

Python: General purpose high level programming language

created by **Guido Van Rossum** from Netherlands(National Research Institute)
python was created in **1989** but went on public on **1991(20/feb/1991)**

Python word is derived on **Monty python's circus Tv show** of BBC from 1969 to 1974 not on the snake.

Why Python: General purpose high level programming language

1. Feature of Python

1. Simple and easy to Learn
2. freeware ← -- Free of charge to use
3. Opensource ← -- source code available and can be improved. Due flavors like Cython, Jython, Pypy(internal JIT compiler), Ironpython(c#), Rubypython, Anacondapython(Bigdata)
4. Portability ← --- can be moved easily across different platforms
5. Platform independent ← ---- Same codes run in all platforms/ write once run everywhere
6. Dynamically typed programming language ← -- x= 10 #data type is given automatically (No need to explicitly declare data type). type(x) ← -- class <int>
7. Embedded ← --- use python codes to others languages
8. Extensible ← -- use codes from other languages Cython,Jython
9. High level language ← ---- like english
10. Extensive library ← -- import and use libraries instead of coding from scratch(Use not implement)
11. procedure and Functional ← --- from c language
12. Object Oriented ← ----- from C++
13. Scripting languages features ← ----- from Perl and shell scripting
14. Modular ← -- from modula 3 (module is a collection of functions, collection of modules is a Package, collection of packages is a Library)
15. Syntax from C and ABC

(collection of statement is a **function**, a collection of functions is a **module** , collection of modules is a **Package**, collection of packages is a **Library**)

Limitation of Python

Interpreter ← --Performance wise Low (Interpreter can only see one line at a time)
Not Much used in Mobile Apps ← -

Where is python used / fields:

1. Desktop Applications
2. web Apps ← ----- Django,Flask
3. Database apps
4. Network apps
5. Computer Games
6. Data analysis
7. Machine learning
8. Image processing
9. web scrapping ← -- Beautifulsoup
10. IOT apps

Which companies using python

Google, Youtube, Nasa, Dropbox, Instagram, Facebook,.....

Python Versions

1.0 ← - January 1994

2.0 ← -- October 2000

3.0 ← -- December 2008

Software version rule ← -- new version should support old version (this is not applicable between python3.0 vs python2.0)

Python3 there is no backward compatibility with python2.0

long datatype are not in python3.0 but are available in python2.0

PIP Pip is a package management system that simplifies installation and management of software packages written in Python such as those found in the Python Package Index (PyPI)

```
sudo apt update
sudo apt install python3-pip
pip3 --version
```

} python3

```
sudo apt update
sudo apt install python-pip
pip --version
```

} python2

Virtualenv ← -- A Virtual Environment, put simply, is an isolated working copy of Python which allows you to work on a specific project without worry of affecting other projects. It enables multiple side-by-side installations of Python, one for each project. It doesn't actually install separate copies of Python, but it does provide a clever way to keep different project environments isolated.

What did Virtualenv will do?

Packages installed here will not affect the global Python installation.

Virtualenv does not create every file needed to get a whole new python environment

It uses links to global environment files instead in order to save disk space and

speed up your virtualenv. Therefore, there must already have an active python environment installed on your system.

Test if virtualenv is installed : `virtualenv --version`

if not installed:

`sudo apt-get install python-virtualenv`

`sudo easy_install virtualenv`

`sudo pip install virtualenv`

`mkdir ~/my_project`

`virtualenv ~/my_project/my_app`

`cd ~/my_projects/my_app/bin`

`source activate`

If you look at the bin directory in your virtualenv, you'll see easy_install which has been modified to put eggs and packages in the virtualenv's site-packages directory. To install an app in your Virtualenv:

`pip install flask`

`pip install django`

to deactivate type:

deactivate

```
virtualenv cnn1
```

```
source cnn1/bin/activate
```

```
#to deactivate
```

```
deactivate
```

```
#to export
```

```
pip freeze --local > requirements.txt
```

```
#to install
```

```
pip install -r requirements.txt
```

Program Fundamentals:

1. Identifiers ← -- Name used for identification purpose

Name of the variable eg: `x= 10`
 `def add():` #function name
 `class Test():` #class name

Identifiers are variable name, method name or class name

Rules to define identifiers:

1. digit(0 to 9)
2. alphabet symbols(lower case and upper case)
3. under score(_)
4. identifier should not start with a digit
5. case sensitive
6. Keywords are not used as identifiers
7. maxlength ← - there is no length limit(but long name are not recommended)

Valid: `Cash = 10, total =20, TOTAL=30, _hello ='Hello'`

Invalid: `ca$h = 10, 123cash=10, If = 30, def=200.`

! if identifiers starts by:

1. `_` (underscore) : it s private
2. `__` (2 under scores): strongly private
3. `__name__` (start with 2 underscore and ends with 2 underscore): language specific identifier(special variable defined by python) eg:[`__name__` , `__add__` ,....]

2. Reserved words ← ----- words used to represent some functionalities.

python has 33 reserved words

True, False, None
and, or, not, is
if, else, elif
for, while, break, continue, in, return, yield
try, except, raise, assert
import, from, as, class, def, pass, global, nonlocal,
lambda, del, with

. assert is used for debugging
. Only alphabet symbols
.except first tree only small letters

To print all the keywords
import keyword
print (keyword.kwlist)

keyword vs reserved word : all keywords are reserved words, however there maybe some reserved words that are not keyword

3. Data types ← ----- what is a data type: a particular kind of data item, as defined by the values it can take, the programming language used, or the operations that can be performed on it.

First 5 are called Inbuilt data types:

1. int X = 10 ← -- number without decimal point
2. float f= 2.5 ← -- decimal point numbers
3. complex c=1+2j ← -c.real, c.imag, real
4. bool t= True, f=False ← -internally True=1, False=0, True+True=2
- 5 str s="python", st='python' ← --sequence of characters single line
 S=''' hello ← --- multiline
 '''
- 6.bytes b=bytes(x) ← -immutable byte representation of int(0 to 256 only)
- 7 bytearray ba = bytearray(x) ← -- mutable
- 8.list l= [1,2,3,3] ← - mutable
- 9.tuple t=(1,1,2,3) ← --immutable list
- 10.set s={1,2,3} ← -- No duplicate, mutable
- 11.frozenset fs=frozenset(s) ← -- immutable set

12.range range(5), range(0,9),range(0,9,2) ← --- range of numbers,immutable

13.dict d = { 'key':value } ← -- mutable, no duplicate allowed

14. None

Char data type is not in python, long is not in python3.0

In python everything is an object internally [means in memory like object]

In python everything is an object internally: eg `x = 10` if you want to know the type of `x`

```
>> type(x)
```

```
<class 'int'>
```

X= 10

 $y=10$

X----->

$$\mathbf{y}_{\rightarrow}$$

10

Some inbuilt functions:

print() ←- to print value of an objects
type() ←-- to know type of object
id() ←----- to know address of object

X , Y are reference variables pointing to the memory where 10 is stored

To know type of the object : `>>type(x)`

To know address of in memory: `>>id(x)`

In python there is no concept of max value:

A=9999999999999999999999999999999999

999

999

```
>>type(A)
```

```
<class 'int'>
```

using (int) we can represent values in 4 ways or

integral numbers

Decimal(base 10: 0 to 9)

```
>>a=111
```

```
>>b= 1000
```

```
>>c= 99
```

Binary(base 2: 0 and 1) start by 0b or 0B

```
>>a=0b1111
```

```
>>a=0B1000
```

Octal(base 8) start by 0o or 0O

```
>>a=0o111
>>a= 0o1000
>>a=0o137
```

Hexa decimal(base 16: 0 to 9 ,a to for A to F) start by Ox or 0X

```
>>a=0x111
>>a=0X111
>>a =0xf
```

However Python will print all (int)values in decimal only:

Eg: a=10, b=0b1111,c=0X111
print(a+b+c)

For complex number real part can be **Decimal,Binary,Octal,Hexadecimal** but imaginary part is always decimal: 0b1111+10j

3.1 Type casting or Type coercion ← --- convert one data type to another

1. **int()** ← --- int(True), int("111"), int(10.51),complex number can't be convert to int gives **TypeError** and int('10.5') gives **valueError**(string should base 10 to be converted to int)
2. **float()** ← -- float(True),float('111'), float(10),complex number can't be convert to float
3. **complex()** ← - complex(x): x+0j or complex(x,y): x+yj,
complex('10'),complex(10,'10.5')
4. **bool()** ← --- 0=False any other number = True,empty string=False any other = True
5. **str()** ← -- str(10), str(10.15), str(True), str(1+2j)

All fundamentals data types are Immutable: once an object is created can't be modified(no changeable nature) WHY: ← -- cause of Object reusability

← ---Since Object creation is very costly operations before creating a new object python will check if not same object(content) is existing already and just point to it(reuse). **Advantage:** memory utilization improved and performance improved. Eg : str ='Hyderabad' , str1='Hyderabad' id(str) is same as id(str1)

```
>>str ='Hyderabad'
>>str1='Hyderabad'
>>str is str1
True
```

Reusable is only for {bool}, int : { 0 to 256} and to {str } cause they are most used.

Reusable is not available for float and complex : No known used common range of float and complex.

Font int the range {0 to 256} is common used

3.2 Escape Character ← --- str =" python\nsoftware "

1. \n new line
2. \t Horizontal tab
3. \r carriage return: goto first position of the line
4. \f form field: goto to the end -page down
5. \'
6. \"
7. \\
8. \v vertical tab
9. \b back space

3.3 Constants ← --- value always fixed(Not applicable in python) programmer are required to take care

4. Operators: ← ----- symbols used to perform certain operation on data.

1. Arithmetic operators : + - * / % // **
2. Relational or Comparison Operators : < <= > >=
3. Equality Operators: == !=
4. Logical Operators: and or not
5. Bitwise Operators : & | ^ ~ << >>
6. Assignment operators =
7. Special Operators is, in

the difference between == and is.

== will compare contents. is will compare references

3.1. Arithmetic operators: a=10, b=2

- | | | |
|---|----------------|---|
| + | addition | >>print(a+b)>>12 |
| - | subtraction | >>print(a-b)>>10 |
| * | Multiplication | >>print(a*b)>>20 |
| / | division | >>print(a/b)>>5.0 division operator always return float value |
| % | modulo | >>print(a%b)>>0 |

// **floor division** >>print(a//b)>>5 return int value if arguments are int or float if arguments are floats
** **exponent or power** >>print(a**b)>>100

+ applicable to string is **concatenation** but both arguments should be **str type**
* applicable to string first argument is **str** second argument should be **int** value

++ and – increment and decrement operators are not available in python

3.2.Relational or Comparison Operators

< **less than** >>print(10<2)>>False, >>print(10<'patto')>>TypeError
<= **less or equal** >>print(10<=2)>>False, >>print('Hyd'<='Hyderabad')>>True
> **greater than** >>print(10>2)>>True
>= **greater than or equal** >>print(10>=2)>>False

Chaning relation operator >>print(10<20<30<40)>>True
>>print(10<20<30<25)>>False

3.3 Equality Operators:

== **never raise any error if same content return True else false**
>>print(10=='patto')>>False
!= **>>print(10!='patto')>>True**

Chaining equality operators >>print(10==20==30)>>False

3.4 Logical Operators: can be applied to boolean and non boolean data types

For Boolean: always return boolean

1. and ← ----if both argument are true return **True** , print(True and False)>>False
2. or ← -----if one of the argument is true return **True**, print(True or False)>>True

3. not ← ----- print(not True)>>False

For Non boolean types:

Zero ← -- False,

none zero ← -- True

empty string ← --False,

any string ← --True

1. x **and** y: if x is not false return y else x

>>print(10 and 20)>>20

>>print(0 and 20)>>0

>>print('Hyd'and 'Mub')>>'Mub'

>>print(1 and 'Mub')>>'Mub'

2. x **or** y: if x is True return x else y

>>print(10 or 20)>>10

>>print(10 or 0)>>0

>>print('Hyd' or 'Mumb')>>'Hyd'

3. **not** x ← --- always return boolean **True** or **False**

>>not 10>> False

>>not 0>>True

3.5 Bitwise Operators :use bits representation to compare applicable for int and bool only

for Bool:

1. **&** and ← --if both bits are 1 otherwise 0

2. **|** or ← - if both bits are 1 or one of the bits is 1

3. **^** exclusive or : if both are different

4. **~** complement ← ----- 00001=11110

5. **<<** shift left ← ----move bits to the left

6. **>>** shift right ← -- move bits to the right

5 ← - in binary 101

4 ← ---in binary 100

Print(5&4) >>4

Print(5|4)>>5

Print(5^4)>>1

Print(~5)>>-6, print(~True)>>-2 since True=1 internally

10=1010

$10 \ll 2 = 40 \leftarrow$ --means move to the left the bits of ten 2 cells
 $101000 = 40$
 $10 \gg 2 = 2 \leftarrow$ --means move to the right the bits of ten 2 cells
 $0010 = 2$
 $\text{print}(\text{True} \ll 2) = 4 \leftarrow$ --01 \leftarrow -1000
 $\text{print}(\text{True} \gg 2) = 0 \leftarrow$ --01 \leftarrow -00

Most significant bit Vacant bit sign bit
 0 positive numbers
 1 negative numbers

3.6 Assignment operators:

$= \leftarrow$ --- $x = 10$
 $a, b, c = 10, 20, 30$

Compound Assignment operators: (Assignment operator mixed with other operators) with **arithmetic** and **bitwise**

$+= \leftarrow$ ---- if $x = 100$ then $x += 10 = 110$
 $-=$
 $/=$
 $*=$
 $//=$
 $**=$
 $\&=$
 $|=$
 $\wedge=$
 $\sim=$
 $\ll=$
 $\gg=$

Ternary Operator in other languages: syntax \leftarrow (condition)?firstvalue:secondvalue

In python ternary operator is not available with same syntax

in python: $x = \text{firstvalue if condition else secondvalue}$

x = 20 if 20 > 10 else 10

Nesting of conditional operator

1. max_value = 30 if 30 > 20 else 20 if 20 > 10 else 10

2. print('both nbr are equal' if a == b else 'a is greater' if a > b else 'a is smaller')

