



# Python Programming

Karthikeyan Palani

karthikeyan@krosum.com

+91 9902084004

# PYTHON

| Session | Module |
|---|---|
| Day 1 | Lesson 1 : Introduction to Python<br>Lesson 2 : Python types & variables<br>Lesson 3 : operators<br>Lesson 4 : conditional & Loop statements |
| Day 2 | Lesson 5 : List ,tuple<br>Lesson 6 : Dictionary & set operations<br>Lesson 7 : File Handling |

# PYTHON

| Session | Module |
|---|---|
| Day 3 | Lesson 8 : Function call and arguments, scopes<br>Lesson 9 : About python module and pip/pip3 |
| Day 4 | Lesson 10 : Regx, Exception Handling<br>Lesson 11 : List Comprehension and lambda<br>Lesson 12 : map, filter, reduce |
| Day 5 | Lesson 13: Object Oriented Programming Concepts<br>Lesson 14: Python objects /methods<br>Lesson 15: Case studies |



# Lesson - 1

# Introduction about Python

- Python is an interpreted, high-level, general-purpose programming language.
- Created by **Guido van Rossum** and first released in 1991.

# Introduction about Python

- Python is an easy to learn, powerful programming language.
- It has efficient **high-level data structures** and a simple but effective approach to object-oriented programming.
- The **Python interpreter** and the extensive standard library are **freely available in source** or binary form for all major platforms from the Python Web site, **https://www.python.org/**, and may be freely distributed.




# Introduction about Python

- The Python interpreter is easily extended with new functions and data types implemented in C or C++ (or other languages callable from C).

- Python is also suitable as an extension language for customizable applications.




# Why python ?

- Easy to Learn and Use
- Interpreted Language
- Cross-platform Language
- Free and Open Source
- Object-Oriented Language
- Large Standard Library
- GUI Programming Support
- Dynamically  typed language



