) T
1%3==0 F
print(1)

2 in (1,6) T
2%3==0 F
print(2)

3 in (1,6) T
3%3==0 T skip

4 in (1,6) T
4%3==0 F
print(4)

5 in (1,6) T
5%3==0 F
print(5)

6 in (1,6) T
6%3==0 T skip

Pass: This statement act as a place holder for the future code.

- when ever the statement is executed nothing will happens but we can avoid getting an error when the empty code is not allowed.

Ex: a=10
b=20
if (a>b):
    print('a is bigger')
if (a<b) (or) else:
    pass

O/P: Empty space

- wap to swap the values with out using a third variable

a=10
b=20
a=a+b = 10+20=30
b=a-b = 30-10=20
a=a-b = 30-20=10
print(f'{a} is a')
print(f'{b} is b')
O/P: a=20
b=10

x=23
y=10
x=x+y = 23+10=33
y=x-y = 33-10=23
x=x-y = 33-23=10
print(f'{x} is x')
print(f'{y} is y')
O/P: x=10
y=23

- wap to swap the values using a third variable

a=10
b=20
c=a+b = 10+20=30
a=c-a = 30-10=20
b=c-b = 30-20=10
print(f'{a} is a')
print(f'{b} is b')
O/P: a=20
b=10

a=10
b=20
temp=b = 20
b=a = 10
a=temp = 20
O/P: a=20
b=10

• Wap to print numbers in a given range using fo loop

```
a=int(input('eav'))
b=int(input('eav'))
fo i in range(a,b):
    print(i)
```

O/P: 1 2 3 4 5 6 7 8 9

```
fo i in range(1,10):
    print(i)
```

• Wap to print numbers in a given range using while loop

```
a=int(input('eav')) a=0
b=int(input('eav'))
while a<=b:
    print(a)
    a+=1
```

O/P: 0 1 2 3 4 5 6 7 8 9 10

```
while a<=10:
    print(a)
    a+=1
```

• Wap to print numbers in a given range in reverse order

```
a=int(input('eav'))
b=int(input('eav'))
fo i in range(b,a-1,-1):
    print(i)
```

O/P: 5 4 3 2 1

```
fo i in range(10,1,-1):
    print(i)
```

O/P: 10 9 8 7 6 5 4 3 2

```
a-1
i=5
5 in range(5,0-1) T
    O/P: 5
i=4
4 in range(5,0,-1) T
    O/P: 4
i=3
3 in range(5,0,-1) T
    O/P: 3
i=2
2 in range(5,0,-1) T
    O/P: 2
i=1
1 in range(5,0,-1) T
    O/P: 1
```

i=0
0 is excluded as it is end value

• Wap to print given numbers in a given range in reverse order using while

```
a= int(input('enter the value'))
b= int(input('enter the value'))
while a>=b:
    print(a)
    a-=1
```

• Wap to reverse a range of value using swap & reverse value

```
x= int(input('enter the value')) 5
y= int(input('enter the value')) 5
if y>=x:    5>=1 T
    x=x+y   x=1+5=6
    y=x-y   y=6-5=1
    x=x-y   x=6-1=5
while x>=y:
    print(x)
    x-=1
```

O/P: 5 4 3 2 1

• Wap to print even numbers in a given range

```
a= int(input('enter the value')) 1
b= int(input('enter the value')) 5
fo i in range(a,b+1):
    if i%2==0   6%2==0
        print(i)   O/P: 2,4,6
```

O/P: 2 4 6 8

1 in range(a,b+1)
1%2==0 False
2 in range(a,b+1)
2%2==0 T
O/P: 2
3 in range(a,b+1)
3%2==0 False
4 in range(a,b+1)
4%2==0 True
O/P: 4

• Wap to print odd numbers in a given range

```
a= int(input('enter a value')) 1
b= int(input('enter a value')) 5
fo i in range(a,b+1):
    if i%2!=0
        print(i)
```

O/P: 1 3 5

1 in range(1,6) T
1%2!=0 T
O/P: 1
2 in range(2,6) T
2%2!=0 F
3 in range(3,6) T
3%2!=0 T
O/P: 3
4 in range(4,6) T
4%2!=0 F
5 in range(5,6) T
5%2!=0 T
O/P: 5

- wap to find the factorial of a number

```
a = int(input('enter a number'))
fact = 1
fb i in range of (1, a+1):
    fact *= i
print(fact)
```

fact = 1

(a) fact *= fact * i

factorial: The number is multiplied if all the int that lies blw one & the number its self.

a = 5

i=1 fb i range (1, a+1 = 6) T
fact = 1×1
fact = 1

i=2 fb i range (1,6) T
fact = 1×2
fact = 2

i=3 fb i range (1,6) T
fact = 2×3
fact = 6

i=4 fb i range (1,6) T
fact = 6×4
fact = 24

i=5 fb i range (1,6) T
fact = 24×5
fact = 120

- wap to find factors of a number

```
a = int(input('enter a number'))
fb i in range of (1, a+1): T 10
    if a%i==0
        print(i)
```

O/P: 1, 2, 5, 10

factor numbers: The numbers that can divide onothers number if the remainder as 0.

i=1 fb i range of (1,11) T
10%1==0 F

i=2 fb i range of (1,11) T
10%2==0 T

i=3 fb i range of (1,11) T
10%3==0 F

i=4 fb i range of (1,11) T
10%4==0 F

i=5 fb i range of (1,11) T
10%5==0 T

i=10 fb i range of (1,11) T
10%10==0

- wap to print some of the numbers from 1 to n

```
a = int(input('enter a number'))
sum = 0
fb i range of (1, a+1): T
    sum += i (B) sum = sum+i
print(sum)
```

5

i=1 fb i range of (1,6) T
sum = 0+1
O/P: 1

i=2 fb i range of (2,6) T
sum = 1+2
O/P: 3

i=3 fb i range of (1,6) T
if 1%2==0 F

i=2
fb i range (1,6) T
if 2%2==0 T
sum = sum+i
= 0+2 = 2

i=3
fb i range (1,6) T
if 3%2==0 F

i=4
fb i range (1,6) T
if 4%2==0 T
sum = sum+i
= 2+4=6

- wap to find sum of the even numbers in a given range

```
a = int(input('enter a value'))
sum = 0
fb i in range (1, a+1):
    if i%2==0:
        sum += i
        print(sum)
```

O/P: 2 6

i=5
fb i range (1,6) T
if 5%2==0 F

- wap to find the product of first natural numbers

Product of natural number program same as factorial number

```
a = int(input('enter a value'))
product = 1
fb i in range (1, a+1):
    product = product * i
print(product)
```

- wap to find to check given number is prime & not. It should be divisible by one and itself.

```
num = int(input('enter a value'))
count = 0
fb i in range (1, a+1): if num%i==0:
    count += 1
if (count == 2):
    print('a is a prime number')
else:
    print('a is not a prime number')
```

num = 3

fb 2 in range of (1,4) T
3%1==0 T
count = 0+1
= 1

fb 2 in range (1,4) T
3%2==0 F

fb 3 in range (1,4) T
3%3==0 T
count = 1+1
= 2

O/P: a is a prime number

• Wap to reverse of the given number

① num = 84
rev = 0
while num != 0 :
rem = num % 10
rev = rev * 10 + rem
num = num // 10
print (rev)
O/P : 48

① 84 != 0 T
10)84(8 → num
80
rem → 4
rem = 84 % 10 = 4
rev = 0 × 10 + 4 = 4
num = 84 // 10
num = 8

② 8 != 0 T
rem = 8 % 10 = 8
rev = 4 × 10 + 8 = 48
num = 8 // 10
num = 0

③ 0 != 0 F

⑤ num = 102
rev = 0
while num != 0 :
rem = num % 10
rev = rev * 10 + rem
num = num // 10
print (rev)

① 102 != 0 T
rem = 102 % 10 = 2
rev = 0 × 10 + 2 = 2
num = 102 // 10
num = 10

10)102(10
100
2

② 10 != 0 T
rem = 10 % 10 = 0
rev = 2 × 10 + 0 = 20
num = 10 // 10
num = 1

③ 1 != 0 T
rem = 1 % 10 = 1
rev = 20 × 10 + 1
= 201
num = 1 // 10
num = 0

④ 0 != 0 F

③ 7892
① 7892 != 0 T
rem = 7892 % 10 = 2
rev = 0 × 10 + 2 = 2
num = 7892 // 10
num = 789

② 789 != 0 T
rem = 789 % 10 = 9
rev = 2 × 10 + 9 = 29
num = 789 // 10
num = 78

③ 78 != 0 T
rem = 78 % 10 = 8
rev = 29 × 10 + 8 = 298
num = 78 // 10
num = 7

④ 7 != 0 T
rem = 7 % 10 = 7
rev = 298 × 10 + 7
= 2987
num = 7 // 10
num = 0

⑤ 0 != 0 F

④ 5483 ① 5483 != 0 T
rem = 5483 % 10 = 3
rev = 0 × 10 + 3 = 3
num = 5483 // 10
num = 548

② 548 != 0 T
rem = 548 % 10 = 8
rev = 3 × 10 + 8 = 38
num = 548 // 10
num = 54

③ 54 != 0 T
rem = 54 % 10 = 4
rev = 38 × 10 + 4 = 384
num = 54 // 10
num = 5

④ 5 != 0 T
rem = 5 % 10 = 5
rev = 384 × 10 + 5
= 3845
num = 5 // 10
num = 0

⑤ 0 != 0 F

• Palindrome of numbers

num = 121
rev = 0
temp = num
while num != 0 :
rem = num % 10
rev = rev * 10 + rem
num = num // 10
if temp == rev :
print('it is Palindrome')
else :
print('it is not palindrome')

① 121 != 0 T
rem = 121 % 10 = 1
rev = 0 × 10 + 1 = 1
num = 121 // 10
= 12

② 12 != 0 T
rem = 12 % 10 = 2
rev = 1 × 10 + 2 = 12
num = 12 // 10
= 1

③ 1 != 0 T
rem = 1 % 10 = 1
rev = 12 × 10 + 1 = 121
num = 1 // 10
= 0

④ 0 != 0 F
temp == rev
121 121
O/P : it is palindrome

• A number of characters that remains same even if the number of characters & immediate are same.

② 243
① 243 != 0 T
rem = 243 % 10 = 3
rev = 0 × 10 + 3 = 3
num = 243 // 10 = 24

② 24 != 0 T
rem = 24 % 10 = 4
rev = 3 × 10 + 4 = 34
num = 24 // 10
= 2

③ 2 != 0 T
rem = 2 % 10 = 2
rev = 34 × 10 + 2 = 342
num = 2 // 10
= 0

④ 0 != 0 F
temp == rev
243 342
O/P : it is not palindrome

• Wap to reverse the given string

a = 'python'
revstr = ''
i = len (a) - 1
while i >= 0 :
revstr = revstr + a[i]
i -= 1
print (revstr)

① i = 6 - 1 = 5
5 >= 0 T
revstr = '' + N
revstr = N
i = 5 - 1 = 4

② i = 4
4 >= 0 T
revstr = 'N' + 'O'
revstr = NO
i = 4 - 1 = 3

③ 3 >= 0 T
revstr = NO + h
revstr = NOh
i = 3 - 1 = 2

④ 2 >= 0 T
revstr = NOh + t
revstr = NOht
i = 2 - 1 = 1

⑤ 1 >= 0 T
revstr = NOht + y
revstr = NOhty
i = 1 - 1 = 0

⑥ 0 >= 0 T
revstr = NOhty + p
revstr = NOhtyp

• wap the given str is palindrome & not.

```
a = 'mom'
seustr = ''
i = len(a)-1
while i>=0:
    seustr = seustr+a[i]
    i-=1
print(seustr):
if a == seustr:
    print('it is palindrome')
else:
    print('it is not palindrome')
```

① i = 3-1 = 2
2>=0 T
seustr = seustr+a[i]
= '' + m
seustr = m
i = 2-1 = 1

② i=1
1>=0 T
seustr = m+o
= mo
i = 1-1 = 0

③ i=0
0>=0 T
seustr = mo+m
= mom
i = 0-0 = 0

• wap to count number of digits in a given number

① a = 124
Count = 0
while a!=0:
a = a//10
Count += 1
print (Count)
o/p: 1 2 3

① 124!=0 T
a=124//10=12
Count = 0+1
=1

② 12!=0 T
a=12//10=1
Count = 1+1
= 2

③ 1!=0 T
a = 1//10=0
Count = 2+1
Count = 3

④ 0!=0 F

② a=48
①48!=0 T
a=48//10 = 4
Count = 0+1
=1

② 4!=0 T
a=4//10=0
Count = 1+1
Count = 2

③ 0!=0 F

• wap to the given no is strong numbers & not
strong no is the special no the sum of all the digits factorial should be equals to a no itself.

```
a = int(input('enter a value'))
temp = a
sum = 0
while (a>0):
    rem = a%10
    fact = 1
    for i in range (1,rem+1):
        fact = fact * i
    sum = sum + fact
```

```
if (sum == temp):
    print('it is strong number')
else:
    print('it is not strong number')
```

o/p:
enter a value
145
it is a strong number

① a = 145
temp = a
sum = 0
145>0: True
while (a>0):
rem = a%10
=145%10
= 5
fact = 1
for i in range (1,6):
fact = 1x1
= 1
i=2 = 2x1
= 2
i=3 = 3x2
=6
i=4 = 4x6
=24
i=5 = 5x24
=120
sum = 0+120
=120
a = 145//10
= 14