3.7 Special Operators

is ← --identity operators ← -- a = 10, b = 10 print(a is b) >> True
is not print(a is not b) >> False

a and b are pointing on same object

l1 = [10, 20, 30], l2 = [10, 20, 30] print(l1 is l2) >> False

is not ← --

in ← --membership operator

not in

l = [10, 20, 30] print(10 in l) >> True
print(1 not in l) >> True
print(30 not in l) >> False

str = "Univ of Hyd"

print('niv' in str) >> True
print(" " in str) >> True space available in str

3.7 Operator precedence

()

**

~

* / // %

+ -

& | ^ << >>

< <= > >=

```
A=30
B=20
C=10
D=5
print((A+B)*C/D)>>100.0
print((A+B)*(C/D))>>100.0
print(A+(B*C)/D)>>70.0
```

== !=
+= -= *= /=
is, is not
in, not in
and or not

3.8 Input from Keyboard

in all pgm languages most used data type is str

python2.0

1. `raw_input()` ← --- every input is considered as str,
we require to use type casting to convert to
int, float, ----

```
x = raw_input("enter data:")
```

2. `input()` ← --- input type is as on keyboard

python3.0

`raw_input()` is not available in python3.0

1. `input()` ← ----- every input is considered as str

```
x = input("enter data:")
```

```
x = int(input("enter data:"))
```

```
x = float(input("enter data:"))
```

eg : sum of 2 numbers in 1 row

```
print('Sum is', int(input('Enter 1st nbr:')) + int(input('Enter 2nd nbr:')))
```

How to read multiple data from Keyboard:

```
x = input("enter data:").split()
```

```
int data
```

```
split() ← -- space is separator  
Split(',') ← -- comma is separator  
split('*') ← -- * is separator
```

```
i = [int(x) for x in input("enter data:").split()]
```

float data

```
f = [float(x) for x in input("enter data:").split('*')]
```

sum:

```
a,b,c,d = [int(x) for x in input("enter 4 numbers:").split()]  
print('sum is',a+b+c+d)
```

eval(): ← -- evaluate to corresponding type of data (No need to use type casting)

```
a = eval("10+20")
```

```
print a>> 30
```

```
s = eval("Univ"+"Hyd")
```

```
print s>> UnivHyd
```

read multiple values different data types:

```
a,b,c= [eval(x) for x in input("Enter values :").split()]
```

```
Enter values: 10 0.5 'Hyd'
```

```
print type(a)>> int
```

```
print type(b)>> float
```

```
print type(c)>> str
```

Command line argument: **from sys import argv**

argv[0] is always program file name

```
add.py ← ----- from sys import argv  
                print (int(argv[1])+int(argv[2]))
```

```
test.py ← ----- from sys import argv  
                sum = 0
```

```
for x in argv:
    n= int(x)
    sum += n
    print(sum)
```

Output: ← ---- print()

`print()` ← -- print a new line

escape character are allowed with `print()`

```
print('Hello \t patto')>> Hello patto
```

```
print('Hello \n patto')>> Hello
                             patto
```

```
print("patto"*3)>> pattopattopatto
```

```
print("Hello","patto")>> Hello patto
```

`print()` with multiple arguments

```
a,b,c= 10,20,30
```

```
print(a,b,c)>>10 20 30
```

```
print(a,b,c,sep=',')>>10,20,30
```

```
print(a,b,c,sep=':')>>10:20:30
```

```
end='... ',sep=':' ← - attributes
```

```
print(a,b,c,end='... ')>>10 20 30...
```

```
print(a,b,c,end=' ', sep ':')>>10:20:30...
```

`print()` formatted string

```
%i or %d ← --int
```

```
%f ← - float
```

```
%s ← - string
```

```
a,b =10, 20.5
```

```
print( 'value of a is %i'%a)
```

```
print( 'value of a is %i'%a 'and value of b is %f'%b))
```

```
name ='patto'
```

```
salary=1000
```

```
print('name:{ },salary:{ }'.format(name,salary))>>name:patto,salary:1000
```

```
print('name:{0},salary:{1}'.format(name,salary))>>name:patto,salary:1000
```

```
print('name:{x},salary:{y}'.format(x=name,y=salary))>>name:patto,salary:1000
```

{ } ← -- replacement operator

4. Flow Control

at runtime in which order the statement are going to be executed is decided by Flow controller.

In Python Flow control are divided in 3 types

1. conditional statements or selection statements

among several available options, only 1 option will be selected and will be executed

if

if - else

if- elif- else

2. Iterative statement or loops

group of statement should be executed iteratively

for

while

3. Transfer statements or

break

continue

pass

do while and switch
statements are not in
python

syntax for conditional / selection statements:

```
if condition
    action1
else:
    action 2
```

```
if condition:
    action1
elif condition:
    action2
else:
    action3
```

syntax for loop/ iterative statements:

for ← ---- used when we know how many loops.
Execute body for every item in the
for x in range(10):
 body

while ← -- used when we don't know how many loops.

Execute the body as long as the condition is true

```
input ← --- n
for x in range(n):
    print(x)
```

```
Print:
0
01
012
0123
01234
for x in range(6):
    for y in range(x):
        print(y,end="")
    print()
```

```
Print:
12345
1234
123
12
1
for x in range(6,0,-1):
    for y in range(1,x):
        print(y,end="")
    print()
```

```
i=0
While i>0:
    print(i)
    i+=1
```

```
a =[1,2,3,4,5,6]
n = len(a)
x=0
while n>x:
    print a[x]
    x+=1
```

```
s =input('Enter string:')
```

Data in forward direction & Data in Backward direction

```
s =input('Enter string:')
```

```
n=len(s)
```

```
i=0
```

```
print('Data in forward direction')
```

```
while i<n:
```

```
    print(s[i],end="")
```

```
    i+=1
```

```
print()
```

```
print("Data in Backward direction")
```

```
i=n-1
```

```
while i>=0:
```

```
    print(s[i],end="")
```

```
    i-=1
```

```
print()
```

```
python >>
```

Output:

Enter string:python

Data in forward direction

python

Data in Backward direction

nohtyp

Print duplicate in string

```
s =input('Enter string:')
```

```
l = len(s)
```

```
for x in range(l):
```

```
    for y in range(x+1,l):
```

```
        if s[x]==s[y]:
```

```
            print(s[x],s[y],end="")
```

```
            print()
```

Output: s='miss'>>ss

Count duplicate in string:

```
s =input('Enter string:')
```

```
l = len(s)
```

```
c=[]
```

```
for x in range(l):
```

```
    count=1
```

```
    for y in range(x+1,l):
```

Enter string:mississipi

m 1

i 4

s 4

p 1

```
    if s[x]==s[y]:
        count+=1
    if s[x] in c:
        continue
    print(s[x],count)
    c.append(s[x])
```

Break

Print Hello 5 times

```
i=1
while True:
    print('Hello')
    if i==5:
        break
    I+=1
```

```
for i in range(10):
    if i==7:
        print('Enough process')
        break
    print(i)
```

Continue

```
Cart = [10,20,30,150,70]
For item in cart:
    If item >100:
        print('can't process item',item)
        Continue
    print('processing item',item)
```

```
for i in range(10):
    if I%2==0:
        continue
    print(i)
```

del statement: ← **--** used to delete a variable after using the variable

```
a=b="python"
```

del a ← **---** **a** will be deleted but not the content. if you also delete **b** and there is no reference to object the garbage collector will deal and destroy the object

Transfer statements:

1. break ← **-----** stop current iteration (stop and exit the loop) remaining item will not be processed

2. continue ← **-----** skip current iteration and continue. Remaining items will be processed

eg : Avoid to divide by zero using **continue**:

```
n=[10,20,0,30,40]
```

```
for number in n:
```

```
    if number ==0:
```

```
        continue
```

```
    print('100/{ }={ }'.format(number,100/number))
```

Using **else** with **for** and **while** loops

using **else** with no loop is not valid , however
else works with **break** but not with **continue**
in python ← ----- **else** can be used with

1. **for-else**
2. **while-else**
3. **try-except-else-finally**

eg: **for** or **while** :

body
body

break ← -- if this break doesn't execute then else body will be processed

break
else:
body

Eg:

marks =[60,70,80,90,30]

for **mark** in **marks**:

if **mark**<50:

print('Failed:',**mark**)

break ← -- if break no executed then else will execute

print('pass:',**mark**)

else:

print('Congraturation you passed')

Pass statement ← ----do nothing

def **f1**():

pass

eg. **class** **P**:

def **f1**():

pass

class **B**(**P**):

```
def f1:  
    statement
```

```
for i in range(100):  
    if i%10==0:  
        print(i)  
    else:  
        pass
```

How to perform switch statement in python

#using if-elif-else

```
if x == 'a':  
    # Do the thing  
elif x == 'b':  
    # Do the other thing  
if x in 'bc':  
    # Fall-through by not using elif, but now the default case includes case 'a'!  
elif x in 'xyz':  
    # Do yet another thing  
else:  
    # Do the default
```

or

```
result = {  
    'a': lambda x: x * 5,  
    'b': lambda x: x + 7,  
    'c': lambda x: x - 2  
}[value](x)
```

or

```
def f(x):  
    return {  
        'a': 1,  
        'b': 2  
    }.get(x, 9)    # 9 is default if x not found
```

6 String:

how to access characters from string: **1. index and 2. slicing**

1.Index:

both +ve and -ve index

+ve index ← -----left to right(forward direction)

-ve index ← -----right to left(backward direction)

s = 'python '

0	1	2	3	4	5
p	y	t	h	o	n
-6	-5	-4	-3	-2	-1

eg:

```
s=input("Enter a string:")
```

```
i=0
```

```
for x in s:
```

```
    print("The character present at +ve index:{} and character  
present at -ve index:{} is :{}".format(i,i-len(s),x))
```

```
    i+=1
```

Enter a string:python

The character present at +ve index:0 and character present at -ve index:-6 is :p

The character present at +ve index:1 and character present at -ve index:-5 is :y
The character present at +ve index:2 and character present at -ve index:-4 is :t
The character present at +ve index:3 and character present at -ve index:-3 is :h
The character present at +ve index:4 and character present at -ve index:-2 is :o
The character present at +ve index:5 and character present at -ve index:-1 is :n

2. by Slicing:

s=[start:end:steps size]

s='python'

s[:] = s[::] = s[0:] >>'python'

s[0:6] ← --s[0:6:1] >>'python'

s[::2] >>'pto'

s[::-1] >>'nohtyp'

s[::-2] >>'nhy'

s[-1:-7:-1]>>'nohtyp'

s[-1::-1]>>'nohtyp'

backward direction: begin to end-1 : -1 to – len(s)+1

mathamatical operations allowed with string + and *

+ ← -- 2 arguments must be str type: 'python'+'python'>>pythonpython

* ← -- 1 argument must be str another 1 must int: 'python'*2>>pythonpython

inbuilt function for string:

string is immutable means once created we can't change the content

find(substring) ← -- return index of the substring location

index() ← ----return index of the substring location

rfind()

rindex()

count substring in the given string:

`s.count(substring)`
`s.count(substring,begin,end)` ← --- surch occurrence of substring in the given range

```
s= 'mississipi'
print(s.count('i'))>>4
print(s.count('i',3,len(s))) >>3
```

Replace substring:

`s.replace(oldstring,newstring)`

`s`='Learning python is very difficult'

`s1 =s.replace('difficult','easy')`

`s2=s.replace(' ','')>>Learningpythonisverydifficult`

splitting of strings:

`s='python is easy'`

`1.split()` ← -- `s.split()` ← -- → > ['python','is','easy']
`s.split(',')>> ['python','is','easy']`

`2.rsplitt()` ← --`s.rsplitt()` ← -->> ['python', 'is', 'easy']

join()

`t=['python','is','easy']`

`s=' '.join(t)`

`print(s)`

Changing case of a string:

`upper()`

```
lower()
swapcase()
title() ← --every word first letter is upper case
capitalize() ← -only first letter is upper case
print(s.upper())
print(s.lower())
print(s.swapcase())
print(s.title())
print(s.capitalize())
```

Checking start and ending of string:

```
1.startswith()
2.endswith()
```

```
t='python is easy'
print(t.startswith('python'))>>True
print(t.endswith('easy'))>>True
```

print characters at odd position and at even position

```
1 ← --use slice operator
2. ← --without slice operators
```

```
S='python'
print('character at odd positin',s[::2]) >>pto
print('character at even positin',s[1::2]) >>yhn
```

Using while:

```
S='python'
i=0
print('character at even positin:')
while i<len(s):
```

```
print(s[i])
i+=2
```

Input: B4A1D3

output:ABD134

```
s='python'
s1=s2=output=''
for x in s:
    if x.isalpha():
        s1=s1+x
    else:
        s2=s2+x
for x in sorted(s1):
    output=output+x
for x in sorted(s2):
    output=output+x
print(output)
```

**Removing space at start(left) or end (right)of string:
strip does not remove space in between substrings**

`strip()` ← -- remove space at right and left

`lstrip()` ← -- remove space at left

`rstrip()` ← -- remove space at right

eg:

`city = ' Hyderabad' ← --- city.lstrip()`

`city = 'Hyderabad '` ← --- `city.rstrip()`

`city = ' Hyderabad '` ← --- `city.strip()`

Unicode ← -- Unlike **ASCII**, which uses **7 bits** for each character, **Unicode** uses **16 bits**, which means that it can represent more than **65,000 unique characters**

1. `chr(unicode)`

2. `ord('character')`

```
chr(97)>>a
chr(98) >>b
```

```
ord('a') >> 97
ord('b') >> 98
```

Print patterns exercises:

```
nbr = int(input('Enter a nbr:'))
for y in range(nbr,0,-1):
    print('*'*y,end=' ')
    print()
for x in range(nbr):
    print('*'*x,end=' ')
    print()
nbr = int(input('Enter a nbr:'))
for y in range(nbr,0,-1):
    print('*'*y,end=' ')
    print()
for x in range(nbr):
    print('*'*x,end=' ')
    print()
```

output:

[illegible]

```
*****
```

```
nbr = int(input('Enter a nbr:'))  
for x in range(1, nbr):  
    print('*'*x)
```

```
*
```

```
**
```

```
***
```

```
****
```

```
nbr = int(input('Enter a nbr:'))  
for x in range(1,1+nbr):  
    print('*'*nbr)
```

```
*****
```

```
*****
```

```
*****
```

```
*****
```

```
*****
```

```
*****
```