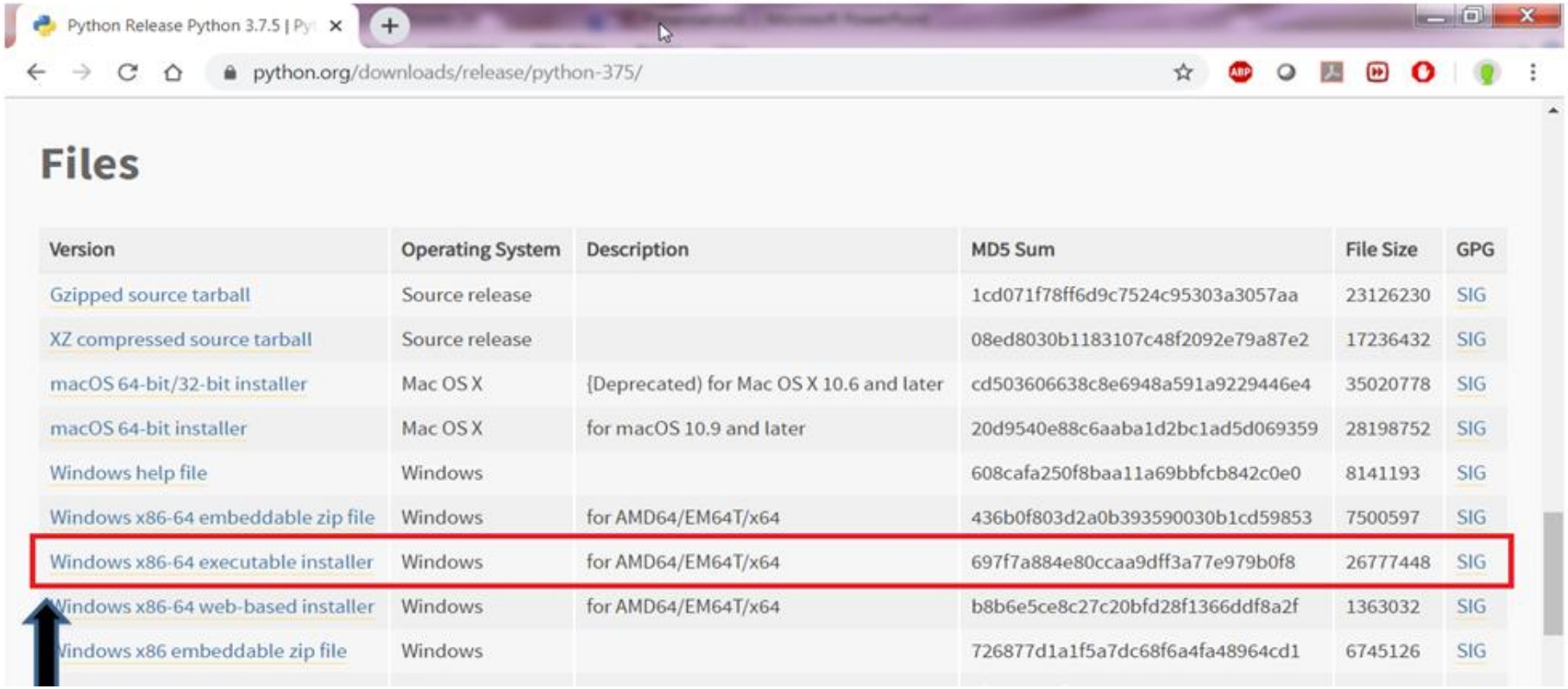

# How to install python ?

Download and install python

