

Introduction to Python

Introduction:

- Using Programming languages and technologies we develop applications.
- Applications are used to store data and perform operations on data.

Types of applications:

Standalone apps:

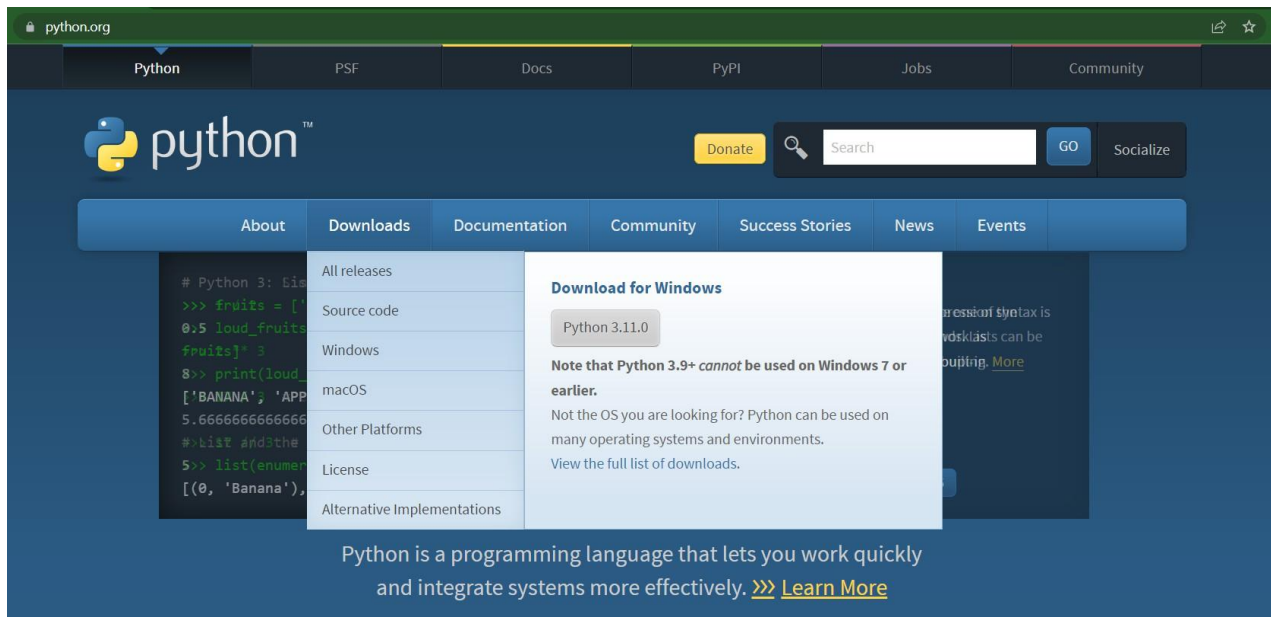
- The application runs from single machine.
- Internet connection is not required to run the application.
- Application needs to be installed on machine.
- **Examples:** VLC, MS-office, Anti-virus, Browser, **Programming languages.**

Web apps:

- The application runs from multiple machines in a network.
- Internet connection is required to run the application.
- Application installed in server and run from the clients.
- **Examples:** Gmail, YouTube, IRCTC, Flip Kart etc.

Download python:

- Python is an Open-Source Technology.
- We can download and install Python software from official website www.python.org

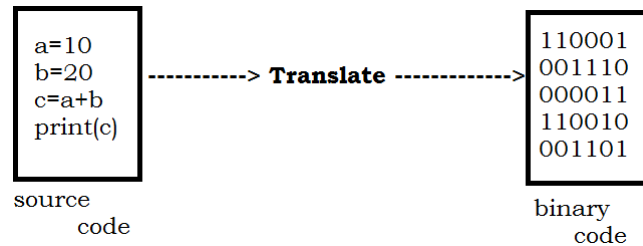


Python is used to develop both Standalone and Web applications:

- Core + Advance python + GUI + DBMS = Standalone app development
- Core + Advance + DBMS + HTML + CSS + JavaScript + Django = Web app develop

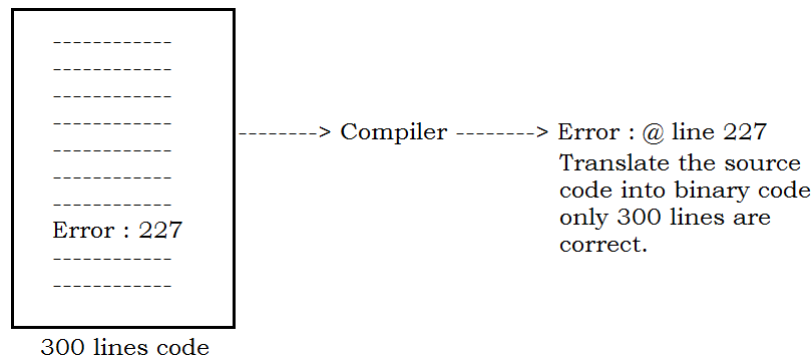
Translators:

- Programmer can define only source code.
- We need to convert the source code into binary code before run.
- We use 2 translators to convert Source code into byte code.
 - Compiler
 - Interpreter



Compiler:

- Compiler checks the source code syntactically correct or not.
- If we define the code correctly, it converts source code into byte code.
- Compiler shows error message with line number if there is a syntax error.



Note: Java programming language use compilation

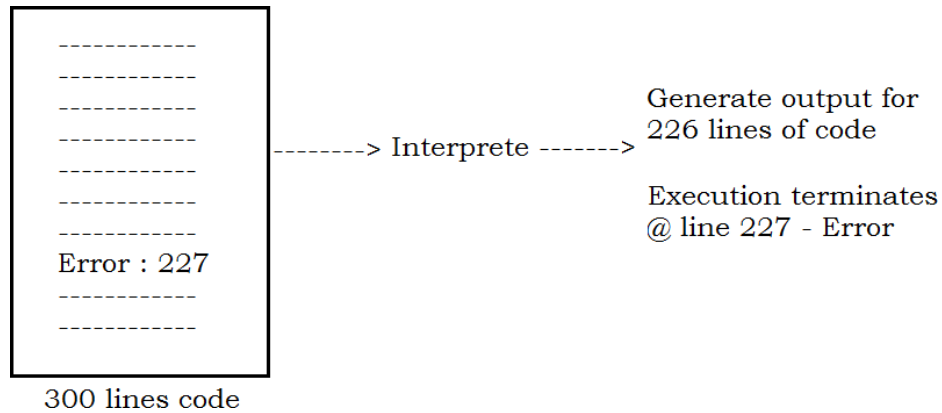
class Program

```
{  
    public static void main(String[] args)  
    {  
        int a=10;  
        System.out.println("a val : " + a);  
        int b=20;  
        System.out.println("b val : " + b);  
        System.out.println("c val : " + c);  
    }  
}
```

Compile: Error @ line – 11 (variable “c” not present)

Interpreter:

- Line by line translation of source code into binary code.
- Python uses interpreter for program execution.

**Note: Python programming uses interpretation**

```
a=10
print("a val :",a)
b=20
print("b val :",b)
print("c val :",c)
```

Output: a val : 10
b val : 20
NameError: name 'c' is not defined

Python(Programming & Scripting):

- Programming language are directly used to develop applications.
 - **Examples:** C, C++, PythonJava, .Net etc.
- Scripting languages always run from another program.
 - **Examples:** JavaScript, TypeScript, Python....

Program:

- A set of instructions.
- Program runs alone.

Script:

- Script is a set of Instructions
- Scripts is a program that always execute from another program.
- JavaScript is the best example of Scritping language.
- JavaScript code always run with HTML program.

web.html

```
<html>
  <head>
    <script>
      java script
      logic
    </script>
  </head>
  <body>
    .....
    .....
  </body>
</html>
```

1. Java Script code cannot run alone.
2. It always execute from HTML file
3. Python code can be used as a script from other applications such as DEVOP, AWS, SELENIUM.....

Python is Dynamic:

- Every programming language is used to develop applications
- Application is used to store and process the data.
- Generally we allocate memory to variables in 2 ways
 1. Static memory allocation
 2. Dynamic memory allocation

Static memory:

- Static means "fixed memory"
- The languages which are supporting primitive types allowed allocating static memory.
- Primitive variable size and type are fixed.
- Primitive variable stores data directly.
- Compiler raises error when data limit or type is deviated from specified.

Primitive : Variable stores the data

In C :

```
int a ;
```

```
a = 10 ;
```

```
a = a+15 ;
```

```
a = 23.45 ; -> Error : only integer allowed
```

```
a = 50000 ; -> Error : Can store a value between -32768 to +32767
```

a (2 bytes)

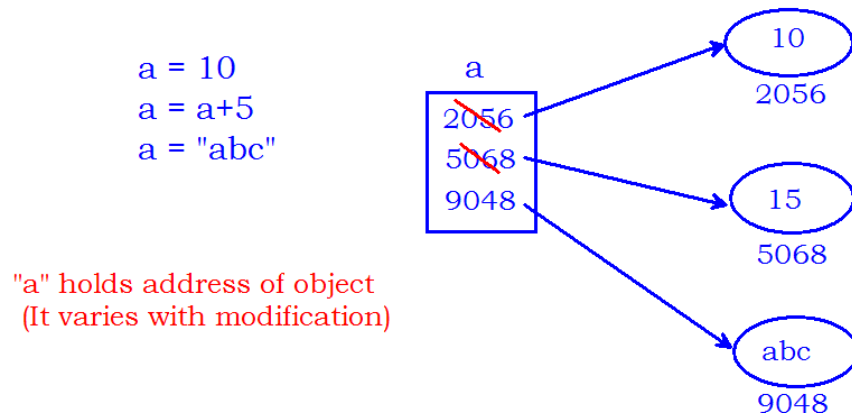
~~10~~ 25

2046

Dynamic memory:

- Python is dynamic.
- Dynamic memory means type and size of data can vary.
- Python can store information in Object format.
- Dynamic variables cannot store the data directly.
- Dynamic variables store the reference of Object which holds data.
- In Python, object location changes every time when we modify the data.

Dynamic : Variable holds the reference of data object.



```
>>> a=10
>>> print(a)
10
>>> print("Address:",id(a))
Address : 1628169120
```

```
>>> a=a+15
>>> print(a)
25
>>> print("Address:",id(a))
Address : 1628169360
```

```
>>> a="python"
>>> print(a)
python
>>> print("Address:",id(a))
Address : 48576832
```

Python Variables

Variable:

- Variable is an identity of memory location.
- Variables used to store values
- You can assign any value to a variable using the "=" operator.

Example:

```
x = 10
```

Variables can be of different types in Python, such as integer, float, string, boolean, etc.

Example:

```
age = 25  
height = 5.7  
name = "Amar"  
is_student = True
```

You can assign the same value to multiple variables at once using the "=" operator.

Example:

```
x = y = z = 0
```

Variables can be updated with new values as the program runs.

Example:

```
x = 10  
x = x + 1
```

Variables can be deleted using the "del" keyword.

Example:

```
x = 10  
del x
```

Python allows you to assign values to variables in a single line

Example:

```
x, y, z = 10, 20, 30
```

Python variables are case-sensitive, which means "a" and "A" are different variables.

Example:

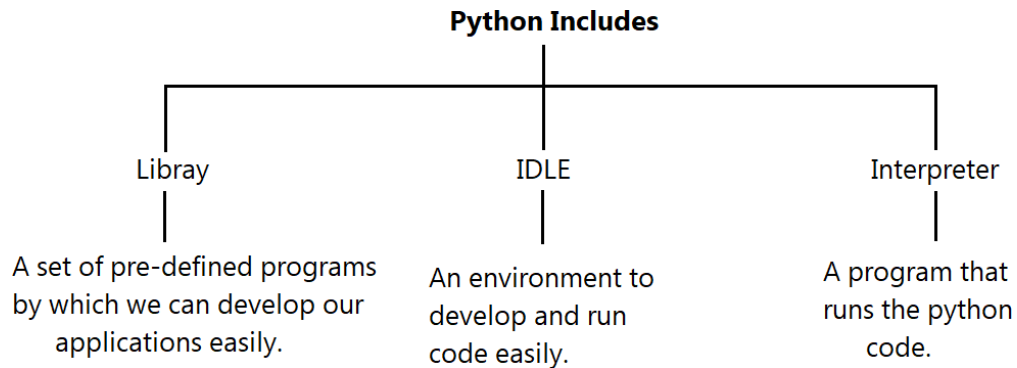
```
a = 10  
A = 20
```

You can use underscores in variable names for better readability.

Example:

```
my_variable = 10
```

Edit and Run python program:



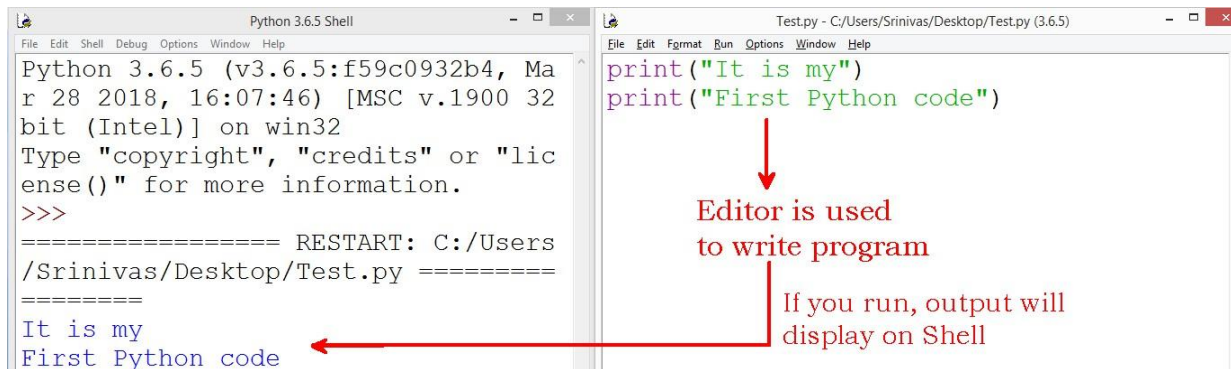
Working with IDLE:

- Once python installed, we can open IDLE by searching its name.
- A shell window will be opened where we can run only commands.

```
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 16:07:46) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> # We cannot write programs here
>>> # we execute simple commands
>>> print("Hello")
Hello
>>> 10+20
30
>>> len("Python")
6
```

Writing and executing programs:

- We can write and execute programs from editor
- Go to File menu
- Select "new" file – Opens an editor
- Write code and Save with .py extension
- Run – with shortcut f5



Operators and Control Statements

Operator:

- Operator is a symbol that performs operation on one or more operands.
- The member on which operator operates is called the operand.
- In the expression $a = 5 + 9$,
a, 5, 9 are operands
=, + are operators

Python supports the following operators:

1. Arithmetic Operators
2. Comparison (Relational) Operators
3. Assignment Operators
4. Logical Operators
5. Bitwise Operators
6. Membership Operators
7. Identity Operators

Arithmetic operators:

Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication etc.

Operators are +, -, *, /, %, //, **

Operator	Meaning	Example
+	Add 2 operands	$X + Y$
-	Subtract right from left	$X - Y$
*	Multiply 2 operands	$X * Y$
/	Divide left with right	X / Y
%	Divide and returns Remainder	$X \% Y$
//	Floor division - division that results into whole number adjusted to the left in the number line	$X // Y$
**	Exponent - left operand raised to the power of right	$X ** Y$

a = 5 , b = 2

$\begin{array}{r} 2) \ 5 \ (2.5 \\ \underline{5} \\ 0 \end{array}$ a/b = 2.5	$\begin{array}{r} 2) \ 5 \ (2 \\ \underline{4} \\ 1 \end{array}$ a%b = 1	$\begin{array}{r} 2) \ 5 \ (2.5 \\ \underline{5} \\ 0 \end{array}$ a//b = 2 ← floor value of 2.5	$5^2 = 25$ a**b = 25
---	---	---	--------------------------------

Division (/) : Operator divide and returns quotient. Result is float value.

Remainder (%): It returns the remainder after division. It performs operation only on integers.

Division	Mod
>>> 5/2 2.5	>>> 5%2 1
>>> 10/3 3.3333333333333335	>>> 10%4 2
>>> 10/5 2.0	>>> 5.0%2.5 0.0
	>>> 5.0%2 1.0

Floor division (/): Returns floor value after divide.

Exponent ():** returns the power value of specified base.

>>> 10/3 3.3333333333333335	>>> 2**2 4
>>> 10//3 3	>>> 2**4 16
>>> 10/4 2.5	>>> 3**3 27
>>> 10//4 2	

print("Arithmetic operations")

print("5+2 :", 5+2)

print("5-2 :", 5-2)

print("5*2 :", 5*2)

print("5/2 :", 5/2)

print("5%2 :", 5%2)

print("5//2 :", 5//2)

print("5**2 :", 5**2)

Complete this work sheet Arithmetic Operators

a=10 a=20 a=30 a=40 a=50 print(a)	a <input type="text"/>	a=5 a=a+1 a=a+1 a=a+1 a=a+1 print(a)	a <input type="text"/>
a=5 a=a+1 a=a+2 a=a+3 a=a+4 print(a)	a <input type="text"/>	a=15 a=a+5 a=a+4 a=a+3 a=a+4 print(a)	a <input type="text"/>

a, x = 5, 1 a=a+x x = x+1 a=a+x x = x+1 a=a+x print(a, x)	<div>a</div> <div></div> <div>x</div> <div></div>	a, b = 5, 1 a=a+b b = b-1 a=a+b b= b-1 a=a+b print(a, b)	<div>a</div> <div></div> <div>b</div> <div></div>
n=2; int s=n*n; print(s);	<div>n</div> <div></div> <div>s</div> <div></div>	n=2; int c=n*n*n; print(c);	<div>n</div> <div></div> <div>c</div> <div></div>
n=2; int s=n*n; int c=n*n*n; print(s+c);	<div>s</div> <div></div> <div>c</div> <div></div>	bal=5000; int amt=3500; bal = bal + amt; print(bal);	<div>bal</div> <div></div> <div>amt</div> <div></div>
a=5, b=3; int c=a+b; print(c);	<div>a</div> <div></div> <div>b</div> <div></div> <div>c</div> <div></div>	a=5, b=3; a=b; b=a; print(a,b);	<div>a</div> <div></div> <div>b</div> <div></div>
a=5, b=3, c; c=a; a=b; b=c; print(a,b);	<div>a</div> <div></div> <div>b</div> <div></div> <div>c</div> <div></div>	a=2, b=3; a=a+b; b=a+a; print(a,b);	<div>a</div> <div></div> <div>b</div> <div></div>
a=2, b=3; a=a+b; b=a-b; a=a-b; print(a,b);	<div>a</div> <div></div> <div>b</div> <div></div>	a=2, b=3; a=a*b; b=a//b; a=a//b; print(a,b);	<div>a</div> <div></div> <div>b</div> <div></div>
n=234; int d=n%10; print(d);	<div>n</div> <div></div> <div>d</div> <div></div>	n=234; int d=n//10; print(d);	<div>n</div> <div></div> <div>d</div> <div></div>
sum=0, i=1; sum=sum+i; i=i+1; sum=sum+i; i=i+1; sum=sum+i; print(sum);	<div>i</div> <div></div> <div>sum</div> <div></div>	fact=1, i=1; fact=fact*i; i=i+1; fact=fact*i; i=i+1; fact=fact*i; print(fact);	<div>i</div> <div></div> <div>fact</div> <div></div>
n=2345, rev=0; rev=rev*10+n%10; n=n//10; Print(rev, n); rev=rev*10+n%10; n=n//10; Print(rev, n);		rev=rev*10+n%10; n=n//10; Print(rev, n); rev=rev*10+n%10; n=n//10; Print(rev, n);	

Python Input()

Reading input from End-user :

- The input() function is used read data from user.
- The function prompts the message to enter the value
 - **input(prompt)**
- The function waits for the user to enter the value followed by pressing the "Enter" key.
- The function reads the input as string.

Reading the name and display:

```
print("Enter your name :")
name = input()
print("Hello,",name)
```

We can give the prompt while reading input

```
name = input("Enter your name : ")
print("Hello,",name)
```

Every input value will be returned in String format only.

```
print("Enter 2 numbers :")
a = input()
b = input()
c = a+b # "5" + "6" = "56"
print("Sum :",c)
```

We need to convert the string type input values into corresponding type to perform operations.

int() :

- It is pre-defined function
- It can convert input value into integer type.
- On success, it returns integer value
- On failure(if the input is not valid, raised error)

Adding 2 numbers

```
print("Enter 2 numbers :")
a = input()
b = input()
c = int(a)+int(b)
print("Sum :",c)
```

Data Conversion Functions

int() :

- It is pre-defined function
- It can convert input value into integer type.
- On success, it returns integer value
- On failure(if the input is not valid, raised error)

```
>>> int(10)
10
>>> int(23.45)
23
>>> int(True)
1
>>> int(False)
0
>>> int("45")
45
>>> int("python") # Error : Invalid input
```

Adding 2 numbers:

```
print("Enter 2 numbers :")
a = input()
b = input()
c = int(a)+int(b)
print("Sum :",c)
```

We can give the prompt directly while calling input() function.

```
x = int(input("First Num :"))
y = int(input("Second Num :"))
print("Sum : ",x+y)
```

float() :

- converts the input value into float type.
- Raise error if the input is not valid.

```
>>> float(2.3)
2.3
>>> float(5)
5.0
>>> float(True)
1.0
>>> float("3.4")
3.4
```

```
>>> float("abc")
ValueError: could not convert string to float: 'abc'
```

bool():

- Returns a boolean value depends on input value.
- boolean values are pre-defined (True, False)

```
>>> bool(True)
True
>>> bool(-13)
True
>>> bool(0.0013)
True
>>> bool(0)
False
>>> bool("abc")
True
>>> bool(" ")
True
>>> bool("")
False
>>> bool(False)
False
>>> bool("False")
True
```

str(): convert any input into string type.

```
>>> str(3)
'3'
>>> str(2.3)
'2.3'
>>> str(True)
'True'
```

bin(): Returns binary value for specified decimal value.

```
>>> bin(10)
'0b1010'
>>> bin(8)
'0b1000'
```

Character System:

- File is a collection of bytes.
- Every symbol occupies 1 byte memory in File.
- Every symbol stores into memory in binary format.
- Symbol converts into binary based on its ASCII value.
- Character system is the representation of all symbols of a language using constant integer values.
- Examples are ASCII and UNICODE.