1.

```
nbr = int(input('Enter a nbr:'))  
s=' '  
for x in range(1,nbr):  
    s=str(x)  
    print(nbr*s)
```

2.

```
nbr = int(input('Enter a nbr:'))  
for i in range(1,nbr+1):  
    for j in range(1,nbr+1):  
        print(i,end="")  
    print()
```

```
11111
```

```
22222
```

```
33333
```

```
44444
```

```
nbr = int(input('Enter a nbr:'))
for i in range(nbr,0,-1):
    for j in range(nbr,0,-1):
        print(i,end="")
    print()
```

```
55555
44444
33333
22222
11111
```

```
nbr = int(input('Enter a nbr:'))
for i in range(1,nbr+1):
    for j in range(1,nbr+1):
        print(j,end="")
    print()
```

```
12345
12345
12345
12345
12345
```

```
nbr = int(input('Enter a nbr:'))
for i in range(nbr,0,-1):
    for j in range(nbr,0,-1):
        print(j,end="")
    print()
```

```
54321
54321
54321
54321
54321
```

```
nbr = int(input('Enter a nbr:'))
for i in range(1,nbr+1):
    for j in range(1,nbr+1):
        print(chr(64+i),end="")
```

```
print()
```

```
AAAAA  
BBBBB  
CCCCC  
DDDDD  
EEEEEE
```

```
nbr = int(input('Enter a nbr:'))  
for i in range(1,nbr+1):  
    for j in range(1,nbr+1):  
        print(chr(64+j),end="")  
    print()
```

```
ABCDE  
ABCDE  
ABCDE  
ABCDE  
ABCDE
```

List:

l=[10,20,30] ← --list are mutable

insertion order is preserved and duplicates are allowed,heterogeneous,growable

```
l=[]  
l=[10,20,30,40]
```

Nested list: **nl=[1,2,[10,20]]**

Traversing element of list with while or for:

using while loop:

```
list =[1,2,3,4,'Hello',100]  
i=0
```



```
while i<len(list):  
    print(l[i])  
    i+=1
```

using for loop:

```
for x in list:  
    print(x)
```

print even number:

```
for x in list:  
    if x%2==0:  
        print(x)
```

print positive and negative indexes:

```
list =[10,20,30,40]  
l =len(list)  
for x in range(l):  
    print('positive index',x,':',list[x],'negative index',':',x-l)
```

Note: ← -- len() ← - start at 1
 ← --range() ← -start at 0

important functions of list:

in python we can use functions or methods (interchangeably) but there is a small difference

```
def f1():                //function is declared outside of a class  
    print("Hello")  
class Student(self):  
    def info(self):      //method is a function declared inside a class  
        print("this is a method")
```

f1() //to call a function

s=Student() //to call a method we use object reference

s.info()

(any function called under object reference is a Method)

1. len(list) ← - length of list start at 1

2. count(element) ← --- number of occurrences of elements

>>list.count(20)

3. index(element) ← - first occurrence of element

>>list.index(20) //if element is not there you get ValueError

eg:

using if else:

l=[1,2,3,4,5,6,]

target=int(input('Enter number to search:'))

if target in l:

print(target,'available and its first occurrence is',l.index(target))

else:

print(target,' not available')

using try and except:

l=[1,2,3,4,5,6,]

target=int(input('Enter number to search:'))

try:

print(target,'available and its first occurrence is',l.index(target))

except ValueError:

print(target,' not available')

To manipulate a list, append(),insert(),extend()

1.append() ← ----l.append(40) ← --- add at last place ← --[10,20,30,40]

2.insert() ← -----l.insert(index,element) ← --l.insert(1,15) ← -insert at given place

>>[10,15,20,30,40]

l.insert(100,50) ← -- when index is higher, insert will always be at last

>>[10,15,20,30,40,50]

`l.insert(-1,555) ← -- when index is -ve, insert will always be at the beginning`
`>>[555,10,15,20,30,40]`

3.extend()

`l1 =[1,2,3] and l2=[4,5,6,7,8,9]`

`l1.extend(l2)`

`print(l1)>>[1,2,3,4,5,6,7,8,9]`

or

`l3 = l1+l2` ← --if you dont l1 to be modified

remove() and pop():

`l3.remove(element)` ← --if element at his first occurrence

`l3.pop()` ← -- remove and return last element

`l3.pop(index)`

reverse() ← -`l3.reverse()`

sort() ← ---`l3.sort()` //list should contain homogeneous elements

Tuple:

is like list but is immutable

`t=(1,)`

`t1 =(1,2,3,'tuple')`

Set: ← --- is like list but no duplicate element and element order

s=set()

1. s.add(x) //always 1 argument

2. s.update(range(x,y)) // always takes a sequence as argument

s.add(10) //always 1 argument

s.update(range(1,10,2))

s=set()

>>> s.add(10)

>>> s.update('patto')

>>> print(s)

set(['a', 'p', 10, 't', 'o'])

3. copy() ← ---cloning //this help in backup

s={10,20,30}

s=s1 ← --means s and s1 are referencing same object

s1.s.copy() ← ----- clone the object(copy of the object or replica)

3. pop() ← -- remove and return element//random element we don't know which

s={10,20,30}

>>s.pop()

>>s.pop()

if set is empty : **KeyError** remove from empty set

4. remove(x) ← --remove and return specified element

if the element is not in set : **KeyError** there is no such element

5. discard(x) ← --to avoid keyError discard will check if element is there before removing it.

6. clear() ← --- remove all elements in the set

Mathematical operation for the set

$s = \{1, 2, 3, 4\}$

$s1 = \{10, 20, 30, 40\}$

1.union() ← --or |

```
>>> s|s1
```

```
>>> s.union(s1)
```

```
set([1, 2, 3, 4, 40, 10, 20, 30])
```

2.intersection() ← -- and &
common elements

```
>>> s&s1
```

```
set([])
```

```
>>> s.intersection(s1)
```

```
set([])
```

```
>>>
```

3.difference() ← --- minus -

element present in s but in s1

```
>>> s.difference(s1)
```

```
set([1, 2, 3, 4])
```

```
>>> s-s1
```

```
set([1, 2, 3, 4])
```

3.symmetric_difference()

symmetric difference $s \wedge s1$ element of both sets minus intersection

```
>>> s^s1
```

```
set([1, 2, 3, 4, 40, 10, 20, 30])
```

```
>>> s.symmetric_difference(s1)
set([1, 2, 3, 4, 40, 10, 20, 30])
```

4. membership operator **in** and **not in**:

check a particular element is available or not

```
>>> s2='python'
>>> s2=set('python')
>>> print('p' in s2)
True
>>> print('z' in s2)
False
>>> print('t' not in s2)
False
```

5.set comprehension:

```
s={x*x for x in range(1,6)}
print(s)
```

s={expression for x in sequence condition}

by default index, slice are not applicable cause element ordering is not important(**TypeError****)**

#write a pgm to eliminate duplicate elements in the list:

just change the list in set

```
l=[1,1,3,3]
s=set(l)
print(s) >>set([1,3])
```

#without using set;

```
l=eval(input("Enter some list of values:"))
l1=[]
for x in l:
    if x not in l1:
        l1.append(x)
print(l1)
```

```
Enter some list of values:['aaa','aaa']
['aaa']
```

different vowels in the word:

```
w=input("Enter some word:")
s=set(w)
v={'i','u','o','a','e'}
d=s.intersection(v)
print('different vowels in the word:',d)
```

Dictionary Data Structure:

List.Tuple,set----- → only to hold individual objects

Dictionary ← --- Key-Value pairs// in perl and Ruby is called hash and in java is hashmap

eg:

rollno:name

mobilenumber:address

d={} or d=dict()

```
>>> d['name']='patto'
```

```
>>> d['salary']=1000
```

```
>>> print(d)
{'salary': 1000, 'name': 'patto'}
```

to access element print(**d['name']**)
d.has_key('name') //python2

To update dictionary if key is already available it will be overwritten otherwise added
d['name']='Raj'

how to delete elements from dictionary

del d[key] ← ----- **del d['name']** if key not available a **KeyError** is returned

d.clear() ← ----- To delete all [key]=value but still we can access d

del d ← ---- variable d will be delete. If you try to access after a **NameError** is returned

list,tuple,set ← -- a key can be associated with a single value object. However, if we want to associate multiple values to a single key, put values in group like list,tuple,set

list ← - **a={1:[10,20,3]}**

tuple ← --**a={1:(10,20,3)}**

set ← -- **a={1:{10,20,3}}**

Dictionary comprehension:

squares={x:x*x for x in range(5)}

print(squares)

```
>>> squares={x:x*x for x in range(5)}
```

```
>>> squares
```

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

important functions/method associated with dictionary:

1.dict() ← ---- d=dict() ← -- create a dictionary

list of tuple dl=dict([(,),(,),(,)]),

tuple of tuple dt=dict(((,),(,),(,))),

set of tuple ds=dict({(,),(,),(,)}),

internal should always be tuple with no multiple values otherwise you get
TypeError:Unhashable


```
d=dict([(100,'pato'),(120,'xoxo'),(200,'tutu')])  
>>{120: 'xoxo', 200: 'tutu', 100: 'pato'}
```

d.get(key) ← ---d.get(100)

if key is not available we get None but we can provide default value to return if key in None

d.get(key,default value) ← ---d.get(1,'Patto')

d.pop(key) ← --remove particular key-value. Pop always expect 1 argument

d.popitem() ← --remove key-value randomly

to get all keys in dictionary:

d.keys()

to get all values in dictionary:

d.values()

to get all key-values(items) in dictionary

d.items() ← ----[(k,v),(k,v)]

```
for k,v in d.items():  
    print(k,.....,v)
```

to get exactly copy

d1=d.copy()

d.setdefault(k,v) ← --while d[key]=value if key is available don't update but return the existing value. If key not available then add the key-value

d.update(x) ← ---if x is a dictionary sequence all elements will be added to dictionary d

update expect atmost 1 argument

#write a pgm to enter name and % of marks in a dictionary and display information:

```
rec={}
n =int(input("Enter number of student:"))
i=1
while i<=n:
    name=input("Enter Student Name:")
    marks=input("Enter % of marks:")
    rec[name]=marks
    i+=1
print("Name of student","\t","\% of marks")
for x in rec:
    print("\t",x,"\t\t",rec[x])
```

output:

```
Enter number of student:2
Enter Student Name:patto
Enter % of marks:100
Enter Student Name:patto
Enter % of marks:90
Name of student    % ofmarks
      patto        90
```

Dictionary pgm

#write a pgm that take dictionary as input and return the sum of all values

```
i =eval(input("Enter dict:"))
```

```
s = sum(i.values())
```

```
print('sum is:',s)
```

//the function `sum()` can be applied to all sequence of data

#write a pgm that take list as input and return the sum of all values

```
L =eval(input("Enter list:"))
```

```
s = sum(L)
```

```
print('sum is:',s)
```

#pgm to print number of occurrences of each letter using dictionary

```
i = eval(input("enter string:"))
d={}
for x in i:
    d[x]=d.get(x,0)+1
print(d)
```

```
>>enter string:'mississippi'
{'p': 1, 'i': 4, 'm': 1, 's': 4}
```

However, if you to sort the above output

```
i = eval(input("enter string:"))
d={}
for x in i:
    d[x]=d.get(x,0)+1
for k,v in sorted(d.items()):
    print("{} occured:{} times".format(k,v))
```

```
>>enter string:'mississippi'
i occured:4 times
m occured:1 times
p occured:1 times
s occured:4 times
```

#number of occurrences of each vowel

```
w = eval(input("Enter word:"))
d={}
a=[]
v=['i','u','o','a','e']
for x in w:
    if x in v:
        a.append(x)
for L in a:
    d[L]=d.get(L,0)+1
print(d)
```

```
>>Enter word:'welcome'
```

```
{'o': 1, 'e': 2}
```

or

```
#number of occurrences of each vowel
word=input('Enter some word:')
vowels={'i','u','o','a','e'}
d={}
for x in word:
    if x in vowels:
        d[x]=d.get(x,0)+1
for k,v in sorted(d.items()):
    print('{} occurred: {} times'.format(k,v))
```

```
Enter some word:'welcome'
e occurred: 2 times
o occurred: 1 times
```

```
#student and marks input
number = int(input("Enter Number of students:"))
d={}
for x in range(number):
    name = input("Enter Student Name:")
    marks =int(input("Enter Marks:"))
    d[name]=marks
print(d)
```

#student and marks checking

```
number = int(input("Enter Number of students:"))
d={}
for x in range(number):
    name = input("Enter Student Name:")
    marks =int(input("Enter Marks:"))
    d[name]=marks
print(d)
while True:
    name =input("Enter student name to get marks: ")
    marks=d.get(name,-1)
    if marks==-1:
```

```

    print("Student Not Found")
else:
    print("the marks of {}:{}".format(name,marks))
    option=input("Do you want to find another student marks?:[yes / No]:")
    if option=="No":
        break
print("Thanks for using our app")

```

Aliasing ← ----

In Python, aliasing happens whenever one variable's value is assigned to another variable, because variables are just names that store references to values.

Eg: Import numpy as np

data in memory that has multiple variable associated with it.
 Aliases don't work on immutable data
 Aliases are used instead of make a copy of the data.

eg:

```

r=range()
a=r(10)
b=r(20)

```

**if a=[]
 b=a vs b=list(a)**

```

>>> a=[1,2,3]
>>> b=a
>>> b is a
True
>>> b.append(4)
>>> b
[1, 2, 3, 4]
>>> a
[1, 2, 3, 4]
>>> c=list(a)

```

```

>>> c
[1, 2, 3, 4]
>>> c is a
False
>>> c.append(5)
>>> c
[1, 2, 3, 4, 5]
>>> a
[1, 2, 3, 4]
>>>

```

```

#alias
class m(str):
    q=lambda a,b,c:m(a.replace(b,c))

```

```

s="patto"
s=m(s).q('a','i').q('o','y')
print(s)

```

```

>>pitty

```

Property	List	Tuple	Set	Dictionary
Immutable	Yes	No	Yes	Yes
Duplicate element	Yes	Yes	No	No
Insertion order	Yes	No	No	No
Indexing	yes	yes	No	yes

slicing	Yes	Yes	No	No
Method to add element (e)	L.append(e) L.insert(index,e)	-----	S.add(e) S.update(range(x,y))	d[key]=value d.update(dict)
Method to delete element	L.pop() L.pop(index) L.remove(e)	-----	S.pop() S.remove(e) S.discard(e) S.clear	d.pop(key) d.popitem() d.clear()

7. function: ← --group of statements repeatedly required.