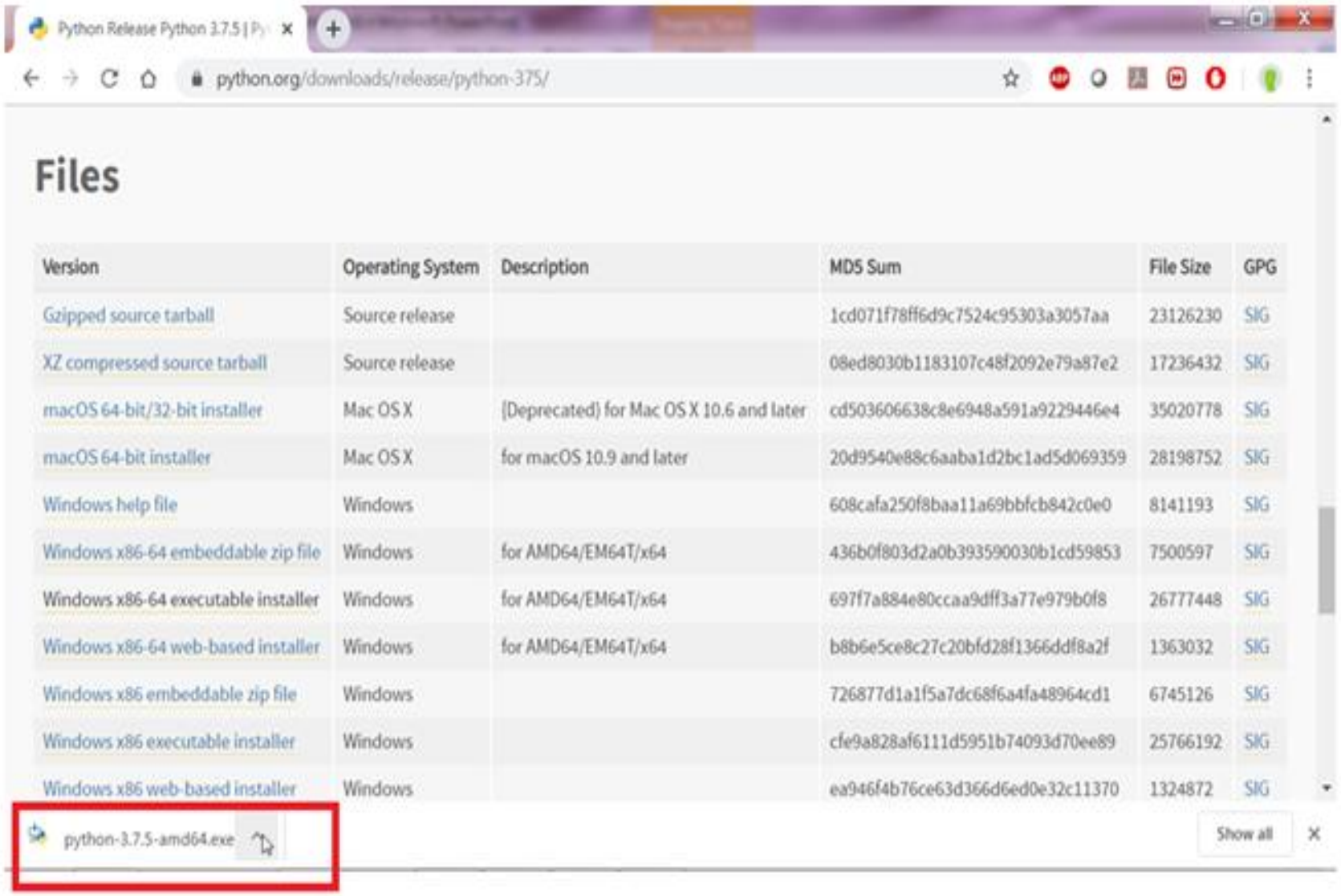
https://www.python.org/

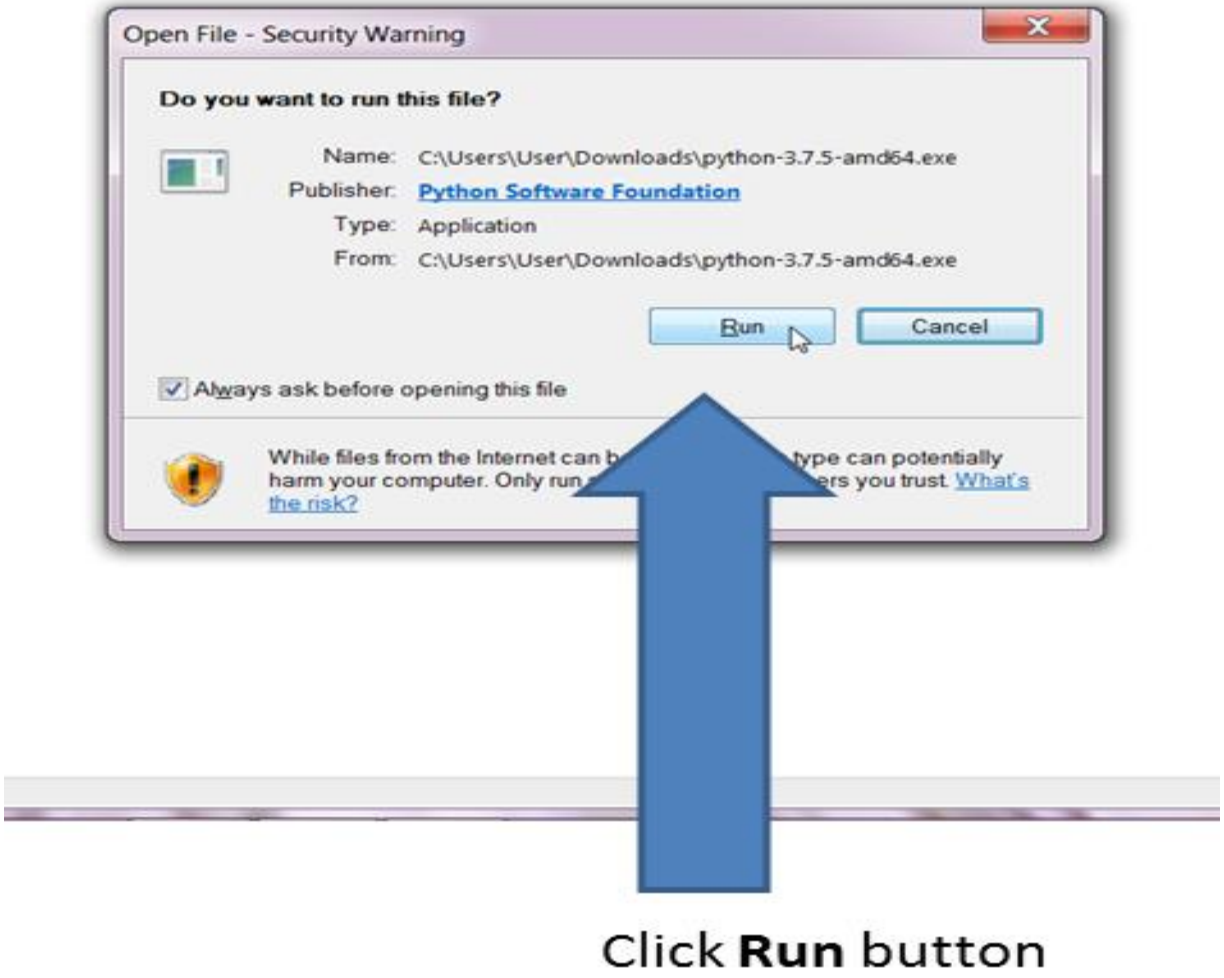