② 14>0: True
rem = 14%10
= 4
fact = 1
for i in range (1,5):
fact = 1x1
= 1
i=2 = 2x1
= 2
i=3 = 3x2=6
i=4 = 4x6
=24
sum = 120+24
= 144
a = 14//10
= 1

③ 1>0: True
rem = 1%10
= 1
for i in range (1,1):
fact = 1x1
= 1
sum = 144+1
= 145
a = 1//10
= 0

④ a>0: 0>0: False
o/p: sum == temp
145 == 145
it is a strong number

- wap to check the given no is armstrong & not:

```
num = 153
temp1 = num
temp2 = num
count = 0
sum = 0
while num != 0
    count += 1
    num = num // 10
while temp1 != 0
    sem = temp1 % 10
    sum = sum + sem ** count
    temp1 = temp1 // 10
if sum == temp2 :
    print ("it is armstrong number")
else :
    print ("it is not armstrong number")
```

O/P: it is armstrong number

153 != 0 T
count = 1
15 != 0 T
count = 2
1 != 0 T
count = 3

① 153 != 0 True
sem = 153 % 10 = 3
sum = 0 + 3^3
sum = 27
temp1 = 153 // 10
temp1 = 15

② 1 != 0 True
sem = 1 % 10 = 1
sum = 152 + 1
= 153
temp1 = 1 // 10
= 0

③ 15 != 0 True
sem = 15 % 10 = 5
sum = 27 + 5^3
sum = 27 + 125 = 152
temp1 = 15 // 10 = 1

④ 0 != 0 False
sum == temp2
153 == 153

- wap to print the armstrong numbers in the given range

```
for i in range (100, 1000):
    num = i
    sum = 0
    temp1 = num
    temp2 = num
    count = 0
    while a > 0 :
        count += 1
        num = num // 10
    while temp1 > 0 :
        sem = temp1 % 10
        sum = sum + sem ** count
        temp1 = temp1 // 10
    if (temp2 == sum):
        print (f'{temp2} it is armstrong')
```

```
num = 153
temp1 = num
count = len (str(num))
sum = 0
while num > 0 :
    sum = num % 10
    sum = sum + sem ** count
    num = num // 10
if temp1 == sum :
    print(f'{temp1} it is an
        armstrong')
```

- wap to check the given number is perfect number & not

```
num = int(input ('enter a value'))
temp = num
sum = 0
for i in range (1, num) :
    if a % i == 0 :
        sum = sum + i
print (sum)
if sum == temp :
    print('it is perfect number')
else :
    print('it is not a perfect number')
```

- wap to print the perfect numbers in a given range

```
for i in range (1,2000):
    num = i
    sum = 0
    for j in range (1, num) :
        if num % j == 0 :
            sum = sum + j
    if sum == num :
        print (f'{sum} it is a perfect number')
```

- wap to check the given number is neon & not

```
num = 9
temp = num
sum = 0
sqr = num * num
while sqr > 0 :
    sem = sqr % 10
    sum = sum + sem
    sqr = sqr // 10
if sum == temp :
    print (f'{sum} it is neon number')
else :
    print('it is not neon number')
```

- wap to check the given no is perfect square number & not

```
a = int(input ("enter a value"))
for i in range (1,a):
    if (a == i * i) :
        print (f'{a} it is perfect square')
```

O/P: enter a value
9
it is a perfect square

36
1 in range (1,36):
1 x 1 = 36 F.
2 in range (1,36):
2 x 2 == 36 F
3 in range (1,36):
3 x 3 == 36 F
6 in range (1,36):
6 x 6 == 36 T
x == 1
break

- wap to check the given number is sunny number & not.

```
a = int (input ('enter a value'))
temp = a
x = 0
for i in range (1,a+1):
    if (a == i*i) :
        x = 1
        break
num = n+1
x = False
for i in range (1, num) :
    if i*i == num
        x = True
        break
```

(8)

n = 35

The number before the perfect square is called sunny number

```
if x == True:
    print(f'{a} is a sunny number')
else:
    print(f'{a} is not a sunny number')
```

- wap to check happy number & not.

```
a = int(input('enter a value'))
while (a!=1 and a!=4):
    sum=0
    while(a>0):
        rem=a%10
        sum=sum+rem**2
        a=a//10
    a=sum
if (a==1):
    print(f'{a} is a happy number')
else:
    print(f'{a} is not a happy number')
```

19

① 19!=1 and 19!=4
sum=0
19!=0
rem=19%10
=9
sum=0+9**2
=81
num=19//10=1
1!=0
rem=81%10=1
sum=81+1**2
=82
num=0
num=sum=82
num=82

② 82!=1 & 82!=4
sum=0
82!=0
rem=82%10=2
sum=0+2**2=4
num=82//10=8
8!=0
rem=87.10=8
sum=4+8**2
=68
num=0
sum=num=68
num=68

③ 68!=1 & 68!=4
sum=0
68!=0
rem=68%10=8
sum=0+8**2=64
num=64//10=6
6!=0
rem=6%10=6
sum=64+6**2
=100
num=0
sum=num=100
num=100

④ 100!=1 & 100!=4
sum=0
100!=0
rem=100%10
=0
sum=0+0=0
num=100//10=10
10!=0
rem=10%10=0
sum=0+0
=0
num=10//10=1
1!=0
rem=1%10=1
sum=0+1**2
=1
num=1//10=0

sum=num=1

Staing Programs

- wap to check to find the vowels in the given staing.

a='python'
j='aeiouAEIOU'
for i in a:
    if i in j:
        print(i)

o/p: o

① for p in python: T
if p in j: F
② for y in python: T
if y in j: F
③ for t in a: T
if t in j: F
④ for h in a: T
if h in j: F
⑤ for o in a: T
if o in j: T
o/p: o
⑥ for n in a: T
if n in j: F

a='LOKESH'

① for L in LOKESH: T
if L in j: F
② for O in LOKESH: T
if O in j: T
o/p: O
③ for K in a: T
if K in j: F
④ for E in a: T
if E in j: T
o/p: E
⑤ for S in a: T
if S in j: F
⑥ for H in a: T
if H in j: F

o/p: O E

- wap to remove the duplicates from a given staing

```
a= input('enter the staing')
stg = ''
for i in a:
    if i not in stg:
        stg = stg+i
print(stg)
```

a='EKESH'

① E in a: T
E not in stg: T
stg = ''+E
= E
② K in a: T
K not in stg: T
stg = E+K = EK
③ E in a: T
E not in stg: F
stg = EK
④ S in a: T
S not in stg: T
stg = EK+S
= EKS
⑤ H in a: T
H not in stg: T
stg = EKS+H
= EKSH

a='LOKESH'

① L in a: T
L not in stg: T
stg = ''+L
= L
② O in a: T
O not in stg: T
stg = L+O
= LO
③ K in a: T
K not in stg: T
stg = LO+K
= LOK
④ E in a: T
E not in stg: T
stg = LOK+E
= LOKE
⑤ S in a: T
S not in stg: T
stg = LOKE+S
= LOKES
⑥ H in a: T
H not in stg: T
stg = LOKES+H
= LOKESH

- Write a program to count number of times a given character is repeated?

```
a='pyspiders'
ch='p'
count=0
for i in a:
    if ch==i:
        count=count+1
print(count)
```

O/P: 2

for i in a = 'pyspiders'
① if ch==i ⓘ i=p
p==p T
count=0+1
=1
② i=y
y==p F
③ i=s
y==p F
④ i=p
p==p T
count=1+1
=2
⑤ i=i
i==p F
⑥ i=d
d==p F
⑦ i=e
e==p F
⑧ i=r
r==p F
⑨ i=s
s==p F

- Wap to find the duplicate characters in the given string

```
str='pyspiders'
s=''
for i in str:
    count=0
    for j in str:
        if i==j:
            count+=1
    if count>1:
        if i not in s:
            s+=i
print(s)
```

i=A A in str
i==j
A==A T
count=1
A==m=F
A==M=F
A==A=T
count=2
count>1 T
A not in s
s=A

Paragrams on List:

- Wap to print all the elements from the given list

```
li=[1,2,3,4,5]
for i in li:
    print(i)
```

O/P: 1 2 3 4 5

- Wap to print only even numbers from the given list

```
li=[1,2,3,11,15,13,14,16]
for i in li:
    if i%2==0:
        print(i)
```

O/P: 2
14
16

- Wap to print only odd numbers from the given list

```
li=[1,2,3,11,15,13,14,16]
for i in li:
    if i%2!=0:
        print(i)
```

O/P: 1
3
11
13
15

- Wap to print sum of all the elements from the list.

```
li=[1,8,6,1,5]
sum=0
for i in li:
    sum=sum+i
    print(sum)
```

O/P: 1
9
15
16
21

- Wap to find product of all the numbers from the given list

```
li=[1,8,6,1,5]
product=1
for i in li:
    product*=i
    print(product)
```

O/P: 1 8 48 48 240

• wap to print the sum of only even numbers from the list

```
li=[1,8,6,2,5,3,6]
sum=0
fo i in li:
  if i%2==0:
    sum+=i
      print(sum)
```

O/P: 8
14
16
22

• wap to remove the duplicates from the list.

```
li=[1,8,6,2,5,3,6,2,1]
l=[]
fo i in li:
  if i not in l:
    l=l+[i]
print(l)
```

O/P: [1,8,6,2,5,3]

① 1 in li T
1 not in l T
[1]

② 8 in li T
8 not in l T
[1,8]

③ 6 in li T
6 not in l T
[1,8,6]

④ 2 in li T
2 not in l T
[1,8,6,2]

⑤ 5 in li T
5 not in l T
[1,8,6,2,5]

⑥ 3 in li T
3 not in l T
[1,8,6,2,5,3]

• wap to print the duplicates from the list

```
li=[1,8,6,2,5,3,6,2,1]
l=[]
fo i in li:
  count=0
  fo j in li:
    if i==j:
      count+=1
  if count>1:
    if i not in l:
      l=l+[i]
print(l)
```

O/P:

• wap to reverse the list using while loop

```
li=[1,2,3,4]
rev=[]
i=len(li)-1
while i>=0:
  rev=rev+[li[i]]
  i-=1
print(rev)
```

O/P: [4,3,2,1]

i=4-1=3
3>0=T
rev=[]+4
=[4]
i=3-1=2
i=2
2>0=T
rev=[4]+[3]
=[4,3]
i=2-1=1
i=1
1>0=T
rev=[4,3]+[2]
=[4,3,2]
i=1-1=0
i=0
0>0=T
rev=[4,3,2]+[1]
=[4,3,2,1]

• wap to reverse the list using fo loop

```
li=[1,2,3,4]
rev=[]
fo i in range(len(li)-1,-1,-1):
  rev=rev+[li[i]]
  print(rev)
```

O/P:

• wap to print a prime numbers from the given list.

• wap to print maximum value from the list

```
l = [1,3,4,6,7]
max = l[0]
for i in l:
    if i > max:
        max = i
print(max)
```

1 in l : T — 1>0 : T — max=1
3 in l : T — 3>1 : T — max=3
4 in l : T — 4>3 : T — max=4
6 in l : T — 6>4 : T — max=6
7 in l : T — 7>6 : T — max=7

O/P: 7

• wap to print the minimum value from the list

```
l = [1,3,4,6,7]
min = l[0]
for i in l:
    if i < min:
        min = i
print(min)
```

1 in l : T — 1<0

O/P: 1

• wap to find second maximum number from the list

```
l = [1,2,3,8,7]
max = l[0]
for i in l:
    if i > max:
        max = i
max2 = 0
for i in l:
    if i > max2 and i != max:
        max2 = i
print(max2)
```

1 in l : T — 1>0 : T — max=1
2 in l : T — 2>1 : T — max=2
3 in l : T — 3>2 : T — max=3
8 in l : T — 8>3 : T — max=8
7 in l : T — 7>8 : F

① 1 in 8 : T
1>0 and 1!=8 T
max2 = 1

② 2 in 8 : T
2>1 and 2!=8 T
max2 = 2

③ 3 in 8 : T
3>2 and 3!=8 : T
max2 = 3

④ 8 in 8 : T
8>3 and 8!=8 : F

⑤ 7 in 8 : T
7>3 and 7!=8 T
max2 = 7

O/P: 7

• wap to swap first and last element in the list

```
l = [1,2,3,8,7]
temp = l[0] = 1
l[0] = l[len(l)-1]   l = 7
l[len(l)-1] = temp   7 = 1
print(l)
```

O/P: [7,2,3,8,1]

• wap to print the prime numbers in a given range

```
a = int(input('enter a value'))  1
b = int(input('enter a value')) 6
for i in range(1,b+1):
    count = 0
    for j in range(1,i+1):
        if i%j == 0:
            count += 1
    if count == 2:
        print(f'{i} is a prime number')
```

1 in (1,5) : T
2 in (1,6) : T
i=1 4%1==0 T
i=2 4%2==0 T
count = 0+1 = 1
1 count == 2 : F

O/P: 2,3,5

2 in (1,5) : T

① Wap to print fibonacci series till the given number.