Write once call several number times

Main advantage ← -- code reusability

in python 2 types of functions

1. inbuilt functions ← --- eg:print(),length() we can use automatically.

Note: Pop(),count(),... are methods

2. user defined functions

How to define a function in python

def fname():
 print("hello function")

calling the function: **fname()**

def fname(name):
 print("hello",name)

calling the function: **fname("patto")**

Parameters:

4 types of **parameters** or **arguments** in python **1.Positional 2.keyword 3.default value 4. variable length**

#we can also pass **function** as **parameter(argument)**

#python function can return many values

def calc(a,b)

1. positional ← --**def calc(100,50)**

2. keyword ← --**def calc(a=100,b=50)**

3.default value ← --**def hello(name="Guest")** #if no name passed it will be "Guest"

Note: no argument follows default argument **def hello(name="Guest", msg)**

should be **def hello(msg, name="Guest")**

def hello(name="Guest"):

print("hello ",name)

hello()

hello('patto')

>>('hello ', 'Guest')

>>('hello ', 'patto')

def hello(msg,name="Guest",):

print("hello ",msg,name)

hello('welcome')

hello('welcome','patto')

>>('hello ', 'welcome', 'Guest')

>>('hello ', 'welcome', 'patto')

4.var ← --**var-leg : variable length** ← ----**def calc(*n)** #any number of arguments

def calc(*n)

def calc(name,*n)

def calc(*n, name) ← --if any argument comes after *n then you have to use positional and keyword argument in function call **calc(10,20,name='patto')**

#sum

def calc(*n):

return sum(n)

>>a=calc(10,20,10,20,30)

b=calc(10,20,30,10,20,30)


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

```
>>90
>>120
```

```
#sum
def calc(*n):
    result=0
    for x in n:
        result=result+x
    print(result)
```

```
>>calc()
>>calc(10,20)
>>calc(10,20,30)
```

```
0
30
60
```

```
#name and sum of max
```

```
def calc(name,*n):
    marks=0
    for x in n:
        marks=marks+x
    print(name,marks)
```

```
calc('patto')
calc('Raj',10,20)
calc('Xoxo',10,20,30)
```

```
>>('patto', 0)
>>('Raj', 30)
>>('Xoxo', 60)
```

python functions can return multiple values

```
def calc(a,b):
    sum=a+b
    prod=a*b
    diff=a-b
    div=a/b
    return sum,prod,diff,div
```

to call:

```
a,b,c,d = calc(100,50)
```

or

```
t= calc(100,50)
```

```
for x in t:
```

```
    print x
```

if function don't return anything the **default** is **None**

(for java or c you have to use **void main** but not in **python**)

```
def wish():
```

```
    print('Hello')
```

```
>>print(wish())
```

output:

```
Hello
```

```
None
```

#pgm to show argument behavior

```
def f(arg1,arg2,arg3=100,arg4=200):
```

```
    print(arg1,arg2,arg3,arg4)
```

```
f(1,2)
```

```
>>1  2  100  200
```

```
f(10,20,arg3=30,arg4=40)
```

```
>>10  20  30  40
```

```
f(arg3=1000,arg1=2,arg4=10,arg2=700)
```

```
>> 2 700 1000 10
```

```
f()
```

```
>>TypeError: f() missing 2 required positional arguments: 'arg1' and 'arg2'
```

```
f(arg3=1000,2,10,arg2=700)
```

```
^
```

```
>>SyntaxError: positional argument follows keyword argument
```

Keyword variable length argument(kwargs):

```
def display(**x):  
    for k,v in x.items():  
        print(k,':',v)  
    print()
```

```
display(name='patto',marks=100,grade='pass')  
display(car='Benz',amount=100000,grade='vip')
```

```
marks : 100  
grade : pass  
name : patto
```

```
car : Benz  
grade : vip  
amount : 100000
```

Global and local variables:

Global variables are the one that are defined and declared outside a function and we need to use them inside a function.

This function uses global variable s

```
def f():  
    print(s)  
# Global scope  
s = "welcome Home"  
f()
```

```
>> welcome Home
```

however we got error if we assign a local s

```
def f():  
    print(s)  
    s="Me too"  
# Global scope  
s = "welcome Home"  
f()  
>> UnboundLocalError: local variable  
's' referenced before assignment
```

Solution:

```
def f():  
    global s  
    print(s)  
    s = "Patto " #local variable  
    print(s)  
# Global Scope  
s = "Welcome Home"  
f()
```

```
>>Welcome home  
>>patto
```

Special functions: recursive, anonymous, decorator, iterator, generator, nested functions

1. recursive ← --function that calls itself

factorial(n)= n*factorial(n-1)

1. Advantage: reduce the length of the code and improves readability

2. very complex problems can be solved easily eg: towers of hanoi

#factorial function by using recursion

def factorial(n):

if n==0:

result=1

else:

result=n*factorial(n-1)

return result

print(factorial(0))

print(factorial(3))

>>1

>>120

2. anonymous (without Name) ← --**lambda**

Nameless functions

instant use(only one time usage)

so why do we need such anonymous function, if it is one time use, (reduce number of lines of codes)

Normal function def squareIt(n): return n*n	Anonymous: s = lambda n:n*n print(s(4))
--	--

```
print(squareIt(n))  
>>16
```

```
>>16
```

```
s = lambda x,y:x+y
```

```
print("The sum of {} and {} is:{}".format(2,3,s(2,3)))
```

```
print("The sum of {} and {} is:{}".format(200,400,s(200,400)))
```

```
bigger = lambda a,b:a if a>b else b
```

```
print(bigger(10,20))
```

Function that take another function as argument

filter()

map()

reduce()

#filter <---filtering a sequence based on a condition

```
number_list = range(-5, 5)
```

```
less_than_zero = list(filter(lambda x: x < 0, number_list))
```

```
print (less_than_zero)
```

```
# Output: [-5, -4, -3, -2, -1]
```

```
def isEven(n):
```

```
    if n%2==0:
```

```
        return True
```

```
    else:
```

```
        return False
```

#using filter and lambda

```
a=[1,2,3,4,5,6,7,8,9]
```

```
f =filter(lambda x:x%2==0,a)  #even number
```

```
print(f)
```

```
f =filter(lambda x:x%2!=0,a)  #odd number
```

```
print(f)
```

#map<-----map(function,sequence)

#map each element of a sequence to a each element in output(input and output same size)

```
def double(x):
```

```
    return 2*x
```

```
print(double(2))
```

#using map

```
m=list(map(double,a))
```

```
print(m)
```

```
ml=list(map(lambda x:2*x,a))
```

```
print(ml)
```

```
b=[10,20,30,40]
```

```
m2=list(map(lambda x,y:x+y,a,b))#if a,b are not same size extra  
numbers will be ignored
```

```
print(m2)
```

```
def multiply(x):  
    return (x*x)
```

```
def add(x):  
    return (x+x)
```

```
funcs = [multiply, add]
```

```
for i in range(5):  
    value = list(map(lambda x: x(i), funcs))  
    print(value)
```

```
# Output:
```

```
# [0, 0]
```

```
# [1, 2]
```

```
# [4, 4]
```

```
# [9, 6]
```

```
# [16, 8]
```

reduce ← ---- It applies a rolling computation to sequential pairs of values in a list. For example, if you wanted to compute the product of a list of integers.

```
from functools import reduce
```



```
product = reduce(lambda x,y:x*y,[1,2,3])
print(product)
```

or

```
p=[1,2,3,4,5]
product = reduce(lambda x,y:x*y,a)
print(product)
```

3. Nested functions:

declaring and calling a function within another functional

#will be used only if we want to access a function inside another

```
def outer():
    def inner():
        inner()
    outrer()
```

```
#eg:
def outer():
    def inner(a,b):
        print('sum is:',a+b,'average is',(a+b)/2)
    inner(10,20)
    inner(30,5)
outer()
```

```
>>('sum is:', 30, 'average is', 15)
>>('sum is:', 35, 'average is', 17)
```

function returning another function

```
def outer():
    print('outer executing')
    def inner():
        print('inner executing')
    print('outer returning inner')
    return inner
```

```
outer()
```

```
>>>outer executing
>>>outer returning inner
```

4. decorator:

helps in making short codes

decorator is a function that takes input another function extends it(without modifying it) and return another functional

function -----> decorator(**add extra capabilities to input** , without modifying the existing function)----->function

```
def decor(func):
    def inner(name):
        if name == 'patto':
            print('No ticket pruchased ',name)
        else:
            func(name)
    return inner
```

```
@decor
def hello(name):
    print("we wish you a wonderful journey with us",name)
```

```
>>>hello("patto")
No ticket pruchased  patto
```

```
>>>hello("paul")
we wish you a wonderful journey with us paul
```

or

#to access **décor** and **hello** separately

```
decorfunction= decor(hello)  
decorfunction("paul")
```

```
hello("paul")  
#inner and division must take always same arguments  
def decor(func):  
    def inner(a,b):  
        if b==0:  
            print('You should not take 0')  
        else:  
            return func(a,b)  
    return inner
```

```
@decor  
def division(a,b):  
    return a/b
```

```
print(division(10,20))  
>>0.5
```

```
print(division(2,0))  
>>You should not take 0  
>>None
```

3.iterator

4.generator

8 Modules:

A group of functions and variables saved to a file

Advantage: Code reusability

Length of the code will be reduced and readability enhanced

Maintainability

How to use Modules:

import module1 <-----: import **numpy**

import module1,module2,... <-----: import **numpy,pandas,scipy**

Aliasing:

import module1 as m1 <-----: import **numpy** as **np**
in pgm you have to use **np** only not **numpy**

since a module is a file and file may be too large,
If we only need to access variables or functions from a module:

from module1 import add,product,x,....

or

from module1 import *

Aliasing;

from module1 import add as a

Then to use `a(10,20)` instead of `module1.add(10,20)`

Reloading a module:

by default for every module compiler file will be generated in **pyc** it will be saved in the folder <-
__pycache__

if you import modules many times it will load once:

eg: **import module1**
import module1
import module1
import module1

Advantage: performance, memory, code execution fast

Drawbacks: once module loaded if the module is update/modified outside is not available during current execution

Solution

even for java, if .class file once loaded by class loader, if the file is modified outside the update file is not available to the jvm. So a customized class loader is needed in this case

The solution in Python there is a module **imp**
in this **imp** module there is a function called **reload()**

to check:

test.py

```
import time
import module1
print(module1.test)
time.sleep(20)
import module1
print(module1.test)
```

module1.py

```
test = 'This module test'
```

run python **test.py** and while time sleeping of 20 seconds modify the **module1.py** and save

```
>>>This module test
>>>This module test
```

we can see that the update module1.py was not loaded

the solution

```
import time
from imp import reload
import module1
print(module1.test)
time.sleep(20) #not always required just used to understand this import and reload concept
reload(module1)
```

print(module1.test)

run python **test.py** and while time sleeping of 20 seconds modify the **module1.py** and save

>>**This module test**

>>**This module test reloaded**

Module execution

we can execute directly **module1.py** ← --- **python module1.py**

also we can execute directly **module1.py** using **__name__=='__main__'**

module1.py

def f():

if **__name__=='__main__':**

print('Module executed directly')

else:

print('Module executed indirectly')

f()

To execute : **python module1.py**

output

>> **Module executed directly**

__name__ : Every module in **Python** has a special attribute called **__name__**. It is a built-in variable that returns the name of the module. **__main__** : Like other programming languages, **Python** too has an execution entry point, i.e., **main**.

Execution from **test.py**

import module1
module1.f()

To execute : **python module1.py**

output

>> **Module executed indirectly**

finding members of a module using `dir()`

`dir()` ← ---list of all current members

`dir(modulename)`

eg: using the **module1.py**

in **test.py**

import module1

print(dir(module1))

```
>>['_builtins_', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__',
    '__spec__', 'test']
```

some members are added automatically by python during creation and execution

'__builtins__' :This module provides direct access to all 'built-in' identifiers of Python; for example, `__builtin__.open` is the full name for the built-in function `open()`

Python gives you several different ways to view module content. The method that most developers use is to work with the `dir()` function, which tells you about the attributes that the module provides.

Function attributes are automatically generated by Python for you. These attributes perform the following tasks or contain the following information:

- **'__builtins__'** : Contains a listing of all the built-in attributes that are accessible from the module. Python adds these attributes automatically for you.

- `__cached__`: Tells you the name and location of the cached file that is associated with the module. The location information (path) is relative to the current Python directory.
- `__doc__`: Outputs help information for the module, assuming that you've actually filled it in. For example, if you type `os.__doc__` and press Enter, Python will output the help information associated with the `os` library.
- `__file__`: Tells you the name and location of the module. The location information (path) is relative to the current Python directory.
- `__initializing__`: Determines whether the module is in the process of initializing itself. Normally this attribute returns a value of `False`. This attribute is useful when you need to wait until one module is done loading before you import another module that depends on it.
- `__loader__`: Outputs the loader information for this module. The *loader* is a piece of software that gets the module and puts it into memory so that Python can use it. This is one attribute you rarely (if ever) use.
- `__name__`: Tells you just the name of the module.
- `__package__`: This attribute is used internally by the import system to make it easier to load and manage modules. You don't need to worry about this particular attribute.

It may surprise you to find that you can drill down even further into the attributes.