Click the URL

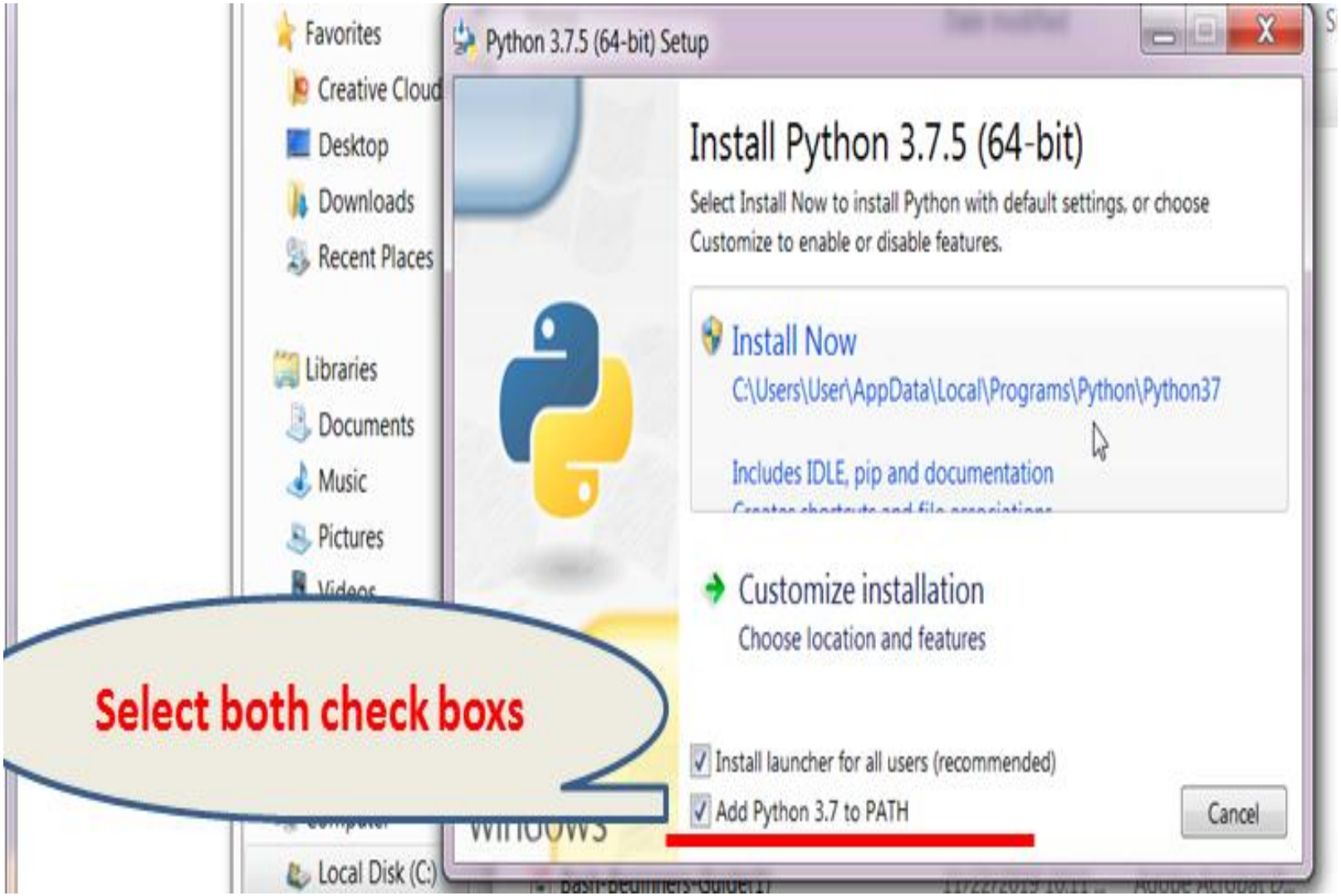
https://www.python.org/downloads/release/python-375/