A series of numbers in which each number is the sum of two preceding numbers.

```
num = 7
n1 = 0
n2 = 1
sum = 0
till i in range (1, num+1):
    print (sum)
    n1 = n2
    n2 = sum
    sum = n1+n2
```

Op: 0 1 1 2 3 5 8

```
n1 = n2
0 = 1
1 = 0 + 1
1 = 0
0 = 1 + 1
0 = 1
1 = 1 + 2
1 = 1
1 = 2 + 3
1 = 2
2 = 3 + 5
2 = 3
3 = 5 + 8
```

• Wap to print first three even numbers from the given range

① Wap to sort the values present in the list in ascending order, using bubble sort:

```
a = [2,3,1,6,8,4]
print ('the original list is:', a)
till i in range (0, len(a)):
    till j in range (1, len(a)):
        if a[j-1] > a[j]:
            temp = a[j-1]
            a[j-1] = a[j]
            a[j] = temp
print ('after sorting', a)
```

till 0 in range (0,6):

I till 1 in range (1,6):
- if a[j-1] > a[j] → 1-1, a[1] → a[0] > a[1] → 2 > 3 : F

II till 2 in range (1,6):
- if a[j-1] > a[j] → 2-1 > a[2] → a[1] > a[2] → 3 > 1 : T
- temp = 3, 3 = 1, 1 = temp → [2,1,3,6,8,4]

III till 3 in range (1,6):
- if a[j-1] > a[j] → 3-1 > a[3] → a[2] > a[3] → 3 > 6 : False

IV till 4 in range (1,6):
- if a[j-1] > a[j] → 4-1 > a[4] → a[3] > a[4] → 6 > 8 : False

V till 5 in range (1,6):
- if a[j-1] > a[j] → 5-1 > a[5] → a[4] > a[5] → 8 > 4 : True
- temp = 8, 8 = 4, 4 = temp → [2,1,3,6,4,8]

till 1 in range (0,6):

till 1 in range (1,6):
- if a[j-1] > a[j] → 1-1 → a[0] > a[1] → 2 > 1 : True
- temp = 2, 2 = 1, 1 = temp → [1,2,3,6,4,8]

till 2 in range (1,6):
- if a[j-1] > a[j] → a[2-1] > a[2] → a[1] > a[2] → 2 > 3 false

till 3 in range (1,6):
- a[2] > a[3] → 3 > 6 false

till 4 in range (1,6):
- a[3] > a[4] → 6 > 4 True
- temp = 6, 6 = 4, 4 = temp → [1,2,3,4,6,8]

till 5 in range (1,6):
- a[4] > a[5] → 6 > 8 false

a = [1,2,3,4,6,8]

Functions: Function is a group of related statements (a) block of code that performs a specific task 'when it is called'.

function declaration: user choice ↓ optional ↓

Syntax: def function name (parameters):

Function definition ⟷ Identation — } function body

statement

function name (arg)

↳ function call

def: it is a keyword which is used to declare the function. it is also called as function definition.

function name: its a name given to the function as per the user choice.

- When ever the function is called the control of the program goes to function definition.
- All the codes inside the function (function body) are executed.
- The control of the program jumps to the next statements after the function call.

Function with no arguments Ex:

```
Ex: def wishes():          ← 2
        print('good afternoon')
        print('had lunch')
    wishes()               ← 1
                           → 3
```

O/P: good afternoon
had lunch

Parameters: These are the variables given to the function when the function is defined.

- Parameters are seperated by ','
- A function can have any no of parameters
- If we create a function with parameters then we need to pass the corresponding values while calling the function.

Ex: def add (x,y):
(x,y) ↓ parameters

Arguments: A function can also have arguments those are the values which are given to the parameters i.e. accepted by a function.

Ex: def add (x,y):
print (x+y)
add (10,20)
↓ arguments

Function with argument Ex:

O/P: 30

Ex: def addition (n1,n2):
sum = n1+n2
print (sum)
addition (10,20)
print ('hello hi bye')

O/P: 30
hello hi bye

Return: It is a keyword it is used to end the execution

- Return should be always used inside the function.
- The return keyword is followed by an optional return value.
- Return keyword returns value to the function caller and that value can be in an variable to perform any other operations.
- When ever we use this keyword it will stop the execution and returns the result i.e. it is the last statement in the function.
- After the return we can't write any code.

Syntax: def function name ():
    return
function name ()

Ex: def add (a,b):
    c = a+b
    return
result = add (3,5)
print (result)

Return value: The value that return to the function call is called as return value.

Ex: def sum (a,b):
print (a+b)
res = sum (10,10)
print (res)

O/P: 9
None

- In the above ex we don't have any return statement so the o/p it will return 9 i.e. from print (a+b)
- along with 9 it will print none.i.e from print (res)

Return: When ever we call return the value next to it will be return when it reached. In order to use that value in program we need to store that's why we are using variable = function call.

Return value: By default when you call the function it does n't have return statement then the value return as 'None'.

- If we want to assign a value to function call then we can assign that value to a variable. (implicit return = 'None').
- wap to print the sum of two numbers. using function.

def add (x,y):
sum = x+y
print (sum)
add (10,20)

O/P: 30

• WAP that compares two numbers and returns the biggest of two numbers.

```
def great (a,b):
    if (a>b):
        return (a)
    else:
        return (b)
see = great (60,50)
Print (see)        O/P: 60
```

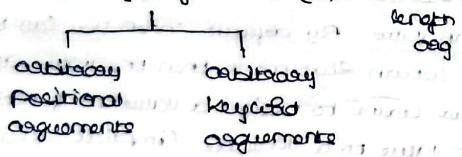
• W.a function program the given is even & odd

```
def even (a):
    if a%2==0:
        return 'even'
    else:
        return 'odd'
see = even (6)          O/P: even
Print (see)
```

Ex: `def add (a,b):`
`return (a+b)` O/P: 4
`Print (add (1,3))`

Functions' arguments in Python:

In Python we can pass the arguments to the function in different ways that are: 1. default arguments
2. positional arguments
3. key called arguments
4. arbitrary arguments (*) variable length arg



1. Default arguments: default arguments are values that are provided while defining functions
• The assignment operator equals to '=' is used to assign a default value to the parameter
• Default arguments become optional during the function calls
• If we provide a value to the default arguments during the function calls it 'overrides the default values'.
• The function can have any numbers of default arguments
• Default arguments should follow non default arguments.

Ex: 1. `def add (a=5,b=6):`
`return (a+b)`
`Print (add())`
O/P: 11

2. `def add (a=5,b=6):`
`return (a+b)`
`Print (add(9,5))`
O/P: 14

3. `def add (a=5,b=6):`
`return (a+b)`
`print (add (6))`
O/P: 10
• 6 a 6 is assigned

4. `def add (c,a=5,b=6):`
`return (a+b+c)`
`print (add (2))`
O/P: 13

5. `def add (a=5, b=6, c):`
`return (a+b+c)`
`Print (add())`
O/P: Error
• default values are not should following non default values

2. positional arguments: These are the arguments that need to be passed to the function call in a perfect order.
• during a function call the values passed through arguments should be in the order of parameters in function definition.
• Key called arguments should follow positional arguments only.

Ex: `def name (fname, lname):`
`return fname+lname`
`see = name ('Venkata', 'Lokesh')` O/P: Venkata Lokesh
`print (see)`

• Here we have called the name function with two arguments
• These arguments are passed to the function are called positional arguments.
• It is because the first argument 'Venkata' is assigned to first parameter and the second argument 'Lokesh' is assigned to the second parameter. These arguments are passed based on their position.

Ex: `def name (fname, lname):`
`return fname+lname`
`see = name ('Venkata')` O/P: Error
`print (see)`

3. Key called arguments: functions can also be called using key called arguments of the form keycalled argument = value (Parameter)

• using this type of arguments we not only pass the arguments to the function based on the position but also using their parameter name.

Ex: `def name (fname, lname):`
`return fname + lname`
`see = name (fname = 'Venkata', lname = 'Lokesh')`
`Print (see)`
O/P: Venkata Lokesh

Ex: def add (a,b):
    return a+b
c = add (a=10, b=20) O/P: 30
print (c)

Important Notes:

1. Default arguments should follow non default arguments

Ex: def add (a=5,c):
    return (a+c)
O/P: Error

2. Keyword argument should follow positional arguments

Ex: def add (a,c):
    return (a+c)
add (a=10,3,4)
O/P: Error

3. All the keyword arguments passed must match one of the argument accepted by the function and their order is not important.

Ex: def add (a,c):
    return (a+c)
add (a=10, c=3, b=4) O/P: Error

4. No arguments should receive a value, more than once.

Ex: def add (a,c):
    return (a+c)
add (a=10, c=3, c=4)
O/P: Error

5. Default arguments are optional arguments because it can ignore it. Ex: def add (a, c=5):
    return (a+c)
add (10,6)
O/P: 16

4. Arbitrary arguments (or) Variable length arguments:

- arbitrary arguments is used when user don't know the number of arguments (or) parameters needed for the function in advance then use use arbitrary arguments.
- The two types of this arguments are:
1. arbitrary positional argument
2. arbitrary keyword argument.

1. Arbitrary positional argument: This argument is used to create a function that accepts n number of positional arguments.
- To this argument is use need to place 'asterisk (*)' before a parameter in function definition.
- This parameter can hold a non keyword variable length arguments
- These arguments will be wrapped up in a tuple hence it is called as 'Tuple packing'.

Ex: def hi (*a):
    return a
res = hi (1,2,4,6,7)
print (res)
O/P: (1,2,4,6,7)

- wap to print sum of all the given arguments.

def add (*a):
    sum=0
    for i in a:
        sum= sum+i
    return sum
res = add (1,2,3,4,5)
print (res)
O/P: 15

Ex: combination of arbitrary positional argument and normal parameter

Ex: ① def hi (*a, b, c):
    print (a)
    print (b,c)
hi (1,2,4,6,7, b=5, c=8)
O/P: (1,2,4,6,7)
    5 8

② def hi (a, b, *c):
    print (a)
    print (b)
    print (*c)
hi (1,2,3,4)
O/P: 1
    2
    (3,4)

③ def hi (a, *b, c):
    print (a)
    print (b)
    print (c)
hi (1,2,4,6, c=7)
O/P: 1
    (2,4,6)
    7

2. Arbitrary Keyword Argument: This argument is used to create a function which accepts n no of keyword arguments.
- In this argument two 'asterisk (**)' is placed infront before a parameter in a function.
- This argument can hold keyword variable length arguments.
- This argument is called as 'dictionary packing'.

Ex: def hi(**a):
    return a
res = hi(a=1, b=2, c=4, d=5, y=1)
print(res)
O/P: {'a':1, 'b':2, 'c':4, 'd':5, 'y':1}

Note: We can't use & we can't provide any parameter after arbitrary keyword argument.

Difference b/w arbitrary positional & arbitrary keyword arguments:

arbitrary positional arguments (arg) single packing	arbitrary keyword arguments (KW args) double packing
• When user don't know the no of parameters to pass to the function definition we use this type of arguments.	• When user have to call n no of arguments then we use this type of argument.
• It is representated by a single 'asterisk (*)'	• It is representated by 'double asterisk (**)'.
• It is also called as 'tuple packing'.	• It is also called as 'dictionary packing'.
• It allows duplicate values.	• It doesn't allows duplicate keys.

Scope of Variables: In python we can declare the variables in two different ways. 1. Local Variable
2. Global Variable

- A variable scope specifies the area where we can access a variable.

1. Local Variable: When user declare the variables 'inside a function' these variables only have local scope. 'We can't access them out side the function'. These type of variables is called as local variables.

Ex: def loc():
    z=10 → local variable
    print(z)
loc()
O/P: 10

Ex: def loc():
    z=10
    print(z)
loc()
print(z)
O/P: Error

- Here the z variable is local to loc function hence we can't access the z variable out side the function.