Typedir(module1.f) and press Enter

eg: using math module

from math import *

```
>>> print(sqrt(16)) #square root
```

4.0

```
>>> print(ceil(16.5)) #next integer value
```

17

```
>>> print(floor(16.5)) #previous integer value
```

16

```
>>> print(fabs(-16.9)) #float absolute value
```

16.9

eg: using random module

To generate random numbers

```
import random
```

```
n=random.random()
```

```
print(n)
```

OR

```
import random as rm
```

```
n= rm.random()
```

```
print(n)
```

To generate random alphabetic symbol

```
chr(randint(65,65+25))
```

functions in random module

1. random() ← -always generate a float random number between 0 and 1 **not inclusive**

$0 < x < 1$

```
import random as rm
for x in range(10):
    print(rm.random())
```

output

```
0.331215739809
0.069261860279
0.15764153409
0.127233415431
0.978116321227
0.112015320212
0.855253315777
0.691363611244
0.15597571355
0.0448707649963
```

2. randint(a,b) ← --to generate **random integer** between a and b inclusive of a or b

```
>>> rm.randint(1,100)
74
```

```
import random as rm
for x in range(10):
    print(rm.randint(0,100))
```

```
18
88
14
16
18
28
50
72
5
55
```

3. uniform() ← -to generate random **float** between a and b **inclusive** of a or b

```
import random as rm
for x in range(10):
    print(rm.uniform(0,100))
```

```
72.7782258528
35.18808483
84.6511617809
1.76529196249
98.0295272271
90.692730549
84.7682104388
52.0614039983
34.4324357338
96.7350268812
```

4. randrange(start,stop,steps) or randrange(start,stop)

start<=x<stop

```
>>> rm.randrange(0,5)
```

```
2
```

```
>>> rm.randrange(0,11,2)
```

```
4
```

```
import random as rm
for x in range(10):
    print(rm.randrange(0,100))
```

```
11
69
79
96
9
```

81
94
37
9
0

```
import random as rm
for x in range(10):
    print(rm.randrange(0,100,20))
```

80
60
20
20
20
80
40
80
60
0

5. choice() ← -- return random object from the collection(**indexable collection** like **list, str, tuple** but not **set, dictionary**)

```
from random import *
list=['Hyd','Mumbai','Delhi']
for x in range(10):
    print(choice(list))
```

Mumbai
Mumbai

Hyd
Mumbai
Delhi
Mumbai
Mumbai
Hyd
Delhi
Mumbai

Eg: write python pgm to generate 6-digit random numbers for otp

```
from random import *  
for x in range(10):  
    print(randint(11111,999999))
```

804861
326700
634397
205685
890520
410748
480239
273471
192536
159176

To generate random alphabetic symbol
chr(randint(65,65+25))

Write a pgm to generate otp random number such 1a3b, 2z5y,....

```
from random import *
for x in range(10):
    print(randint(0,10),chr(randint(65,90)),randint(0,10),chr(randint(65,90)),sep='')
```

```
5Y8I
3T9D
9S2P
8X0U
3N9M
5B8P
2J5B
0C9S
9W3K
2B7J
```

9. try except finally

The **try** block lets you test a block of code for errors.

The **except** block lets you handle the error.

The **finally** block lets you execute code, regardless of the result of the try- and except bloc (always executed)

The **try** block will generate an exception, because **x** is not defined:

```
| try| :
|     print(x)
| except:
|     print("An exception occurred")
```

Since the try block raises an error, the except block will be executed.

Without the try block, the program will crash and raise an error:

Many Exceptions

You can define as many exception blocks as you want, e.g. if you want to execute a special block of code for a special kind of error:

Example

Print one message if the try block raises a `NameError` and another for other errors:

```
try :  
    print(x)  
except NameError:  
    print("Variable x is not defined")  
except:  
    print("Something else went wrong")
```

#The try block will generate an exception, because x is not defined:

```
try:  
    print(x)  
except:  
    print("exception occurred")
```


flow control in try except finally:

if there is error in second try and the except can't handle it the finally is executed ,then the command is passed to the next except

try:

print("outer block")

try:

print("inner block")

print(10/0)

except ZeroDivisionError:

print("inner except block")

finally:

print("inner finally block")

print("Will not be printed if error in try block and except can't handle it")

except:

print("outer except block")

finally:

print("outer finally block")

print(" will not be printed if there is error in 1st try block and exception can't handle it")

try:

stmt-1

stmt-2

try:

stmt-3

stmt-4

except xxxx:

stmt-5

finally:

stmt-6

except yyyy:

stmt-7

finally:

stmt-8
stmt-9

try except else finally:

else block will always execute if **except block** is not executed
if **except block** is executed **else block** is not executed

```
x=10
```

```
try:
```

```
    print("try block")
```

```
    print(x)
```

```
except NameError:
```

```
    print("X is not defined")
```

```
else:
```

```
    print("x is defined")
```

```
finally:
```

```
    print("finally Block")
```

```
>>try block
```

```
>>100
```

```
>>x is defined
```

```
>>finally Block
```

```
try:
    print("try block")
    print(x)
except NameError:
    print("X is not defined")
else:
    print("x is defined")
finally:
    print("finally Block")
```

```
>>try block
>>x is not defined
>>finally Block
```

Eg:

```
x=100
try:
    print("first try block")
    print(x)
    try:
        print("2nd try block")

    except KeyError:
        print("x is not defined")
    finally:
        print("1st finally block")
    print("first Non block")
except NameError:
```

```
print("X is not known")
else:
    print("x is known")
finally:
    print("2nd finally Block")
print("2nd Non block")
```

```
first try block
100
2nd try block
1st finally block
first Non block
x is known
2nd finally Block
2nd Non block
```

```
eg1:
#x=100
try:
    print("first try block")
    print(x)
    try:
        print("2nd try block")

    except KeyError:
        print("x is not defined")
    finally:
        print("1st finally block")
        print("first Non block")
except NameError:
    print("X is not known")
else:
    print("x is known")
```

finally:

```
print("2nd finally Block")  
print("2nd Non block")
```

first try block
X is not known
2nd finally Block
2nd Non block

eg2:

~~#x=100~~

try:

```
print("first try block")
```

try:

```
print("2nd try block")
```

```
print(x)
```

except :

```
print("x is not defined")
```

finally:

```
print("1st finally block")
```

```
print("first Non block")
```

except NameError:

```
print("X is not known")
```

else:

```
print("x is known")
```

finally:

```
print("2nd finally Block")
```

```
print("2nd Non block")
```

first try block
2nd try block
x is not defined

1st finally block
first Non block
x is known
2nd finally Block
2nd Non block

eg3:

~~#x=100~~

try:

print("first try block")

try:

print("2nd try block")

print(x)

except KeyError:

print("x is not defined")

finally:

print("1st finally block")

print("first Non block")

except NameError:

print("X is not known")

else:

print("x is known")

finally:

print("2nd finally Block")

print("2nd Non block")

first try block
2nd try block

1st finally block
X is not known
2nd finally Block
2nd Non block

Various 25 combinations of try except else finally blocks:

1. try:

`print("this is try")` //this is **invalid**, try block should have except or finally block
always

2. except:

`print("this is except")` //this is **invalid**, except block always need try block

3. else:

`print("this is else")` //this is **invalid**

4. finally:

`print("this is finally")` //this is **invalid**

5. try:

`print("this is try")`
except:
`print("this is except")` // this is valid

6. try:

`print("this is try")`
finally:
`print("this is finally")` // this is valid, eg: exception don't know how to handle error, on final block close database

7. try:

```
print("this is try")
```

except:

```
print("this is finally")
```

else:

```
print("this is else")
```

 / this is **valid**, eg: if no **exception**, try followed by **else**.
If there is **exception**, try followed by **else**.

8. try:

```
print("this is try")
```

else:

```
print("this is else")
```

 / this is **invalid**, **except** block must be there for **else** block

9. try:

```
print("this is try")
```

except:

```
print("this is finally")
```

finally:

```
print("this is finally")
```

 / this is **valid**,

10. try:

```
print("this is try")
```

except:

```
print("this is except")
```

else:

```
print("this is else")
```

else:

```
print("this is else")
```

 / this is **invalid**,

11. try:


```
print("this is try")
except:
    print("this is except")
finally:
    print("this is else")

finally:
    print("this is else") / this is invalid,
```

12. try:

```
print("this is try")
print("this is try 2") /independent print block
except:
    print("this is except") / this is invalid, the statement outside of try block is
making try block alone
```

13. try:

```
print("this is try")
except:
    print("this is except")
print("this is except") /independent print block between 2 except blocks is invalid
except:
    print("this is except") / this is invalid,
```

14. try:

```
print("this is try")
except:
    print("this is except")
print("this is except") /independent print block between except and finally blocks is invalid
finally:
    print("this is except") / this is invalid
```

15. try:

```
    print("this is try")
except:
    print("this is except")
try:
    print("this is try")
except:
    print("this is except") / this is valid
```

16. try:

```
    print("this is try")
except:
    print("this is except")
try:
    print("this is try")
finally:
    print("this is except") / this is valid
```

17. try:

```
    print("this is try")
except:
    print("this is except")
if 10>20:
    print("this is if")
else:
    print("this is else") / this is valid, if else block is not related to try-except block
```

18. try:

```
    print("this is try")
try:
    print("this is try")
except:
    print("this is except")
finally:
```

```
        print("this is finally")
except:
    print("this is except") / this is valid
```

19. try:

```
    print("this is try")
except:
    print("this is except")
finally:
    try:
        print("this is try")
    except:
        print("this is except")
    finally:
        print("this is finally") /valid
```

Various possible combination

1. try block compulsory we should write either except or finally block
2. except without try is invalid
3. finally without try is invalid
4. we can take multiple except blocks for the same try but we cannot take multiple else or finally
5. else without except is invalid
6. try-except-else-finally order is very important
7. Nesting of try-except-else-finally is possible

Types of exceptions:

all exceptions in python are categorized in 2 types;

1. predefined exceptions /In Built
2. User Defined exceptions /Customized

1. predefined exceptions /In Built

exceptions which are raised automatically by python whenever a particular event occurs...

eg: `print(10/0)` ZeroDivisionError
`x=int("ten")` <---ValueError

2. User Defined exceptions /Customized

eg1: bank account balance is 1000 but try withdraw 2000 //InsufficientFundsException

eg2: recharge cellphone airtime and you enter wrong number from card
//InvalidCouponCodeException

How to define and raise customized exceptions

```
class TooYoungException(Exception):  
    def __init__(self,arg):  
        self.msg=arg
```

```
class TooOldException(Exception):  
    def __init__(self,arg):  
        self.msg=arg
```

```
age =int(input("Enter age : "))  
if age<18:  
    raise TooYoungException("Too Young for house loan")  
elif age>60:  
    raise TooOldException("Too Old for house loan")  
else:  
    print("Thanks for registration")
```

output1

```
>>Enter age :20  
Thanks for registration
```

output2

```
>>>Enter age :10
Traceback (most recent call last):
  File "test.py", line 10, in <module>
    raise TooYoungException("Too Young for house loan")
__main__.TooYoungException
```

output3

```
>>>Enter age :70
Traceback (most recent call last):
  File "test.py", line 12, in <module>
    raise TooOldException("Too Old for house loan")
__main__.TooOldException
```

10. Logging and Debugging:

*eg: real life like in internet cafe when write your details in **log book** before using computers, or in office or labs you register arrival and leave time. Like this for app we need to record every activate in **log file***

*for every activity it s recommended to record it.
For app each and every exception must be recorded for the future purpose*

Advantage : *1. keep record / track of all information.*

Eg: mouse missing on system we can check when it git disconnected which user was using the system.....

2. if something goes wrong we can use this log file for debugging

3.we can provide statistics from log files.

Eg: how many requests per day.

Log file is different from database. Database will store app data like customer names whereas log files will contain activities of the app

To implement logging python contains an in built module called **logging**

depending on type of information logging is divided in 6 levels

logging 6 levels:

1	CRITICAL	50	By default (high levels) which are warning , Error and CRITICAL only will be saved in log file for python pgm
2	ERROR	40	
3	Warning	30	
4	Info	20	
5	DEBUG	10	
6	NOTSET	0	

1. How to implement logging

1. first we have to create the file (name of the file)

2. we have to specify which level of infos to store(level messages)

we use **basicConfig()** of **logging** module

```
logging.basicConfig(filename="log.txt",level=logging.WARNING)
```

#log.txt is the file to write in, if it is not there it will be created automatically and from **WARNING** and higher level will be written to the file, this same as

```
logging.basicConfig(filename="log.txt").
```

If we set **level=logging.DEBUG** means from **DEBUG** and all higher levels will be written to the file.

after creating the file you can write to the file using the following methods:

```
logging.debug(message)
```

```
logging.info(message)
```

```
logging.warning(message)
```

```
logging.error(message)
```

```
logging.critical(message)
```

#Write a python pgm to create a log file and write higher level messages

```
import logging
```

```
logging.basicConfig(filename='log.txt',level=logging.DEBUG)
```

```
print("python logging demo")
```

```
logging.debug("debug message")
```

```
logging.debug("info message")
```

```
logging.debug("warning message")
```

```
logging.debug("error message")
```

```
logging.debug("critical message")
```

output

DEBUG:root:debug message
INFO:root:info message
WARNING:root:warning message
ERROR:root:error message
CRITICAL:root:critical message

To write exception from python app to log file

logging.exception(msg)

import logging

logging.basicConfig(filename='log.txt',level=logging.DEBUG)

logging.info("A new request")

try:

x=int(input("Enter first number:"))

y=int(input("Enter second number:"))

print(x/y)

except ZeroDivisionError as msg:

print("Can not divide with zero")

logging.exception(msg)

except ValueError as msg:

print("Enter only int values")

logging.exception(msg)

logging.info("Request processing Completed")

log.txt

INFO:root:A new request
INFO:root:Request processing Completed
INFO:root:A new request
ERROR:root:division by zero
Traceback (most recent call last):
 File "test.py", line 8, in <module>
 print(x/y)


```
ZeroDivisionError: division by zero
INFO:root:Request processing Completed
INFO:root:A new request
ERROR:root:invalid literal for int() with base 10: 'five'
Traceback (most recent call last):
  File "test.py", line 7, in <module>
    y=int(input("Enter second number:"))
ValueError: invalid literal for int() with base 10: 'five'
INFO:root:Request processing Completed
```