**Click the above link**


Python Release Python 3.7.5 | Py
python.org/downloads/release/python-375/
Files
Version
Operating System
Description
MD5 Sum
File Size
GPG
Gzipped source tarball
Source release
1cd071f78ff6d9c7524c95303a3057aa
23126230
SIG
XZ compressed source tarball
Source release
08ed8030b1183107c48f2092e79a87e2
17236432
SIG
macOS 64-bit/32-bit installer
Mac OS X
(Deprecated) for Mac OS X 10.6 and later
cd503606638c8e6948a591a9229446e4
35020778
SIG
macOS 64-bit installer
Mac OS X
for macOS 10.9 and later
20d9540e88c6aaba1d2bc1ad5d069359
28198752
SIG
Windows help file
Windows
608cafa250f8baa11a69bbfcb842c0e0
8141193
SIG
Windows x86-64 embeddable zip file
Windows
for AMD64/EM64T/x64
436b0f803d2a0b393590030b1cd59853
7500597
SIG
Windows x86-64 executable installer
Windows
for AMD64/EM64T/x64
697f7a884e80ccaa9dff3a77e979b0f8
26777448
SIG
Windows x86-64 web-based installer
Windows
for AMD64/EM64T/x64
b8b6e5ce8c27c20bfd28f1366ddf8a2f
1363032
SIG
Windows x86 embeddable zip file
Windows
726877d1a1f5a7dc68f6a4fa48964cd1
6745126
SIG
Windows x86 executable installer
Windows
cfe9a828af6111d5951b74093d70ee89
25766192
SIG
Windows x86 web-based installer
Windows
ea946f4b76ce63d366d6ed0e32c11370
1324872
SIG
python-3.7.5-amd64.exe
Show all


Open File - Security Warning
Do you want to run this file?
Name: C:\Users\User\Downloads\python-3.7.5-amd64.exe
Publisher: Python Software Foundation
Type: Application
From: C:\Users\User\Downloads\python-3.7.5-amd64.exe
Run
Cancel
Always ask before opening this file
While files from the Internet can b
type can potentially
harm your computer. Only run
ers you trust. What's
the risk?
Click Run button


Favorites
Creative Cloud
Desktop
Downloads
Recent Places
Libraries
Documents
Music
Pictures
Videos
Local Disk (C:)
Python 3.7.5 (64-bit) Setup
Install Python 3.7.5 (64-bit)
Select Install Now to install Python with default settings, or choose
Customize to enable or disable features.
Install Now
C:\Users\User\AppData\Local\Programs\Python\Python37
Includes IDLE, pip and documentation
Customize installation
Choose location and features
Select both check boxs
Install launcher for all users (recommended)
Add Python 3.7 to PATH
Cancel

# Test your python version

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\Karthikeyan>python -V
Python 3.7.6

C:\Users\Karthikeyan>python --version
Python 3.7.6

C:\Users\Karthikeyan>
```

After completion of successful installation, **open a new command line shell** and type the above commands.

**Note: python –V ( V uppercase char)**

# Linux

- root@hostname~]# yum install **python3 {Enter}**
- root@hostname~]# apt-get install **python3 {Enter}**

# Test your python – in Linux

- **root@hostname~]#** python –V
  python 2.7.5

- **root@hostname~]#** python3 –V
  python 3.7.5

- **root@hostname~]#** python3 - -version
  python 3.7.5

**Note: python3 –V ( V uppercase char)**

# How to run python program?

1. Python subshell
2. Editor(notepad,notepad++) or IDEs (Eclipse,pycharm,padre etc.,)

# Understanding the python program execution.

C:\> **python** filename.py

root@hostname~]# **python** filename.py

root@hostname~]# **python3** filename.py

# Python comments

- Single line comment #
- Multiline comment

```
‘’’
Multiline
comments
‘’’
```

# print(); type();dir()

- **print() – display message to monitor**
- print(named_variable)
- print("user defined string")
- Ex: print("Hello")
- print(10)

- **type() – To determine python type/class**
- type(named_variable) (or) type(value)
- Ex: type(10) -> <class 'int'>

- **dir()** - which returns list of the attributes and methods of any object (say functions , modules, strings, lists, dictionaries etc.)
- dir("named_variable") (or) dir(value)
- Ex: dir(10)

python™

rosum labs
fulfilling your yearning for learning

# Quiz

**Q1. How to check python version?**

A. python –v

B. python - -version

C. python –V

D. option B and C both

E. Option B only

python™

Krosum labs
fulfilling your yearning for learning

# Quiz

**Q2. Is python, a case sensitive language?**

A. Yes

B. No

python™

Krosum labs
fulfilling your yearning for learning

# Quiz

**Q3. How to check the type of a value in python?**

A. print()

B. type()

C. dir()


# Lesson - 2

# Data types in Python

- Numbers  (int,float,complex)
- String (str)
- Bytes (bytes)
- Boolean( bool )
- NoneType(None)

- Python containers (collections)
  - List  (list)
  - Tuple (tuple)
  - Dictionary (dict)
  - Set (set)

# Variable

- variable – namespace – it's holding a value
- **Syntax**:-

  **variablename = value**

  var=10

  name='root'

  cost=14.53

  status=True

Here **var, name , cost and status** are variables

python™

Krosum labs
fulfilling your yearning for learning

# Activity

Q1. what are declarations are invalid declarations

A. $var=10

B. 5var=10

C. VAR=10

D. Var=10

E. var=10

F. F_name=“”

G. s  name=“”

# Activity

Write a python program

Step 1: open an IDLE or any std editor

Step 2: create a file p1.py

Step 3: Within p1.py, declare variables and initialize them with value corresponding to employee details ( empName,empID,empCost)

Step 4: use print() to display employee details.

# String(str)

- String (str) – A string is a sequence of chars. – immutable

- Strings can be created by enclosing characters inside a single quote or double-quotes.

  Ex: ‘Welcome’ “Welcome”

- Triple quotes can be used in Python but generally used to represent multiline strings and docstrings.
- Ex: ‘’’Sample

  Python

  Test code’’’

# String(str)