ASCII : (Americans Standard Code for Information Interchange)

- Represents all symbols 1 language using constants
- The range is 0 - 255
- A language is at most having 256 symbols.
- 1 byte range is (0-255) - 2^8 value
- Hence we represent a symbol using 1 byte memory.

A-65	a-97	0-48	#-35	
B-66	b-98	1-49	\$-36	
..
..
Z-90	z-122	9-57	..	
<hr/>				
26	+	26	+	10
<hr/>				
150 < 256 symbols				

chr(): Return the symbol for specified integer value.

ord(): Returns the integer for specified symbol.

```
>>> chr(65)
'A'
>>> chr(50)
'2'
>>> ord('a')
97
>>> ord('$')
36
>>> ord('1')
49
```

Programs On Arithmetic Operators

Adding 2 numbers:

```
num1 = float(input("Enter the first number: "))
num2 = float(input("Enter the second number: "))
sum = num1 + num2
print("The sum of", num1, "and", num2, "is", sum)
```

Arithmetic Operations:

```
num1 = float(input("Enter the first number: "))
num2 = float(input("Enter the second number: "))

sum = num1 + num2
difference = num1 - num2
product = num1 * num2
quotient = num1 / num2
remainder = num1 % num2
floordiv = num1 // num2

print("Sum:", sum)
print("Difference:", difference)
print("Product:", product)
print("Quotient:", quotient)
print("Remainder :", remainder)
print("Floor Division :", floordiv)
```

Program to display the last digit of given number:

```
num = int(input("Enter a number: "))
last_digit = num % 10
print("The last digit of", num, "is", last_digit)
```

Program to remove last digit of given number:

```
num = int(input("Enter a number: "))
num = num//10
print("The number with the last digit removed is", num)
```

Find Total and Average of 4 numbers:

```
mark1 = float(input("Enter the first mark: "))
mark2 = float(input("Enter the second mark: "))
mark3 = float(input("Enter the third mark: "))
mark4 = float(input("Enter the fourth mark: "))
average = (mark1 + mark2 + mark3 + mark4) / 4
print("The average of the four marks is", average)
```

Find sum of square and cube of given number:

```
num = int(input("Enter a number: "))
square = num ** 2
cube = num ** 3
sum = square + cube
print("The sum of the square and cube of", num, "is", sum)
```

Calculate Total Salary for given basic Salary:

```
basic_salary = float(input("Enter the basic salary: "))

# Calculate the allowances and deductions
hra = 0.2 * basic_salary
da = 0.1 * basic_salary
pf = 0.05 * basic_salary

# Calculate the gross and net salary
gross_salary = basic_salary + hra + da
net_salary = gross_salary - pf

# Print the result
print("Basic salary:", basic_salary)
print("HRA:", hra)
print("DA:", da)
print("PF:", pf)
print("Gross salary:", gross_salary)
print("Net salary:", net_salary)
```

Swapping 2 numbers:

```
num1 = float(input("Enter the first number: "))
num2 = float(input("Enter the second number: "))

# Before swapping
print("Before swapping:")
print("num1 =", num1)
print("num2 =", num2)

# Swap the values
temp = num1
num1 = num2
num2 = temp

# After swapping
print("After swapping:")
print("num1 =", num1)
print("num2 =", num2)
```

Swapping 2 number without third variable:

```
# Take input from the user
num1 = float(input("Enter the first number: "))
num2 = float(input("Enter the second number: "))

# Swap the values without using a third variable
num1, num2 = num2, num1

# After swapping
print("After swapping:")
print("num1 =", num1)
print("num2 =", num2)
```

Another Way:

```
// Swap the values without using a third variable
num1 = num1 + num2;
num2 = num1 - num2;
num1 = num1 - num2;
```

Relational operators:

- Operators are > , < , >= , <= , == , !=
- These operators validate the relation among operands and return a boolean value.
- If relation is valid returns True else False

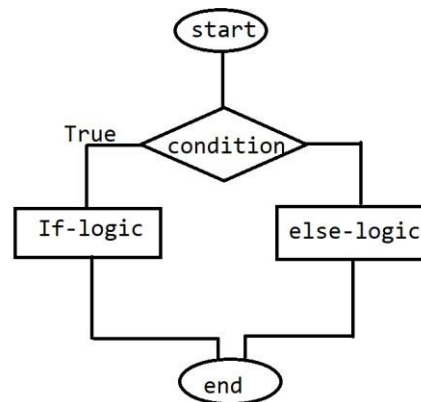
Program to understand Relational operators:

```
print("Relational operations")
print("5>3 :", 5>3)
print("5==3 :", 5==3)
print("5<3 :", 5<3)
print("5!=3 :", 5!=3)
print("5>=3 :", 5>=3)
print("5<=3 :", 5<=3)
```

If-else Conditional Statement:

Syntax :

```
if(condition) :
    .....
    if-logic
    .....
else:
    .....
    else-logic
    .....
```



Check the Number is Zero or Not

```
n = int(input("Enter number : "))
if(n==0):
    print("Number equals to zero")
else:
    print("Number is not zero")
```

Check the Number is Positive or Not

```
n = int(input("Enter number : "))
if(n>=0):
    print("Positive Number")
else:
    print("Negative Number")
```

Check the 2 numbers equal or not

```
n1 = int(input("Enter First num : "))
n2 = int(input("Enter Second num : "))
if(n1==n2):
    print("Equal numbers")
else:
    print("Not equal Numbers")
```

Check the first number greater than second number or not

```
n1 = int(input("Enter First num : "))
n2 = int(input("Enter Second num : "))
if(n1>n2):
    print("First Number is big")
else:
    print("Second Number is big")
```

Check the person eligible for vote or not

```
age = int(input("Enter age : "))
if(age>=18):
    print("Eligible for vote")
else:
    print("Not eligible for vote")
```

Check the number is divisible by 7 or not

```
num = int(input("Enter number : "))
if(num%7==0):
    print("Divisible by 7")
else:
    print("Not divisible by 7")
```

Check the number is even or not

```
num = int(input("Enter number : "))
if(num%2==0):
    print("Even Number")
else:
    print("Not Even")
```

Check the last digit of number is zero or not

```
num = int(input("Enter number : "))
if(num%10==0):
    print("Last digit is zero")
else:
    print("Last digit is not zero")
```

Check the sum of 2 numbers equal to 10 or not

```
n1 = int(input("Enter First number : "))
n2 = int(input("Enter Second number : "))
if(n1+n2==10):
    print("Equal to 10")
else:
    print("Not equal to 10")
```

Check last digits of given 2 numbers equal or not

```
n1 = int(input("Enter First number : "))
n2 = int(input("Enter Second number : "))
if(n1%10 == n2%10):
    print("Equal")
else:
    print("Not equal")
```

Check the average of 3 numbers greater than 60 or not

```
print("Enter 3 numbers :")
n1 = int(input())
n2 = int(input())
n3 = int(input())

if((n1+n2+n3)/3 > 60):
    print("avg Greater than 60")
else:
    print("Not")
```

Check the last digit of number is divisible by 3 or not

```
n = int(input("Enter num : "))
if((n%10)%3==0):
    print("Last digit divisible by 3")
else:
    print("Not divisible")
```

Logical operators: These operators returns True or False be validating more than one expression

Operator	Meaning	Example
and	True if both the operands are true	x and y
or	True if either of the operands is true	x or y
not	True if operand is false (complements the operand)	not x

And examples:	Or examples:	Not examples:
>>> True and True True >>> 5>3 and 3>2 True >>> True and False False >>> False and True False >>> False and False False >>> 5>3 and 5!=5 False	>>> False or False False >>> False or True True >>> True or False True >>> True or True True >>> 3>5 or 5>2 True	>>> not True False >>> not False True >>> not 5>3 False >>> not 3!=3 True

Check the Number Divisible by 3

$N\%3==0$ (3, 6, 9, 12....)

Check the Number Divisible by 5

$N\%5==0$ (5, 10, 15, 20....)

Check the Number Divisible by both 3 and 5

$N\%3==0$ and $N\%5==0$ (15, 30, 45, 60....)

Check the Number Divisible by either 3 or 5

$N\%3==0$ and $N\%5==0$ (3, 5, 6, 9, 10, 12, 15...)

Program to check the Number Divisible by both 3 and 5:

```
n = int(input("enter number : "))
if n%3==0 and n%5==0:
    print("Divisible by 3 and 5")
else:
    print("Not divisible")
```

Check the person age between 20 and 50:

```
age = int(input("enter age : "))
if age >= 20 and age <= 50:
    print("Age between 20 and 50")
else:
    print("Not in between")
```

Check the Number is Single Digit or Not:

```
n = int(input("enter num : "))
if n >= 0 and n <= 9:
    print("Single Digit")
else:
    print("Not Single Digit")
```

Check the Number is Two Digit or Not:

```
n = int(input("enter num : "))
if n >= 10 and n <= 99:
    print("Two Digit")
else:
    print("Not Two Digit")
```

Check the Character is Upper case Alphabet or Not:

```
ch = input("enter character : ")
if ch >= 'A' and ch <= 'Z':
    print("Upper case Alphabet")
else:
    print("Not")
```

Check the Character is Lower case Alphabet or Not:

```
ch = input("enter character : ")
if ch >= 'a' and ch <= 'z':
    print("Lower case Alphabet")
else:
    print("Not")
```

Check the Character is Digit or Not:

```
ch = input("enter character : ")
if ch >= '0' and ch <= '9':
    print("Digit")
else:
    print("Not")
```

Character is Vowel or Not:

```
ch = input("enter character : ")
if ch=='a' or ch=='e' or ch=='i' or ch=='o' or ch=='u':
    print("Vowel")
else:
    print("Not")
```

Check the Character is Alphabet or Not:

```
ch = input("enter character : ")
if((ch>='A' and ch<='Z') or (ch>='a' and ch<='z')):
    print("Alphabet")
else:
    print("Not")
```

Check the Student passed in all 3 subjects or not with minimum 35 marks:

```
subj1 = int(input("Enter subj1 score: "))
subj2 = int(input("Enter subj2 score: "))
subj3 = int(input("Enter subj3 score: "))
if subj1 >= 35 and subj2 >= 35 and subj3 >= 35:
    print("Pass")
else:
    print("Fail")
```

Check A greater than both B and C:

```
print("Enter 3 numbers : ")
x = int(input())
y = int(input())
z = int(input())
if(x>y and x>z):
    print("Yes")
else:
    print("No")
```

Check given 3 numbers equal or not:

```
print("Enter 3 numbers : ")
x = int(input())
y = int(input())
z = int(input())
if(x==y and y==z and z==x):
    print("Equal numbers")
else:
    print("Not equal numbers")
```

Check given 3 numbers unique (not equal):

```
print("Enter 3 numbers : ")
x = int(input())
y = int(input())
z = int(input())
if(x!=y and y!=z and z!=x):
    print("Unique numbers")
else:
    print("Not unique numbers")
```

Check any 2 numbers are equal among the given 3 numbers:

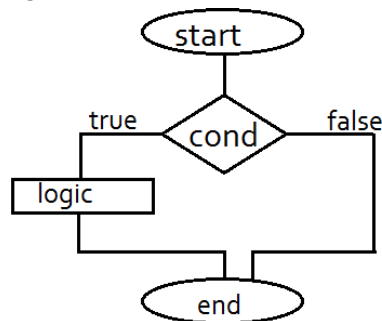
```
print("Enter 3 numbers : ")
x = int(input())
y = int(input())
z = int(input())
if(x==y or y==z or z==x):
    print("Any 2 equal")
else:
    print("Not equal numbers")
```

If-block: Execute a block of instructions only if the given condition is true

Syntax :

if (condition) :

.....
logic
.....

**Program to give 20% discount to customer if the bill amount is > 5000**

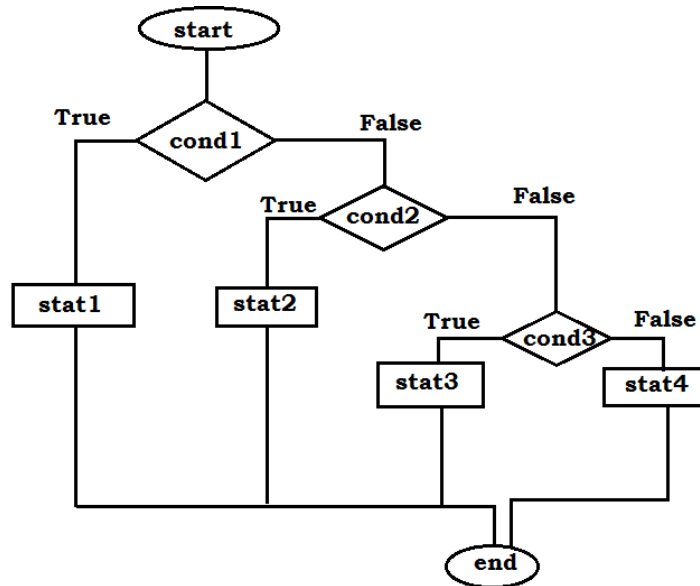
```
print("Enter bill amount :")
bill = float(input())
if(bill>5000):
    discount = 0.2*bill
    bill = bill-discount

print("Plz pay : ", bill)
```

if-elif-else: if-elif-else is a control flow structure in programming that allows a program to execute different blocks of code based on one or more conditions.

Syntax :

```
if(cond1) :  
    stat1  
  
elif(cond2) :  
    stat2  
  
elif(cond3) :  
    stat3  
  
else :  
    stat4
```



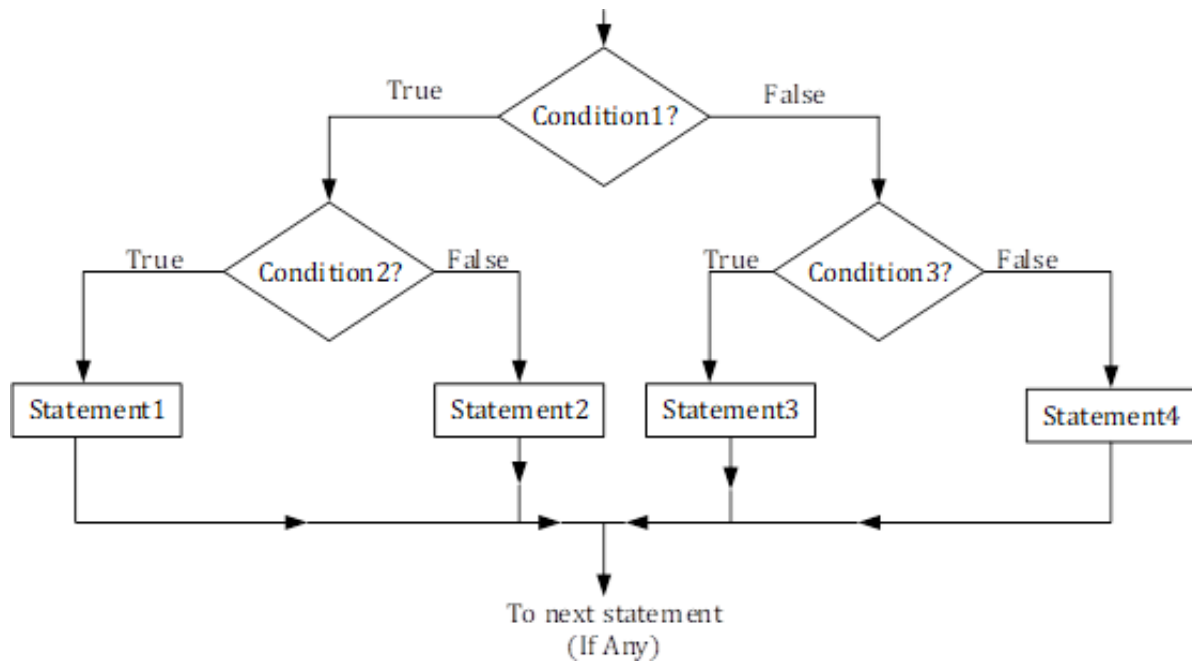
Check the Given number is Single Digit or Two Digit or Three Digit or Other

```
n = int(input("Enter number :"))  
if(n>=0 and n<=9):  
    print("Single digit")  
elif(n>=10 and n<=99):  
    print("Two digit")  
elif(n>=100 and n<=999):  
    print("Three digit")  
else:  
    print("Other digits number")
```

Check the given character is Upper case or Lower case or Digit or Symbol:

```
ch = input("Enter character :")  
if(ch>='A' and ch<='Z'):  
    print("Upper case alphabet")  
elif(ch>='a' and ch<='z'):  
    print("Lower case alphabet")  
elif(ch>='0' and ch<='9'):  
    print("Digit")  
else:  
    print("Symbol")
```

Nested-If: Writing if block inside another if block



Check the number is even or not only if the Number is positive

```
n = int(input("Enter number :"))
if n >= 0:
    if n % 2 == 0:
        print("Even number")
    else:
        print("Not even number")
else:
    print("Negative")
```

Check the biggest of 2 numbers only if the 2 numbers are not equal:

```
print("Enter 2 integers :")
a = int(input())
b = int(input())
if(a != b):
    if(a > b):
        print("a is big")
    else:
        print("b is big")
else:
    print("equal numbers given")
```

Display Student Grade only if the Student passed in all subjects:

```
print("Enter 3 subject marks :")
m1 = int(input())
m2 = int(input())
m3 = int(input())

if(m1>=40 and m2>=40 and m3>=40):
    avg = (m1+m2+m3)/3
    if(avg>=75):
        print("Distinction")
    elif(avg>=60):
        print("A-Grade")
    elif(avg>=50):
        print("B-Grade")
    else:
        print("C-Graade")
else:
    print("Fail")
```

Bitwise operators:

- Bitwise operators act on operands as if they were string of binary digits. It operates bit by bit, hence the name.
- For example, 2 is 10 in binary and 7 is 111.
- In the table below: Let x = 10 (0000 1010 in binary) and y = 4 (0000 0100 in binary)

Operator	Meaning	Example
&	Bitwise AND	x&y = 0 (0000 0000)
	Bitwise OR	x y = 14 (0000 1110)
~	Bitwise NOT	~x = -11 (1111 0101)
^	Bitwise XOR	x ^ y = 14 (0000 1110)
>>	Bitwise right shift	x>> 2 = 2 (0000 0010)
<<	Bitwise left shift	x<< 2 = 40 (0010 1000)

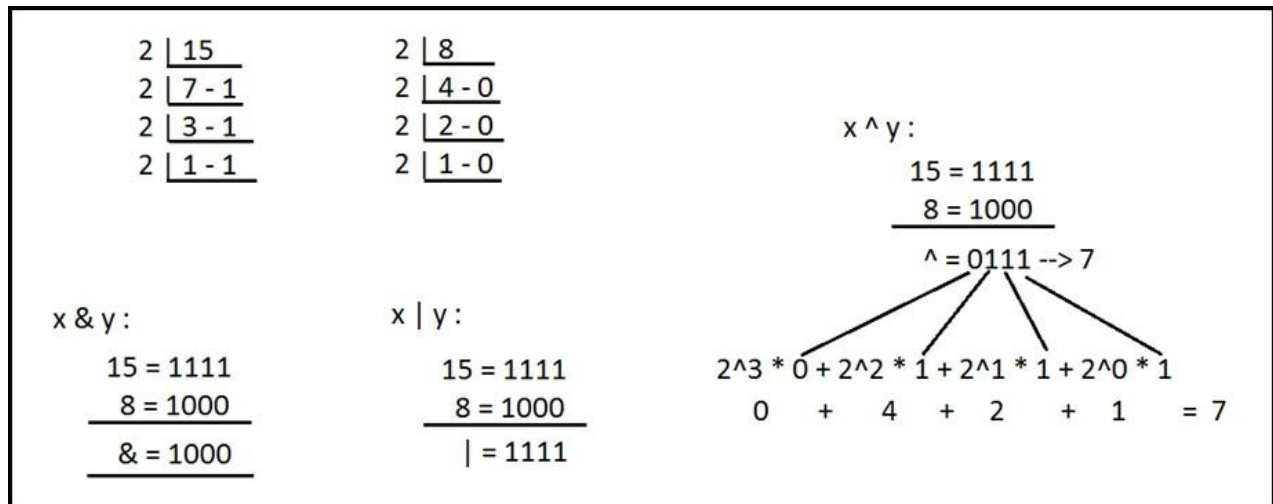
Bitwise truth table:

x	y	x&y	x y	x^y
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

```

>>> x=15
>>> y=8
>>> x&y
8
>>> x|y
15
>>> x^y

```



Shift operators:

- These are used to move the bits in the memory either to right side or to left side.
- Moving binary bits in the memory change the value of variable.
- These operators return the result in decimal format only.
- Operators are Right shift (>>) and Left shift (<<)

```

>>> x=8
>>> x>>2
2
>>> x<<2
32

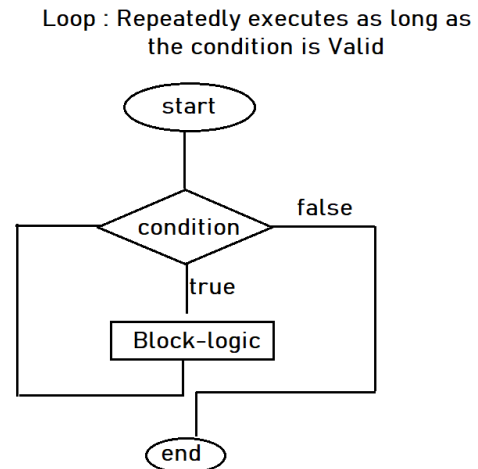
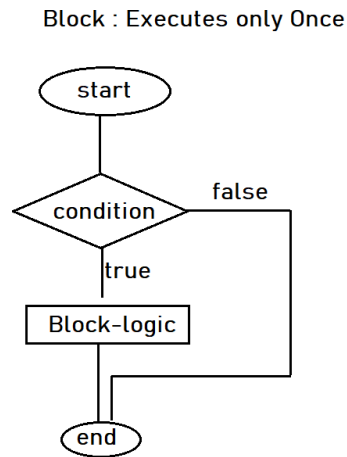
```

Right shift: $n/2^s \rightarrow 8/2^2 \rightarrow 8/4 \rightarrow 2$

Left shift : $n*2^s \rightarrow 8*2^2 \rightarrow 8*4 \rightarrow 32$

Introduction to Loops

Loop: A Block of instructions execute repeatedly as long the condition is valid.