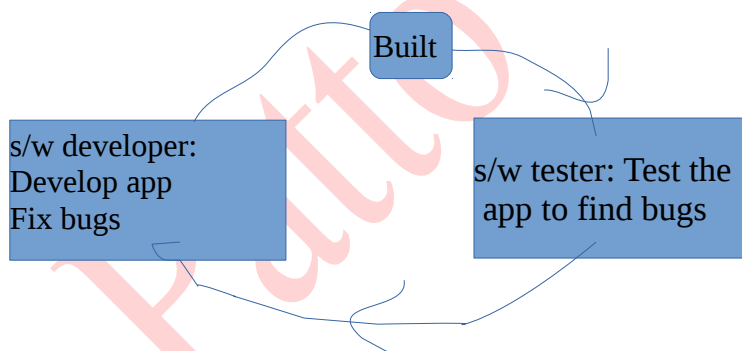
Debugging

Debugging ← ----- fixing/removing bugs in software

Defect / Bug --- → *Mismatch between **expected result** and **original/current result***

eg: **def squareit(x):**
 return 2*x it should be **return x*x**

the result is coming but current result don't match expected result



Built Process

after development of app, it s given to s/w tester team for testing. If any bug is found is immediately reported to s/w development team and they have to fix the bug and forward the app to testing team for a new test

for fixing bug in large number of codes often **print()** statement is used. It is placed in codes just to verify the output. However it is not recommended since after debugging you have to remove many **print()** if you have large number of codes

for debugging in python **assert** is used

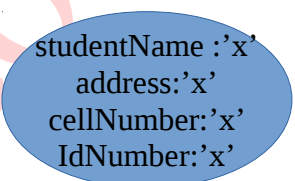
2 types of assert:

1. Simple version
2. very Simple Version(Augmented)

11.Pickling and Unpickling

Pickling or **serialization**(in other languages) ← is the process of **writing/saving** state of an object to a file

state of an object



```
studentName : 'x'  
address: 'x'  
cellNumber: 'x'  
IdNumber: 'x'
```

UnPickling ← is the process of reading the state of object from the file

State of an object: Values for instance variables

the module used is **pickle**

Pickling:

in pickle use function **dump()**

pickle.dump(object,file)

UnPickling:

in pickle use function load()

pickle.load(file)

#eg 1 object:

import pickle

class Employee:

def __init__(self,empId,emName,emSalary,emAddress):

self.empId=empId

self.emName=emName

self.emSalary=emSalary

self.emAddress=emAddress

#display the state of the object

def display(self):

print(self.empId,"\t",self.emName,"\t",self.emSalary,"\t",self.emAddress)

#if emp.dat file not exist it will be created automatically

with open("emp.dat","wb") as f:

e = Employee(1,'patto',1000,'Kgl')

pickle.dump(e,f)

print("Pickled successful")

#the file will be in not readable form we hv to unpickle to read data

with open("emp.dat","rb") as f:

obj=pickle.load(f)

print("Employee Info")

obj.display()

#eg Many objects:

create 3 files pick.py , unpick.py , obj.py

pick.py

```
import emp,pickle
```

```
f=open("emp.dat","wb")
```

```
#wb will overwrite existing, to append use "ab"
```

```
n = int(input("Enter nbr of employees :"))
```

```
for x in range(n):
```

```
    empId= int(input("id :"))
```

```
    emName= input("Name :")
```

```
    emSalary=float(input("Salary"))
```

```
    emAddress=input(" Address :")
```

```
    e=emp.Employee(empId,emName,emSalary,emAddress)
```

```
    pickle.dump(e,f)
```

```
f.close()
```

obj.py

```
class Employee:
```

```
    def __init__(self,empId,emName,emSalary,emAddress):
```

```
        self.empId=empId
```

```
        self.emName=emName
```

```
        self.emSalary=emSalary
```

```
        self.emAddress=emAddress
```

```
    #display the state of the object
```

```
    def display(self):
```

```
        print(self.empId,"\t",self.emName,"\t",self.emSalary,"\t",self.emAddress)
```

unpick.py

```
import emp,pickle
```

```
f=open("emp.dat","rb")
```

```
print("Emp Details...")
```

```
while True:
```

```
    try:
```

```
        obj=pickle.load(f)
```

```
        obj.display()
```

```
        print()
```

```
    except EOFError:
```

```
        print("All details printed")
```

```
        break
```

```
f.close()
```

To unpick a particular employee based on id

```
import emp,pickle
f=open("emp.dat","rb")
print("Emp Details...")
while True:
    try:
        obj=pickle.load(f)
        if obj== 1:
            obj.display()
            break
    except EOFError:
        print("All details printed")
        break
f.close()
```

12.Object oriented

what is a class

what is object

what is reference variable

How to create object

How we can invoke method

meaning of self ←---variable always point to current object

what is constructor

instance = object

#define a class test and define one method among and create an object and call that method.

class Emp:

#construct is executed automatically when we create object, not required to call like we call a method

def __init__(self,name,salary):

self.name=name #instance variable

self.salary=salary #instance variable

print("constructor invoked")

def display(self):

print("employee name:{}, salary:{}".format(self.name,self.salary))

e1 = Emp('patto',1000) #object creation, e1 is reference variable

e1.display() #invoking the method

Method	vs	constructor
Any name		<u>__init__</u>
You have to call to execute		Auto execution whenever we create object
Per object, method can be called many times		Per object constructor is executed once
Inside method we can write business logic: 1. print() 2. insert data in database 3. validate users		Declare and initialize instance variable

Types of variables allowed inside python class?

3 types:

1. instance variables /non-static ← ----- object related variables

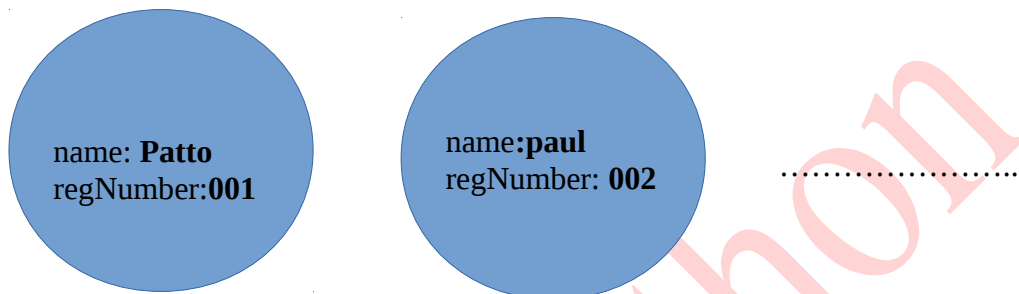
2. static variables /class level variables

3. local variables /inside methods

1. instance variable

1. If the value of variable is varied from object to object is instance variable

eg: if we have class student we can define any number of objects,



we can see that from object to object value of variable name and regNumber are different so we to declare as instance variable.

2. for every object a separate copy of instance will be created:

eg: if there are 1000 student, we have to create 1000 objects

3. In general inside constructor we have to declare by using **self**

4. how to access instance variables:

- .within the class by using self

- .outside of class by using reference variable

eg:

class Student:

def __init__(self,name,regNumber,marks):

self.name=name

self.regNumber=regNumber

self.marks=marks

def display(self):

print("student name:{}, Reg number:{}, marks:

{}".format(self.name,self.regNumber,self.marks))

s1 = Student('patto','00001',90)

s2 = Student('Raj','00002',88)

```
s1.display()
```

```
s2.display()
```

```
#To access the object outside of the class use reference variable
```

```
print(s1.name, s1.regNumber,s1.marks)
```

```
print(s2.name, s2.regNumber,s2.marks)
```

```
#To see related instance variable in form of dictionary ← --- objectreference.__dict__
```

```
print(s1.__dict__)
```

```
print(s2.__dict__)
```

output

```
student name:patto, Reg number:00001, marks:90
```

```
student name:Raj, Reg number:00002, marks:88
```

```
('patto', '00001', 90)
```

```
('Raj', '00002', 88)
```

```
{'regNumber': '00001', 'name': 'patto', 'marks': 90}
```

```
{'regNumber': '00002', 'name': 'Raj', 'marks': 88}
```

```
#for every instance variable separate copy proof
```

```
class Test:
```

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

```
        self.x=10
```

```
t1 =Test()
```

```
t2 =Test()
```

```
print(t1.x,t2.x)
```

```
t1.x=100
```

```
print(t1.x,t2.x)
```

10 100

10

output

```
(10, 10)
```

```
(100, 10)
```


#instance variable outside of constructor

class Test:

def __init__(self):

self.a=10

self.b=20

def n(self):

self.c=100

self.d=200

t1 =Test()

print(t1.__dict__)

t1.n()

print(t1.__dict__)

print(t1.a,t1.b,t1.c,t1.d)

output

{'a': 10, 'b': 20}

{'a': 10, 'c': 100, 'b': 20, 'd': 200}

(10, 20, 100, 200)

Where we can declare instance variable:

1. inside constructor
2. Inside method
3. outside of the class using reference variable(available only for the particular boject)

class Test:

def __init__(self):

self.a=10

def f(self):

self.b=777

```

t1 =Test()
print(t1.__dict__)
t1.a=20
t1.b=30 #instance varibale creation using reference variable outside the class
t1.c=40 # instance varibale creation using reference variable outside the class
print(t1.__dict__)
t1.f()
print(t1.__dict__)
t1.b=888 #to access the reference variable b from f()
print(t1.a,t1.b,t1.c,)

```

output

```

{'a': 10}
{'a': 20, 'c': 40, 'b': 30}
{'a': 20, 'c': 40, 'b': 777}
(20, 888, 40)

```

```

class Dev:
    def __init__(self,name,salary):
        self.name=name
        self.salary=salary

```

```

dev1 = Dev('patto',1000)
dev2 = Dev('Raj',2000)
print(dev1.__dict__)
print(dev2.__dict__)
dev1.pgm='python'
dev1.age=20
dev2.pgm='java'
print(dev1.__dict__)
print(dev2.__dict__)

```

output

```
{'salary': 1000, 'name': 'patto'}  
{'salary': 2000, 'name': 'Raj'}  
{'salary': 1000, 'age': 20, 'pgm': 'python', 'name': 'patto'}  
{'salary': 2000, 'pgm': 'java', 'name': 'Raj'}
```

How to remove instance variable from object:

1. **del** self.variable_name
2. **del** objectreference.variable_name #outside the class

```
class Test:  
    def __init__(self):  
        self.a=10  
        self.b=20  
        self.c=30  
    def f(self):  
        self.x=100
```

```
t1 =Test()  
t1.f()  
print(t1.__dict__)  
del t1.b #delete instance variable b for t1. This does not affect t2  
t2=Test()  
t2.f()  
print(t2.__dict__)  
print(t1.b)
```

output

```
{'a': 10, 'x': 100, 'c': 30, 'b': 20}  
{'a': 10, 'x': 100, 'c': 30, 'b': 20}  
Traceback (most recent call last):  
  File "test.py", line 16, in <module>  
    print(t1.b)  
AttributeError: Test instance has no attribute 'b'
```

!! if value of a variable is different for all objects we have to declare as instance variable(there is a separate copy for every object)

!! if value of a variable is fixed for all objects is never recommended to declare as instance variable(cause of separate copy : memory waste, performance degradation) we have at **class level as static variable**

only one copy of object will be created and shared among all objects

class Student:

collegename="Univ of Hyd" #static variable / class level variable

def __init__(self):

self.name=name

#instance variable / object level variable

self.regNumber=regNumber

eg:

class Test:

a=10

def __init__(self):

self.b=20

t1=Test()

t2 =Test()

print(t1.a,t1.b)

print(t2.a,t2.b)

Test.a= 100

t1.b=40

print(t1.a,t1.b)

print(t2.a,t2.b)

(10, 20)

(10, 20)

(100, 40)

(100, 20)

To print dict related to class variable:

print(classname.__dict__)

place to declare static variabe:

1. class level

```
class Test:  
    a=10
```

2. Inside constructor : using **class name**

```
class Test:  
    a=10  
    def __init__(self):  
        self.b=2  
        Test.c =3 #inside constructor
```

3. Inside instance method: using **class name**

```
class Test:  
    a=10  
    def __init__(self):  
        self.b=2  
        Test.c =3 #inside constructor  
    def f(self):  
        self.d=100  
        Test.e=1000 #inside instance method
```

4. Inside class method: using **cls** variable name or **class name**

```
class Test:  
    a=10  
    def __init__(self):  
        self.b=2  
        Test.c =3 #inside constructor  
    def f(self):
```

```
self.d=100  
Test.e=1000 #inside instance method
```

```
@classmethod #decorator  
def fc(cls): #cls is reference to current class, any name can be used.  
    cls.f=30  
    Test.g=40
```

```
t=Test()  
t.f()  
Test.fc()  
print(Test.__dict__)
```

```
{'__module__': '__main__', 'a': 10, '__init__': <function Test.__init__ at 0x7f24ead5e510>, 'f': 30,  
'fc': <classmethod object at 0x7f24ead63b00>, '__dict__': <attribute '__dict__' of 'Test' objects>,  
 '__weakref__': <attribute '__weakref__' of 'Test' objects>, '__doc__': None, 'c': 3, 'e': 1000, 'g': 40}
```

```
class Test:  
    a=10  
    def __init__(self):  
        self.b=2  
        Test.c =3 #inside constructor  
    def f(self):  
        self.d=100  
        Test.e=1000 #inside instance method  
    @classmethod  
    def fc(cls):  
        cls.f=30  
        Test.g=40
```

```
t=Test()  
t.f()  
Test.fc()  
print(Test.__dict__)
```

5. Inside static method: using **class name**

```
class Test:  
    a=10  
    def __init__(self):
```

```

    self.b=2
    Test.c =3 #inside constructor
def f(self):
    self.d=100
    Test.e=1000 #inside instance method
@classmethod
def fc(cls):
    cls.f=30
    Test.g=40

```

```

@staticmethod #this s option #it must be here for access using object reference, if not use classname
def st():
    Test.s=1000

```

```

t=Test()
t.f()
Test.fc()
Test.st()
print(Test.__dict__)

```

```

{'__module__': '__main__', 'a': 10, '__init__': <function Test.__init__ at 0x7fc182810510>, 'f': 30, 'fc': <classmethod object at 0x7fc182815be0>, 'st': <function Test.st at 0x7fc1828108c8>, '__dict__': <attribute '__dict__' of 'Test' objects>, '__weakref__': <attribute '__weakref__' of 'Test' objects>, '__doc__': None, 'c': 3, 'e': 1000, 'g': 40, 's': 1000}

```

6. Outside the class by using class **name.variable name**

```
Test.i=100
```

all static variable can be accessed using object reference

```

print(t.a,t.b,t.c,t.d,t.e,t.f,t.g,t.s)
>>8000000 2000000 333333 6000000 1000000 3000000 4000000 5000000