```
s='Welcome to python'
print(type(s))
<class 'str'>
print(len(s)) # length of python string
17
print(s)
Welcome to python
```




# String(str)

**s='aF4^k  G'** # it alpha,number,space,specialchars

**s='abcaba'** # string allows duplicate chars

**S='line1\nline2\nline3'** # string can hold **escape chars**

**S='''line1**
**Line2**
**Line3'''** # string can hold **multiline** statement ( multiline string )

# String(str)

- **String Indexing**
- We can access individual characters using indexing and a range of characters using slicing.
- S='abcd'  # s | a | b | c | d |

  # | 0 | 1 | 2 | 3 | ← index
- String index starts from 0 (zero)

# String(str)

- **How to access individual index ?**
- String Name [index] => Value / IndexError
- S='abcd'  # s | a | b | c | d |
  # | 0 | 1 | 2 | 3 | ← index

- S[1] => 'b'
- S[5] => IndexError

# String(str)

- Trying to access a character out of index range will raise an **IndexError**.
- The index must be an integer.
- We can't use floats or other types, this will result into TypeError
- Python allows negative indexing for its sequences.
- The index of -1 refers to the last item, -2 to the second last item and so on.
- S[-1] => ‘d’

# String(str)

- **String Slicing**
- We can access a range of items in a string by using the slicing operator **:**(colon)
- String_name[n:m] **# from nth string into m-1 string**

S=‘abcdefg’ # s | a | b | c | d | e | f |

0 1 2 3 4 5

S[1:4] # from 1st index to 3rd (4-1) index

|__ result : ‘bcd’

# String(str)

- **String methods**
- help(str) – help docs about string (str) document
- help(str.upper) – help docs about particular method
- Object.function() # method call
- **“abc”.upper() => ‘ABC’**
- **“abc”.title() => ‘Abc’**
- **“abc”.isupper() => False**

# Activity

Q1. Given a string

S="Sample python code"

From the given string, extract "code" and display it to the console.

Q2. Given a string

S="x:y:z"

(i) Display last 2 chars

(ii) Calculate string total length

python™

Krosum labs
fulfilling your yearning for learning

# Activity

**Q3. Given a String**

**S1="root:x:bin:bash\n"**

**S2="root:x:bin:bash\t"**

**S3="root:"**

Is it possible to remove **\n** **\t** and **:** chars from the above string? If so, How?

# typecasting

- Changing one type to another type
- a=10
- type(a) -><class ‘int’>

convert to float -> float(a) -> 10.0

convert to string -> str(a) -> ‘10’

Convert to boolean -> bool(a) ->True

bool(0) ->False

# typecasting

- Given type is float -> convert to int
- int(10.0) -> 10

- Given type is str -> convert to int /float
- S=‘45’
- int(S) ->45
- float(s) ->45.0
- V=‘ab’
- int(V) ->ValueError

# Activity

Q1. Given:

**V1=100**

**V2=245.34**

**V3='56'**

**V4=0**

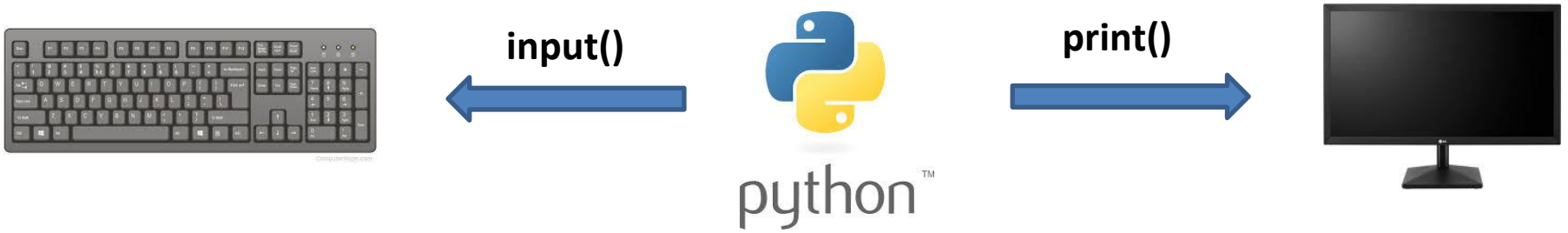
**(i)** convert V1 to string type

**(ii)**convert V2 to int type

**(iii)**convert V3 to float type

**(iv)**convert V4 to boolean

# Basic I/O operation




Python 2.x - raw_input()
Python 3.x - input()

python™

Krosum labs
fulfilling your yearning for learning

# input()

input("prompt message:")

input() – program can prompt the user for input.

All input is stored as a **string**.

# input()

**Prompting for a value**

**Syntax :-**

variable=input("prompt message")

**>>> name=input("Enter your name:")**

Enter your name:Karthik ← user input

# print()

- The **print()** function **prints** the specified message to the monitor (STDOUT)

- **print("message")**

# Example 1