2. Global Variable: In Python the variables declare out side the function is called as global variable. This means those variables can be access by both inside (&) out side the function.

Ex: z=10
def loc():
    print(z)
loc()
print(z)
O/P: 10
10

Note: Global variable we can access anywhere in the main space of program, but we cannot modify.

*Global Keyword: This keyword is used to make a local variable which is present inside a function as global variable. So that we can access the variable out side the function also.
- If we want to make the local variable to global variable then we have to define (&) declare the keyword 'global'. and we want to declare the global keyword before initializing a variable.

Ex: def loc():
    global z
    z=10
    print(z)
loc()
print(z)
O/P: 10
10

Ex: def loc():
    z=10
    global z
    print(z)
loc()
print(z)
O/P: Error

- If we declare the global keyword after initializing the variable then you will get an error.

Different types of functions: There are two types of functions
1. User defined function 2. Built in function

1. User defined function: These are the function which will get created based on user requirement.
- In python we can define the user defined functions in four ways:
  1. Function with no argument & no return value
  2. Function with return value & no argument
  3. Function with argument & no return value
  4. Function with argument & return value

1. Function with no argument & no return value:

- In this type of function while defining, declaring (&) calling them we can't pass any arguments to them. This type of function can't return any value when we call them.

Ex: def loc ()
z=10
print(z) o/p: 10
loc ()

2. Function with return value & no argument:

- In this type of function we are not passing any arguments while defining, (&) calling the function. when we call this type of function it returns some value.

Ex: def multi ()
x=10
y=5
z=x*y
return z
ans = multi () o/p: 50
Print (ans)

3. Function with argument & no return value:

- In this type of function passing argument should be required but function will not return any values back.

Ex: def ad(a,b):
print diffs
c=a*b
Print (c) o/p: 84
ad (12,7)

4. Function with argument and return value:

- In this type of function it allows to pass the arguments while calling the function and this type of function returns some value when we call them.
- This type of user defined methods is called as fully dynamic functions because it provides maximum control to use.

Ex: def sub (a,b):
m=a-b
Return m
ans = sub(55,10)
Print (ans)
o/p: 45

2. Built in Functions: Built in functions are pre defined functions.

- In this type of function already the task is assigned by the developers.
- we can't change the built in functions

Ex: float () - Returns a floating point number
eval () - Evaluates and executes an expression
- filter () - use a filter function to exclude items in an iterable object.
bool () - Returns the boolean value of the specified object
bin () - Returns the binary version of a number.
dict () - Returns a dictionary (Array)
id () - Returns the id of an object
input () - Allowing the user input
int () - Returns an integer number
len () - Returns the length of an object.
list () - Returns a list
- map () - Returns the specified iterable with the specified function applied to each item
max () - Returns the largest item in an iterable
min () - Returns the smallest item in an iterable
print () - prints to the standard output device
range () - Returns a sequence of numbers, starting from 0 & increments by 1 (by default).
set () - Returns a new set object.
tuple () - Returns a tuple
type () - Returns the type of an object.
append () - To insert the value from the last.

map() - map is a function applies a given function to each element of an iterable (list, tuple) and returns an iterable containing results.

Syntax: map(function, iterable)

- map function takes two arguments 1. function
2. Iterable - it is iterable like (list, set, tuple)

Ex: def sq(n):
return n*n
num=(1,2,3,3,4,14)
ans = map (sq, num)
print (ans)
nsq = set (ans)
print (nsq)

filter() – use a filter function to exclude items in an iterable object.

Syntax: filter(function, iterable)

```
Ex: def eg(n):
        return n%2
    num = (1,2,3,3,4,4)
    res = filter(eg, num)
    print(res)
    neg = set(res)
    print(neg)
```

Nested Functions: Function written in an another function is called as nested function.

Syntax:
```
def fun1():
    def fun2():
        ...
    fun2()
fun1()
```

Ex:
```
def out():
    print('I')
    def inn():
        print('happy weekend')
    inn()
out()
```
O/P: happy weekend

- In the above example we have define the inn() function inside the out() function.
- Here inn() function is called as nested function
- This type of functions works similar to normal functions and it executes when inn() his called inside the function out():

Unpacking: It is the process of accessing a value from the collection like list, tuple, set & dict into distinct variables.

- unpacking is helpful in accessing the values from the collection without using indexes

Syntax: var1, var2 .... = collection

Ex: ① `a,b,c = 1,2,3`
```
print(a,b,c)
```
O/P: 1,2,3

② `n1,n2 = {1:4,4:8}.items()`
```
print(n1,n2)
```
O/P: {1:4, 4:8}
(1,4) (4,8)

Lambda Functions: Lambda are special type of functions in python without the function name.

- These are called as simple one line functions and there is no need of def keyword. It is also called as 'anonymous'.
- We can use lambda as a keyword to create the lambda function.

Syntax: Lambda argument : Expression

Ex: `S = Lambda x : x**2`
```
print(S(3))
```
O/P: 6

- **Arguments:** Here any value passing to the lambda function is arguments
- **Expression:** It is executed and it will return.

Ex: ① `a = lambda x,y : x+y`
```
print(a(3,4))
```
O/P: 7

② `a = lambda g,h: g if g>h else h`
```
print(a(10,20))
```
O/P: 20

- WAP to count no of Palandrome words in a given str.

```
a = 'maraj likes malayalam'.split()
b = ''
c = 0
for i in a:
    if i==i[::-1]:
        b+=1
        b+=''
        c+=1
print(b,c)
```
O/P: malayalam 1

Non Local Variable: In python non local variables are used in nested functions whose local scope is not defined. This means the variable can be neither in the local nor in the global scope.

- We have to use the keyword 'non local' to create this variable.

Ex:
```
def outer():
    msg = 'local'
    def inner():
        nonlocal msg
        msg = 'srinivas'
        print('inner:', msg)
    inner()
    print('outer:', msg)
outer()
```
O/P: inner: non local
outer: non local

OOPS (Object Oriented Programming System):

With the pillars of object oriented programming are:
Abstraction, inheritance, polymorphism, encapsulation.

Class: class is a blue print which consists of properties & functionalities of some world entity (or) objects. (or)
class is a container where we can store the data and we can access the data.

How to define a class.

- To create a class in python we can make use of the keyword 'class'.

Syntax: class classname:

Ex: class bird:
    name = ''
    colr = ''
    sound = ''
    age = 0

- Here bird is the name of the class
- name, colr, sound and age are the properties inside the class with default values.

Note: The variables inside a class are called as 'attributes'.

Object: object is a instance of a class

How to create an object:

Syntax: Object name = class name ( )

- Suppose bird is a class then we can create objects like bird 1, bird 2, bird 3 etc. from the class.

Ex: Bird 1 = bird ( )

- Here Bird 1 is the object of the class bird
- Now we can use this object to access the class attributes

Access class attributes using object:

- In order to access the attributes of the class we can make use of '. notation'

Syntax: object name . attribute

Ex: bird 1 . name = 'parrot'

- Here we use bird 1 . name as to change & access the value of name present in the attribute.

- WAP to create a class bird with min four attributes in it.

```
class bird:
    name = ''
    colr = ''
    sound = ''
    age = 0
bird 1 = bird ( )
bird 1. name = 'parrot'
bird 1. colr = 'green'
bird 1. sound = 'chipchip'
bird 1. age = 8
print ( bird 1. name, bird 1. colr, bird 1. sound, bird 1. age)
```

Types of Members (or) Types of properties:

1. Static members (or) class members (or) generic
2. object members (or) specific members

1. Static members: These are the members of the class which is common to each & every object.

Ex: class bird:
    name = ''
    sound = ''
    colr = ''

obj 1
obj 2
obj 3

2. object members: These are the members of an object which could be different (or) specific to each & every object.

Ex: class CN:
    name = ''
    colr = ''
  object 1
    sound = ''
  object 2
    age = 5

Self: Self allows us to access the variables (or) the attributes.

- Self represents the instance of a class.
- When ever we defined a function 'method' inside the class then the first parameter to the function is 'Self'.
- After the Self we can pass any no of parameters.

Ex: ① Class emp:

```
    empid = 123
    def seed():
        empno = 345
        print('empno', empno)
e1 = emp()
print('empid', empid)
e1.seed()
```

O/P: 123
345

Type error: emp.seed() takes a 0 positional arguments but 1 was taken

② Class emp:

```
    empid = 123
    def seed(self):
        empno = 345
        print('empno', empno)
e1 = emp()
print('empid', empid)
e1.seed()
```

O/P: 123
345

- With the help of self we can access the properties and methods of a defined class.

Ex: (with out declaring a properties to class & accessing the properties from the object itself.

```
Class emp:
    Pass
e1 = emp()
e1.empid = 123
e1.name = 'Lokesh'
e1.sal = 30000
print('empid', e1.empid)
print('ename', e1.ename)
print('sal', e1.sal)
```

O/P: empid 123
ename Lokesh
sal 30000

Constructor: Constructor is used to initilize & assigned a value to the data members of a class when an object in the class is created.

```
def __init__(self):
```

init: init is a type of function, init method is called as a constructor and it is always called when an object is created.

Rules of constructor:

- Constructor always starts with the keyword def
- It is followed by the word '__init__' which is prefix and suffix with two underscore with a pair of brackets '__init__()'.
- It takes an argument called 'self' to assigning a values to the variables.

Different types of constructor: when ever there are 3 types of constructor 1. default constructor
2. parameterized constructor
3. non-parameterized constructor

1. default constructor: When the user doesn't write the constructor but the class is created python itself creates the constructor during the compilation of the program i.e. is called as default constructor.

Ex: Class Std:

```
    sname = 12345
    def display(self):
        print(self.sname)
s1 = Std()
s1.display()
```

O/P: 12345

2. Parameterized Constructor: When the constructor accepts the parameters along with self is called as parameterized constructor.

Syntax: def __init__(self, var1, var2, ... varn):

Ex: def __init__(self, name, age):

```
class Std:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def display(self):
        print(self.name, self.age)
s1 = Std('Varun', 16)
s2 = Std('Samya', 14)
s3 = Std('Shri', 17)
```

S1. display ( )
S2. display ( )
S3. display ( )

o/p: Varun 16
Samya 14
Shri 17

5. Non-parameterized constructor: When the constructor does not accept any parameter from the object and as only 1 parameter i.e. self in the constructor is known as non parameterized constructor.

```
Ex: Class Std (
      def __init__ (self):
          Print ('i am from non parameterized constructor')
      def display (self, name):
          Print ('my name is', name)

    S1 = Std ( )
    S1. display ('hemanthi')
```

o/p: i am from non parameterized constructor
my name is hemanthi

Inheritance: It the process of creating a new class from the existance class.

- When one class inherits from another class the inheriting class is known as 'child class, subclass, derived class' and the class it inherits from is called as 'Parent class, base class (&) super class'.

base class: The class that does not inherit from another class is called as base class.

```
Ex: Class Parent :
        def Pdisplay (self):
            Print ('i am from Parent Class')
```

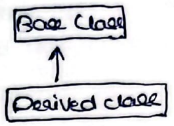
derived class: The class that is derived from the existing class is called as derived class.

```
Ex: Class Child (Parent):
        def Cdisplay (self):
            Print ('i am from Child Class')
```

Different types of inheritance:

1) Single inheritance
2) multi level inheritance
3) multiple inheritance
4) Hierarchical inheritance
5) Hybrid inheritance

1) Single inheritance: inheriting the properties from single parent & base class into single child class. (&) derived class.



Syntax:
```
Class Parent :
    Pass
Class Child (Parent) :
    Pass
```

- By inheriting parent class properties to child class, the child class can access all the data members, methods of parent class.

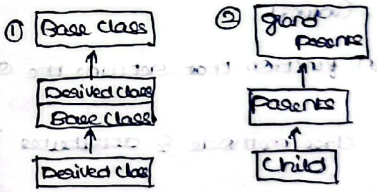
```
Ex: Class Parent :
        def Pdisplay (self):
            Print ('i am from parent class')
    Class Child (Parent):
        def Cdisplay (self):
            Print ('i am from child class')

    C1 = Child ( )
    C1. Pdisplay ( )
    C1. Cdisplay ( )
```

o/p: i am from parent class
i am from child class

2) multi level inheritance: In this type of inheritance a class inherits from another class which in turn inherits from another class. (&)
- It is a process of inheriting property from a single parent class to single child class and single child class to grand child class.