```

all static variable can be accessed using class name

```

print(Test.a,Test.c,Test.e,Test.f,Test.g,Test.s)
>>8000000 333333 1000000 3000000 4000000 5000000

```

AttributeError: !!!! if you try to access instance variable using class name

```
print(Test.b,Test.d)
```

Traceback (most recent call last):

File "test.py", line 21, in <module>

```
    print(Test.b,Test.d)
```

AttributeError: type object 'Test' has no attribute 'b'

How to access static variable:

```
class Test:
```

```
    a=8000000
```

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

```
        print(self.a)
```

```
        print(Test.a) #inside constructor
```

```
    def f(self):
```

```
        print(self.a)
```

```
        print(Test.a) #inside instance method
```

```
    @classmethod
```

```
    def fc(cls):
```

```
        print(cls.a)
```

```
        print(Test.a) #inside class method
```

```
    @staticmethod
```

#it must be here for access using object reference, if not use classname

```
    def st():
```

```
        print(Test.a) #inside static method
```

```
t=Test()
```

```
t.f()
```

```
t.fc()
```

```
t.st()
```

```
print(Test.a) #outside class
```

```
print(t.a)
```


8000000
8000000
8000000
8000000
8000000
8000000
8000000
8000000

where can you modify the static variable:

**every where using class name or inside class method
using cls variable**

!!you should not use self or object reference

if you use **object reference** it will create the **instance variable** with **same name** as **static variable** instead of modifying **static variable**

```
class Test:
```

```
    a=10
```

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

```
        self.b
```

```
t1=Test()
```

```
t1.a=2000 # this will not modify static variable a, this will create instance variable a  
for t1
```

eg:

```
class Test:
```

```
    a=10
```

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

```
        self.b=20
```

```
t1=Test()
```

```
t2=Test()
```

```
t1.a+=1    #11 , but it ll be saved as instance variable for t1 only
```

```
print(t1.a)
```

```
print(t2.a)
```

output

11

10

13. Regular expression:

if we want to represent a group of strings according to a particular pattern, we should use a **regular expression**

pattern ---- to represent a pattern we can write a **regular expression**

regular expression ← -- declarative mechanism to represent a group of string according to a particular pattern

eg: e-mail id : all email ids follow a particular pattern

zxxx12_ee.ee@domain.com

now we can write a regular expression to represent all valid email ids

whenever a group of strings follow a particular pattern we can use a regular expression to represent them eg: **country phone code, email ids, identifiers,**

Where we have to use regular expression:

1. validations framework
2. To check whether a particular pattern in text : find command, ctrl f, grep
3. Translators like compilers, interpreters, assemblers,....

Compiler design steps:

1. **lexical Analysis** ← **scanning(text scanner) : Tokenization : Regular expression**

2.Syntax Analysis <----parsing

3.Semantic Analysis

4.ICG ← --intermediate code generation

5.CO ← --code optimization

6.Symbol table

4. To develop digital circuit

FA with output ← -Moore and Mula

Binary incrementor

Binary adder

5. Communication protocol (set of rules): **Tcp/ip,.....**

6. Generate otp

if you want to use Regular expression in apps development python provide a module

re ← ---module: contains several functions to use

re ← --- module

compile()

finditer()

start()

end()

group()

} functions

1step: **compiler()** ← --- change a pattern to python object (**regex object**)

if you have to search the word **python**

pattern = **re.compile('python')**

import re

pattern = re.compile('python')

print(type(pattern))

output

<class '_sre.SRE_Pattern'>

2nd step: **finditer()** ← ----- how many matches are available

```
matcher = pattern.finditer('Learning python is very easy')
```

if you find matches:

3rd step: **start()** ← -----index of the match

end() ← -----end+1 index of the match

group() ← -- returns matched string(eg: return all digits in a file)

eg: word : **ab**

target : **abababccccab**

```
import re
```

```
count = 0
```

```
pattern = re.compile('ab')
```

```
matcher = pattern.finditer('abababccccab')
```

```
for match in matcher:
```

```
    count += 1
```

```
    print('Match is available at start index:',match.start())
```

```
print('print number of occurrences:',count)
```

output

Match is available at start index: 0

Match is available at start index: 2

Match is available at start index: 4

Match is available at start index: 11

print number of occurrences: 4

can you write a python program to check if a particular word is available or not, if it is available, where is it and how many times it is available(pattern matching app using functions)

```
import re
count = 0

pattern = re.compile('ab')
matcher = pattern.finditer('abababccccab')
for match in matcher:
    count += 1
    print('start:{} end:{} group:{}'.format(match.start(),match.end(),match.group()))

print('print number of occurrences:',count)
```

output

```
start:0 end:2 group:ab
start:2 end:4 group:ab
start:4 end:6 group:ab
start:11 end:13 group:ab
print number of occurrences: 4
```

How to search any type or group of symbols:

use of character classes:

[abc] ← ---- either a or b or c

[^abc] ← ---- except a and b and c

[a-z] ← --- any **lower** case alphabet symbols

[A-Z] ← ---- any **upper case** alphabet symbols

[a-zA-Z] ← any alphabet symbol(lower case or upper case)

[0-9] ← any digit

[a-zA-Z0-9] ← any alphanumeric character

[^a-zA-Z0-9] ← special characters(all character except alphanumeric)

Predefined character classes:

\s ← space

\S ← except space characters

\d ← any digit

\D ← except digits

\w ← any word(alphanumeric)

\W ← (special symbols)any character except word

. ← every character

eg:

```
import re
matcher = re.finditer('[abc]','abcdfetc')
for match in matcher:
    print(match.start(), '...', match.group())
```

output

0 a

1 b

2 c

7 c

```
import re
```

```
matcher = re.finditer('[^abc]','abcdfetc')
for match in matcher:
    print(match.start(), '...', match.group())
```

output

```
3 .... d
4 .... f
5 .... e
6 .... t
```

```
import re
```

```
matcher = re.finditer('[0-9]','abcdf10tc')
for match in matcher:
    print(match.start(), '...', match.group())
```

output

```
6 .... 1
7 .... 0
```

Quantifiers:

number of occurrences

^a ← check if string starts with **a** or not

a\$ ← -- check if string ends with **a** or not

a ← --- exactly one **'a'**

a+ ← --- at least one **a**

'a'

'aaa'

'aa'

```
import re
```

```
matcher = re.finditer('a+', 'abcaadfe10tcaaa')
```

```
for match in matcher:
```

```
    print(match.start(), '...', match.group())
```

```
0 .... a
```

```
3 .... aa
```

```
12 .... aaa
```

a* ← -- any number of 'a' including zero number of a's

```
import re
matcher = re.finditer('a*','abcaadfe10tcaaa')
for match in matcher:
    print(match.start(), '...', match.group())
```

```
0 .... a
3 .... aa
12 .... aaa
0 .... a
1 ....
2 ....
3 .... aa
5 ....
6 ....
7 ....
8 ....
9 ....
10 ....
11 ....
12 .... aaa
15 ....
```

a? ← ---- atmost one a, either one a or zero number of a's

```
import re

matcher = re.finditer('a?','abcaadfe10tcaaa')
for match in matcher:
    print(match.start(), '...', match.group())
```

```
output
0 .... a
1 ....
2 ....
3 .... a
4 .... a
5 ....
6 ....
```



```
7 ....
8 ....
9 ....
10 ....
11 ....
12 .... a
13 .... a
14 .... a
15 ....
```

a{n} ← -- exactly n number of a's

```
import re
```

```
matcher = re.finditer('a{3}','abcaadfe10tcaaa')
for match in matcher:
    print(match.start(), '...', match.group())
```

output
12 aaa

a{m,n} ← minimum **m** number of a's and **maximum n** numbers of a's

```
import re
```

```
matcher = re.finditer('a{2,3}','abcaadfe10tcaaa')
for match in matcher:
    print(match.start(), '...', match.group())
```

output
3 aa
12 aaa

a{2}a* ← -- minimum 2 a then after any number of a's

important functions of re module:

- 1.match()
- 2.fullmatch()
- 3.search()
- 4.findall()
- 5.finditer()
- 6.sub()
- 7.subn()
- 8.split()
- 9.compile()

1.match() ← -- to check the given pattern at the beginning of target string, if it is available return Match object else return None

```
import re
s=input("Enter pattern to check:")
m=re.match(s,'welcome to python class')
if m!=None:
    print('Match is available at the beginning')
    print('start index:{} end index:{}'.format(m.start(),m.end()))
else:
    print("Match is not available")
```

output

```
Enter pattern to check:welco
Match is available at the beginning
start index:0 end index:5
```

2.fullmatch() ← -- complete pattern should match the target else return None

3.search() ← --- search a given pattern to a string, if it is there return match object at first occurrence else return None

```
import re
```

```
s=input("Enter pattern to check:")
m=re.search(s,'welcome')
if m!=None:
    print('Match is available at the beginning')
    print('start index:{} end index:{}'.format(m.start(),m.end()))
else:
    print("Match is not available")
```

output

Enter pattern to check:me
Match is available at the beginning
start index:5 end index:7

4.findall() ← -- find all matches and return a list

```
import re
l =re.findall('[0-9]','11mymail09@gmail.com')
print(l)
```

output

['1', '1', '0', '9']

14. Multithreading:

Multitasking: Executing several tasks simultaneously is the concept of multitasking

Rule I: wherever a group of independent jobs are there don't ever execute one by one

Rule II: wherever dependency is there we should never use the concept of multitasking

!!!!!!avoid leader or manager attitude to apply multitask everywhere

eg: Baby ----> 1 mother --> 9 months

9 mothers -----1 month--->baby

adv: reduce execution time and improve performance

there are 2 types of multitasking:

1. process based multitasking
2. Thread based multitasking

1. process based multitasking:

multitasking where each task is a separate independent process
most of the time at Os level

Downloading a file from internet at same time listen to audio music

2. Thread based multitasking

multitasking at programmatic level

each task is separate independent part of same program.

Each independent part is called a **thread**

eg: if you hv to search in all drivers and there are many files , you can make a thread to search in D:\ another to search in c:\ that will be executed at same time

multithread concept used in fields:

1. Animations
2. Multi media graphics
3. Video games
4. Web servers and applications servers

python provides module **threading**

by default every python pgm contains 1 thread : MainThread

Wat is a thread

1. Thread is a flow of execution
2. for every thread, there is some job available
3. Job

single threaded program: only one thread. one by one. One flow of execution

multi threaded pgm: multiple flow of execution. Each flow is responsible to do some task

```
current_thread().getName()
```

```
import threading  
print('Current thread:',threading.current_thread().getName())
```

output

Current thread: MainThread

Java

```
class Test  
{  
    public static void main(String args[] )  
    {  
        System.out.println(Thread.currentThread().getName());  
    }  
}
```

output

main

if current_thread will return current thread obj then in case of multithread environment also it will give always the current thread name.... Because this code will be executed by any one thread at a time

we can create thread in python using 3 ways:

- 1. creating a thread without using any class**
- 2. creating a thread by extending Thread class**
- 3. creating a thread without extending thread class**

1. creating a thread without using any class

```
from threading import *  
def display():  
    print('Current thread:',current_thread().getName())
```

#MainThread creates child thread

#Thread is predefined class

```
t=Thread(target=display)
```

#MainThread starts ChildThread

```
t.start()
print('This code executed by thread:',current_thread().getName())
```

output

Current thread: Thread-1

This code executed by thread: MainThread

eg2:

```
from threading import *
def display():
    for i in range(10):
        print('display')
```

```
#MainThread creates child thread
#Thread is predefined class
t=Thread(target=display)
#MainThread starts ChildThread
t.start()
for i in range(5):
    print('Not display')
```

output

display
Not display
Not display
display
display
Not display
display
Not display
Not display
display
display
display
display
display
display

we can not guarantee order of execution

2. creating a thread by extending Thread class

```
class A:  
    stmt-1  
    stmt-2
```

```
class B(A):  
    stmt-1  
    stmt-2
```

inheritance : class A is parent of B. B is child class of A and inherit properties of A

in java :

class A extends B

```
{  
}
```

```
from threading import *
```

```
#define child class for thread class
```

```
class myThread(Thread):
```

```
    #override run method
```

```
    #you can't change the method name run, if you do child class will not be executed
```

```
    def run(self):
```

```
        for i in range(10):
```

```
            print('child thread')
```

```
#MainThread creates child thread
```

```
t=myThread()
```

```
#MainThread starts child thread
```

```
t.start()
```

```
for i in range(10):
```

```
    print('main Thread')
```

output

child thread

main Thread

child thread

main Thread

main Thread

child thread

```
child thread
main Thread
child thread
main Thread
child thread
main Thread
child thread
main Thread
child thread
main Thread
child thread
main Thread
main Thread
child thread
```

3. creating a thread without extending thread class

```
from threading import *
class Test:
    def display(self):
        for i in range(5):
            print('Child Thread')

obj=Test()
t=Thread(target=obj.display)
t.start()
```

```
for i in range(5):
    print('Main Thread')
```

output

```
Child Thread
Main Thread
Child Thread
Main Thread
```


Main Thread
Main Thread
Main Thread
Child Thread
Child Thread
Child Thread

Many treads

```
from threading import *  
class Test:  
    def display(self):  
        for i in range(5):  
            print('executed by thread:',current_thread().getName())
```

```
obj=Test()  
t1=Thread(target=obj.display)  
t2=Thread(target=obj.display)  
t3=Thread(target=obj.display)  
t4=Thread(target=obj.display)  
t1.start()  
t2.start()  
t3.start()  
t4.start()
```

output

executed by thread: Thread-1
executed by thread: Thread-2
executed by thread: Thread-1
executed by thread: Thread-3
executed by thread: Thread-2
executed by thread: Thread-1
executed by thread: Thread-3
executed by thread: Thread-2
executed by thread: Thread-1
executed by thread: Thread-1
executed by thread: Thread-3
executed by thread: Thread-4

executed by thread: Thread-2
 executed by thread: Thread-3
 executed by thread: Thread-3
 executed by thread: Thread-4
 executed by thread: Thread-2
 executed by thread: Thread-4
 executed by thread: Thread-4
 executed by thread: Thread-4