Note: Block executes only once whereas Loop executes until condition become False

Python Supports 3 types of Loops:

1. For Loop
2. While Loop
3. While - else Loop

For Loop: We use for loop only when we know the number of repetitions. For example,

- Print 1 to 10 numbers
- Print String elements
- Print Multiplication table
- Print String character by character in reverse order

While loop: We use while loop when we don't know the number of repetitions.

- Display contents of File
- Display records of Database table

While – else : while-else loop is a type of loop that combines a while loop with an else statement that is executed after the loop has completed. The else block is executed only if the while loop completed normally

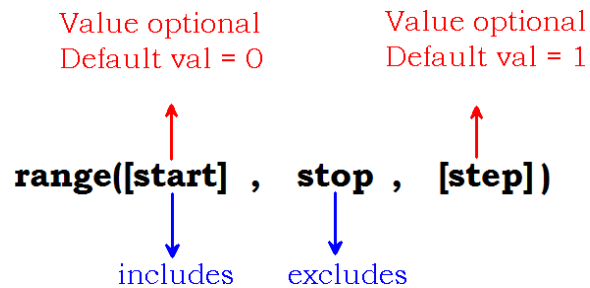
For Loop

for loop:

- "for" is a keyword.
- for loop executes a block repeatedly as long as the condition is valid.
- for loop uses range() function for loop repetition.

range():

- The range() function in Python generates a sequence of numbers within a given range.
- It takes up to three arguments: start (optional), stop (required), and step (optional).
- The sequence generated by the range() function is commonly used in for loops.



Print the numbers from 0 to 9:

```
for i in range(10):  
    print(i)
```

Print the numbers from 3 to 8:

```
for i in range(3,9):  
    print(i)
```

Print the numbers from 10 to 1:

```
for i in range(10, 0, -1):  
    print(i)
```

Print every third number from 0 to 20:

```
for i in range(0, 21, 3):  
    print(i)
```

Print the even numbers from 0 to 10:

```
for i in range(0, 11, 2):  
    print(i)
```

Print the odd numbers from 1 to 9:

```
for i in range(1, 10, 2):  
    print(i)
```

Print the even numbers from 10 to 0:

```
for i in range(10, -1, -2):  
    print(i)
```

Display values side by side:

```
for i in range(1,6):  
    # print(i)  
    # print(i, end=' ') -> print values with spaces  
    # print(i, end='\t') -> print values with tab spaces  
    print(i, end='\n') # print values in new lines
```

Sum of First N numbers:

```
n = int(input("Enter a positive integer: "))  
sum = 0  
for i in range(1, n+1):  
    sum += i  
print("The sum of the first", n, "natural numbers is:", sum)
```

Find factorial for given number:

```
n = int(input("Enter a positive integer: "))  
factorial = 1  
for i in range(1, n+1):  
    factorial *= i  
print("The factorial of", n, "is:", factorial)
```

Multiplication table program:

```
n = int(input("Enter a positive integer: "))  
for i in range(1, 11):  
    product = i * n  
    print(n, "x", i, "=", product)
```

Print even numbers from 1 to 10:

```
n = int(input("Enter a positive integer: "))  
print("Even numbers from 1 to", n, "are:")  
for i in range(1, n+1):  
    if i % 2 == 0:  
        print(i)
```

Print factors for given number:

```
n = int(input("Enter a positive integer: "))
print("Factors of", n, "are:")
for i in range(1, n+1):
    if n % i == 0:
        print(i)
```

Prime number program:

```
n = int(input("Enter a positive integer: "))
is_prime = True
for i in range(2, n):
    if n % i == 0:
        is_prime = False
        break
if is_prime:
    print(n, "is a prime number")
else:
    print(n, "is not a prime number")
```

Perfect number program:

```
n = int(input("Enter a positive integer: "))
sum = 0
for i in range(1, n//2 + 1):
    if n % i == 0:
        sum += i
if sum == n:
    print(n, "is a perfect number")
else:
    print(n, "is not a perfect number")
```

Fibonacci Series program:

```
n = int(input("Enter the number of terms: "))
a, b = 0, 1
for i in range(1, n+1):
    print(a)
    c = a + b
    a, b = b, c
```

Multiplication tables in the given range:

```
for n in range(5, 11):  
  
    for i in range(1,11):  
        print(n,'*',i,'=',n*i)
```

Factorials in the Given range:

```
for n in range(1,8):  
  
    fact=1  
    for i in range(1,n+1):  
        fact=fact*i  
    print("factorial of",n,"is",fact)
```

Prime numbers in given range:

```
for n in range(1,51):  
  
    factors=0  
    for i in range(1,n+1):  
        if(n%i==0):  
            factors=factors+1  
  
    if(factors==2):  
        print(n,"is prime")
```

Perfect numbers in given range:

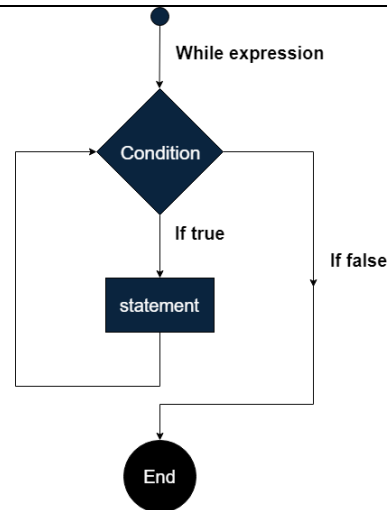
```
for n in range(1,10000):  
  
    sum=0  
    for i in range(1,n):  
        if(n%i==0):  
            sum=sum+i  
  
    if(n==sum):  
        print(n,"is perfect")
```

while loop:

- We use while loop when we don't know the number of iterations.
- **Examples:** Reading a File, Display records in database.

Syntax:

```
while(condition):  
    statements;
```



Check the even numbers until user quits:

```
while(True):  
    n = int(input("enter num : "))  
    if(n%2==0):  
        print(n,"is even")  
    else:  
        print(n,"is odd")  
  
    print("Do you check another num(y/n) :")  
    ch = input()  
    if(ch=='n'):  
        break
```

Display Multiplication tables until user quits:

```
while(True):  
    n = int(input("Enter table num : "))  
    for i in range(1,11):  
        print(n,'*',i,'=',n*i)  
  
    print("Do you print another table(y/n) :")  
    ch = input()  
    if(ch=='n'):  
        break
```

Menu Driven Program to perform all arithmetic operations:

```
while(True):
    print("1. Add")
    print("2. Subtract")
    print("3. Multiply")
    print("4. Divide")
    print("5. Quit")

    ch = int(input("enter choice :"))
    if(ch==1):
        print("Enter 2 numbers:")
        a=int(input())
        b=int(input())
        print("Res : ", a+b)
    elif(ch==2):
        print("Enter 2 numbers:")
        a=int(input())
        b=int(input())
        print("Res : ", a-b)
    elif(ch==3):
        print("Enter 2 numbers:")
        a=int(input())
        b=int(input())
        print("Res : ", a*b)
    elif(ch==4):
        print("Enter 2 numbers:")
        a=int(input())
        b=int(input())
        print("Res : ", a/b)
    elif(ch==5):
        break
    else:
        print("Invalid choice")
```

Count digits in the given number:

```
n = int(input("enter num : "))
count=0
while(n!=0):
    n=n//10
    count=count+1
print("Digits count : ", count)
```

Sum of digits in the given number:

```
n = int(input("Enter a number: "))
sum = 0
while n > 0:
    digit = n % 10
    sum += digit
    n //= 10
print("The sum of digits is:", sum)
```

Display only Even digits:

```
n = int(input("Enter a number: "))
while n > 0:
    digit = n % 10
    if digit % 2 == 0:
        print(digit)
    n //= 10
```

Sum of Even digits:

```
n = int(input("Enter a number: "))
sum = 0
while n > 0:
    digit = n % 10
    if digit % 2 == 0:
        sum += digit
    n //= 10
print("The sum of even digits is:", sum)
```

Largest digit in the given number:

```
n = int(input("Enter a number: "))
max_digit = 0
while n > 0:
    digit = n % 10
    if digit > max_digit:
        max_digit = digit
    n //= 10

print("The largest digit is:", max_digit)
```

Display First digit in the given number:

```
n = int(input("Enter a number: "))
while n >= 10:
    n //= 10

print("The first digit is:", n)
```

Sum of First and Last Digits in the given number:

```
n = int(input("Enter a number: "))
last_digit = n % 10
first_digit = n
while first_digit >= 10:
    first_digit //= 10

sum = first_digit + last_digit
print("The sum of the first and last digits is:", sum)
```

Reverse number program:

```
n = int(input("Enter a number: "))
reverse = 0

while n != 0:
    digit = n % 10
    reverse = (reverse * 10) + digit
    n //= 10

print("The reversed number is:", reverse)
```

Strong number program:

```
n = int(input("Enter a number: "))
temp = n
sum = 0

while temp != 0:
    digit = temp % 10
    fact = 1
    for i in range(1, digit + 1):
        fact *= i
    sum += fact
    temp //= 10

if sum == n:
    print(n, "is a strong number")
else:
    print(n, "is not a strong number")
```

ArmStrong Number program:

```
num = int(input("Enter a number: "))
sum = 0
temp = num

# find the number of digits
n = len(str(num))

# calculate sum of cubes of each digit
while temp > 0:
    digit = temp % 10
    sum += digit ** n
    temp //= 10

# display the result
if num == sum:
    print(num, "is an Armstrong number")
else:
    print(num, "is not an Armstrong number")
```

Break and Continue

break:

- It is a keyword
- It is called branching statement
- It is used to terminate the flow of a loop(for or while)

```
while True:  
    print("Hi")  
    print("Hello")  
    break
```

Break loop on condition:

```
for i in range(10):  
    if i==5:  
        break  
    print("i val :",i)
```

continue:

- It is a keyword.
- It is used to skip the current iteration in loop execution.

```
for i in range(1,11):  
    if i==5:  
        continue  
    print("i val :",i)
```

```
for i in range(1,11):  
    if i==3 or i==6:  
        continue  
    print("i val :",i)
```

Patterns Programming

Code:	Pattern:
<pre>for i in range(1,6): for j in range(1,6): print("*", end=" ") print()</pre>	<pre>* *</pre>

Code:	Pattern:
<pre>for i in range(1,6): for j in range(1,6): print(i, end=" ") print()</pre>	<pre>1 1 1 1 1 2 2 2 2 2 3 3 3 3 3 4 4 4 4 4 5 5 5 5 5</pre>

Code:	Pattern:
<pre>for i in range(1,6): for j in range(1,6): print(j, end=" ") print()</pre>	<pre>1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5</pre>

Code:	Pattern:
<pre>for i in range(1,6,1): for j in range(1,i+1,1): print(j, end=' ') print()</pre>	<pre>1 1 2 1 2 3 1 2 3 4 1 2 3 4 5</pre>

Code:	Pattern:
<pre>for i in range(5,0,-1): for j in range(1,i+1,1): print(j, end=' ') print()</pre>	<pre>1 2 3 4 5 1 2 3 4 1 2 3 1 2 1</pre>

Code:	Pattern:
<pre>for i in range(5,0,-1): for j in range(5,i-1,-1): print(j, end=' ') print()</pre>	<pre>5 5 4 5 4 3 5 4 3 2 5 4 3 2 1</pre>

Code:	Pattern:
<pre> for i in range(1,10): for j in range(1,10): if(i==j or j==10-i): print("+", end=" ") else: print(" ", end=" ") print() </pre>	<pre> + + + + + + + + + + + + + + + + + + </pre>

Code:	Pattern:
<pre> for i in range(1,10): for j in range(1,10): if(i==1 or i==9 or i==j or j==10-i): print("+", end=" ") else: print(" ", end=" ") print() </pre>	<pre> +++++++ + + + + + + + + + + + + + + + + +++++++ </pre>

Code:	Pattern:
<pre> for i in range(1,10): for j in range(1,10): if((i==1 and j<=5) or j==5 or (i==9 and j>=5) or (j==9 and i<=5) or i==5 or (j==1 and i>=5)): print("+", end=" ") else: print(" ", end=" ") print() </pre>	<pre> +++++ + + + + + + + +++++ + + + + + + + + + + + </pre>

Functions in Python

Functions:

- A block of instructions that performs a task.
- Function takes input, process the input and returns the output.
- "def" is a keyword is used to define functions in python programming.

Syntax:

```
def identity(arguments) :
```

```
.....  
Logic  
.....
```

Ex:

```
def add(a, b):  
    c=a+b  
    return c
```

Every function consists:

1. Function definition: Function definition is a block which contains the logic.
2. Function call : It is a single statement used to access the function definition.

Definition:

```
def add(a, b):  
    c=a+b  
    return c
```

Call:

```
res = add(10,20)
```

Basic functional programming:

- Function execute only when we call that function.
- Calling is a single statement used to access the function logic

```
def fun():  
    print("Hello...")  
    return
```

fun() # calling

Note : We cannot call a function before it has defined.

```
fun() # error :
def fun():
    print("Hello...")
    return
```

The main advantage of functions is code re-usability. We can call the function many times once we defined.

```
def test():
    print("logic..")
    return

test()
test()
test()
```

One source file(.py file) can have more than one function definition. Functions get executed in the order we invoke.

```
def m1():
    print("m1 ....")
    return

def m2():
    print("m2 ....")
    return

m2()
m1()
```

We can access one function from another function.

```
def m1():
    print("control in m1...")
    return

def m2():
    print("control in m2...")
    m1() #calling
    print("control back to m2 from m1...")
    return

print("Program starts...")
m2() #calling
print("Program ends...")
```

Classification of functions: The way of passing input and returning output, functions classified into

1. No arguments and No return values
2. With arguments and No return values
3. With arguments and With return values
4. No arguments and With return value
5. Recursion

No arguments and No return values:

- Defining a function without arguments. No need to pass input values while calling the function. 'return' statement is optional.

```
def sayHi():  
    print("Hi to all...")  
    return
```

```
sayHi()  
sayHi()  
sayHi()
```

With arguments and No return values:

- Defining a function with arguments(variables)
- Function takes input and it can be any type.
- We need to pass parameter values while calling the function.

```
def printMessage(msg):  
    print("Message is :",msg)  
    return
```

```
printMessage("Live Tutition")  
printMessage("Python")  
printMessage("Tutorials")
```

With arguments and with return values:

- Defining a function with arguments and return values
- The value returned by function need to collect into variable.
- Function returned value back to the calling function.

```
def add(a,b):  
    c=a+b  
    return c  
print("sum :",add(3,5))  
print("sum :",add(5,8))
```

No arguments and with return values:

- Defining a function with no arguments
- Function returns a value from its definition.

```
def getPI():  
    PI = 3.142  
    return PI  
  
print("PI value :",getPI())
```

Recursion:

- Function calling itself is called recursion.
- Calling the function from the definition of same function.
- Function executes from the allocated memory called STACK.
- While executing the program, if the stack memory is full, the program execution terminates abnormally.

```
def tutorials():  
    print("Keep reading...")  
    tutorials()  
    return  
  
tutorials()
```

Factorial of given number using recursion:

```
def fact(n):  
    res=0  
    if(n==0):  
        res=1  
    else:  
        res=n*fact(n-1)  
    return res  
  
n = int(input("Enter n val :"))  
print("Factorial val :", fact(n))
```

Argument type functions: Depends on the way of passing input and taking input, functions in python classified into

1. Default arguments function
2. Required arguments function
3. Keyword arguments function
4. Variable arguments function

Default arguments function:

- Defining a function by assigning values to arguments.
- Passing values to these arguments is optional while calling the function.
- We can replace the values of default arguments if required.

```
def default(a,b=20):  
    print("a val :",a)  
    print("b val :",b)  
    return  
  
default(10)  
default(50,100)  
default(10,"abc")
```

- Argument assigned with value is called default argument
- A Non default argument cannot follow default argument

```
def default(a=10,b):  
    print("a val :",a)  
    print("b val :",b)  
    return
```

Required arguments function:

- Function definition without default arguments.
- We need to pass input value to all positional arguments of function.

```
def required(a,b):  
    print("a val :",a)  
    print("b val :",b)  
    return  
  
#required(10) -> Error:  
required(10,20)  
required(10,"abc")
```

Keyword arguments function:

- Calling the function by passing values using keys.
- We use arguments names as keys.

```
def keyword(name, age):  
    print("Name is :",name)  
    print("Age is :",age)  
    return
```

```
keyword("Annie",21)  
keyword(23,"Amar")
```

We can change the order of arguments while passing values using keys.

```
def keyword(name, age):  
    print("Name is :",name)  
    print("Age is :",age)  
    return
```

```
keyword(age=23,name="Amar")
```

Keyword functions are useful mostly in default arguments function.

```
def default(a=10,b=20,c=30):  
    print("a :",a)  
    print("b :",b)  
    print("c :",c)  
    return
```

```
default()  
# I want to change value of b=50  
default(10,50,30)  
# It is easy to use keyword arguments  
default(b=50)
```

Variable arguments function:

- Passing different length of values while calling the function.
- We can collect these arguments into pointer type argument variable.

```
def variable(*arr):  
    print("Length is :",len(arr))  
    print("Elements :",arr)  
    return
```

```
variable()
```

We can pass different types of data elements also

```
def variable(*arr):  
    print("Elements :",arr)  
    return  
  
variable(10,20,30,40,50)  
variable(10,2.3,"abc")
```

```
def variable(*arr):  
    print("Elements :")  
    for ele in arr:  
        print(ele)  
    return  
  
variable(10,20,30,40,50)
```

We can process the elements easily using for loop

Local and Global Variables:

Local variables:

- Defining a variable inside the function.
- Local variables can be accessed only from the same function in which it has defined.
- We access the local variable directly.

```
def test():  
    a=10 #local  
    print("Inside :",a)  
    return  
  
test()  
print("Outside :",a) #error :
```

Arguments(parameters):

- Variables used to store input of the function.
- Arguments are working like local variables.
- Arguments can access within that function only.

```
def test(a, b): #a,b are local  
    print("Inside :",a)  
    print("Inside :",b)  
    return  
  
test(10,20)  
print("Outside :",a) #error
```

Global variables:

- Defining a variable outside to all the functions.
- We can access the variable directly.
- It is available throughout the application.

```
a=10 #global
def m1():
    print("Inside m1 :",a)
    return

def m2():
    print("Inside m2 :",a)
    return

m1()
m2()
print("Outside :",a)
```

- We can define local & global variables with the same name.
- We access both the variables directly using its identity.
- When we access a variable inside the function, it gives the first priority to local variable.
- If local is not present, then it is looking for Global variable.

```
a=10 #global
def m1():
    a=20 #local
    print("Inside m1 :",a)
    return

def m2():
    print("Inside m2 :",a)
    return

m1()
m2()
print("Outside :",a)
```

global:

- It is a keyword.
- It is used to define, access, modify & delete global variables from the function.
- global statement must be placed inside the function before the use of that variable.

Note: We can access global variable inside the function. We cannot modify the global variable from function directly.

```
a=10
```

```
def test():  
    print("a val :",a)  
    a=a+20 #error :  
    print("a val :",a)  
    return
```

```
test()
```

We can use global keyword to modify the global variable from the function:

```
a=10
```

```
def test():  
    global a  
    print("a val :",a)  
  
    a=a+20  
    print("a val :",a)  
  
    a=a+30  
    return
```

```
test()  
print("a val :",a)
```

We can define global variables from the function using "global" :

```
def test():  
    global a  
    a=10  
    print("Inside :",a)  
    return
```

```
test()  
print("Outside :",a)
```

Introduction to OOPS

Application:

- Programming Languages and Technologies are used to develop applications.
- Application is a collection of Programs.
- We need to design and understand a single program before developing an application.

Program Elements: Program is a set of instructions. Every Program consists,

1. Identity
2. Variables
3. Methods

1. Identity:

- Identity of a program is unique.
- Programs, Classes, Variables and Methods having identities
- Identities are used to access these members.

2. Variable:

- Variable is an identity given to memory location.
or
- Named Memory Location
- Variables are used to store information of program(class/object)

Syntax	Examples
identity = value;	name = "amar"; age = 23; salary = 35000.00; is_married = False;

3. Method:

- Method is a block of instructions with an identity
- Method performs operations on data(variables)
- Method takes input data, perform operations on data and returns results.

Syntax	Example
identity(arguments): body	add(int a, int b): c = a+b return c

Object Oriented Programming:

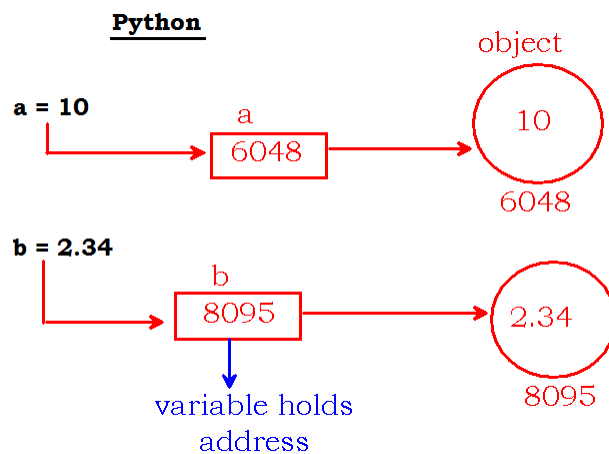
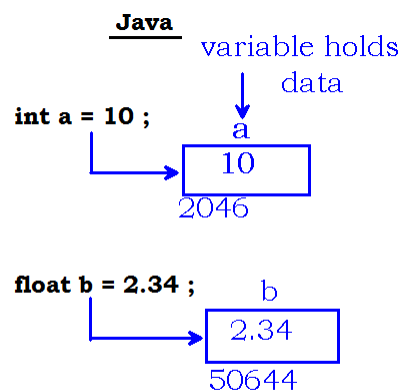
- Python is Object Oriented.
- Python is Fully Object Oriented because it doesn't support primitive data types.
- Languages which are supporting primitive types are called, Partial Object oriented programming languages.
- for example C++, Java, .Net...