Syntax:




```
Class grandparent :
    Pass
Class Parent (grandparent):
    pass
Class Child (Parent):
    pass
```

```
Ex: Class grandparent :
        def g display (self):
            Print ('antiques')
    Class Parent (grandparent):
        def P display (self):
            Print ('house')
    Class Child (Parent):
        def C display (self):
            Print ('job')

    obj = child ( )
    obj. g display ( )
    obj. P display ( )
    obj. C display ( )
```

o/p: antiques
house
job

5) class grand parent (self):
    def __init__ (self, land):
        self.land = land
    def g display (self):
        print ('antique')

class parent (grand parent):
    def P display (self):
        print ('house')

class child (parent):
    def c display (self):
        print ('job')
        print (self.land)

obj = child ('lace')
obj.g display()
obj.p display()
obj.c display()

O/P: antique
house
job
land lace

Constructor chaining: The process of calling a parent class constructor method into the child class constructor method in order to access the object properties of parent class.

Syntax: class name.__init__ (self, values)
    super().__init__ (values)

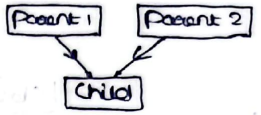
super(): super() is a built in function that returns the object that represents a parent class
• It allows to access the parent class methods & attributes in the child class

Ex: class gparent:
    def __init__ (self, land, house):
        self.land = land
        self.house = house
class parent (gparent):
    def __init__ (self, gold):
        self.gold = gold
        #gparent.__init__ (self, 'lace', 'bnk')
        super().__init__ ('lace', 'bnk')
obj = parent ('kg')
print (obj.land, obj.house, obj.gold)

O/P: lace, bnk, kg

3) multiple inheritance: When the child class is derived from more than one parent class is called as multiple inheritance.
• In this type of inheritance we have two parent classes and one child class that inherits from both parent1 and parent2 properties.

Syntax: class P1:
    ==
    class P2:
    ==
    class C(P1, P2):
    ==



Ex: class P1:
    def P display (self):
        print ('i am from parent1')
class P2:
    def C display (self):
        print ('i am from parent 2')
class child (P1, P2):
    def PC display (self):
        print ('i am from child')
obj = child()
obj.P display()
obj.C display()
obj.PC display()

O/P: i am from parent1
i am from parent 2
i am from child

with Constructor chaining:

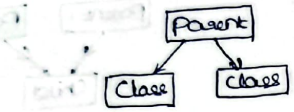
Ex: class father:
    def __init__ (self, name):
        self.name = name
class mother:
    def __init__ (self, mname):
        self.mname = mname
class child (father, mother):
    def __init__ (self):
        print ('my name is abd')
        father.__init__ (self, 'Varun')
        mother.__init__ (self, 'Varajakshi')
obj = child()
print (obj.name, obj.mname)

O/P: my name is abd
Varun Varajakshi

4) Hierarchical inheritance: inheriting (8) deriving more than one child class from a single parent class is called as hierarchical inheritance.

Syntax:
```
class P:
    pass
class C1(P):
    pass
class C2(P):
    pass
obj1 = C1()
obj2 = C2()
```

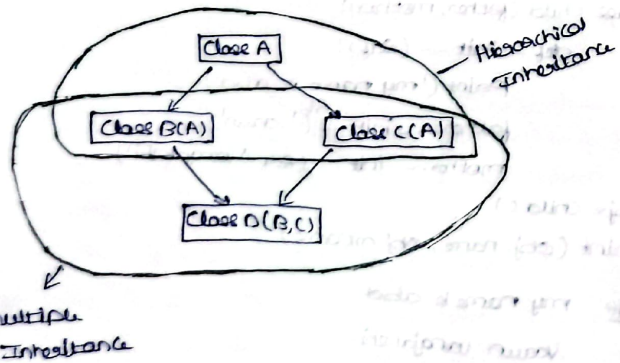
block diagram:



Ex:
```
class father:
    surname = 'gubba'
    def show(self):
        print('my family name is', self.surname)
class son(father):
    def sons(self):
        print('my name is dinga', self.surname)
class daughter(father):
    def dau(self):
        print('my name is dingi', self.surname)
obj1 = son()
obj1.show()
obj1.sons()
obj2 = daughter()
obj2.dau()
```

5) Hybrid inheritance: in this type of inheritance we can use more than one form of inheritance.

- It makes consists of all the types of inheritance & it make consists of combinations of single level, multiple level, multiple (8) hierarchical



Ex:
```
class A:
    def display(self):
        print('i am from class A')
class B(A):
    def show(self):
        print('i am from class B')
class C(A):
    def die(self):
        print('i am from class C')
class D(B,C):
    def see(self):
        print('i am from class D')
obj = D()
obj.display()
obj.show()
obj.die()
obj.see()
```

Advantages of inheritance:

1. It improves the code reusability coherence we use inheritance once we don't need to write the same code again and again.
2. Using inheritance we can inherits the features of the other classes and also add more features to the derived class.
3. By using the concepts of inheritance the program looks a simple and structure there by which makes easy to read a program by these inheritance improves readability of the code.

Method: If we use a function with in a class is called as method

- method operates a data present in the class
- The different types of methods are

1. Object method / instance method
2. Class method
3. Static method

1. Object method: object methods are written inside the class & in the methods we can access object properties (8) data members.

Ex:
```
class student:
    name = 'siva'
    def __init__(self):
        self.a = 'kumar'
        ## object method
    def die(self):
        print(self.a)
```

```
        print (self.name)
        print ('i am from object method')
obj = student()
obj.die()
```

O/P: Kumar
Siva
I am from
object method

2. Class method: A class method is a method i.e. is applied to the class and not to the object of the class.

- They have the access to the state of the class as it takes a class parameter that points to the class.
- It can modify the class that calls apply access all the instance of the class.

Syntax:
```
@classmethod
def mname (cls, v1, v2):
    ==
```

Ex:
```
class Student:
    grade = 11
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def Std_details (self):
        Print ({'name': self.name, 'age': self.age,
                'grade': self.grade})
    @classmethod
    def newgrade (cls, grade):
        cls.grade = grade
S1 = Student ('manoj', 21)
S2 = Student ('Prasad', 21)
Student.newgrade (12)
S1.Std_details()
S2.Std_details()
```

O/P: {'name': manoj, 'age': 21, 'grade' = 12}

3. Static method: Static method is a method it calls take some parameters & it calls some task on those parameters.

- using static method we can't access & we can't modify any properties.

Syntax:
```
@staticmethod
def mname (vcs):
    ==
```

- Static method is not pointing to any of the class property (or) object property. It is used to perform 'independent task' inside the class.

Ex:
```
class Student:
    grade = 11
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def Std_details (self):
        print ({'name' = S.name, 'age' = self.age, 'grade' =
                self.grade})
@ static method
    def Sage (age):
        if age>18:
            print ('Student is eligible')
        else:
            print ('Student should be complete the age 18')
S1 = Student ('manoj', 18)
S2 = Student ('Lokesh', 21)
Student.Sage (18)
S1.Std_details()
S2.Std_details()
```

O/P: Student should be complete the age 18
{'name': manoj, 'age': 18, 'grade': 11}
{'name': Lokesh, 'age': 21, 'grade': 11}

Object method: object method can be modify by using object name (or) class name.

Syntax:
1. By using object name
```
objname.methodname (arg):
```
2. By using class name
```
classname.methodname (objname, arg):
```

Ex:
```
class Bank:
    bname = 'SBI'
    ifsc = '10002sbi'
    loc = 'hebbal'
    def __init__(self, a, b, c):
        self.name = a
        self.age = b
        self.accno = c
    def display (self):
        print ('acc holder name is :' self.name)
        print ('the age the person is :' self.age)
        print ('accno' is' :, self.accno)
        print ('bank name is :' self.bname)
```

```
print('ifsc code is:', self.ifsc)
print('bank location is:', self.loc)
def rname(self, newname):
    self.name = newname

a1=Bank('Pawan', 23, 100658)
a2=Bank('Siva Kumar', 22, 220658)
a3=Bank('Lokesh', 21, 650658)
a1.rname('Pawan')
a1.display()
a2.display()
a3.display()
```

Polymorphism: Polymorphism is from the greek word whose poly means 'many' and morphism means 'forms'.

- In python polymorphism is having the ability of an object to take many forms. 'One task is performed in many forms'.
- It allows us to perform the same action in many different ways.

Ex:
```
a=5                 c='Kavya'
b=6                 d='Sharu'
print(a+b)          print(c+d)
```
O/P: 11 — O/P: Kavya Sharu

- We can achieve polymorphism in three different ways
1. operator over loading
2. method over loading
3. method over riding

1. operator over loading: operator over loading refers to a single operator which performs several operations based on the class of operands.

- operator over loading is the process of utilizing of an operator in multiple ways depending on operands.

Ex: To add two objects in the class

```
class Addition:
    def __init__(self,x,y,z):
        self.x=x
        self.y=y
        self.z=z
obj1=Addition(1,2,3)
obj2=Addition(4,5,6)
n=obj1+obj2
```
O/P: Error

In the above example interpreter produces an error because it is not possible to perform sum of two objects.

- Here we have to 'over load' '+' operator.
- For that in order to over load the '+' operator we have to use add function

Syntax: `def __add__(self,obj):`

- `n=obj1+obj2` when ever this statement is executed the add method will call.
- In all assignment and arithmetic operator we can perform

Ex: Class Addition:

```
def __init__(self, x, y, z):
    self.x = x
    self.y = y
    self.z = z
def __add__(self, new):
    x = self.x + new.x
    y = self.y + new.y
    z = self.z + new.z
    ee = Addition(x, y, z)
    return ee
obj1 = Addition(1,2,3)
obj2 = Addition(4,5,6)
n = obj1 + obj2
print(n.x, n.y, n.z)
```

O/P: 5, 7, 9

Ex: to operation over loading in comparision operators

Ex: ① class Addition:

```
def __init__(self, x, y, z):
    self.x = x
    self.y = y
    self.z = z
def __gt__(self, new):
    a1 = self.x + self.y + self.z
    a2 = new.x + new.y + new.z
    if a1 > a2:
        return True
    else:
        return False
obj1 = Addition(1,2,2)
obj2 = Addition(4,5,6)
print(obj1 > obj2)
```

O/P: false

② Class Sum:

```
def __init__(self, addi):
    self.addi = addi
def __add__(self, Py):
    return self.addi + Py.addi
ob1 = Sum(100)
ob2 = Sum(200)
print(ob1 + ob2)
```

O/P: 300

- When ever we use '+' operator for ob1 in Sum class this method is going to execute. and this add method return value and that value will be output of obj1 + ob2.
- ob1 value will be assigned to self & ob2 value will be assigned to argument

- Operator over loading internally implemented by using `def __add__(self, arg):` by using that special method.

2. Method over loading: Method over loading means creating multiple methods with the same name but with different parameters.
- using this method over loading we can perform different operations with same method name by passing different parameters (or) arguments.
- In python it doesn't support method over loading like other programming languages like java, C++.
- If we have you want to over load the methods with different arguments you will get an error. for that error ex:

```
class Sum:
    def add(self, a, b):
        print(a+b)
class addition:
    def add(self, a, b, c):
        print(a+b+c)
ob = addition()
ob.add(10,20)
```

O/P: Error

- if we want to achieve method over loading in python we need to create a methods using 'default arguments'.

Ex: ① class Sum

```
def add(self, a, b):
    print(a+b)
class addition:
    def add(self, a, b, c=0):
        print(a+b+c)
ob = addition()
ob.add(10,20)
```

O/P: 10+20=30

Method over loading by using variable length (or) arbitrary positional arguments:

Ex: ① class Method loading:

```
def display(self, *a):
    print(a)
o1 = Method Over loading()
o1.display()
o1.display(10)
o1.display(10,20)
o1.display(10,20,30)
```

O/P: ()
(10)
(10,20)
(10,20,30)

Addition of numbers by using method over loading:

Ex: ① Class Moverloading:

```
def display (self, a=None, b=None, c=None):
    if a!=None and b!=None and c!=None:
        print (a+b+c)
    elif a!=None and b!=None:
        print (a+b+c)
    else:
        print (a)
```

```
o1 = Moverloading ()
o1.display (10)
o2.display (10,0)
o3.display (10,20,30)
```

O/P: 10
10
60

Addition using arbitrary positional arguments using method over loading:

Ex: Class example:

```
def addi (self, *a):
    sum=0
    for i in a:
        sum+=i
    print ('addition of numbers:', sum)
```

```
ob= example ()
ob.addi ()
ob.addi (10)
ob.addi (10,20)
ob.addi (10,20,30,40)
```

O/P: addition of num: ()
addition of num: (10)
addition of num: (30)
addition of num: (100)

② Class example:

```
def addi (self, *a):
    sum=''
    for i in a:
        sum+=i
    print ('addition of strs:'sum)
```

```
ob= example ()
ob.addi ()
ob.addi ('venkata')
ob.addi ('venkata','Lokesh')
ob.addi ('venkata','Lokesh','lankamsetty')
```

O/P: addition of strs: ()
addition of strs: (venkata,)
addition of strs: (venkata Lokesh)
addition of strs: (venkata Lokesh lankamsetty)

③ Class Moverloading:

```
def m1(self):
    print ('i have 0 arg')
def m1 (self,a):
    print ('i have 1 arg',a)
def m1 (self,a,b):
    print ('i have 2 arg',a,b)
def m1 (self,a,b,c):
    print ('i have 3 arg',a,b,c)
```

```
ob= moverloading ()
ob.m1 (10,2,3)
```

O/P: 10,2,3

Note: if you define multiple methods of same name with different arguments in one class then we can use only the latest defined method.