Pgm (no multithreading)	Pgm (multithreading)
<pre> import time def double(numbers): for n in numbers: time.sleep(1) print('Double :',2*n) def square(numbers): for n in numbers: time.sleep(1) print('square',n*n) numbers=[1,2,3,4,5] starttime=time.time() double(numbers) square(numbers) endtime=time.time() print('Total_time :',endtime-starttime) output Double : 2 Double : 4 Double : 6 Double : 8 Double : 10 square 1 square 4 square 9 square 16 square 25 </pre>	<pre> import time from threading import * def double(numbers): for n in numbers: time.sleep(1) print('Double :',2*n) def square(numbers): for n in numbers: time.sleep(1) print('square',n*n) numbers=[1,2,3,4,5] starttime=time.time() #target variable always expect function name #args always takes tuple as argument t1=Thread(target=double,args=(numbers,)) #args always takes tuple as argument t2=Thread(target=square,args=(numbers,)) t1.start() t2.start() t1.join() #Main thread wait till child execute t2.join() #Main thread wait till child execute endtime=time.time() print('Total_time :',endtime-starttime) output Double : 2 square 1 Double : 4 </pre>

Total_time : 10.011852025985718	square 4 square 9 Double : 6 Double : 8 square 16 Double : 10 square 25 Total_time : 5.00614857673645
--	---

How to get Thread name and to change thread name:

t.getName() or t.name

t.setName()

eg:

```
from threading import *
print(current_thread().getName())
current_thread().setName('patto_thread')
print(current_thread().name)
```

output

MainThread
patto_thread

16. Python Database Programming(PDBC):

It is very common to save data for future purpose

1. Temporary storage area

2. permanent storage area

1. Temporary storage area

#once pgm completed data will not be available

dict={}

```
for i in range(5):
    inp = input('Enter Name:')
    dict[i]=inp
print(dict)
output
{0: 'tutu', 1: 'yoyo', 2: 'roro', 3: 'wowo', 4: 'xoxo'}
#once pgm completed data will not be available
```

2. permanent storage area

. file systems

.databases

. file systems : best suitable to store less amount of data

Limitations:

1. No Ql support
2. security ← --- many files can be accessed with no password
3. No mechanism to prevent duplicate data, redundancy : data inconsistency
4. Huge data handling very difficult, opening or process huge file is time consuming.

.databases:

- 1.store huge amount of data
- 2.Ql support
- 3.security is more
4. tables structure, no duplication

Limitations:

1. databases can't hold very huge/large amount of data
2. only structured data
3. No support for semi structure data ← --xml
4. No support for unstructure data ← --video . Audio

unstructured : usually refers to information that doesn't reside in a traditional row-column **database.** For example, **data** stored in XML and **JSON** documents, CSV files, and Excel files is all **unstructured**.

To overcome limitation of database we use **Advanced data storage Technology**

Python Database connectivity [PDBC:](#)

To talk to the database **SQL commands** are used. **Python** is used to send those SQL commands to databases

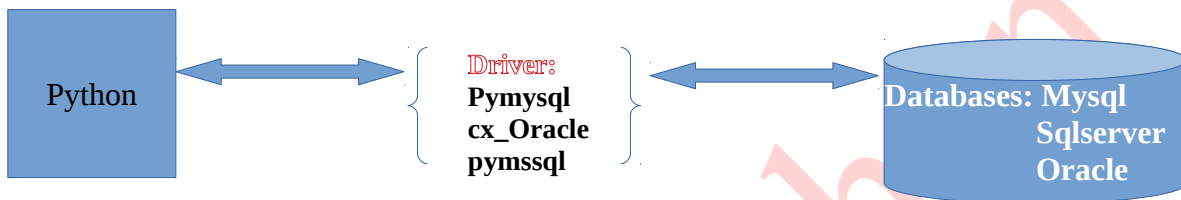
Python provides inbuilt support(modules) for several databases:

modules

cx_Oracle ← oracle database

pymssql ← Microsoft sqlserver

pymysql ← Mysql



General standard steps to communicate with databases:

1. import that database module

eg: `import pymysql`
`import pymssql`
`import cx_Oracle`

2. with that module, establish connection between python and the database

eg: `con = cx_Oracle.connect(database information)` : `con = cx_Oracle.connect('username/password@localhost')`
`con = pymysql.connect(database information)`

3. Cursor Object ←-- to execute SQL queries

eg:
`cu = con.cursor()`

4. Execute SQL query

`cu.execute(sqlquery)` ← --- to execute a single query

`cu.executescript(sqlqueries)` ← to execute a string of sql queries separated by coma

`cu.executemany()` ← to execute a parameterized query

5. Fetch ←--- to get the result

`cu.fetchone()` ← to fetch only one result

`cu.fetchall()` ← to fetch all results. (All rows)

`cu.fetchmany(n)` ← to fetch n rows. (only n rows)

6. commit() ← - confirm the result to be reflected in database(in jdbc autocommit is enable but not in python)

cu.commit() ← save permanently to database, you can not rollback

rollback() ← this is like undo all modifications if you did not commit

7. close connection

cu.close() ← close the cursor

con.close() ← close the connection

all commands

connect()
cursor()
execute()
executescript()
executemany()
fetchone()
fetchall()
fetchmany()
commit()
rollback()
close()

Working with Oracle database:

install driver/module : **sudo pip install cx_Oracle**

to check all modules >> **print(help('modules'))**

#write a pgm to connect to oracle database and print its version

```
import cx_Oracle as cx
```

```
con= cx.connect('username/password@localhost')
```

```
if con != None:
```

```
    print('connection established successfully')
```

```
    print('version is :', con.version)
```

```
else:
```

```
    print('connection refused!!')
```

```
con.close()
```

```

#write a pgm to create employees table oracle database
import cx_Oracle as cx
try:
    con= cx.connect('username/password@localhost')
    cu = con.cursor()
    query=""" create table employees(eid number,ename varchar2(10),esalary number, eaddress
varchar2(10)) """
    cu.execute(query)
    print("Table created successfully")
except cx_Oracle.DatabaseError as e:
    if con:
        con.rollback()
        print("there is a problem",e)

finally:
    if cu:
        cu.close()
    if con:
        con.close()

```

Sql language statements

DDL ← **data definition language** (used to define the database structure. Any CREATE, DROP and ALTER)

DML ← **data manipulation language** (is used to access, modify or retrieve the data from the database, INSERT, UPDATE ,DELETE,SELECT)

in python all **DML** commands require **commit()** after to save it to database

insert multiple record into database ← - **executemany()** method

query=insert into employees values(:eid, :ename, :esalary, :eaddress)

records =[(1,'patto',1000,'Hyd'),(2,'paul',2000,'Mumbai'),(3,'Raj',3000,'Bangalore')]

#execute query for every record in list

cursor.executemany(query,records)

cursor.commit()

#write a pgm to enter multiple records in employees table oracle database

```
import cx_Oracle as cx
```

```
try:
```

```
    con= cx.connect('username/password@localhost')
```

```
    cursor = con.cursor()
```

```
    query=insert into employees values(:eid, :ename, :esalary, :eaddress)
```

```
    records =[(1,'patto',1000,'Hyd'),(2,'paul',2000,'Mumbai'),(3,'Raj',3000,'Bangalore')]
```

```
    #execute query for every record in list
```

```
    cursor.executemany(query,records)
```

```
    cursor.commit()
```

```
    print("Records inserted successfully")
```

```
except cx_Oracle.DatabaseError as e:
```

```
    if con:
```

```
        con.rollback()
```

```
        print("there is a problem",e)
```

```
finally:
```

```
    if cursor:
```

```
        cursor.close()
```

```
    if con:
```

```
        con.close()
```

#write a pgm to **insert** multiple records in employees table oracle database
from keyboard(dynamic input)

```
import cx_Oracle as cx
```

```
try:
```

```
    con= cx.connect('username/password@localhost')
```

```
    cursor = con.cursor()
```

```
    while True:
```

```
        eid=int(input("Enter emp ID:"))
```

```
        ename=input("Enter emp name :")
```

```
        esalary=int(input("Enter emp salary :"))
```



```

eaddress=input("Enter emp address :")
query="insert into employees values(%d, '%s', %d, '%s')"
#execute query for every record in list
cursor.execute(query %(eid,ename,esalary,eaddress))
cursor.commit()
print("Records inserted successfully")
check = input("Do you want to insert another record [Yes | No] :")
if check == 'No' or 'no':
    break
except cx.DatabaseError as e:
    if con:
        con.rollback()
        print("there is a problem",e)

finally:
    if cursor:
        cursor.close()
    if con:
        con.close()

```

#write a pgm to **update** salary in employees table oracle database from keyboard(dynamic input) where salary is under given range.

```

import cx_Oracle as cx
try:
    con= cx.connect('username/password@localhost')
    cursor = con.cursor()
    increment=float(input("Enter increment amount:"))
    salrange=float(input("Enter salary range :"))
    query="update employees set esalary=esalary+%f where esalary<%f"
    cursor.execute(query %(increment,salrange))
    cursor.commit()
    print("Records updated successfully")
except cx.DatabaseError as e:
    if con:
        con.rollback()
        print("there is a problem",e)

finally:
    if cursor:
        cursor.close()

```

```
if con:
    con.close()
```

#write a pgm to **Delete** salary in employees table oracle database where salary is under given range(dynamic input).

```
import cx_Oracle as cx
try:
    con= cx.connect('username/password@localhost')
    cursor = con.cursor()
    cutoff=float(input("Enter the cutoff salary :"))
    query="delete from employees where esalary>%"
    cursor.execute(query %cutoff)
    cursor.commit()
    print("Records updated successfully")
except cx.DatabaseError as e:
    if con:
        con.rollback()
        print("there is a problem",e)

finally:
    if cursor:
        cursor.close()
    if con:
        con.close()
```

#write a pgm to **Select** all data in employees using **fetchone()**

```
import cx_Oracle as cx
try:
    con= cx.connect('username/password@localhost')
    cursor = con.cursor()
    query="select * from employees"
    cursor.execute(query)
    row=cursor.fetchone()
    while row is not None:
```

```
        print(row)
        row=cursor.fetchone()
except cx.DatabaseError as e:
    if con:
        con.rollback()
        print("there is a problem",e)
```

```
finally:
    if cursor:
        cursor.close()
    if con:
        con.close()
```

#write a pgm to **Select** all data in employees using **fetchall()**

```
import cx_Oracle as cx
try:
    con= cx.connect('username/password@localhost')
    cursor = con.cursor()
    query="select * from employees"
    cursor.execute(query)
    rows=cursor.fetchall()
    #fetchall returns a list
    for row in rows:
        print('eid:',row[0])
        print('ename:',row[1])
        print('esalary:',row[2])
        print('eaddress:',row[3])
except cx.DatabaseError as e:
    if con:
        con.rollback()
        print("there is a problem",e)

finally:
    if cursor:
        cursor.close()
    if con:
        con.close()
```

#write a pgm to **Select** all data in employees using **fetchmany()**

```
import cx_Oracle as cx
try:
    con= cx.connect('username/password@localhost')
    cursor = con.cursor()
    query="select * from employees"
    cursor.execute(query)
    n=int(input('Enter number of rows :'))
    rows=cursor.fetchmany(n)
    #fetchall returns a list
    for row in rows:
        print('eid:',row[0])
        print('ename:',row[1])
        print('esalary:',row[2])
        print('eaddress:',row[3])

except cx.DatabaseError as e:
    if con:
        con.rollback()
        print("there is a problem",e)

finally:
    if cursor:
        cursor.close()
    if con:
        con.close()
```

Deep copy an Shallow copy:

```
l1=[10,20,30,40]
l2=l1 #l1 and l2 point to the same object (duplicate reference)(if you modify one it will affect
the other)
print(l1)
print(l2)
```

```
>>[10,20,30,40]
>>[10,20,30,40]
```

```
l2[0]=333
print(l1)
print(l2)
>>[333, 20, 30, 40]
>>[333, 20, 30, 40]
```

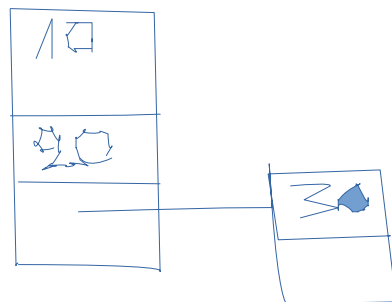
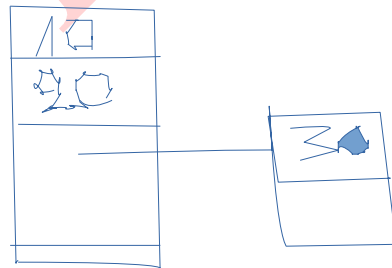
However.

```
l3=l2.copy() #cloning l2 and l3 are pointing to different object
l3[0]=111
print(l2)
print(l3)
>>[333, 20, 30, 40]
>>[111, 20, 30, 40]
```

```
import copy
l1=[10,20,[30,40],50]
l2=copy.copy(l1)
l1[2][0]=111
print(l1)
print(l2)
>>[10, 20, [111, 40], 50]
>>[10, 20, [111, 40], 50]
```

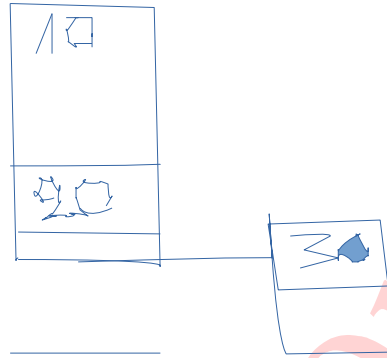
But

```
import copy
l1=[10,20,[30,40],50]
l2=copy.copy(l1)
l1[2]=111
print(l1)
```



```
print(l2)
>>[10, 20, 111, 50]
>>[10, 20, [30, 40], 50]
```

```
import copy
l1=[10,20,[30,40],50]
l2=copy.deepcopy(l1)
l1[2][0]=111
print(l1)
print(l2)
>>[10, 20, [111, 40], 50]
>>[10, 20, [30, 40], 50]
```



shallow Copy:

if the original object contains any reference to mutable object, just duplicate reference variable(No duplicate object creation).

some references:

https://www.python-course.eu/why_python.php

https://www.python-course.eu/object_oriented_programming.php

<https://realpython.com/python-virtual-environments-a-primer/>

Patto Python

3 4 /