Note: Primitive types such as int, float, char occupies fixed memory size and stores specific type of data.

Python is Dynamic:

- Python is dynamic
- Python variable stores address instead of data directly.
- Python stores information in the form of Objects.
- Depends on the data, the size of memory grows and shrinks dynamically.
- A python variable accepts any type of data assignment.

Note: In python, if we modify the value of variable, it changes the location.



id(): A pre-defined function that returns memory address of specified variable.

```
a=10
print("Val :",a)
print("Addr :",id(a))

a=a+5
print("Val :",a)
print("Addr :",id(a))
```

Object Oriented Programming features are:

1. Encapsulation
2. Inheritance
3. Polymorphism
4. Abstraction

Note: We implement Object-Oriented functionality using Classes and Objects

Class: Class contains variables and methods. Java application is a collection of classes

Syntax	Example
<pre>class ClassName: Variables ; & Methods ;</pre>	<pre>class Account: num; balance; withdraw(): logic; deposit(): logic;</pre>

Object: Object is an instance of class. Instance (non static) variables of class get memory inside the Object.

Syntax: `refVariableName = ClassName();`

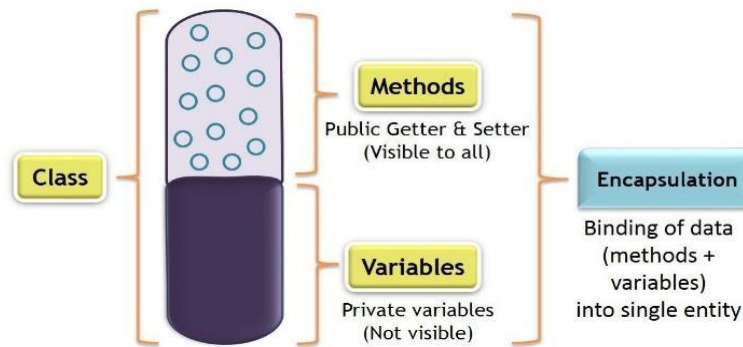
Example: `acc = Account();`

Note: Class is a Model from which we can define multiple objects of same type



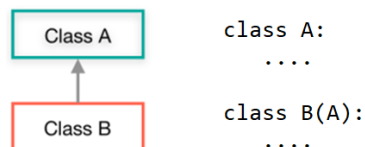
Encapsulation:

- The concept of protecting the data within the class itself.
- **Implementation rules:**
 - Class is Public (to make visible to other classes).
 - Variables are Private (other objects cannot access the data directly).
 - Methods are public (to send and receive the data).



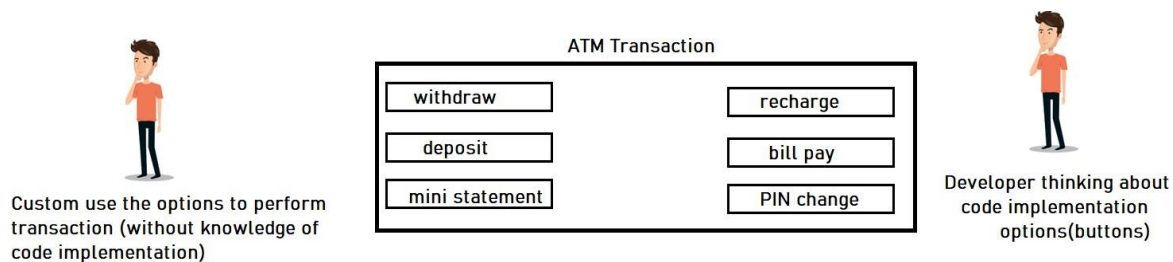
Inheritance:

- Defining a new class by re-using the members of other class.
- We can implement inheritance using "extends" keyword.
- Terminology:**
 - Parent/Super class:** The class from which members are re-used.
 - Child/Sub class:** The class which is using the members



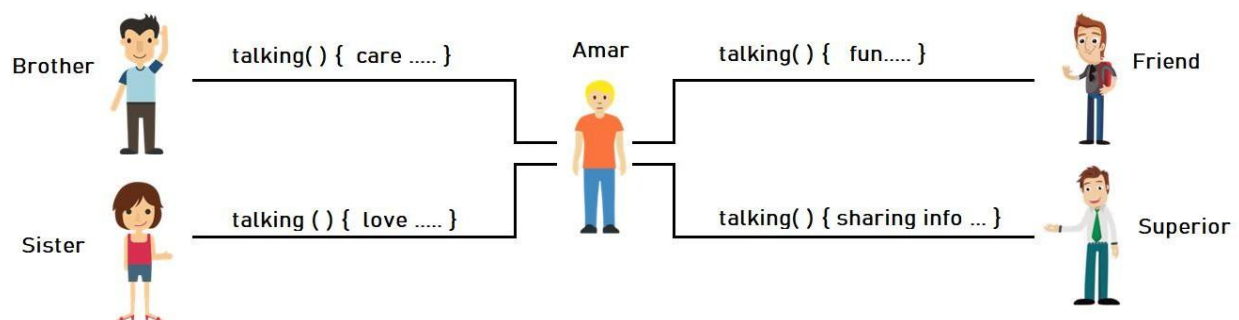
Abstraction:

- Abstraction is a concept of hiding implementations and shows functionality.
- Abstraction describes "What an object can do instead how it does it?".



Polymorphism:

- Polymorphism is the concept where object behaves differently in different situations.



Class Members or Static members:

- Defining variable or method inside the class.
- We can access these members using class name.

Note: Generally, we access the variables and functions directly after definition.

```
a=10
def test():
    print("test")
    return
print("a val :",a)
test()
```

- In Object Oriented application, variables and methods must be defined inside the class.
- "class" is a keyword.
- Defining variable and method inside the class called "static members"
- We need to access static members using identity of class.

```
class Test:
    a=10 # static
    def fun():
        print("static fun")
        return
print("a val :",Test.a)
Test.fun()
```

Connect classes:

- One python file allowed to define any number of classes
- We can access the members of these classes using "class names"

```
class First:
    def fun():
        print("First class fun")
        return
class Second:
    def fun():
        print("Second class fun")
        return
class Access:
    def main():
        print("starts @ main")
        First.fun()
        Second.fun()
        return

Access.main()
```

Local and Static variables:

- Defining a variable inside method is called local variable
- We access local variables directly but only from the same block in which it has defined.
- Defining a variable inside the class and outside to all methods is called static variable.
- We can access static variable using class name.

```
class Access:
    a=10 #static
    def main():
        a=20 #local
        print("local a :", a)
        print("static a :", Access.a)
        return

Access.main()
```

Global variables:

- Defining variables outside to all classes.
- We access global variables directly.
- When we access variable inside the method, it is looking for local variable first. If the local variable is not present, it accesses the global variable.

```
a=10 #Global
class Access:
    a=20 #Static
    def m1():
        a=30 #Local
        print("Inside m1")
        print(a)
        print(Access.a)
        return

    def m2():
        print("Inside m2")
        print(a)
        print(Access.a)
        return

    def main():
        Access.m1()
        Access.m2()
        return
Access.main()
```

Dynamic members

Dynamic Members:

- The specific functionality of Object must be defined as dynamic.
- We access dynamic members using object.
- We can create object for a class from static context.

Dynamic method:

- Defining a method inside the class by writing "self" variable as first argument.
- "self" is not a keyword.
- "self" is a recommended variable to define Dynamic methods.
- Definition of dynamic method as follows.

```
class Test:
    def m1():
        # static method
        return

    def m2(self):
        # dynamic method
        return

    def __init__(self):
        # constructor
        return
```

self:

- It is used to define dynamic methods and constructor.
- It is not a keyword but it is the **most recommended** variable to define dynamic functionality.
- It is an argument(local variable of that function)
- We can access "self" only from the same function or constructor.
- "self" variable holds object address.

Constructor:

- A special method with pre-defined identity(__init__).
- It is a dynamic method(first argument is self)
- It invokes automatically in the process of Object creation.

Creating object in main():

- Application execution starts from static context(main).
- We can access non static members using object address.
- We create object(take permission) in static context(main)

```
class Test:
    def __init__(self):
        print("Constructor")
        return

    def main():
        Test() #access constructor
        return

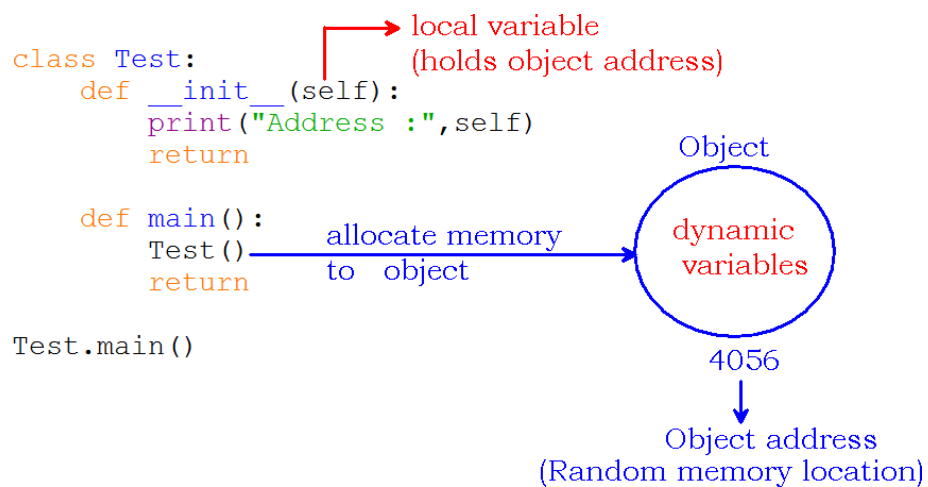
Test.main()
```

Constructor executes every time when we create the object.

```
class Test:
    def __init__(self):
        print("Constructor")
        return

    def main():
        for i in range(10):
            Test()
        return

Test.main()
```



```
class Test:
    def __init__(self):
        print("Address :",self)
        return

    def main():
        Test()
        return

Test.main()
```

id(): A pre-defined method that returns integer value of specified Object.

type(): Returns the class name of specified object.

```
class Test:
    def __init__(self):
        print("Address :",id(self))
        print("Name :",type(self))
        return

    def main():
        Test()
        return

Test.main()
```

We cannot access 'self' in main() method to work with object.

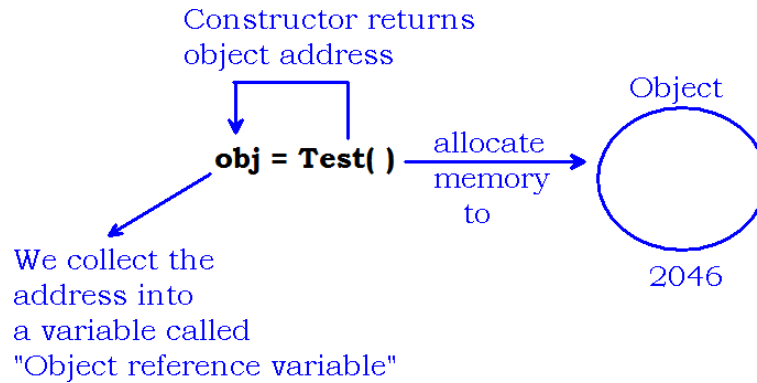
```
class Test:
    def __init__(self):
        print("In Constructor :",id(self))
        return

    def main():
        Test()
        print("In main :", id(self))
        return

Test.main()
```

Object reference variable:

- Constructor returns address of Object after creation.
- We can collect the address assigning to any variable.
- The variable stores object address is called "Object reference variable"



We can display object address in main using "object reference variable.

```
class Test:
    def __init__(self):
        print("In Constructor :",id(self))
        return

    def main():
        addr = Test()
        print("In main :", id(addr))
        return

Test.main()
```

Access static and dynamic methods from main:

```
class Test:
    def m1():
        # static method
        return

    def m2(self):
        # dynamic method
        return
```

Default constructor:

- In object creation process, constructor executes implicitly.
- When we don't define any constructor, a default constructor will be added to the source code.
- Default constructor has empty definition – no logic.

```
class Test:

    def main():
        x = Test()
        y = Test()
        print("Address of x : ", id(x))
        print("Address of y : ", id(y))
        return

Test.main()
```

We can define the default constructor explicitly to understand the object creation process.

```
class Test:
    def __init__(self):
        print("Object Created :", id(self))
        return

    def main():
        x = Test()
        y = Test()
        print("Address of x : ", id(x))
        print("Address of y : ", id(y))
        return

Test.main()
```

Accessing static and dynamic methods:

```
class Test:
    def main():
        obj = Test()
        Test.m1()
        obj.m2()
        return

    def m1():
        print("Static method")
        return

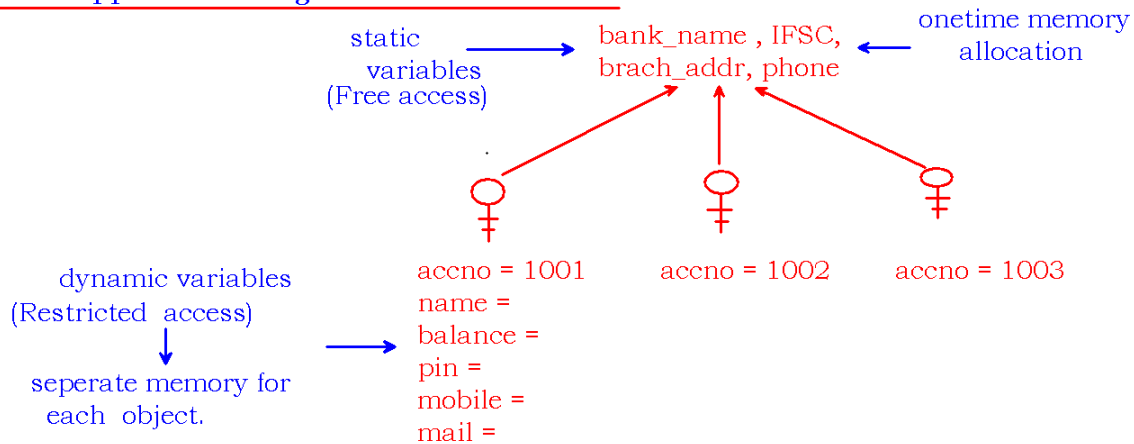
    def m2(self):
        print("Dynamic method")
        return

Test.main()
```

Dynamic variables:

Static Variables	Dynamic Variables
Static variables store common information of Object.	We create dynamic variables inside the object using 'self' variable.
Static variables get memory only once.	Variables creation must be from constructor.
We can access static variables using class Name	We can access the variables through object address.

Bank App : Processing account holders data

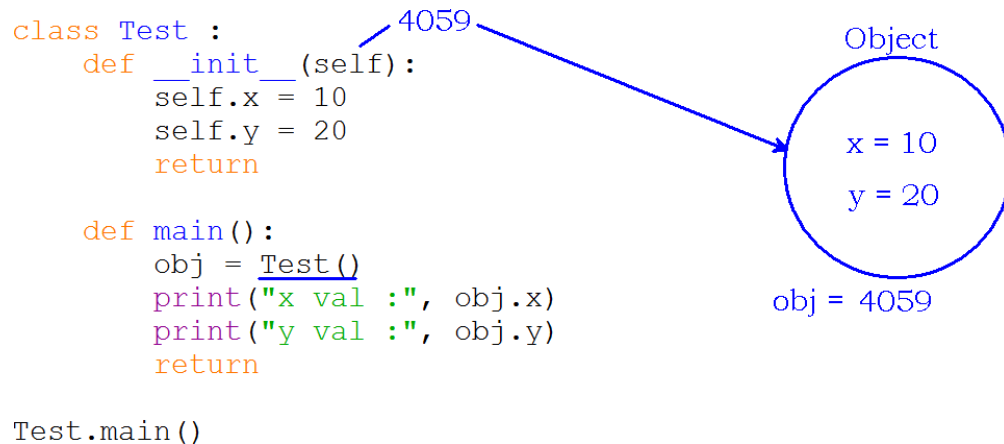


Constructor is used to define dynamic variables:

- Constructor is a special method
- Execute every time automatically in object creation process.
- Constructors used to create dynamic variables.
- As soon as object memory is ready, constructor allocates memory to dynamic variables inside the object.

Access instace variables:

```
class Test :  
    def _init_(self):  
        self.x = 10  
        self.y = 20  
        return  
  
    def main():  
        obj = Test()  
        print("x val :", obj.x)  
        print("y val :", obj.y)  
        return  
Test.main()
```



Note: In the above program, if we assign values directly to dynamic variables, all objects initializes with same set of values.

```
class Test:
    def _init_(self):
        self.a = 10
        return

    def main():
        t1 = Test()
        print("t1 a val :", t1.a)

        t2 = Test()
        print("t2 a val :", t2.a)
        return

Test.main()
```

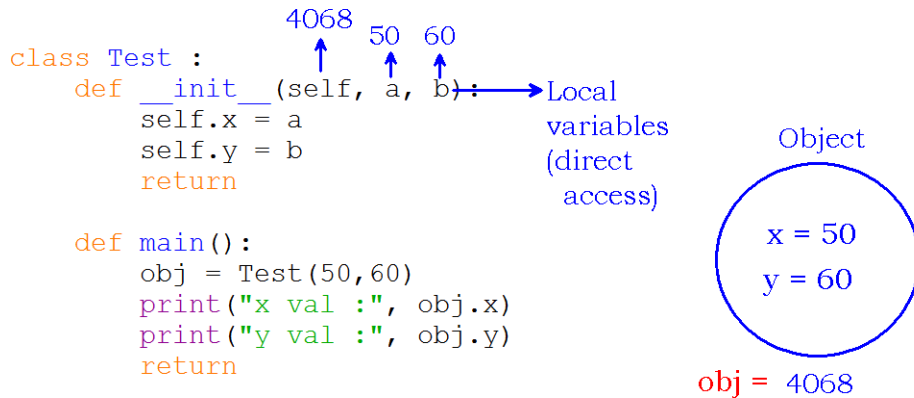
Arguments constructor:

- Constructor taking input values to initialize dynamic variables in Object creation process.
- Using arguments constructor, we can set different values to different object variables.
- Arguments are local variables and we can access directly.

```
class Test :
    def __init__(self, a, b):
        self.x = a
        self.y = b
        return
```

```
def main():
    obj = Test(50,60)
    print("x val :", obj.x)
    print("y val :", obj.y)
    return
```

Test.main()



Test.main()

self.x = a --> local variable "a" value
assigning to dynamic variable "x"

- Local variables and dynamic variables can have the same identity.
- We access the local variable directly where as dynamic variable using object address.

```
class Test:
    def __init__(self,a,b):
        self.a = a
        self.b = b
        return

    def main():
        obj = Test(10,20)
        print("a val :", obj.a)
        print("b val :", obj.b)
        return
```

Test.main()

WAP to construct the object by taking user input:

```
class Test:
    def __init__(self,a,b):
        self.a = a
        self.b = b
        return

    def details(self):
        print("a val :", self.a)
        print("b val :", self.b)
        return

    def main():
        print("Enter a, b values :")
        a = input()
        b = input()
        obj = Test(a,b)
        obj.details()
        return

Test.main()
```

Read Emp details and Construct object:

```
class Emp:
    def __init__(self, num, name):
        self.num = num
        self.name = name
        return

    def details(self):
        print("Emp num is :", self.num)
        print("Emp name is :", self.name)
        return

class Access:
    def main():
        print("Enter emp details :")
        num = int(input())
        name = input()
        obj = Emp(num, name)
        obj.details()
        return

Access.main()
```

Account Operations program:

```
class Account:
    def __init__(self,amt):
        self.balance = amt
        return

    def deposit(self,amt):
        self.balance=self.balance+amt
        print(amt,"deposited")
        return

    def withdraw(self,amt):
        print("Withdrawing :",amt)
        print("Avail bal :",self.balance)
        if(amt<=self.balance):
            print("Collect cash :",amt)
            self.balance=self.balance-amt
        else:
            print("Error : Low balance")
        return

class Bank:
    def main():
        amt = int(input("Enter initial bal : "))
        acc = Account(amt)
        print("Balance is :",acc.balance)

        amt = int(input("Enter deposit amt :"))
        acc.deposit(amt)
        print("After deposit :",acc.balance)

        amt = int(input("Enter withdraw amt :"))
        acc.withdraw(amt)
        print("Final balance :",acc.balance)
        return

Bank.main()
```

Access Modifiers

Access Modifiers:

- Access modifiers are used to set permissions to access the data.
- Python supports 3 access modifiers.
 - private(`_ _var`)
 - protected (`_var`)
 - public (`var`)

Note: Protected members can be discussed with Inheritance concept.

public:

- We can access public members (variables or methods) directly.
- All programs discussed before contains public variables and methods.

```
class Test:
    a=10
    def __init__(self,b):
        self.b = b
        return

class Access:
    def main():
        obj = Test(20)
        print("a val :", Test.a)
        print("b val :", obj.b)
        return

Access.main()
```

Private members:

- A member definition preceded by `_ _` is called private member.
- One object(class) cannot access the private members of another object directly.