- if we call the other method then it will throw an error

Constructor over loading:

Ex: Class Method over loading:

```
def __init__ (self):
    print ('i have 0 arg')
def __init__ (self,a):
    print ('i have 1 arg',a)
def __init__ (self,a,b):
    print ('i have 2 arg', a,b)
def __init__ (self, a,b,c):
    print ('i have 3 arg', a,b,c)
ob= method over loading (1,2,3)
```

O/P: i have 3 arg, 1,2,3

Operator Overloading: (Special Methods): (Magic Methods):

FB arithmetic operators:

```
def __add__():
def __sub__():
def __mul__():
def __floordiv__():
def __div__():
def __mod__():
def __pow__():
```

FB Assignment:

```
def __iadd__():
def __isub__():
def __imul__():
def __ifloordiv__():
def __idiv__():
def __ipow__():
def __imod__():
```

FB Relational:

```
def __gt__():
def __lt__():
def __ge__():
def __le__():
def __eq__():
def __ne__():
```

Method Over Riding: Method Over riding in Object Oriented programming allows us to change the implementation of a method in the child class i.e defined in the Parent Class.

- It is the ability of the child class to change the implementation of any method which is already provided in their Parent class.
- If you want to method over riding inheritance should be there so this we need to derive child class from a parent class.

Ex:①

```
Class Parent:
    def show(self):
        Print('money+gold+land')
    def massy(self):
        print('Varajakshi')
Class Child(Parent):
    def massy(self):
        print('Kamashi')

ob = child()
ob.show
ob.massy
```

O/P: money+gold+land
Kamashi

②

```
Class Parent:
    def show(self):
        Print('money+gold+land')
    def massy(self):
        Print('Varajakshi')
Class Child(Parent):
    Super().massy()
    def massy(self):
        Print('Kamashi')

ob = child()
ob show
ob massy
```

O/P: money+gold+land
Varajakshi
Kamashi

Encapsulation: Encapsulation is the process of clapping up a variable and methods into a single entity.

- It provides security by hiding the data from the out side class.
- The need of encapsulation is

1. Encapsulation allows us to restrict accessing a variables and methods directly.
2. It prevents data modifications by creating private data members and private methods inside the class.

Access modifiers: Access modifiers are used to restrict the access of variables and methods out side the class.

- In Python we can achieve it provides three types of access modifiers

1) Public
2) Private
3) Protected

1) public access modifiers: The public members of a class can access to every one. So they can be accessed from out side the class & inside the class and also by their child class too.

① accessing public attributes with in the class.

```
Class A:
    def __init__(self):
        self.x = 25
    def test(self):
        print(self.x)
a = A()
a.test()
```

O/P: 25

② accessing public attributes inside with in the child class

```
Class A:
    def __init__(self):
        self.x = 25
Class B(A):
    def test(self):
        print(self.x)
ob = B()
ob.test()
```

O/P: 25
25

③ accessing public attributes out side the class

```
Class A:
    def __init__(self):
        self.x = 25
a = A()
print(a.x)
```

O/P: 25

① Accessing public method outside the class

```
Class A:
    def m(self):
        print('i am from public method')

    def show(self):
        self.m()

a = A()
a.show()
```

② Accessing public method inside outside in the child class

```
Class A:
    def m(self):
        print('i am from public method')

Class B(A):
    def show(self):
        self.m()

a = B()
a.show()
```

③ Accessing public method out side the class

```
Class A:
    def m(self):
        print('i am from public method')

a = A()
a.m()
```

2. Protected access modifiers: protected members of a class can be accessed by outside class and also access to their sub classes.
- No other class can access these protected members
- In Python we define protected access modifiers with _ single underscore. This prevents it usage by out side the class.

Note: In Python Protected access modifiers doesn't perform this functionalities.
- In python protected members can be access out side that class also

① Accessing protected attribute inside a class

```
Class A:
    def __init__(self):
        self._x = 50

    def test(self):
        print(self._x)

a = A()
a.test()
```

O/P: 50

② Accessing protected attribute inside the child class

```
Class A:
    def __init__(self):
        self._x = 50

Class B(A):
    def test(self):
        print(self._x)

a = B()
a.test()
```

O/P: 50
50

③ Accessing protected attribute out side the class

```
Class A:
    def __init__(self):
        self._x = 50

a = A()
print(a._x)
```

O/P: 50

3. Private access modifiers: private members of a class are only access outside in a class. in Python private member can be defined by using a prefix '__ double underscore'.
- In private modifiers are cannot access out side the class and inside the child class.

Note: We can access private attributes out side the class outside with the help of 'data mangling (or) name mangling'.
- Every member outside with '__ double under score' could be changed to "object._classname__variable"

① Accessing attribute inside a class

```
class A:
    def __init__(self):
        self.__x = 25
    def show(self):
        print(self.__x)

a = A()
a.show()
```

O/P: 25

② accessing attribute inside the child class

```
Class A:
    def __init__(self):
        self.__x=25
Class B(A):
    def __init__(self):
        print(self.__x)

a = B()
a.show()
```

O/P: Error

③ accessing attribute out side the class

```
class A:
    def __init__(self):
        self.__x = 25

a = A()
print(a.__x)
```

O/P: Error

using method:

① accessing method inside the class

```
Class A:
    def __m(self):
        print('hi i am private method')

    def test(self):
        self.__m()

ob = A()
ob.test()
```

② accessing method inside the child class

```
class A:
    def __m(self):
        print('i am private')

class B(A):
    def test(self):
        self.__m()

ob = B()
ob.test()
```

O/P: Error

③ accessing method outside the class

```
class A:
    def __m(self):
        print('i am private')

ob = A()
print(ob.__m)
```

O/P: Error

- To access private members outside the class we have to use data mangling.

```
class A:
    def __init__(self):
        self.__x = 50

ob = A()
print(ob._A__x)
```

O/P: 50

```
class A:
    def __m(self):
        print('i am private')

ob = A()
print ob.m()
```

O/P: i am private

Setter and getter method:

Syntax: `def Set Variable (Self, Variable name):`
`Self.Variable name = Variable name`

Ex:
```
class encap:
    __a = 10
    def SetA(self, b):
        self.__a = b
    def getA(self):
        return self.__a

ob = encap()
ob.SetA(30)
Print(ob.getA())
```

Setter method: A setter method used to set the value of the property.

- We can set the value with the method declaration:

Syntax: `def Set Variable (Self, Variable name):`
`Self.Variable name = new Variable name`

Getter method: getter method is used to retrieve a value of the property.

- We can retrieve a value with a method declaration

Syntax: `def get Variable name (Self):`
`return Self.__Variable name`

Abstraction: Abstraction is the process of hiding the implementation and showing (B) using the functionalities.

- In python to achieve abstraction we need to import abstract base class and abstract method.

abstract method: When even we don't know about implementation but still we can declare a method. Such type of methods are called as abstract method.

- abstract method as only declaration but not implementation.
- In python we can declare abstract method by using '@abstract method' decorator.
- Abstract method decorator present in 'abc module'.
- Hence mandatorily we should import abc module.

Ex: from abc import abstract method
```
class Animal():
    @abstract method
    def getNoOfLegs(self):
        pass
```

- child class are responsible to provide implementation for parent class abstract

abstract class: When even implementation of a class is not complete such type of 'Partially implemented classes' are called as abstract class.

- Every abstract class in python should be a child class of 'ABC' class which is present in abc module.
↓
Abstract base class

Ex: from abc import ABC, abstract method
```
class Animal(ABC):
    @abstract method
    def getNoOfLegs(self):
        pass
```

Note: 1. The advantage of declaring abstract method in parent class we can provide guidelines to the child class such that which method compulsorily they should implementation.

2. If a class contains atleast one abstract method then and if we are extending ABC class then instantiation i.e creating objects not possible.

* If abstract class with abstract method creating object is not possible.

3. If we are creating a child class of abstract class then for every abstract method of parent class compulsorily we should provide implementation in the child class. otherwise child class is also abstract and we cannot create a object. for child class.

Note 4: We can create animal class object because it is a concrete class and it does n't contain any abstract method.

```
Ex: class Animal ():
        pass
    Ob= Animal ()          O/P: Empty
```

Note 5: We can create an object even it is abstract method is present because it is a not an abstract class

- If a class contains atleast one abstract method and if we are extending ABC class then object creation is not possible but the below example contains only abstract method so it will not print error.

```
Ex: from abc import abstract method
    class Animal ():
        @abstract method
        def get no of legs (self):      O/P: Empty
            pass
    ob= Animal()
```

Note 6: We can create an object even it is abstract class because it doesn't contains any abstract method.

- Abstract class can contain zero number of abstract methods

```
Ex: from abc import ABC, abstract method
    class Animal (ABC):
        def get no of legs (self):
            pass                    O/P: Empty
    Ob= Animal()
```

Note 7: Type Error can't instiate abstract class from animal with abstract method get no of legs.

```
Ex: from abc import ABC, abstract method
    class Animal (ABC):
        @abstract method
        def get no of legs (self):
            pass
    Ob= Animal()                    O/P: Error
```

Note 8:

```
from abc import ABC, abstract method
class Animal (ABC):
    @abstract method
    def get no of legs (self):
        pass
class Tiger (Animal):
    def get no of legs (self):
        return 4                    O/P: 4
class Snake (Animal):               0
    def get no of legs (self):
        return 0
Ob= Tiger()
print (Ob.get no of legs())
Ob1 = Snake()
print (Ob1.get no of legs())
```

Note 9:

```
from abc import *
class test (ABC):
    @abstract method
    def m1 (self):
        pass
    @abstract method
    def m2 (self):
        pass
class Subtest (test):
    def m1 (self):
        print('i am from m1')
    def m2 (self):
        print('i am from m2')
Ob= Subtest()                       O/P: i am from m1
Ob.m1()                                  i am from m2
Ob.m2()
```

Note 10:

```
from abc import *
class test (ABC):
    def m1 (self):
        pass
```

```
@abstract method
def m2 (self):
    pass
class subtest (test):
    def m2 (self):
        print('i am from m1')
ob = subtest ()                    o/p: i am from m1
ob.m2 ()
```

- If a class contains both abstract and non abstract method the child class is responsible to provide implementation only to abstract method.

Note 1: Type error can't instiate class subtest with abstract method m2 ().

```
Ex: from abc import *
    class Test (ABC):
        @abstract method
        def m1 (self):
            pass
        @abstract method
        def m2 (self):
            pass
    class subtest (Test):
        def m1 (self):
            print ('i am from m1')
    ob = subtest ()
    ob.m1 ()
    ob.m2 ()
```

o/p: Error

Concrete class: A fully implemented class is called as concrete class.

Interface: An abstract class contains only abstract method such type of abstract class is called as 'interface'.

- Interface acts as service requirement specification
- If user don't knows anything about implementation & just user have requirement specification then user should go to interface.

```
Ex: from abc import *
    class Emp(ABC):
        @abstract method
        def sal (self):
            pass
        @abstract method
        def idemp (self):
            pass
        @abstract method
        def calking days (self):
            pass
    class pyspiders (Emp):
        def sal (self):
            print ('employe sal is 2000')
        def idemp (self):
            print ('employe id is 12345')
        def calking days (self):
            print ('NO calking days')
    ob = pyspiders ()
    ob.sal ()
    ob.idemp ()
    ob.calking days ()
```

Interface

o/p: employe sal is 2000
employe id is 12345
NO calking days

Decorators: Decorators are functions that take another function as a argument and extends its behavior (or) functionalities without modifying a original program.

Syntax:
```
def funcname (arg):
    def inner (arg):
        ==
        ==
        return
    @funcname
    def funname (arg):
        ==
        ==
    funname( )
```

Ex:
```
def great (name):
    print ('hi', name, 'Id')
great ('Naveen')
great ('pavani')
great ('pallavi')
```
o/p: hi Naveen Id
hi pavani Id
hi pallavi Id

- Without modifying the great function we are providing extended functionality to the great function with the help of decorators.

```
def decB (func):
    def inner (name):
        if name == 'pallavi':
            print('happy birthday pallavi')
        else:
            func(name)
    return inner
@decB
def great (name):
    print ('hi', name, 'Id')
great ('Naveen')
great ('pavani')
great ('pallavi')
```
o/p: hi Naveen Id
hi pavani Id
happy birthday pallavi

Ex: ②
```
def div (func):
    def inner (a,b):
        if b==0:
            return 'hey we cannot divisible by 0'
        else:
            return func (a,b)
    return inner
@div
def division (a,b):
    return a/b
print (division (10, 2))
print (division (100,2))
print (division (10,0))
```
o/p: 5.0
50.0
hey we cannot divisible by 0

Decorators chaining: Declaring more than one decorators inside a function is called decorators chaining

Ex:
```
def decB1(func):
    def inner (name):
        print ('i am from decB 1')
        func(name)
    return inner
def decB 2 (func):
    def inner (name):
        print ('i am from decB 2')
        func(name)
    return inner
@decB 1
@decB 2
def great (name):
    print ('hi', name, 'welcome to decBators')
great('Lokesh')
```
o/p:
```
i am from decB 1
i am from decB 2
hi Lokesh welcome to decBators
```

- In decBators chaining execution always starts from first to last.