Note: A class itself can access the private members.

```
class First:
    a = 10 #public
    __b = 20 #private

    def main():
        print("a val :", First.a)
        print("b val :", First.__b)
        return

First.main()
```

Accessing private members of another class results Error:

```
class First:
    a = 10 #public
    __b = 20 #private

class Second:
    def main():
        print("a val :", First.a)
        print("b val :", First.__b)
        return

Second.main()
```

Private dynamic variables creation and Access:

```
class First:
    def __init__(self,x,y):
        self.a = x # 'a' is public
        self.__b = y # 'b' is private
        return

    def main():
        obj = First(10,20)
        print("a val : ", obj.a)
        print("b val : ", obj.__b)
        return

First.main()
```

Accessing private variables from other class results error:

```
class First:
    def __init__(self,x,y):
        self.a = x
        self.__b = y
        return

class Second:
    def main():
        obj = First(10,20)
        print("a val : ", obj.a)
        print("b val : ", obj.__b)
        return

Second.main()
```

Accessing private variables from another class:

- One class cannot access the private information from another class directly.
- A class(object) is allowed to share(send or receive) the private data in communication.
- Communication between objects is possible using methods.
- Two standard methods get() and set() to share the information.

```
class First:
    __a = 10
    def getA():
        return First.__a

class Second:
    def main():
        # print("a val : ", First.__a)
        print("a val : ", First.getA())
        return

Second.main()
```

Accessing dynamic private variable using dynamic get() method:

```
class First:
    def __init__(self,a):
        self.__a = a
        return

    def getA(self):
        return self.__a

class Second:
    def main():
        obj = First(10)
        # print("a val : ", obj.__a)
        print("a val : ", obj.getA())
        return

Second.main()
```

Modifying private variables data:

- We cannot set values directly to private variables.
- We use set() method to modify the data.

Note: When we try set the value directly to private variable, the value will be omitted.

```
class First:
    _a=10
    def getA():
        return First._a
    def setA(a):
        First._a = a
        return

class Second:
    def main():
        print("a val :", First.getA())
        First._a=20
        print("a val :", First.getA())
        First.setA(20)
        print("a val :", First.getA())
        return

Second.main()
```

Setting values to dynamic private variables:

```
class First:
    def _init_(self,a):
        self._a = a
        return
    def getA(self):
        return self._a
    def setA(self,a):
        self._a = a
        return

class Second:
    def main():
        obj = First(10)
        print("a val :", obj.getA())
        obj.setA(20)
        print("a val :", obj.getA())
        return

Second.main()
```

Encapsulation: The concept of protecting object information.

Rules:

1. Class is public – Object is visible to other objects in communication.
2. Variables are private – One object cannot access the information of other object directly.
3. Communication between objects using get() and set() methods to share the information.

```
class Emp:
    def __init__(self,num,name,salary):
        self._num = num
        self._name = name
        self._salary = salary
        return

    def getNum(self):
        return self._num
    def getName(self):
        return self._name
    def getSalary(self):
        return self._salary

    def setNum(self,num):
        self._num = num
        return
    def setName(self,name):
        self._name = name
        return
    def setSalary(self,salary):
        self._salary = salary
        return

class Access:
    def main():
        print("Enter Emp details :")
        num = int(input("Emp Num : "))
        name = input("Emp Name : ")
        salary = float(input("Emp Sal : "))
        obj = Emp(num, name, salary)
        print("Name :",obj.getName())
        return
Access.main()
```

Inheritance in Python

Inheritance:

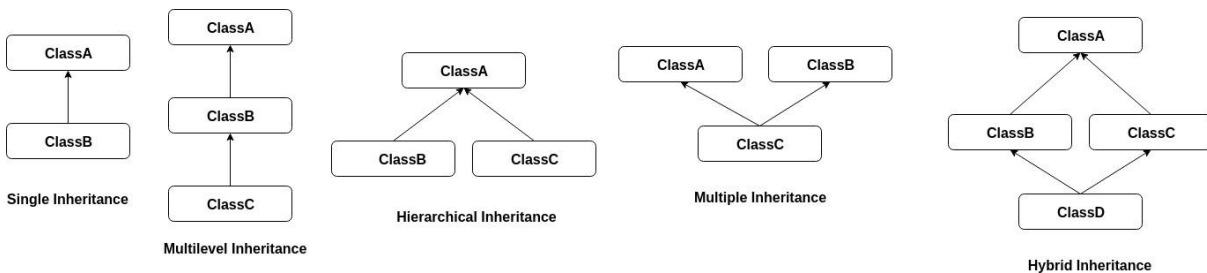
- Defining an object by re-using the functionality of existing object.
- The main advantage of inheritance in code re-usability.

Terminology of inheritance :

- Parent – Child class
- Super – Sub class
- Base – Derived class

Types of Inheritance:

- Python supports all types of inheritance supported by Object Oriented Programming.
- The following diagram represents all types



Single Inheritance:

- In parent-child relation, we can access the functionality of Parent through child.
- We cannot access Child functionality using Parent.
- Accessing static members in Parent and Child as follows:

```
class Parent:
    def m1():
        print("Parent's m1")
        return
```

```
class Child(Parent):
    def m2():
        print("Child's m2")
        return
```

```
class Inheritance:
    def main():
        Child.m1()
        Child.m2()
```

```
Inheritance.main()
```

Accessing dynamic functionality in Parent-Child relation:

```
class Parent:
    def m1(self):
        print("Parent's m1")
        return

class Child(Parent):
    def m2(self):
        print("Child's m2")
        return

class Inheritance:
    def main():
        c = Child()
        c.m1()
        c.m2()

Inheritance.main()
```

Method overriding:

- Defining a method in the Child class with same name and same set of arguments of Parent class method.
- When two methods in Parent and Child with the same identity, it gives the first priority to Child object.

```
class Parent:
    def fun(self):
        print("Parent's fun()")
        return

class Child(Parent):
    def fun(self): #override
        print("Child's fun()")
        return

class Inheritance:
    def main():
        obj = Child()
        obj.fun()
        return

Inheritance.main()
```

Advantage of overriding:

- Overriding is the concept of updating existing object functionality when it is not sufficient to extended object.
- Re-writing the function logic with the same identity.

Complete Inheritance with Code program:

- # 1. Accessing existing functionality
- # 2. Adding new features
- # 3. Update(override) existing features

```
class Guru:
    def call(self):
        print("Guru - Call")
        return
    def camera(self):
        print("Guru - Camera - 2MP")
        return

class Galaxy(Guru):
    def videoCall(self):
        print("Galaxy - Video Call")
        return
    def camera(self):
        print("Galaxy - Camera - 8MP")
        return

class Inheritance:
    def main():
        g1 = Galaxy()
        g1.call()# Access existing
        g1.videoCall()# new feature
        g1.camera() #updated

        g2 = Guru()
        g2.call()
        g2.camera()
        g2.videoCall() # error:
        return

Inheritance.main()
```

Accessing overridden functionality of Parent class:

super():

- It is pre-defined method.
- It is used to access Parent class functionality(super) from Child(sub).

```
class Grand:
    def fun(self):
        print("Grand")
        return

class Parent(Grand):
    def fun(self):
        super().fun()
        print("Parent")
        return

class Child(Parent):
    def fun(self):
        super().fun()
        print("Child")
        return

class Inheritance:
    def main():
        obj = Child()
        obj.fun()
        return

Inheritance.main()
```

- We can access the functionality of all classes in the hierarchy from one place using super() method.
- We need to specify the Class type along with object reference variable.
- If we specify the Child type, it access Parent functionality.

```
class Grand:
    def fun(self):
        print("Grand")
        return

class Parent(Grand):
    def fun(self):
        print("Parent")
```

```

return

class Child(Parent):
    def fun(self):
        print("Child")
        return

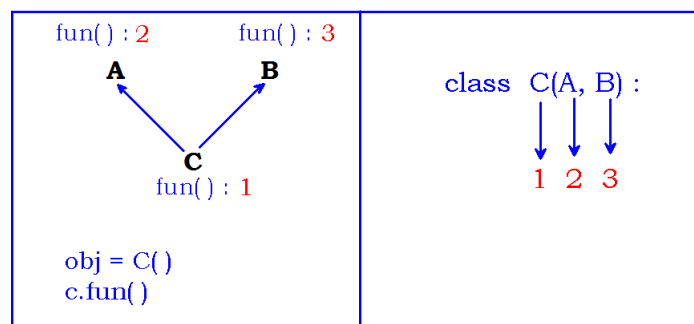
class Inheritance:
    def main():
        obj = Child()
        obj.fun()
        super(Child, obj).fun()
        super(Parent, obj).fun()
        return

Inheritance.main()

```

Accessing the functionality in Multiple Inheritance:

- While accessing the functionality of Multiple Inheritance or Hybrid Inheritance, we need to understand the concept of MRO(Method Resolution Order).
- MRO is the concept of how the Interpreter searching for methods while accessing using Child object address.



```

class A:
    def m1(self):
        print("A-m1")
        return

    def m3(self):
        print("A-m3")
        return

```

```

class B:
    def m1(self):
        print("B-m1")
        return

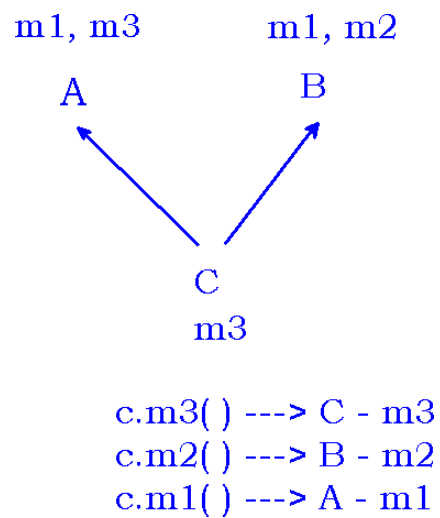
    def m2(self):
        print("B-m2")
        return

class C(A,B):
    def m3(self):
        print("C-m3")
        return

class Multiple:
    def main():
        c = C()
        c.m3()
        c.m2()
        c.m1()
        return

```

Multiple.main()



Accessing the complete functionality of all objects in multiple inheritance:

```
class A:
    def m1(self):
        print("A-m1")
        return

    def m3(self):
        print("A-m3")
        return

class B:
    def m1(self):
        print("B-m1")
        return

    def m2(self):
        print("B-m2")
        return

class C(A,B):
    def m3(self):
        print("C-m3")
        return

class Multiple:
    def main():
        obj = C()
        obj.m1()
        obj.m2()
        obj.m3()
        super(C,obj).m1()
        super(C,obj).m2()
        A.m3(obj)
        B.m1(obj)
        return

Multiple.main()
```

```

class A:
    def m1(self):
        print("A-m1")
        return

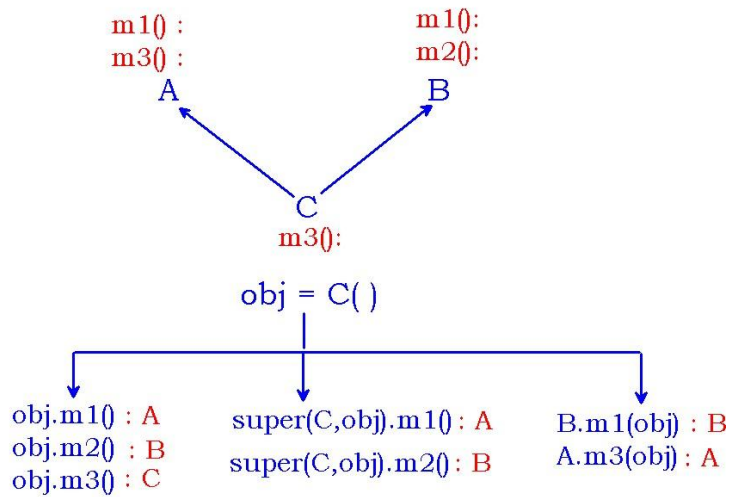
    def m3(self):
        print("A-m3")
        return

class B:
    def m1(self):
        print("B-m1")
        return

    def m2(self):
        print("B-m2")
        return

class C(A,B):
    def m3(self):
        print("C-m3")
        return

```



Hybrid inheritance:

```

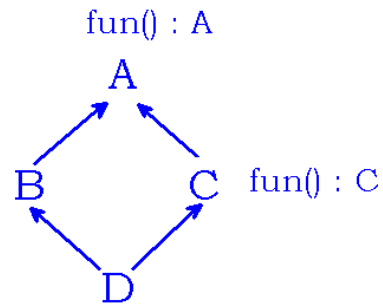
class A:
    def fun(self):
        print("A")
        return

class B(A):
    pass

class C(A):
    def fun(self):
        print("C")
        return

class D(B,C):
    pass

```



```

class Hybrid:
    def main():
        obj = D()
        A.fun(obj) --> A
        B.fun(obj) --> A
        C.fun(obj) --> C
        D.fun(obj) --> C

```

Polymorphism in Python

Polymorphism:

- Defining an object(class) that shows different behavior(methods) with the same identity.
- Polymorphism is of 2 types
 - Compile time polymorphism
 - Runtime polymorphism

Compile time polymorphism:

- It is also called "Static binding".
- It is "Method Overloading" technique.
- Defining multiple methods with same name and different set of arguments is called "Overloading".
- Depends on input values, corresponding method executes.

Note: Python doesn't support method overloading. When we define a duplicate method, existing method will be replaced.

```
class Calc:
    def add(a,b):
        res=a+b
        print("Sum of 2 num's :", res)
        return

    def add(a,b,c): # replace existing add() method
        res=a+b+c
        print("Sum of 3 num's :", res)
        return

class Overload:
    def main():
        Calc.add(10,20,30)
        Calc.add(10,20) # error
        return

Overload.main()
```

How can we implement overloading in python?

- In other programming languages, we can define overloading by writing multiple methods either with different types of input or with different length of input.
- As python doesn't support data types, we can pass different types of data into same variables.

```
class Calc:
    def add(a,b):
        res=a+b
        print("Sum :", res)
        return

class Overload:
    def main():
        Calc.add(10,20)
        Calc.add(2.3,4.5)
        Calc.add("Python","Class")
        return

Overload.main()
```

Python supports variables arguments function. We can pass different length of arguments to a single method.

```
class Calc:
    def add(*arr):
        l = len(arr)
        sum = 0
        for ele in arr:
            sum = sum+ele
        print("Sum of",l,"elements :",sum)
        return

class Overload:
    def main():
        Calc.add(10,20)
        Calc.add(10,20,30)
        Calc.add(10,20,30,40,50)
        return

Overload.main()
```

Runtime Polymorphism:

- It is called "dynamic binding".
- It is method "overriding technique".
- Overriding is the concept of defining a method in Child class with the same name and same set of arguments in Parent class method.
- A Child object can shows the functionality of Parent and Child. Hence it is called Polymorphic

```
class Grand:
    def fun(self):
        print("Grand")
        return

class Parent(Grand):
    def fun(self):
        print("Parent")
        return

class Child(Parent):
    def fun(self):
        print("Child")
        return

class Override:
    def main():
        obj = Child()
        obj.fun()
        super(Child,obj).fun()
        super(Parent,obj).fun()
        return

Override.main()
```

Exception Handling in Python

Exception Handling:

- Exception is a "class"
- Exception is a "Runtime Error"
- When exception rises, the flow of execution will be terminated with informal information to the user.

```
class Add:  
    def main():  
        print("Enter 2 integers :")  
        x = int(input())  
        y = int(input())  
        z = x+y  
        print("Sum : ", z)  
        print("End...")  
        return  
  
Add.main()
```

Note the followings:

- Exception handling is the concept of working with failure cases of logic.
- As a programmer, we need to analyze success cases and failure cases in the transaction to implement the code.

Default Exception handler:

- It is a pre-defined program.
- When exception raises, an object will be created with the complete information of that Error is called "Exception Object".
- It is recommended to handle every exception in the application.
- If we don't handle, the object will be transferred to "Default Exception Handler"

Handling the exceptions:

Try:

- Try block is used to place the doubtful code that may raise exception.
- If exception raises in try block, an exception object will be raised with the error information.

Except:

- Except block is used to collect and handle the exception object which is raised in try block.

Syntax:

```
try :  
    ->Doubtful code...  
except <Exception_type> var :  
    ->Handling logic...
```

Reading 2 numbers and perform addition operation: Chance of getting ValueError if the user entered invalid input

```
class Demo:  
    def main():  
        try:  
            print("Enter 2 numbers :")  
            x = int(input())  
            y = int(input())  
            z = x+y  
            print("Sum :",z)  
        except ValueError:  
            print("Exception : Invalid input")  
        print("End")  
        return  
Demo.main()
```

try with multi except: One try block can have more than one except blocks to handle different types of exceptions occur in different lines of code. Only one except block executes among we defined.

```
class Division:  
    def main():  
        try:  
            print("Enter 2 numbers :")  
            x = int(input())  
            y = int(input())  
            print("Result :", x/y)  
  
        except ValueError:  
            print("Exception : Invalid input")  
  
        except ZeroDivisionError:  
            print("Exception : Denominator should not be zero")  
        print("End")  
        return  
Division.main()
```

Exception class:

- "Exception" is pre-defined class.
- "Exception" is the Parent class of all other exception classes.
- Instead of handling multiple exceptions with number of except blocks, we can specify "Exception" type to handle all.
- We need to provide the common the error information to handle using "Exception" class.

Note: We can display the message of Exception object by collecting into variable in "except" block.

```
class Division:
    def main():
        try:
            print("Enter 2 numbers :")
            x = int(input())
            y = int(input())
            z = x/y
            print("Result :",z)

        except Exception as msg:
            print("Exception :",msg)

        return

Division.main()
```

Finally block: It is used to provide "Resource releasing logic". All resources (connected to program) must be closed from finally block.

Note: Finally block executes whether or not an exception has raised in the try block.

```
class Finally:
    def main():
        try:
            x = 10/5
            print("Try block")
        except Exception:
            print("Except block")
        finally:
            print("Finally block")
        return

Finally.main()
```

Finally block executes though exception occurs

```
class Finally:
    def main():
        try:
            x = 10/0
            print("Try block")

        except Exception:
            print("Except block")

        finally:
            print("Finally block")

        return

Finally.main()
```

Open and Close the File using Finally block:

- open() is pre-defined and it is used to open the file in specified path.
- If the file is not present, it raises Exception.

```
class Finally:
    def main():
        try:
            file = open("sample.txt")
            print("File opened...")
        except FileNotFoundError:
            print("Exception : No such file")
        return

    Finally.main()
```

- When we connect any resource to the program, we must release that resource.
- close() function is pre-defined and it is used to release any resource.
- Closing statements must be in finally block.

```
class Finally:
    file = None
    def main():
        try:
            Finally.file = open("sample.txt")
            print("File opened...")
```

```

except FileNotFoundError:
    print("Exception : No such file")

finally:
    Finally.file.close()
    print("File closed...")
return

```

```
Finally.main()
```

- In the above application, if the file is not present, close() function raises exception as it invokes on "None" value.
- We need to close the file only by checking whether it is pointing to any address on None.

```

class Finally:
    file = None
    def main():
        try:
            Finally.file = open("sample.txt")
            print("File opened...")

        except FileNotFoundError:
            print("Exception : No such file")

        finally:
            if Finally.file != None:
                Finally.file.close()
                print("File closed...")
        return

```

```
Finally.main()
```

Custom Exceptions:

- Python library is providing number of exception classes.
- As a programmer, we can define custom exceptions depends on application requirement.
- Every custom exception should extends the functionality of pre-defined Exception class.

raise:

- It is a keyword.
- It is used to raise Custom Exception explicitly by the programmer.

- Pre-defined exceptions will be raised automatically when problem occurs.
- If we don't handle the exception, Default Exception Handler handles.

```
class CustomError(Exception):
    def __init__(self,name):
        self.name=name
        return

class RaiseException:
    def main():
        obj = CustomError("Error-Msg")
        raise obj
        return

RaiseException.main()
```

Handling Custom Exception:

- Generally exception rises inside the function.
- We need to handle the exception while calling the function.
- When exception has raised, the object will be thrown to function calling area.

```
class CustomError(Exception):
    def __init__(self,name):
        self.name=name
        return

class Test:
    def fun():
        obj = CustomError("Message")
        raise obj
        return

class Access:
    def main():
        try:
            Test.fun()
        except CustomError as e:
            print("Exception :",e)
        return

Access.main()
```

Exceptions in Banking application:

- Runtime error is called Exception.
- When we perform the transaction, in case of any error in the middle of transaction called "Runtime error".
- We must define every error as Exception.
- The following example explains how to define Runtime Errors in Banking trasactions.

```
class LowBalanceError(Exception):
    def __init__(self,name):
        self.name = name
        return

class Account:
    def __init__(self,balance):
        self.balance = balance
        return

    def withdraw(self,amount):
        print("Trying to withdraw :",amount)
        print("Avail bal :",self.balance)
        if amount <= self.balance:
            print("Collect cash :",amount)
            self.balance=self.balance-amount
        else:
            err=LowBalanceError("Low Balance")
            raise err
        return

class Bank:
    def main():
        amount = int(input("enter amount :"))
        acc = Account(amount)
        print("Balance is :",acc.balance)

        amount = int(input("Enter withdraw amt :"))
        try:
            acc.withdraw(amount)
        except LowBalanceError as e:
            print("Exception :",e)
        print("Final Balance is :",acc.balance)
        return

Bank.main()
```

Inner classes

Inner class: Defining a class inside another class. It is also called Nested class.

Syntax:

```
class Outer:
    .....
    logic
    .....
    class Inner:
        .....
        logic
        .....
```

Accessing static functionality:

- Static members we access using class name.
- Inner class can be accessed using Outer class name.

```
class Outer:
    def m1():
        print("Outer-m1")
        return

    class Inner:
        def m2():
            print("Inner-m2")
            return

class Access:
    def main():
        Outer.m1()
        Outer.Inner.m2()
        return

Access.main()
```

Accessing dynamic functionality:

- We can access dynamic member through object address.
- We create object for inner class with the reference of outer class only.

```
class Outer:
    def m1(self):
        print("Outer-m1")
        return

    class Inner:
        def m2(self):
            print("Inner-m2")
            return

class Access:
    def main():
        obj1 = Outer()
        obj1.m1()

        obj2 = obj1.Inner()
        obj2.m2()
        return

Access.main()
```

We create object directly to inner class as follows:

```
class Outer:
    class Inner:
        def fun(self):
            print("Inner-fun")
            return

class Access:
    def main():
        #obj1 = Outer()
        #obj2 = obj1.Inner()
        #obj2.fun()
        Outer().Inner().fun()
        return

Access.main()
```

Local inner classes:

- Defining a class inside the method.
- To access that class, first control should enter into that function.
- We invoke the functionality of local inner class from that function.

```
class Outer:
    def fun():
        print("Outer-fun")

        class Local:
            def fun():
                print("Outer-Local-fun")
            return

        Local.fun()
        return

class Access:
    def main():
        Outer.fun()
        return

Access.main()
```

We can define duplicate classes in different functions:

```
class Outer:
    def m1():
        print("Outer-m1")

        class Local:
            def fun():
                print("m1-Local-fun")
            return

        Local.fun()
        return

    def m2():
        print("Outer-m2")

        class Local:
            def fun():
```

```
        print("m2-Local-fun")
        return
```

```
    Local.fun()
    return
```

```
class Access:
    def main():
        Outer.m1()
        Outer.m2()
        return
```

```
Access.main()
```

We can define a local inner class inside another local inner class. But it is complex to access the functionality.

```
class Outer:
    def fun(self):
        print("Outer-fun")

        class Local:
            def fun(self):
                print("Outer-Local-fun")

                class Inner:
                    def fun(self):
                        print("Outer-Local-Inner-fun")
                        return
```

```
        Inner().fun()
        return
```

```
    Local().fun()
    return
```

```
class Access:
    def main():
        Outer().fun()
        return
```

```
Access.main()
```

Python Modules

Importance of Modularity:

- When people write large programs they tend to break their code into multiple different files for ease of use, debugging and readability.
- In Python we use modules to achieve such goals.
- Modules are nothing but files with Python definitions and statements.
- The name of the file should be valid Python name (think about any variable name) and in lowercase

Pre-defined modules:

- **threading** : To implement Parallel processing
- **gc** : For Garbage collection
- **tkinter** : To implement GUI programming
- **time** : To find system time and data and to display in different formats
- **numpy** : One, two and Multi dimensional
- **re** : Regular expressions
- **mysql** : Python – MySQL database connectivity

connecting modules using import:

- import is a keyword.
- import is used to connect the modules to access their members.

Function based modules:

- A module cannot contain classes(object oriented programming).
- We have such type of library modules like time, data and so on.
- We access the functions using module name after import.

arithmetic.py:

```
def add(a,b):  
    c=a+b  
    return c
```

```
def subtract(a,b):  
    c=a-b  
    return c
```

```
def multiply(a,b):  
    c=a*b  
    return c
```

calc.py:

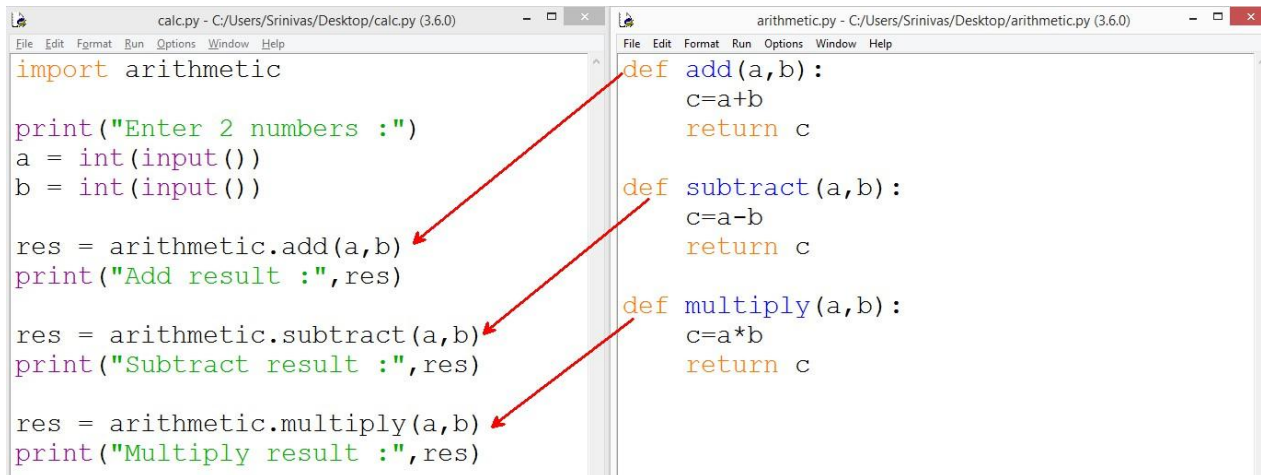
```
import arithmetic

print("Enter 2 numbers :")
a = int(input())
b = int(input())

res = arithmetic.add(a,b)
print("Add result :",res)

res = arithmetic.subtract(a,b)
print("Subtract result :",res)

res = arithmetic.multiply(a,b)
print("Multiply result :",res)
```



Why we need to call the function using module name after import?

- We can define members with same identity in different modules
- We access the duplicate members from different modules by using the identity of module while accessing.

```
def f1():
    print("one-f1()")
    return

def f2():
    print("one-f2()")
    return
```

one.py:

two.py:

```
def f1():
    print("two-f1()")
    return

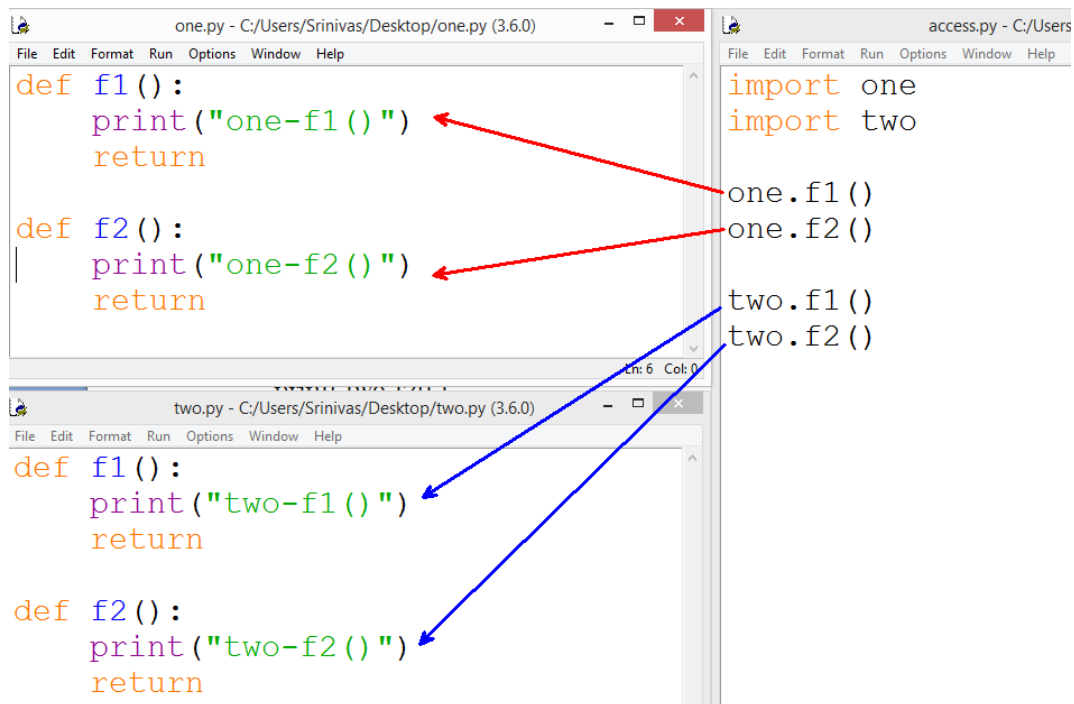
def f2():
    print("two-f2()")
    return
```

access.py:

```
import one
import two

one.f1()
one.f2()

two.f1()
two.f2()
```

**Class Based modules:**

- A module is a collection of classes.
- A class is a collection of variables and methods
- To access the class members we need to import the module.

one.py:

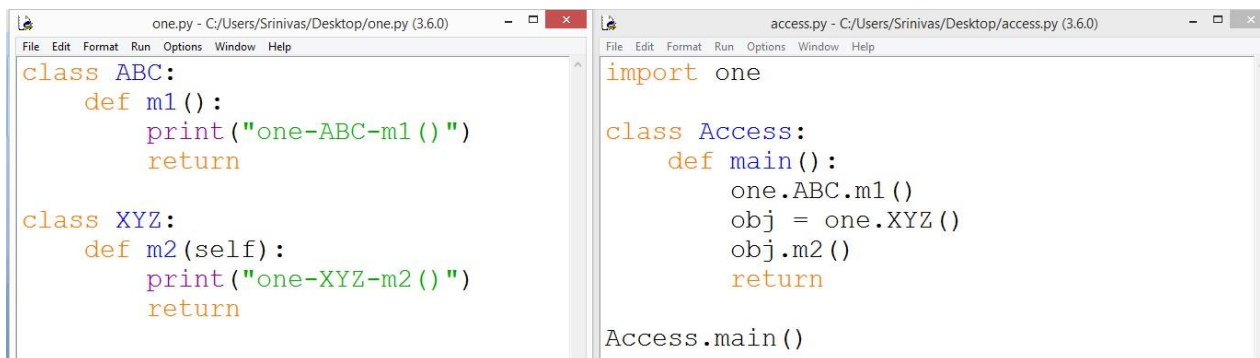
```
class ABC:
    def m1():
        print("one-ABC-m1()")
        return

class XYZ:
    def m2(self):
        print("one-XYZ-m2()")
        return
```

access.py:

```
import one
class Access:
    def main():
        one.ABC.m1()
        obj = one.XYZ()
        obj.m2()
        return

Access.main()
```

**from:**

- "from" is a keyword used to access one or more classes from the specified module.
- Note that, no need to specify the module name along with class name if we import using "from"

Syntax:

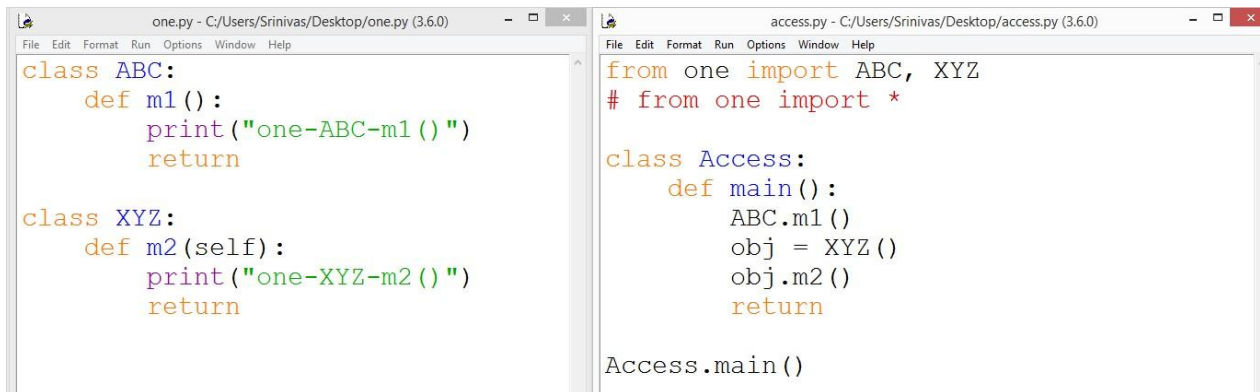
from module import class

Or

from module import *

Or

from module import class1, class2, class3



```
one.py - C:/Users/Srinivas/Desktop/one.py (3.6.0)
class ABC:
    def m1():
        print("one-ABC-m1()")
        return

class XYZ:
    def m2(self):
        print("one-XYZ-m2()")
        return

access.py - C:/Users/Srinivas/Desktop/access.py (3.6.0)
from one import ABC, XYZ
# from one import *

class Access:
    def main():
        ABC.m1()
        obj = XYZ()
        obj.m2()
        return

Access.main()
```

Accessing duplicate classes from different modules:

One.py:

```
class ABC:
    def m1():
        print("one-ABC-m1()")
        return

class XYZ:
    def m2():
        print("one-XYZ-m2()")
        return
```

two.py:

```
class ABC:
    def m1():
        print("two-ABC-m1()")
        return

class XYZ:
    def m2():
        print("two-XYZ-m2()")
        return
```

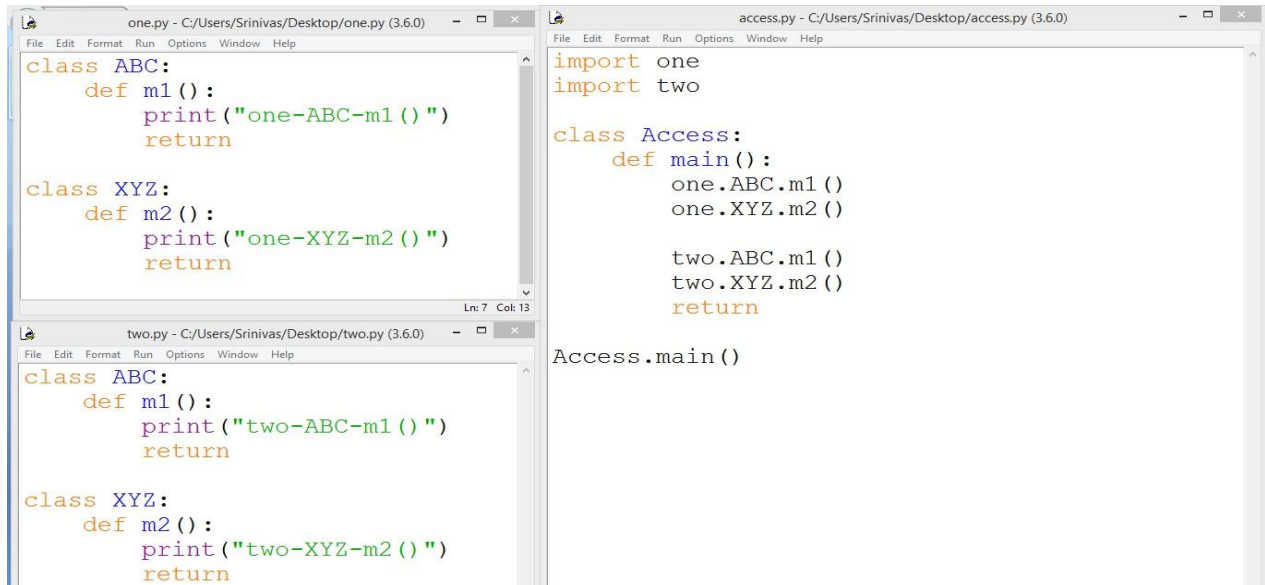
access.py:

```
import one
import two

class Access:
    def main():
        one.ABC.m1()
        one.XYZ.m2()
```

```
two.ABC.m1()
two.XYZ.m2()
return
```

```
Access.main()
```



```
one.py - C:/Users/Srinivas/Desktop/one.py (3.6.0)
class ABC:
    def m1():
        print("one-ABC-m1()")
        return

class XYZ:
    def m2():
        print("one-XYZ-m2()")
        return

two.py - C:/Users/Srinivas/Desktop/two.py (3.6.0)
class ABC:
    def m1():
        print("two-ABC-m1()")
        return

class XYZ:
    def m2():
        print("two-XYZ-m2()")
        return

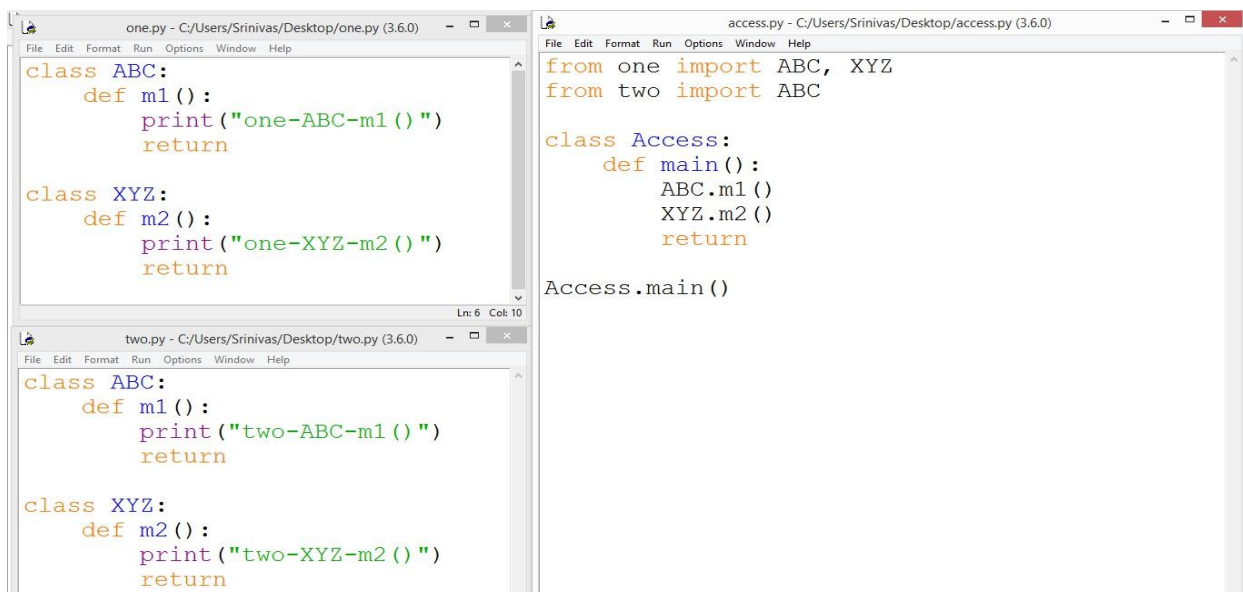
access.py - C:/Users/Srinivas/Desktop/access.py (3.6.0)
import one
import two

class Access:
    def main():
        one.ABC.m1()
        one.XYZ.m2()

        two.ABC.m1()
        two.XYZ.m2()
        return

Access.main()
```

- When we import classes using 'from' keyword, we cannot work with duplicate classes from different modules.
- Duplicate class replace the existing class as follows



```
one.py - C:/Users/Srinivas/Desktop/one.py (3.6.0)
class ABC:
    def m1():
        print("one-ABC-m1()")
        return

class XYZ:
    def m2():
        print("one-XYZ-m2()")
        return

two.py - C:/Users/Srinivas/Desktop/two.py (3.6.0)
class ABC:
    def m1():
        print("two-ABC-m1()")
        return

class XYZ:
    def m2():
        print("two-XYZ-m2()")
        return

access.py - C:/Users/Srinivas/Desktop/access.py (3.6.0)
from one import ABC, XYZ
from two import ABC

class Access:
    def main():
        ABC.m1()
        XYZ.m2()
        return

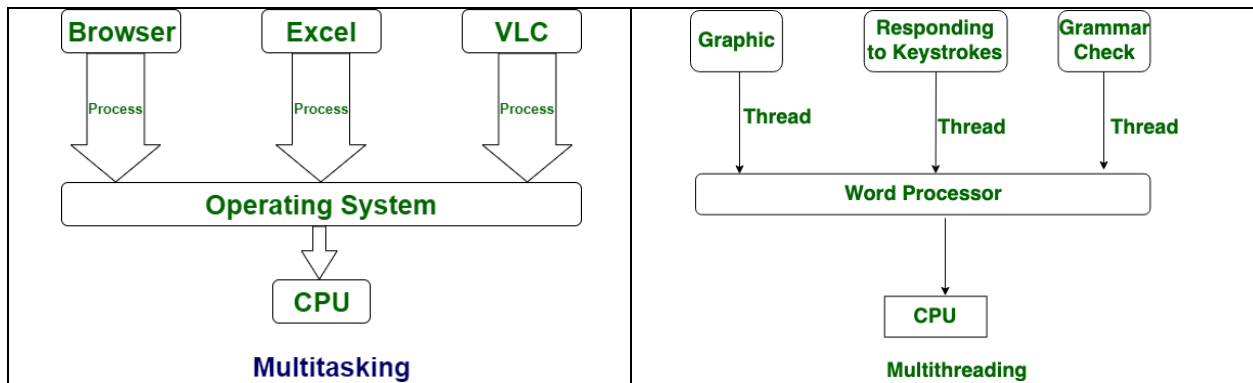
Access.main()
```

Multi threading

Multi-threading is a process of executing two or more threads(programs) simultaneously to utilize the processor maximum.

Multi-tasking:

- We can implement multi-tasking in two ways.
 - Program(process) based
 - Sub program(thread) based.



Note: Multi tasking is the concept of maximum utilization of Processor.

Multi threading:

- Task is a program(or process)
- Thread is a sub program (or sub process)
- We can execute multiple threads from single process.
- Using multi-threading, we can reduce the stress on processor.

Creating Thread:

- Python provides a threading module that allows you to create and manage threads in your program.
- To create a new thread, you can define a new class that extends the threading.Thread class and override the run() method.