Comprehensions

- Comprehensions is the process of creating a new sequence from an existing sequence.
- Comprehensions make an easy way of creating an output list, tuple, dict by applying expressions & conditions.
- In Python we can apply comprehensions to different sequences
  1. List comprehensions
  2. Set comprehensions
  3. dict comprehensions
  4. generators comprehensions

1. **List comprehensions:** List comprehensions create a list from the existing sequence

Syntax: ① var = [Expression for var in sequence] ② var = [Expression for var in sequence if condition]

Ex: ① l = [1,2,3,4,5]
li = [ix i for i in l]
print(li)
o/p: [1,4,9,16,25]

- Each time the variable (i) read one value from sequence after reading one value from the sequence it will apply expression (ixi). After that result will be stored in the list.

② l = [1,2,4,5]
li = [i+10 for i in l]
print(li)
o/p: [11,12,14,15]

③ l = [1,2,4,5,6,7,8]
li = [i for i in l if i%2==0]
print(li)
o/p: [2,4,6,8]

2. **Set comprehensions:** It will create a set from the existing sequence.

Syntax: ① var = {Expression for var in sequence} ② var = {Expression for var in sequence if condition}

Ex: ① l = {1,2,4,5}
li = {i+10 for i in l}
print(li)

② l = {1,2,4,5}
li = {i+10 for i in l}
print(li)

③ l = {1,2,4,5,6,7,8}
li = {i for i in l if i%2==0}
print(li)

3. **Dict comprehensions:** It will create a dict comprehension from the given sequence

Syntax: {key:value for var in sequence}

Ex: l = {1,2,4,5}
li = {i:ix i for i in l}
print(li)

o/p: {1:1, 2:4, 4:16, 5:25}

② l = [1,2,4,5]
ll = ['a','b','c','d']
d = {key:value for key,value in zip(l,ll)}
print(d)

o/p: {1:'a', 2:'b', 3:'c', 4:'d'}

4. **Generator comprehensions:** is a one line specification for defining generator in Python.

Ex: ① l = [1,2,4,5]
li = (ix i for i in l)
print(li)
o/p: <generator object <gen expr> at 0x000>

② l = [1,2,4,5]
li = (ix i for i in l)
print(li)
for i in li:
print(i)

o/p: 1
4
16
25

It is possible to print the o/p in generator comprehension but it is not in any of the format.

Modules: A group of functions and variables saved to a file is called as module.

- module helps in reusing of code instead of writing the same code in different files.
- In Python module makes the program more readability.
- To use code from one module to another module we can make use of the keyword import (s) from.
- Syntax to import one module to another module
  1. import module name
  2. import module name as aliasing name
  3. from module name import var1, var2
  4. from module name import *

packages: collection of modules

- The different types of modules are
  1. user defined module
  2. Pre defined / [illegible] built in module.

1. **User defined module**: These are the modules defined by (8) created by users in order to reuse the code.

cal.py
```
n=10
m=20
def add(a,b):
    return a+b
def sub(a,b):
    return a-b
def mul(a,b):
    return a*b
def div(a,b):
    return a/b
```

Ex: ① import cal
```
print(cal.add(4,8))
print(cal.mul(4,8))
print(cal.sub(8,4))
print(cal.div(10,2))
```

② import cal as c
```
print(c.add(4,8))
print(c.mul(4,8))
print(c.sub(4,8))
print(c.sub(8,4))
```

③ from cal import add, mul
```
print(add(4,5))
print(mul(4,5))
```

from cal import add as a, mul
```
print(a(4,5))
print(mul(4,5))
```

④ from cal import *
```
print(add(4,5))
print(mul(4,5))
```

2. **Built in module**: These are the modules which are already present in the Python when ever user want to use no need to create manually just we have to import and we can reuse.

Ex: Math module
Random module
os module
Operator module
abc module
time module

math module: It is used to perform mathematical operations such as finding a square root, finding a cube root etc.

Ex: import math
```
print(math.sqrt(5))   # 2.23
print(math.cbrt(9))   # 2.08
print(math.factorial(6))  # 720
print(math.ceil(5.6))  # 6
print(math.floor(5.6))  # 5
print(math.trunc(5.6))  # 5
print(math.pow(5,2))  # 25
print(math.remainder(100,2))  # 50.0
```

Random module: This module is used to generate random values like OTP's, Captcha's etc.

Random(): using this function it will generate the decimal values between 0 and 1 and it will not include 0 and 1.

Ex: from random import *
```
for i in range(4):
    print(random())
```
O/P: 0.48409400
0.71463586
0.09723459

randint(): It will generate the values in the form of int data type.

Ex: from random import *
```
for i in range(4):
    print(randint(1,10))
```
O/P: 8
3
4
8

. It includes the given values also

uniform(): It will generate the float values and it will not include the specified values.

Ex: from random import *
```
for i in range(4):
    print(uniform(1,5))
```
O/P: 4.226391995
3.552121038
3.761668548

randrange(): It is used to generate random integer values between the start index and end index.

Syntax: 1) import random
random.randrange(SI, EI, step)

from random 2) import *
randrange(SI, EI, step)

Ex: 1) from random import *
```
for i in range(10):
    print(randrange(10))
```

2) from random import *
```
for i in range(10):
    print(randrange(1,4,2))
```

choice(): choice is used to select a random element from a specified sequence.

Ex: from random import *
```
l1=['umesh','pawan','vinod','sachin']
for i in range(10):
    print(choice(l1))
```

operator module:

```
from operator module *
print (add(2,3))    // 5
print (sub(3,2))    // 1
print (mul(3,2))    // 6
print (floordiv(2,5)) // 0
print (mod(12,3))   // 0
```

OS module: This module is used to know the current working directory & to create new directory & to remove any directory & to rename a directory

i) To know the current working directory:

```
ex: from os import *
    c = getcwd()
    print('my current working directory is', c)
```

ii) To create a subdirectory in a current working directory:

Syntax: mkdir('filename')

Note: if you try to create same directory again and again it will through an error.

iii) To create a particular sub directory in a subdirectory:

```
ex: from os import *
    mkdir('Python module/os module')
    print('new directory created')
```

To create multiple subdirectory:

```
ex: from os import *
    makedirs('Pymod/osm/opm')
```

iv) To remove individual directory:

```
Ex: from os import *
    rmdir('Python module/os module') // os module removed
```

rmdir: this function remove the directory i.e. recently created.

removedirs: using this function we can remove multiple directories at a time

```
Ex: from os import *
    removedirs('Pymod/osm/opm')
```

rename: using this function we can rename the directory name.

```
Ex: from os import *
    rename('python module', 'pmodule')
```

v) To run any other program in a present python program we can make use of 'system'.

```
Ex: from os import *
    system('calc')
```

Time module: This module provides a function to perform time related operations like to display the current time, to display the month & sleeping for specified number of seconds etc.

```
Ex①: from time import *
     print(localtime())
```

② from time import *

```
a = localtime().tm_hour
b = localtime().tm_min
c = localtime().tm_sec
print(f'{a}:{b}:{c}')
```

o/p: 13:51:17

③ from time import *

```
a = localtime().tm_day
b = localtime().tm_mon
c = localtime().tm_year
print(f'{a}:{b}:{c}')
```

o/p: 22:8:2023

sleep(): sleep function will delay the execution of the current statements for the given number of seconds.

```
Ex: from time import *
    print('before statement')
    sleep(5) -> we should provide only in seconds
    print('after sleep')
```

File handling: File handling is the process of performing operations on files like opening a file, reading a file, and writing on it & closing the files.

- files are used to store the data temporarily & permanently.
- A file is the collection of data on a disk as a one unit and identified by file name.
- The different types of files are 1. text file
  2. Binary file
  3. CSV files
- To perform any file handling operations we should follow three steps
  1. Open the file
  2. Perform some operations on the file
  3. Close the file

1. **Open()**: It is used to open the files with file handling modes such as read, write, append.

   Syntax: Variable name = open('filename', 'mode')

2. **Read()**: It is used to read the files and we need to open the existing file with 'r' mode.

   Syntax: Variable name = open('filename', 'r')

- The different modes for reading the file are read(), readline(), readlines()

  Ex: fn = open('cal.py', 'r')
  print(fn.read()) → print(fn.readlines()) → It will print all lines
  print(fn.readline()) → it will print only first line

3. **Close()**: It is used to close the file. Once we close the file we can't perform any operations on the closed file.

   Syntax: Var name.close()

   Ex: ① fn = open('cal.py', 'r')
   print(fn.read())
   fn.close()

   ② fn = open('cal.py', 'r')
   fn.close()
   print(fn.read()) → value Error
   O/P: I/O operation on closed file

- Once we open the file and it is not closed properly then system resources will wasted and some other users may get error.

Read(): If the specified file is present in the directory then it will display the data in the existing file.

- If the specified file is not present in the directory then it will throw an error like 'file not found.'
- **read()**: by this function we can read entire data given in the specified file
- **readline()**: by this function we can read only the first line of the specified file.
- **readlines()**: by this function we can read each line seperately, and data is present in the list format

```
fn = open('c.py','r')
print(fn.readlines()[0])
print(fn.readlines()[3])
fn.close()
```

- In read cases cursor will always points in the beginning of the file.

Write(): Write mode is used to write the data in the specified file. This mode over write the existing files.

Syntax:
```
var = open('filename','w')
var.write('data')
```

Ex:
```
fn = open('c.py','w')
print(fn.write('hi from py spiders hobbai'))
fn.close()
```

append(): using this mode data will be added to the last.

- append() will not over write the data in the given file.
- If the file is not exist then it will be create the file then data will be appended.

Note: In write mode if the specified file is not exist then it will create the file.

Syntax:
```
var = open('filename','a')
var.write('data')
```

Ex:
```
fn = open('c.py','a')
print(fn.open('hi from j spiders hobbai'))
fn.close()
```

Exclusive mode: This mode creates a new file and write the data in the newly existing file

- If we try to write a data in the existing file it will throw an error called 'file exists error.'

Syntax:
```
var = open('filename','x')
var.write('data')
```

Ex:
```
fn = open('c1.py','x')
print(fn.open('hello'))
fn.close()
```

r+ mode: using this mode one can read the file and one can write the data to the existing file.
- Reading and writing both the operations can be done only to the existing files.
- In this type of mode data present in the file will not be over side.

Syntax: var = open('filename', 'r+')

Ex: n = open ('mod.py', 'r+')
n.read()
n.write ('pqr\spiders')
n.close()

w+ mode: using this mode one can write and reading the files
- In this mode the data present in the file will be over written by the new data then one can read
- w+ mode can be used to both existing and non existing file.

Syntax: var = open('filename', 'w+')

Ex: n = open ('car.py', 'w+')
n.read()
n.write ('pqr\spiders')
n.close()

a+ mode: using this mode one can append and read the files.
- One can use this mode to both existing and non existing file.
- Data will be over written if will append the data with previous data.

Syntax: var = open('filename', 'a+')

Ex: n = open('car.py', 'a+')
n.write ('pqr\spiders')
n.read()
n.close()

- The various properties of the file object are
1. var.mode 0/p: It is used to check the mode of the file (read, write, append, r+, w+, a+)
2. var.name: It is used to specify the file name in which user currently using
3. var.readable(): This function is used to check whether the given file is readable or not
- It gives output in the form of boolean expression
- If the given file is in r, r+, w+, a+ these modes then the output is true. Other wise false.
4. var.writable(): using this user can check the given file is writable or not.
- It gives the output in the form of boolean expression.
5. var.closed: using this one can check whether the file is closed or not.
- It gives the output in the form of boolean expression.
- If the file is closed the o/p is true else false.

Ex: n = open ('car.py', 'a')
n.read()
n.write ('from hebbal')
print (n.mode) // a
print (n.name) // car.py, f.py
print (n.readable()) // False
print (n.writable()) // True
print (n.closed 0/p) // False

write(): If you want to write any string type of data one have to use write()

writelines(): If you want to write a sequence of elements then one have to use writelines():

6. with: with statement works as with open function to open a file and with statement closes the file with out telling & with out using close()

Syntax: with open ('filename', 'mode') as alising name :

7. tell(): This function is used to know the current position of the cursor.