Example:

```
import threading
```

```
class MyThread(threading.Thread):
```

```
    def run(self):
```

```
        # Code to be executed in the new thread
```

You can start a new thread by creating an instance of the class and calling its start() method.

```
# Example:
my_thread = MyThread()
my_thread.start()
```

Single threaded application:

- Every python program is single threaded.
- In single threaded application, execution starts at main() and ends at same.
- Main() thread is called default thread.

```
class Numbers:
    def display(self):
        for i in range(50):
            print("i val :",i)
        return

class Default:
    def main():
        print("Starts @ main")
        obj = Numbers()
        obj.display()
        for j in range(50):
            print("j val :",j)
        print("Ends @ main")
        return
```

```
Default.main()
```

Multi threaded application:

- Define a Custom thread and execute along with Default thread.
- Every Custom thread inherits the functionality from "Thread" class.
- "Thread" class is belongs to "threading" module.
- We must define thread logic by overriding run() method of Thread class.

```
from threading import Thread
class Custom(Thread):
    def run(self):
        for i in range(50):
            print("Custom :" + str(i))
        return

class Default:
```

```
def main():
    obj = Custom()
    obj.start()
    for j in range(50):
        print("Default :" + str(j))
    return
```

```
Default.main()
```

You can also create a thread using the `threading.Thread()` constructor and passing in a target function as an argument.

```
# Example:
import threading

def my_function():
    # Code to be executed in the new thread

my_thread = threading.Thread(target=my_function)
my_thread.start()
```

You can pass arguments to the target function using the `args` parameter.

```
# Example:
import threading

def my_function(arg1, arg2):
    # Code to be executed in the new thread

my_thread = threading.Thread(target=my_function, args=(arg1, arg2))
my_thread.start()
```

`sleep()`:

- A pre-defined method belongs to "time" module.
- `Sleep()` method is used to stop the current thread execution for specified number of seconds.
- The following example explains clearly about `sleep()` method.

```
from threading import Thread
import time

class Custom(Thread):
    def run(self):
        for i in range(1,11):
```

```

        print("custom : " + str(i))
        time.sleep(1)
    return

class Default:
    def main():
        obj = Custom()
        obj.start()
        for i in range(1,11):
            print("default : " + str(i))
            time.sleep(1)
        return

Default.main()

```

- When multiple threads are execution, any thread can complete the execution first.
- The following example explains that Default thread execution may complete after Custom thread execution.
- The thread execution completely depends on the logic we defined in that thread.

```

from threading import Thread
import time
class Custom(Thread):
    def run(self):
        for i in range(1,11):
            print("custom : " + str(i))
            time.sleep(1)
        print("Custom thread execution completed")
        return

class Default:
    def main():
        print("Starts at Default thread")
        obj = Custom()
        obj.start()
        for i in range(1,11):
            print("default : " + str(i))
            time.sleep(0.3)
        print("Default thread execution completed")
        return

Default.main()

```

Can we call run() method directly?

- run() method is pre-defined in Thread class with empty definition.
- Run() method override by programmer with custom thread logic.
- Start() method allocates separate thread space to execute run() logic parallel.
- If we call run() method directly, it will execute from the same thread space sequentially.

```
from threading import Thread
import time

class Custom(Thread):
    def run(self):
        for i in range(1,11):
            print("custom : " + str(i))
            time.sleep(1)
        return

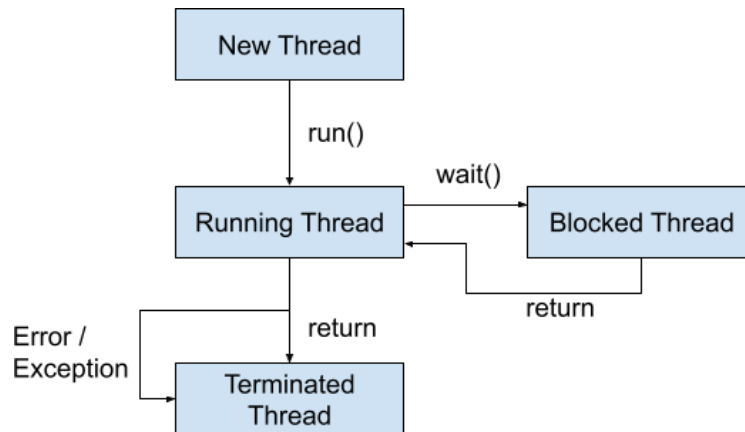
class Default:
    def main():
        obj = Custom()
        obj.run()
        for i in range(1,11):
            print("default : " + str(i))
            time.sleep(1)
        return

Default.main()
```

Life cycle of Thread:

- Every thread executes according to Life cycle.
- Life cycle consists
 - Creation phase by programmer
 - Execution phase by scheduler(OS)
- As a programmer, we need to define and start the thread.
- All threads scheduled by the processor.
- Thread has many states
 - Runnable state
 - Running state
 - Waiting state
 - Dead state.
- Every thread should wait in the Queue until processor allocates the memory is called Runnable state.
- Once the memory has been allocated to thread, it moves to running state. In running state, thread execute independently.

- From the running state, the thread may fall into waiting state to read user input, sleep and join methods.
- Once the thread execution completed, it moves to dead state.



How to execute different logics from different threads?

- We can define run() method only once in a Thread class.
- To execute different logics, different run() methods need to override.
- It is possible by defining more than one thread class and override run() method in each class with different logic.

```

from threading import Thread
import time

class Custom1(Thread):
    def run(self):
        for i in range(1,11):
            print("custom1 :" + str(i))
            time.sleep(1)
        return

class Custom2(Thread):
    def run(self):
        for i in range(50,71):
            print("custom2 :" + str(i))
            time.sleep(1)
        return

class Custom3(Thread):
    def run(self):

```

```
    for i in range(100,90,-1):
        print("custom3 :"+str(i))
        time.sleep(3)
    return
```

```
class Default:
    def main():
        c1 = Custom1()
        c1.start()
        c2 = Custom2()
        c2.start()
        c3 = Custom3()
        c3.start()
        return
```

```
Default.main()
```

join() method:

- It is a dynamic method belongs the Thread class.
- It is used to stop current thread execution until joined thread moved to dead state.

```
from threading import Thread
import time
```

```
class Custom(Thread):
    def run(self):
        for i in range(1,11):
            print("custom : " + str(i))
            time.sleep(1)
        return
```

```
class Default:
    def main():
        print("Starts @ Default")
        c = Custom()
        c.start()
        c.join()
        print("Ends @ Default")
        return
```

```
Default.main()
```

Thread synchronization:

- The concept of allowing threads sequentially when these threads trying to access same resource parallel.
- "threading" module is providing "Lock" class to implement thread synchronization.
- Lock class is providing dynamic methods to lock specific logic in the function.
 - obj = Lock()
 - obj.acquire()
 - obj.release()

```
import time
from threading import Thread, Lock
class Numbers:
    x=0
    lock=Lock()
    def incrementX():
        Numbers.lock.acquire()
        Numbers.x = Numbers.x+1
        Numbers.lock.release()
        return

class Custom1(Thread):
    def run(self):
        for i in range(100000):
            Numbers.incrementX()
        return

class Custom2(Thread):
    def run(self):
        for i in range(100000):
            Numbers.incrementX()
        return

class Default:
    def main():
        t1 = Custom1()
        t2 = Custom2()
        t1.start()
        t2.start()
        t1.join()
        t2.join()
        print("Final x val : "+str(Numbers.x))
        return
Default.main()
```

Strings

String:

- String is a sequence of characters.
- We can specify strings using single, double or triple quotes.

Accessing strings: We can access the strings using indexing and slicing

Indexing:

- String array elements index starts with 0
- Using indexing, we can process each character.
- Python supports negative indexing from the end.
- Negative indexing starts with -1 from the end.

s = "python"					
0	1	2	3	4	5
p	y	t	h	o	n
-6	-5	-4	-3	-2	-1

```
s = "python"
print("s[2] :", s[2])
print("s[-3] :", s[-3])

print("Length :", len(s))
print("s[len(s)-1] :", s[len(s)-1])
print("s[-len(s)] :", s[-len(s)])
```

Slicing: Accessing more than one character from the string.

Syntax : [lower : upper : step]

- Specifying the above values are optional
- Lower value by default 0
- Upper value by default len(string)
- Step value is 1 by default.

```
s = "python"
print("String is :",s)
print("s[:] :", s[:])
```

```
print("s[: :] :", s[: :])
print("s[0:6:1] :", s[0:6:1])
print("s[0:len(s)] :", s[0:len(s)])
```

```
s = "python"
print("String is :",s)
print("s[2:5] :", s[2:5])
print("s[-5:-2] :", s[-5:-2])
print("s[2:-2] :", s[2:-2])
```

```
s = "python"
print("String is :",s)
print("s[5:2:-1] :", s[5:2:-1])
print("s[-1:-5:-1] :", s[-1:-5:-1])
```

Boolean methods - String validation methods:

- String is an Object
- String object is providing pre-defined methods to validate the string.
- These methods are called Boolean methods, these methods always return a Boolean value (True or False)
- Some of the methods :

```
s1 = "abc"
print(s1,"is alpha :", s1.isalpha())
print(s1,"is alpha numeric :", s1.isalnum())
print(s1,"is digit :", s1.isdigit())
```

```
s1 = "1234"
print(s1,"is alpha :", s1.isalpha())
print(s1,"is alpha numeric :", s1.isalnum())
print(s1,"is digit :", s1.isdigit())
```

```
s1 = "abc123"
print(s1,"is alpha :", s1.isalpha())
print(s1,"is alpha numeric :", s1.isalnum())
print(s1,"is digit :", s1.isdigit())
```

```
s1 = "abc123$#"
print(s1,"is alpha numeric :", s1.isalnum())
print(s1,"is lower :", s1.islower())
```

Displaying Strings with Quotes: Generally a string can be represented in python using single or double quotes.

#Output : It is 'Python' online session

```
line = "It is 'Python' online session"
print(line)
```

#Output : It is "Python" tutorial

```
line = 'It is "Python" tutorial'
print(line)
```

Output : She said, "It's a good one"

```
# line = "She said, "It's a good one"" -> Error :
# line = 'She said, "It's a good one"' -> Error :
line = ""She said, "It's a good one""
print(line)
```

#Escape characters

```
# \n = new line
# \t = tab space
# \\ = \
# \' = '
# \" = "
```

Output : This is "Python" session

```
line = "This is \"Python\" session"
print(line)
```

Output : Python's tutorial

```
line = "Python's tutorial"
print(line)
```

Output : She said, "It's a good one"

```
line = 'She said, "It\'s a good one"'
print(line)
```

Output : She said, "It's a good one"

```
line = "She said, \"It's a good one\""  
print(line)
```

Python Comments:

1. Single line comments
2. Multi line comments - Not supported

Single line comment:

- We use comments to describe the instruction of a program.
- We use # symbol preceded by description

Single line comment

Multi line comment:

- Python doesn't support multi line comments.
- Generally they call triple quotes as multi line comment but not correct.
- For example,

```
'''This  
is  
Multi line  
comment'''
```

- The above declaration is a string and memory.
- If we collect that String value into any variable, we can display the content of that String.
- **For example:**

```
line = '''  
This is  
Multi line  
Comment  
in Python'''  
print(line)
```

Comments do not get memory in any programming language as follows.

```
line = #single line comment  
print(line)
```

Old Style formatting:

- We can display string after formatting
- Python supports C style of formatting using access specifiers.
- Format specifiers like int(%d), float(%f), String(%s) can be used.
- We can also concatenate strings using plus(+) operator.

String representation:

```
name = input("Enter your name : ")
print("Hello" , name)
print("Hello " + name)
print("Hello %s" %name)
```

String and Integer concatenation:

```
name = input("Enter your name : ")
age = int(input("Enter your age : "))
print(name , "age is : " , age)
print(name + " age is : " + str(age))
print("%s age is : %d" %(name,age))
```

Code:

```
print("Enter Emp details : ")
no = int(input())
name = input()
salary = float(input())

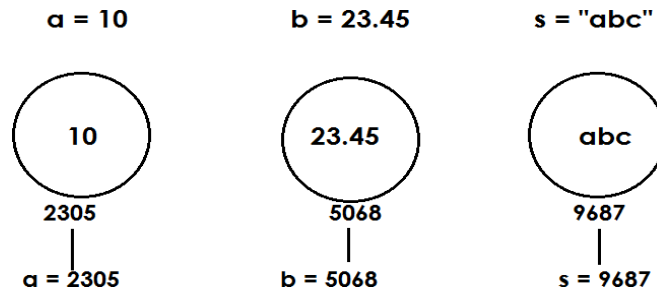
print(name , "with id : " , no , "is taking salary : " , salary)
print(name + " with id : " + str(no) + " is taking salary : " + str(salary))
print("%s with id : %d is taking salary : %f" %(name, no, salary))
```

Float representation:

```
val = 34.567
print("Value is :", val)
print("Value is : %f" %val)
print("Value is : %.3f" %val)
print("Value is : %.2f" %val)
```

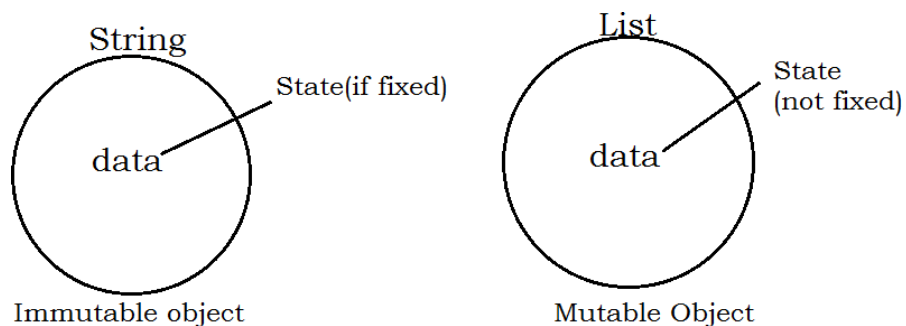
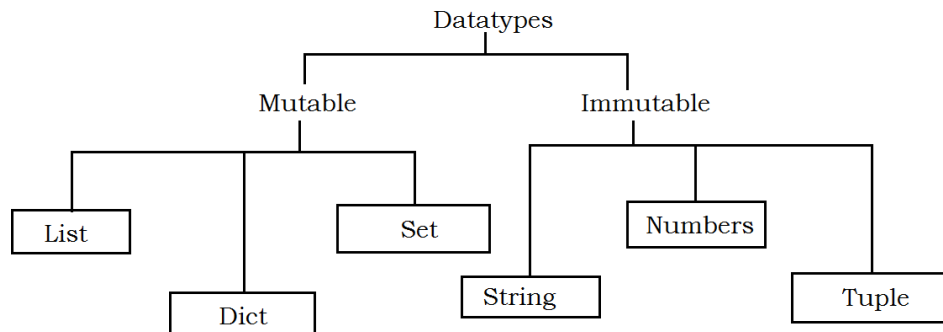
Immutability:

- Python is Object oriented programming language.
- Any information(Number, String or Collection) stores in Object format only.



- Object behavior is either Mutable or Immutable.
- **Immutable objects** : Numbers , Strings and Tuples
- **Mutable Objects** : Lists, Sets and Dictionaries

Data types divided into Mutable and Immutable objects as follows:



- Immutable objects nothing but constants. We cannot modify the contents of Object once created.

Numbers:

- Integers and Float values comes under Number type Objects.
- Once we assign any number to a variable that cannot be modified.
- If we try to modify the value, a new object will be creating with modified content.

Code:

```
a=10
```

```
print("a val :", a)
```

```
print("Loc :", id(a))
```

```
a=a+5
```

```
print("Modified a val :", a)
```

```
print("Loc :", id(a))
```

```
a=10
```

```
print("a val :", a)
```

```
print("Loc :", id(a))
```

```
a=a+5
```

```
print("Modified a val :", a)
```

```
print("Loc :", id(a))
```

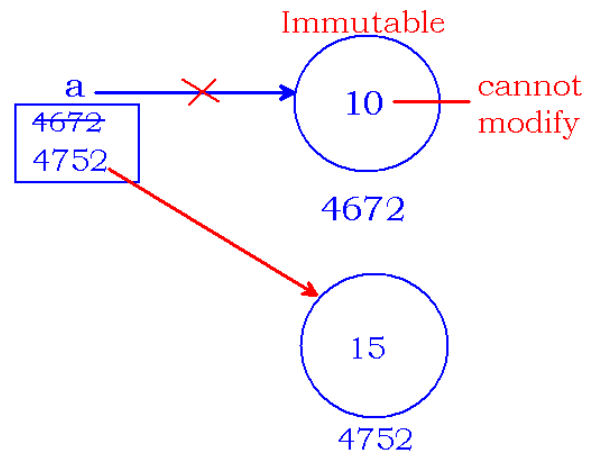
Output:

```
a val : 10
```

```
Loc : 1631904672
```

```
Modified a val : 15
```

```
Loc : 1631904752
```



String Immutability:

```
s1="Hello"
```

```
s2="World"
```

```
print("s1 val :",s1)
```

```
print("s1 loc :",id(s1))
```

```
print("s2 val :",s2)
```

```
print("s2 loc :",id(s2))
```

```
s1=s1+s2
```

```
print("Modified s1 val :",s1)
```

```
print("s1 loc :",id(s1))
```

```
print("s2 val :",s2)
```

```
print("s2 loc :",id(s2))
```

Output:

s1 val : Hello

s1 loc : 59401472

s2 val : World

s2 loc : 59401376

Modified s1 val : HelloWorld

s1 loc : 59410896

s2 val : World

s2 loc : 59401376

```
s1="Hello"
s2="World"
print("s1 val :",s1)
print("s1 loc :",id(s1))
print("s2 val :",s2)
print("s2 loc :",id(s2))
s1=s1+s2
print("Modified s1 val :",s1)
print("s1 loc :",id(s1))
print("s2 val :",s2)
print("s2 loc :",id(s2))
```

Output:

s1 val : Hello

s1 loc : 59401472

s2 val : World

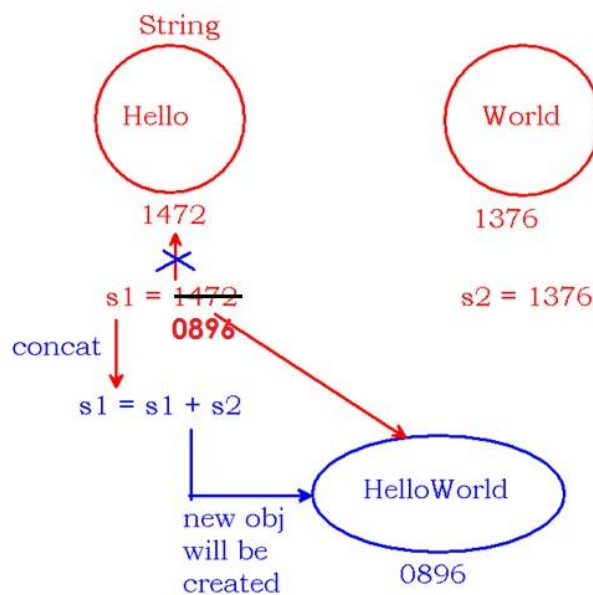
s2 loc : 59401376

Modified s1 val : HelloWorld

s1 loc : 59410896

s2 val : World

s2 loc : 59401376



When we try to create multiple String objects with the same content, duplicate objects will not be created. The address of object will be shared.

```
s1="Hello"
s2="Hello"
print("s1 val :",s1)
print("s1 loc :",id(s1))
print("s2 val :",s2)
print("s2 loc :",id(s2))
```

```

s1="Hello"
s2="Hello"
print("s1 val :",s1)
print("s1 loc :",id(s1))

print("s2 val :",s2)
print("s2 loc :",id(s2))

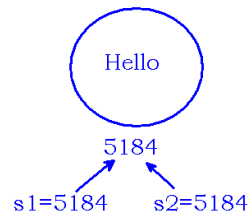
```

Output:

```

s1 val : Hello
s1 loc : 58025184
s2 val : Hello
s2 loc : 58025184

```



Does s2 effects when we modify s1 content in the above code?

- No, String object is Immutable. When the contents are same then only locations are same. When we modify the content, a new object will be created in another location.

```

s1="Hello"
s2="Hello"
print("s1 val :",s1)
print("s1 loc :",id(s1))
print("s2 val :",s2)
print("s2 loc :",id(s2))

s1 = s1+'$'
print("s1 val :",s1)
print("s1 loc :",id(s1))
print("s2 val :",s2)
print("s2 loc :",id(s2))

```

```

s1="Hello"
s2="Hello"
print("s1 val :",s1)
print("s1 loc :",id(s1))
print("s2 val :",s2)
print("s2 loc :",id(s2))

```

```

s1 = s1+'$'
print("s1 val :",s1)
print("s1 loc :",id(s1))
print("s2 val :",s2)
print("s2 loc :",id(s2))

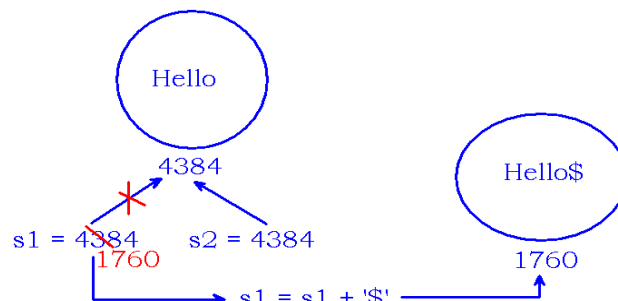
```

Output:

```

s1 val : Hello
s1 loc : 57304384
s2 val : Hello
s2 loc : 57304384
s1 val : Hello$
s1 loc : 57221760
s2 val : Hello
s2 loc : 57304384

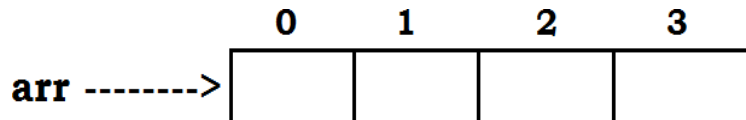
```



Collection data types

Introduction:

- We use programming languages to develop applications.
- Applications are used to store and process the information.
- In the application, when we use set of elements, it is recommended to use collection type.
- If we can structure the data while storing, can access more effectively while processing.
- Basic data structure that store elements in consecutive locations is called array.



- Different algorithms were proposed to structure the information in application called Data structures and Algorithms.
- In some of the languages like C or C++, we need to implement these algorithms manually to structure the information in application.
- Python collections are pre defined implementations of these algorithms to structure the data.

Note: The complete functionality of all data structures and algorithms given as 4 simple collection objects.

1. List
2. Tuple
3. Set
4. Dictionary

Array v/s List:

- Python is not allowed the programmer to create traditional arrays for data processing.
- Python doesn't support Arrays.
- Array is static (Size is fixed) where as Python is dynamic. This is the one reason that why python is not allowed arrays.
- In any programming language only arrays can process one dimensional, two dimensional and multi dimensional data easily using indexing.

As Python doesn't support Arrays, we use third party modules (Numpy & Pandas) to process elements like arrays.

List Collection

List:

- List is a linear data structure.
- List is an Object contains multiple data items (elements).
- Square brackets are used to enclose the elements of List.
- The elements in List are separated with comma (,) operator.
- List can accept heterogeneous data elements.
- List is allowed to store duplicate elements.