Syntax: var.tell()

8. seek(): It is used to change the position of cursor to a specific position.

Syntax: var.seek()

Ex: with open ('car.py', 'w') as a:
print ('before seek', a.tell())
a.seek(5)
print ('after seek', a.tell())
a.write ('welcome to python class')

CSV files: CSV means comma seperated values

- Here a file is a text file that stores the data i.e., seperated by comma's.
- CSV files helps to store data in simple organised ways which can be directly used in data bases and spread sheets.
- CSV files are opened same as text files, but if you want you use CSV first one have to 'import CSV'

Writer(): This function in CSV module returns 'writer object that converts the data into strings and store the data in a file object.

Reader(): It is used to read the file which returns and an iterable readable object.

Write row(): Using this function one can write a single row at a time into row, CSV file.

Ex: import CSV
```
with open('emp.csv','w') as f:
    w = csv.writer(f)
    w.writerow(['empno','ename','sal','deptno'])
    n = int(input('enter the number of employee:'))
    for i in range(n):
        empno = eval(input('enter the empno:'))
        ename = input('enter the empname:')
        sal = eval(input('enter the sal:'))
        deptno = eval(input('enter the deptno:'))
        w.writerow([empno, ename, sal, deptno])
print('employee data in CSV file')
```

Ex: Readable CSV file
```
import csv
f = open('emp.csv','r')
r = csv.reader(f)
d = list(r)
for i in d:
    for j in i:
        print(i,'\t',end='')
    print()
```

Binary files: These are the type of files which stores the information in the format in which the information is present in the memory.

- Examples of binary files are images, videos and compiled files.
- To handle the binary files we should follow these steps.

1. open the file
2. perform operations
3. closing the file

1. open the file: In order to perform any operation, first we have to open the file.

Syntax: var = open('filename','mode')

(or)

with open ('filename','mode') as aliasing name:

file mode: A file mode is used to specified the type of the operation.

- Binary files modes are
  - rb → read
  - wb → write
  - ab → append
  - r+b → read & write
  - w+b → write & read
  - a+b → append & read

Pickle in Python: Pickle is used in python to read & write the objects like list, tuple and dict etc.,

- type of objects in to a file.
- If one want to use pickle first one have to import pickle
- functions related to pickle module.

1. dump(): This function is used to write an object into a binary file.

Syntax: pickle.dump('Python obj to be written', file object)

dump → convert python object to machine readable form

2. load(): This function is used to load an object to the binary file.

Syntax: var = pickle.load(file object)

load ← convert machine readable format to human readable form

Ex: 1 Binary files To write the program

```
import pickle
l=[1,2,3,4,5]
x= open ('pic.dat','wb')
pickle.dump (l,x)
print ('data written in binary format')
x.close()
```

To read the program

```
import pickle
x= open('pic.dat','rb')
y= pickle.load(x)
print(y)
x.close()
```

2. To write and read in a single program

```
import pickle
l= eval( input('enter the list'))
x=open('pick.dat','w+b')
pickle.dump(l,x)
print ('data written in binary format')
x.seek(0)
y= pickle.load(x)
print(y)
x.close()
```

- Pickling: The process of converting the python object into a byte string is called pickling.

Unpickling: The process of converting a byte string back to its original form is called as unpickling.

Exception Handling: It's a phenomenon where unstructured events occurs while executing in the program which stop the flow of execution and throws Run time error, To overcome this Run time error we use exception handling where the output will be the exception message.

- There are 3 types of exceptions:

1. Specific 2. general 3. Default

1. Specific exception handling: in these exception handling we know the error name.

- we can use n number of exceptions in specific exception handling.
- we use try and except to achieve the execution.

Syntax:

```
try:
    [ ]
except error name:
    [ ]
except error name:
    [ ]
```

Ex: ① a = int(input('enter the num:'))
b = int(input('enter the num:'))
c = a/b
print(c)

O/P: 10/2 = 5.0
10/0 = zero division error
10/'hii' = Value error

② def div():
    try:
        a = int(input('enter the num:'))
        b = int(input('enter the num:'))
        c = a/b
        print(c)
    except Zero division error:
        print('b should n't equal to zero')
    except Value error:
        print('b always should be integer')
div()

O/P: 10/2 = 5.0
10/0 = b should n't equal to zero
10/'hii' = b always should be integer

2 (General Exceptions): In these exception case don't know the exact name to over the exceptions case use the general exception.
- General exceptions don't allow keyboard problems.

Syntax:
```
try:
    [ ]
except exception:
    [ ]
```

Ex: def div():
```
    try:
        a = int(input('enter the num:'))
        b = int(input('enter the num:'))
        c = a/b
        print(c)
    except exception:
        print('problem solved....')

div()
```
O/P: 10/2 = 5.0
10/0 = problem solved...

```
def div():
    try:
        i=1
        while True:
            print(i)
            i+=1
    except exception:
        print('problem solved!...')
div()
```
O/P: in loop — break the loop - ctrl+c
keyboard Interrupt

Finally: Finally is mainly used to clean up the resources
- Finally block executes in both the cases of try block or even as except block. it means if there is an exception & if there is no exception in both of these cases finally block will execute.

Syntax: try = except exception handling code = finally:

3. Default Exception: To overcome all keyboard interruption use use default interruption (BI) exception.

Syntax:
```
try:
    [ ]
except:
    [ ]
```

Ex: def div():
```
    try:
        a = int(input('enter the num:')) & i=1
        while True:
            print(i)
            i+=1
    except:
        print('problem solved....')

div()
```
O/P: in loop
problem solved.....

Recursion: It is a phenomena of calling a function by itself until the termination code become True.
- Use use recursion to avoid looping to do the operation.
- With the help of recursion it is possible to increase the efficiency of the program.
- Prog to find the factorial of a given number using recursion

```
def fact(n):
    if n==0 & n==1:
        return 1
    return n*fact(n-1)
print(fact(5))
```
= 5x4x3x2x1
= 120

n=5: 5 x fact(n-1) → 5 x fact(4)
n=4: 4 x fact(3)
n=3: 3 x fact(2)
n=2: 2 x fact(1)

Note: By executing recursion problem general code provide 1000 block of memories defaultly.

```
import sys
print(sys.getrecursionlimit())
```
O/P: 1000

- How to set recursion block size

```
import sys
sys.setrecursionlimit(2000)
def fact(n):
    if n==0 & n==1:
        return 1
    return n*fact(n-1)
Print(fact(1200))
```

- How to convert looping program to get recursion program
  - Initialization of all the looping variable should done in formal argument
  - Case should have a termination code which is opposite to the looping condition.
  - Ca should have a condition by which return should give the o/p to the program.
  - Incremental, decremental should be done in the recursion argument.
- Prog to find all the upper case alphabets using while loop.

```
a = 'Lokesh'
out = ''
i=0
while i<len(a):
    if 'A'<=a[i]<='Z':
        out+=a[i]
    i+=1
Print(out)
```

→ Normal Program

```
def upper(a, out='', i=0):
    if i>=len(a):
        return out
    if 'A'<=a[i]<='Z'
        out+=a[i]
    return upper(a,out,i+1)
Print(upper(input('enter the string:')))
```

→ By using recursion

Exception handling:

Ex:

```
try:
    Print('Hi')
    Print(100/2)
    Print('good morning')
except ZeroDivisionError:
    Print('except block')
finally:
    Print('final')
Print('bye')
```

o/p: Hi
50.0
good morning
final
bye

```
try:
    Print('Hi')
    Print(100/0)
    Print('good morning')
except ZeroDivisionError:
    Print('except block')
finally:
    Print('final')
Print('bye')
```

o/p: Hi
except block
final
bye

```
try:
    Print('Hi')
    Print(100/ten)
    Print('good morning')
except ZeroDivisionError:
    Print('except block')
finally:
    Print('final')
Print('bye')
```

o/p: Hi
final

Recursion: fibonacci series

```
def fibo(i):
    if i<=1:
        return i
    else:
        return fibo(i-1)+fibo(i-2)
for i in range(10):
    Print(fibo(i))
```

} Base case → To Terminate the Condition
Ca use base case.

o/p: 0112358

There are two types of exceptions

1. In-built exception
2. User-defined exception (Customised exception).

Built-in: The exception which are raised automatically by python whenever a particular event occurs.

Ex: Zero division error
Name error
Value error etc.

Customized exception: In python a user can create their own exception class in order to raise the exception based on the requirements.

- A customized exception class should inherit the properties from base exception class.

Syntax:
```
class classname (exception):
    pass
```

Ex: ①
```
class invalidage error(exception):
    def __init__(self, msg):
        self.msg = msg

try:
    age = int(input('enter the age:'))
    if age >= 18:
        print('you are eligible to voting')
    else:
        raise invalidage error('you are not eligible')
except exception as e:
    print('error', e)
```

op: enter the age: 21
you are eligible to voting

enter the age: 17
error you are not eligible

②
```
class tooeasily exception(Exception):
    pass
class too late exception(Exception):
    pass
age = int(input('enter a age'))
if age > 50:
    raise too late exception('you are too late')
elif age < 18:
    raise to easily exception('you are too easily')
else:
    print('you are eligible')
```

Assertion in Python: Assertions are used to evaluate the expression to check the condition.

- To perform assertion we have to make use of the keyword 'Assert'.
- Assertion check a given condition if the given condition is true. Then it will not raise any kind of exceptions.
- If condition is false then it will raise an assertion error.

Syntax: Assert <condition>, 'error msg' (optional)

Ex:
①
```
try:
    a = int(input('enter a number:'))
    assert a%2==0, 'a is not even'
    print(a)
except exception as e:
    print('error', e)
```

②
```
try:
    age = int(input('enter the age:'))
    assert age >= 18, 'age is not eligible'
    print(age)
except exception as e:
    print('error', e)
```

Types of errors: There are two types of errors
1. Syntax error 2. Run time error

1. Syntax error: The error which occurs because of invalid syntax (s) any spelling mistakes such type of errors are called as syntax error.

- Programmers are responsible for these kind of errors.

Ex: ①
```
a=10
b=20
if a>b        # missed semicolon
    print(a)
```
② print 'Hello' # parenthesis
③ Pelin ('Hi') # spelling

2. Run time error: Run time errors are also called as exceptions. While executing the program at run time if something goes wrong because of end-user input (s) due to some wrong program etc. Such type of errors we call it as run time error.

Ex: File not found error
Zero division error
Name error etc.

Iterable: Iterable is an object from that one can access the elements sequentially without knowledge of memory.

- Iterable object could provide a single element from the collection of instead of iterating on the elements at a time. By this operation are memory efficient and type efficient.
- Iterable object can be created by a iter method `iter()`.
- To access elements from the iterable object sequentially one can use `next()`: (8) -- next--().

Ex: `l = [1,2,3]`

```
v = l.__iter__()
i1 = v.__next__()
print(i1)
i2 = v.__next__()
print(i2)
i3 = v.__next__()
print(i3)
i4 = v.__next__()
print(i4)
```

O/P: 1 2 3 Stop Iteration

```
num = [1,2,3,4]
i = iter(num)
while True:
    try:
        e = next(i)
        print(e)
    except StopIteration:
        break
```

O/P: 1 2 3 4

```
n = [1,2,3,4]
i = iter(n)
i1 = next(i)
print(i1)
i2 = next(i)
print(i2)
i3 = next(i)
print(i3)
i4 = next(i)
print(i4)
```

O/P: 1 2 3 4

Note: If one try to give next() to the fifth time on the iterable object it raises an exception called 'StopIteration'.

Iterable: The object with which one can iterate over & one can traverse through a loop is called as iterable.

Generator: Generator are used to generate the values in a sequence.

- Generator function returns generator object which acts as an iterable
- To represent generator one use the keyword, called 'yield'.

Ex:
```
def gen():
    yield 'a'
    yield 'b'

g = gen()
print(type(g))
print(next(g))
print(next(g))
```

O/P: <generator>
a
b

(2)
```
def fib():
    a,b = 0,1
    while True:
        yield a
        a,b = b,a+b

for n in fib():
    if n>100:
        break
    print(n)
```

O/P: 0 1 1 2 3 5 8

reduce(): reduce is the function which is used to reduce the collection into a single value.

- reduce() is very handy to processing iterables without explicit for loops.

→ reduce(function, collection) **Syntax:**

- reduce() in Python is a part of functools (functools) module which has to be imported before calling the function in a program.

- from functools import reduce (8) *

Ex:
```
from functools import reduce
def product(x,y):
    return x*y
a = reduce(product,[1,2,3])
print(a)
```

O/P: 6

(2)
```
from functools import reduce
a = [1,2,3,4,5]
b = reduce(lambda x,y: x+y, a)
```