Code:

```
l1 = [10,20,30,40,50]
print("L1 list :",l1)

l2 = [10,20,10,"abc",34.56]
print("L2 list :",l2)
```

Accessing Elements: We can access elements of List in 2 ways such as **Indexing & Slicing**.

Indexing:

- Python index starts from 0 to size-1.
- We can access only one element through indexing.
- Python supports negative indexing also
- Some of the examples as follow.

l = [10,20,30,40,50]

0	1	2	3	4
10	20	30	40	50
-5	-4	-3	-2	-1

Code:

```
l = [10,20,30,40,50]
print("List is :", l)
print("Length :", len(l))
print("l[2] :", l[2])
print("l[-2] :", l[-2])
print("l[len(l)-2] :", l[len(l)-2])
print("l[-(len(l)-3)] :", l[-(len(l)-3)])
```

Output:

```
List is : [10, 20, 30, 40, 50]
Length : 5
l[2] : 30
l[-2] : 40
l[len(l)-2] : 40
l[-(len(l)-3)] : 40
```

Slicing:

- To access more than one element at a time from the collection.
- Slicing representation is as same as range() object

Syntax:

[start-index : stop-index : step]

- Start index default value is 0
- Stop index default value is length-1
- Step value modified by 1
- When we don't specify the values of start, stop and step, it will consider default values and process all elements of List.
- Start index is includes where as stop index value excludes.

Code:

```
l = [10,20,30,40,50]
print("List is :", l)
print("Length :", len(l))
print("l[:]:", l[:])
print("l[:]:", l[:])
print("l[0:len(l):1]:", l[0:len(l):1])
print("l[0:5:2]:", l[0:5:2])
print("l[-5:-2]:", l[-5:-2])
```

Output:

```
List is : [10, 20, 30, 40, 50]
Length : 5
l[:] : [10, 20, 30, 40, 50]
l[:]: [10, 20, 30, 40, 50]
l[0:len(l):1] : [10, 20, 30, 40, 50]
l[0:5:2] : [10, 30, 50]
l[-5:-2] : [10, 20, 30]
```

List methods:

- List is a pre-defined Object (class).
- List class is providing set of pre-defined methods(dynamic), we access using list object address.
- Using list functionality, we can process the list elements easily.
- The functionality is used to append, insert, remove, pop, sort, reverse and update the list as follows.

append(): Function that adds an element at the end of the List and returns the complete list with appended element.

```
l = [10,20,30]
print("List :", l)

l.append(40)
print("After append 40 :", l)
```

count(): Function that returns the specified element count in the List

```
l = [10, 20, 10, 40, 10]
print("List :", l)
print("Count of 10 :", l.count(10))
print("Count of 40 :", l.count(40))
print("Count of 50 :", l.count(50))
```

index():

- Function that returns index value of specified element.
- List supports duplicate elements.
- If the specified element is duplicated, it return first occurrence of index.
- Raises an exception if the specified element is not present in the list.

```
l = [10, 20, 10, 40, 20, 50]
print("List :", l)
print("index of 40 :", l.index(40))
print("index of 20 :", l.index(20))
print("index of 70 :", l.index(70))
```

Output:

```
List : [10, 20, 10, 40, 20, 50]
index of 40 : 3
index of 20 : 1
ValueError: 70 is not in list
```

Insert():

- Function that is used to insert element at specified index.
- After insertion, other elements in the list will shift to right.
- Element will be appended if the specified index is not present in that List.

```
l = [10,20,10,40]
print("List :", l)

print("insert 50 @ index : 2 :")
l.insert(2,50)
print("List :", l)

print("insert 70 @ index : -4 :")
l.insert(-4,70)
print("List :", l)

print("insert 90 @ index : 12 :") # append if out of bounds
l.insert(12,90)
print("List :", l)
```

Output:

```
List : [10, 20, 10, 40]
insert 50 @ index : 2 :
List : [10, 20, 50, 10, 40]
insert 70 @ index : -4 :
List : [10, 70, 20, 50, 10, 40]
insert 90 @ index : 12 :
List : [10, 70, 20, 50, 10, 40, 90]
```

Deleting elements:

- We can remove the elements of List in many ways.
- List object is providing different methods to remove elements.

pop([index]) :

- Removes the last element from the List when we don't specify the index.
- Index argument is optional
- We can remove the element by specifying its index also
- Raises Exception if the specified index is not present.

Code:

```
l = [10, 20, 30, 40, 50]
print("List is :", l)
```

```
print("pop() :", l.pop())
print("List is :", l)

print("pop(1) :", l.pop(1))
print("List is :", l)

print("pop(4) :", l.pop(4))
print("List is :", l)
```

Output:

```
List is : [10, 20, 30, 40, 50]
pop() : 50
List is : [10, 20, 30, 40]
pop(1) : 20
List is : [10, 30, 40]
IndexError: pop index out of range
```

remove(element):

- Removes specified element from the List.
- List allows duplicates.
- Remove the first occurrence of element if the element is duplicated.
- Raises Exception if the specified element is not present in the list.

```
l = [10, 20, 30, 40, 50, 20]
print("List is :", l)
print("remove(30) :")
l.remove(30)
print("List is :", l)
print("remove(20) :")
l.remove(20)
print("List is :", l)
print("remove(70) :")
l.remove(70)
print("List is :", l)
```

Output:

```
List is : [10, 20, 30, 40, 50, 20]
remove(30) :
List is : [10, 20, 40, 50, 20]
remove(20) :
List is : [10, 40, 50, 20]
remove(70) : ValueError: list.remove(x): x not in list
```

clear():

- Removes all elements of List.
- Return an empty List.
- The memory allocated to list will not be de-allocated.

Code:

```
l = [10,20,30,40,50]
print("List is :", l)

print("clear list :")
l.clear()
print("List is :", l)

print("append(20) :")
l.append(20)
print("List is :", l)
```

Output:

```
List is : [10, 20, 30, 40, 50]
clear list :
List is : []
append(20) :
List is : [20]
```

del:

- It is a keyword.
- It is used to release the memory of Object(here list variable)

Code:

```
l = [10,20,30,40,50]
print("List is :", l)

print("del list :")
del l
print("List is :", l)
```

Output:

```
List is : [10, 20, 30, 40, 50]
del list :
NameError: name 'l' is not defined
```

extend():

- It is used to merge lists.
- One list will be appended to another list
- List1 will be modified where as List2 remains same.

Code:

```
l1 = [10, 20, 30]
print("L1 is :", l1)

l2 = [23.45, 56.78]
print("L2 is :", l2)

print("l1 extends l2 :")
l1.extend(l2)
print("L1 after extend :", l1)
```

copy():

- A function that is used to copy all elements of List to another list.
- Returns a duplicate copy of specified list.
- Copied list get different memory location.
- List is Mutable (we will discuss later). Hence the location must be different for duplicate Lists.

Code:

```
src = [10,20,30]
print("Source list is :", src)
print("Source list location :", id(src))

cpy = src.copy()
print("Copied list is :", cpy)
print("Copied list location :", id(cpy))
```

Output:

```
Source list is : [10, 20, 30]
Source list location : 58932360
Copied list is : [10, 20, 30]
Copied list location : 14314432
```

Note: Only Lists get different memory locations with same set of elements. The duplicates in both the Lists have same memory locations as follows.

Code:

```
src = [10,20,30]
print("Source list location :", id(src))

cpy = src.copy()
print("Copied list location :", id(cpy))

print("src[1] ele :", src[1])
print("src[1] loc :", id(src[1]))

print("cpy[1] ele :", cpy[1])
print("cpy[1] loc :", id(cpy[1]))
```

Output:

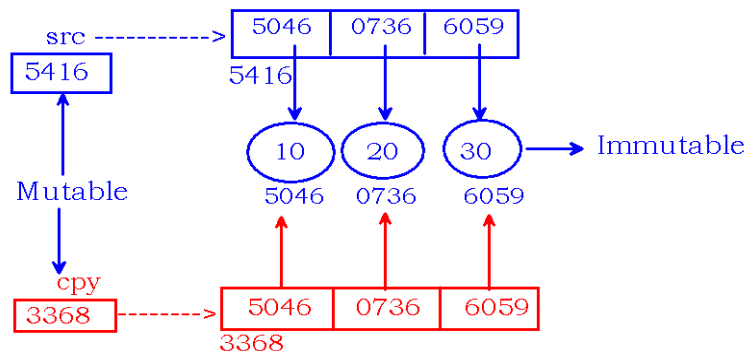
```
Source list location : 60595416
Copied list location : 60663368
src[1] ele : 20
src[1] loc : 1634460736
cpy[1] ele : 20
cpy[1] loc : 1634460736
```

List is Mutable :

Hence, when we copy the list,
an object creates at different
location.

Number is Immutable:

Hence, the content of both
lists having same numbers.
So that locations remains same.

**List Operations using operators:**

- Lists can be concatenated using + operator. List repetition is possible using * operator.
- "in" operator is used to test whether the element is present or not in the list.
- "in" operator can be used to iterate the list using 'for' loop.
- "not in" operator can be used to check whether the element is not present or not in the List.

Code:

```
print("List operations using operators :")
a1 = [1,2,3]
a2 = [4,5,6]
print("a1 list :", a1)
print("a2 list :", a2)

print("a1=a1+a2 :")
```

```
a1=a1+a2
print("a1 list :", a1)

print("a2 list :", a2)
print("a2=a2*3 :")
a2 = a2*3
print("a2 list :", a2)

print("a1 list :", a1)
print("5 is present :", 5 in a1)
print("7 is present :", 7 in a1)

print("20 is not present :", 20 not in a1)
print("6 is not present :", 6 not in a1)
```

Output:

```
List operations using operators :
a1 list : [1, 2, 3]
a2 list : [4, 5, 6]
a1=a1+a2 :
a1 list : [1, 2, 3, 4, 5, 6]
a2 list : [4, 5, 6]
a2=a2*3 :
a2 list : [4, 5, 6, 4, 5, 6, 4, 5, 6]
a1 list : [1, 2, 3, 4, 5, 6]
5 is present : True
7 is present : False
20 is not present : True
6 is not present : False
```

sort():

- sort all elements in the specified list
- All elements in the list should be of same type
- Heterogeneous elements list cannot sort.

Code:

```
a1 = [10, 40, 20, 50, 30]
print("a1 List is :",a1)
a1.sort()
print("Sorted a1 list is :", a1)

a2 = [34.56, 23, "abc"]
```

```
print("a2 List is :",a2)
a2.sort() # sorting possible only on homogenous data elements
print("Sorted a2 list is :", a2)
```

Output:

```
a1 List is : [10, 40, 20, 50, 30]
Sorted a1 list is : [10, 20, 30, 40, 50]
a2 List is : [34.56, 23, 'abc']
TypeError: '<' not supported between instances of 'str' and 'int'
```

reverse(): Reverse the elements in specified list.

```
a1 = [10, 40, 20, 50, 30]
print("a1 List is :",a1)
a1.reverse()
print("Reversed a1 list is :", a1)
a2 = [34.56, 23, "abc"]
print("a2 List is :",a2)
a2.reverse()
print("Reversed a2 list is :", a2)
```

Built in Functions can be used with List as follows:

all() : return True if all elements of the List are true (or if the list is empty)

```
>>> a = [1, 2, 0]
>>> all(a)
False
>>> a = [1, 2, 3]
>>> all(a)
True
>>> a = []
>>> all(a)
True
```

any() : Return True if any one of List element is True. Returns False if the List is empty.

```
>>> arr = [1,2,0]
>>> any(arr)
True
>>> arr = [1,2,3]
>>> any(arr)
True
>>> arr = []
>>> any(arr)
False
```

list():

- List is a class.
- list() is a constructor
- list() is used to construct any empty list object.

```
arr = list()
print("List is :", arr)
print("Length :", len(arr))
print("Any :", any(arr))
```

Output:

```
List is : []
Length : 0
Any : False
```

We can append elements to empty list by reading at runtime.

```
arr = list()
print("Initially :", arr)
print("Enter 5 elements :")
for i in range(5):
    ele = input()
    arr.append(ele)
print("Later :", arr)
```

Output:

```
Initially : []
Enter 5 elements :
10
23.42
g
abc
10
Later : ['10', '23.42', 'g', 'abc', '10']
```

Sum of list elements:

```
print("Sum of list elements")
arr = list()
n = int(input("Enter size : "))
print("Enter %d elements : " %n)
for i in range(n):
    ele = int(input())
    arr.append(ele)
```

```
#logic
res=0
for ele in arr:
    res=res+ele
print("Sum is :", res)
```

Output:

```
Sum of list elements
Enter size : 5
Enter 5 elements :
6
4
2
8
3
Sum is : 23
```

max() : Returns the largest element from the List

min() : Returns minimum element from List

sorted() : Returns a new sorted list of elements (doesn't sort the list itself)

sum() : Returns the sum of all elements in the List

```
>>> List = [6, 3, 8, 2, 9]
>>> min(List)
2
>>> max(List)
9
>>> sum(List)
28
>>> sorted(List)
[2, 3, 6, 8, 9]
>>> print(List)
[6, 3, 8, 2, 9]
```

Note: We can process elements of collection using Iterators.

Using for loop:

```
arr = [10,20,30,40,50]
```

```
print("Elements are :")
```

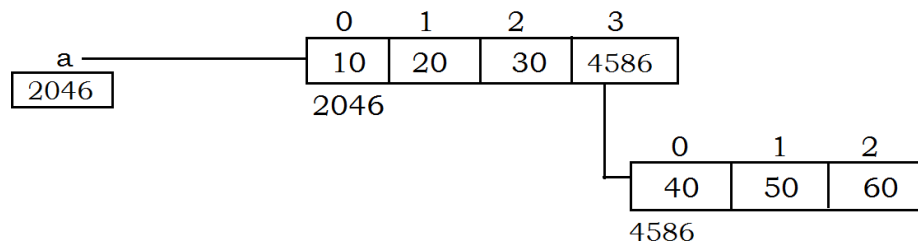
```
for ele in arr :
```

```
    print(ele)
```

Nested Lists:

- Defining a List as an element inside another list.
- List is an object; hence we connect the nested list by storing the address.

```
a = [10,20,30,[40,50,60]]
```



```
a = [10,20,30,[40,50,60]]
print("List is :", a)

# access using indexing
print("a[1] :",a[1])
print("a[3] :",a[3])

# negative indexing
print("a[-2] :", a[-2])
print("a[-1] :", a[-1])

# using len()
print("len(a) :", len(a))
print("a[len(a)-1] :", a[len(a)-1])
```

```
a = [10,20,30,[40,50,60]]
print("List is :", a)

# access sub list
print("a[3][0] :", a[3][0])
print("a[-1][0] :", a[-1][0])
print("a[3][-3] :", a[3][-3])
print("a[-1][-3] :", a[-1][-3])

# using len()
print("len(a[-1]) :", len(a[-1]))
print("a[len(a)-1] :", a[len(a)-1])
print("a[len(a)-1][len(a[-1])-1] :",a[len(a)-1][len(a[-1])-1])
```

- Using index, we can access only one element at a time.
- Through slicing, we can access multiple elements either from list or sub list.

```
a = [10,20,30,[40,50,60]]
```

```
print("List is :", a)
```

```
# list slicing
```

```
print("a[:]:", a[:])
```

```
print("a[:]:", a[:])
```

```
print("a[0:len(a):1]:", a[0:len(a):1])
```

```
print("a[2:len(a)]:", a[2:len(a)])
```

```
a = [10,20,30,[40,50,60]]
```

```
print("List is :", a)
```

```
# sub list slicing
```

```
print("a[-1][:]:", a[-1][:])
```

```
print("a[-1][1:3]:", a[-1][1:3])
```

```
print("a[len(a)-1][0:len(a[-1])]:", a[len(a)-1][0:len(a[-1])])
```

- Every element in python is an Object.
- An address will store into the location instead of value.

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

```
print(id(a))
```

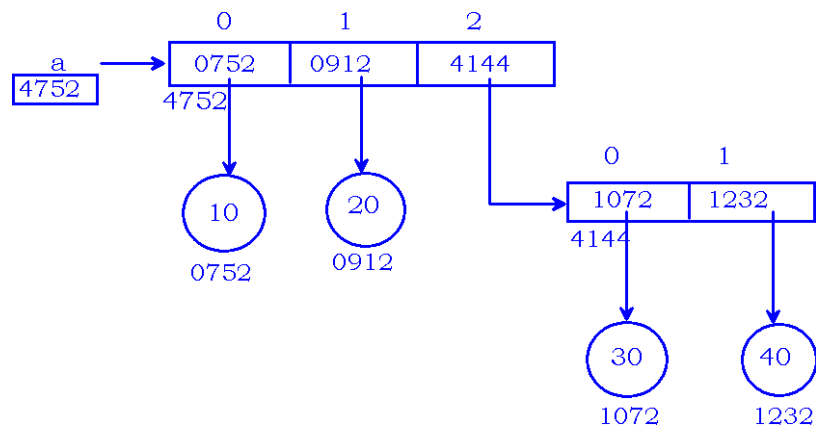
```
print(id(a[0]))
```

```
print(id(a[1]))
```

```
print(id(a[2]))
```

```
print(id(a[2][0]))
```

```
print(id(a[2][1]))
```

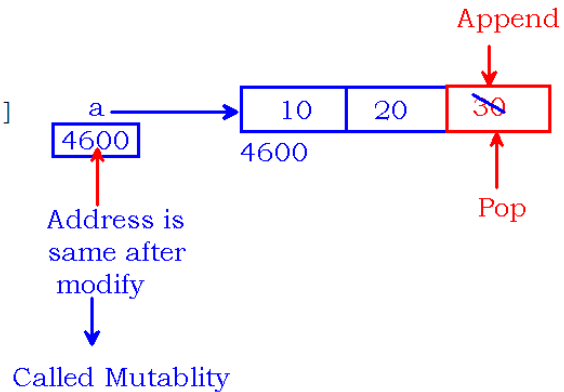


List Mutability:

- List is Mutable
- Mutable object can be modified.
- Address remains same after modification of Mutable object

```
a = [10, 20]
print("List is :",a)
print("Address :",id(a))
a.append(30)
print("After append, List is :",a)
print("Address :",id(a))
a.pop()
print("After pop, List is :",a)
print("Address :",id(a))
```

```
Output:
List is : [10, 20]
Address : 61554600
After append, List is : [10, 20, 30]
Address : 61554600
After pop, List is : [10, 20]
Address : 61554600
```



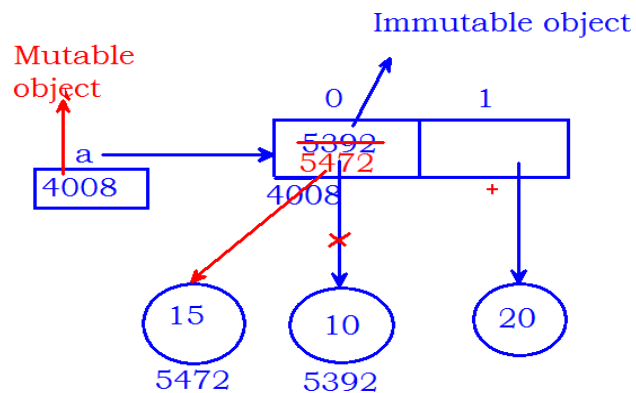
- In Python, Strings, Integers, Floats all elements are Immutable objects.
- Immutable object get another location when we modify.

```
a = [10,20]
print("List is :", a)
print("Address of a :",id(a))
print("a[0] :", a[0])
print("a[0] address :", id(a[0]))

a[0]=a[0]+5
print("List is :", a)
print("Address of a :",id(a))
print("a[0] :", a[0])
print("a[0] address :", id(a[0]))
```

Output:

```
List is : [10, 20]
Address of a : 54494008
a[0] : 10
a[0] address : 1647965392
List is : [15, 20]
Address of a : 54494008
a[0] : 15
a[0] address : 1647965472
```



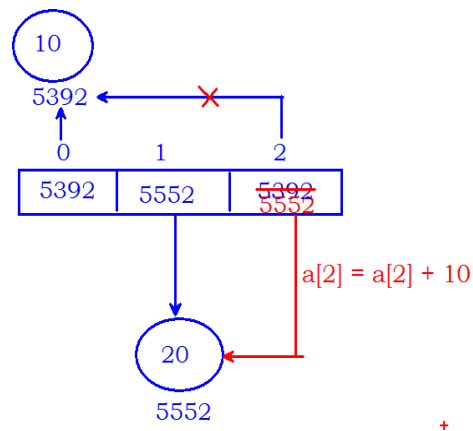
- Immutable objects cannot be duplicated.
- Contents with same values will share the memory.

```
a = [10, 20, 10]
print("List is :", a)
print("id(a[0]) :", id(a[0]))
print("id(a[1]) :", id(a[1]))
print("id(a[2]) :", id(a[2]))

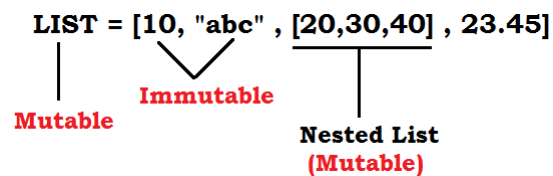
a[2]=a[2]+10
print("id(a[1]) :", a[2])
print("id(a[2]) :", id(a[2]))
```

Output:

```
List is : [10, 20, 10]
id(a[0]) : 1647965392
id(a[1]) : 1647965552
id(a[2]) : 1647965392
id(a[1]) : 20
id(a[2]) : 1647965552
```



- List is Mutable - We can modify the List
 - List elements either Mutable or Immutable
- LIST = [10, "abc" , [20,30,40] , 23.45]**



tuple:

- It is an ordered collection
- It allows duplicates
- It allows heterogeneous elements
- tuple can be represented using ()

```
t = (10, 20, 30, 40, 50)
print(t)
t = (10, 20, 10, "abc", 23.45)
print(t)
```

Output:

```
(10, 20, 30, 40, 50)
(10, 20, 10, 'abc', 23.45)
```

Tuple elements can be processed using indexing or slicing.

```
t = (10, 20, 30, 40, 50)
print("Tuple :",t)
print("t[2] :", t[2])
print("t[-4] :", t[-4])
```