```
>>> name=input("Enter your name:")
Enter your name:karthik

>>> print(name)
karthik
>>>
>>> type(name)
<class 'str'>
>>>
>>> print("Hello...{}".format(name))
Hello...karthik
```

# Example 2

```
>>> N=input("Enter any two digits:")
Enter any two digits:56

>>> print(N)
56
>>>
>>> type(N)
<class 'str'>
>>>N
>>>'56'
```




# Prompting for numeric input

```
>>> N=int(input("Enter any two digits:"))
Enter any two digits:56

>>> print(N)
56
>>>
>>> type(N)
<class 'int'>
>>>N
>>>56
```

# Prompting for numeric input

```
>>> pi=float(input("Enter  pi value:"))
Enter  pi value: 3.15

>>> print(pi)
3.15

>>> type(pi)
<class 'float'>
```

# Activity

Q1. write a python program (Modify **p1.py** file )

Step 1: Create a new file : p2.py

Step 2: Read the employee details from <STDIN>

Step 3:Use **type()** & display input types <STDIN>

Step 4:Use **print()** & display employee details line by line

# Activity

Q2. write a python program

Step 1 : create a filename p3.py

Step 2: Read an application name, application port number and service name from <STDIN>

Step 3: Use print()  to display application details

Note : use escape chars to display application details line by line


# Lesson - 3

# Operators

- + addition
- - subtraction
- /
- // ( floor division – whole number)
- ** exponentiation
- % modulus (remainder after division)
- == != < <= > >= - Comparison operators
- and or not – logical operators
- in not in - membership operators
- is is not identity operators

# Operators

- Example operators.py  # python 2.x – examples

```
print 2*2
print 2**3
print 10%3
print 1.0/2.0
print 1/2
```

Output:

```
4
8
1
0.5
0
```

- Note the difference between floating point division and integer division in the last two lines


# += but not ++


- Python has incorporated operators like +=, but ++ (or --) do not work in Python

# Activity

Write a python program

Step 1: create a filename p4.py

Step 2 : read any two disks partition name from <STDIN>

Step 3 : read an individual partition size from <STDIN>

Step 4: calculate sum of partition size

Step 5: use multiline statement & display input details in below format

**Expected Result:**

**Python p4.py**

**Enter a disk partition: /dev/sda1**

**Enter /dev/sda1 partition Size: 100**

**Enter a disk partition:/dev/sda2**

**Enter /dev/sda2 partition Size:200**

**Partition  /dev/sda1  Size : 100**

**Partition  /dev/sda2  Size : 200**

**----------**

**Total  Partition Size:  300**

**----------**

# String operators

- s1="Welcome"
- s2="Python"
- print(s1+"to"+s2) # WelcometoPython

- s1="welcome"
- print(s1*3) # WelcomeWelcomeWelcome

- count=1230
- print("Total Sales count is:"+str(count))
- # Total Sales count is:1230

# Examples

- s1= "sales"
- **s1 == "sales"**

  True

- **s1 == "SALES"**

  False

- **s1 != "SALES"**

  True

# Activity

Predict the output of below expressions

Q1. “Admin” = = “admin”

Q2. 5062 > 5000

Q3. int(“60”) < int(“75”)

Q4. “Raj” != “raj”

Q5. float(“1.34”) > 0.045

# Logical operators

- Test more than one condition

Logical **and** operator

| Condition1 | Condition2 | Result |
|---|---|---|
| **True** | **True** | **True** |
| True | False | False |
| False | True | False |
| False | False | False |

Logical **or** operator

| Condition1 | Condition2 | Result |
|---|---|---|
| **True** | **True** | **True** |
| **True** | **False** | **True** |
| **False** | **True** | **True** |
| False | False | False |

Logical **not** operator

**not** True => False
**not** False => True

# Logical operators

- counter=560
- counter >500 **and** counter<600

- service=“apache2”
- service == “apache2” **or** service == “httpd”

- S=“root”
- **not** S == “root”

# Membership operators

- **in  not in  => True / False**
- “searchPattern”  in  inputString => True/False
- “e” **in** “hello” => True
- “E” **in** “hello” => False
- “E” **not in** “hello” => True

# Identity operators

- **'is' operator –** Evaluates to true if the variables on either side of the operator point to the same object and false otherwise

- x=100
- type(x) **is** int

  True

- **'is not' operator –** Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.
- type(x) **is** str

  False
- type(x) **is not** str

  True

# Activity

**Predict the result**

Q1. sname=“xerox”

sname == “xerox” **and** sname == “XEROX”

Q2. port=6590

port >6000 **and** port <7000

Q3. app=“TestApp”

app=“testapp1” **or** app == “testapp” **or** app== “TestApp”



# Python Conditional Statement

- Testing (or) Validation (or) Decision making
- Conditional code block will execute only one time.
- Conditional statements are handled by **if** statements.

python™


# Python Conditional Statement

If statement we can write 3 ways

I) if only style

II) if ..else style

III) if .. elif…else style

# Python Conditional Statement

**What is if statement ?**

- **if** is a python keyword.
- **if** statement is used for testing (or) decision making (or) validation.
- **if** only style code block will run the body of code only when if statement is **True**.

**How to use if only style?**

**Syntax:-**

```
if(condition):
<---->True only block
```

# Python Conditional Statement

```
>>> name="root"
>>> name == "root"
True
>>> if(name == "root"):
...     print("Login is success")
...
Login is success
>>>
>>> name = "admin"

>>> name == "root"
False
>>> if(name == "root"):
...     print("Login is success")
...
>>>
```

# Python Conditional Statement

```
>>> count=50
>>>
>>> count>10
True
>>> if(count>10):
...     print("valid count:{}".format(count))
...
valid count:50
>>>

>>> count<10
False
>>> if(count<10):
...     print("valid count:{}".format(count))
...
>>>
```

# if ..else statement

**if(condition):**

**True block**

**else:**

**False block**

```
>>> count=50
>>> count>100
False
>>> if(count>100):
...     print("valid count:{}".format(count))
...   else:
....     print(“invalid count:{}”.format(count))
invalid count:100
>>>
```

# if ..elif statement

**Multi conditional statement**

```
if(condition1):
        Trueblock1
elif(condition2):
        Trueblock2
elif(condition3):
        True block3
…
elif(condition N):
        True block N
else:
        False block
```

# if ..elif statement

```
>>> name="root"
>>> if(name == "root"):
...     print("Login name is:{}".format(name))
... elif(name == "userA"):
...     print("Login name is:{}".format(name))
... elif(name == "userB"):
...     print("Login name is:{}".format(name))
... else:
...     print("Invalid login name")
...
Login name is:root
>>>
```

# if ..elif statement

```
>>> name=“userA"
>>> if(name == "root"):
...     print("Login name is:{}".format(name))
... elif(name == "userA"):
...     print("Login name is:{}".format(name))
... elif(name == "userB"):
...     print("Login name is:{}".format(name))
... else:
...     print("Invalid login name")
...
Login name is:userA
>>>
```

# if ..elif statement

```
>>> name=“userB"
>>> if(name == "root"):
...     print("Login name is:{}".format(name))
... elif(name == "userA"):
...     print("Login name is:{}".format(name))
... elif(name == "userB"):
...     print("Login name is:{}".format(name))
... else:
...     print("Invalid login name")
...
Login name is:userB
>>>
```

# if ..elif statement

```
>>> name=“userC"
>>> if(name == "root"):
...     print("Login name is:{}".format(name))
... elif(name == "userA"):
...     print("Login name is:{}".format(name))
... elif(name == "userB"):
...     print("Login name is:{}".format(name))
... else:
...     print("Invalid login name")
...
Invalid login name
>>>
```

# Activity

Step 1: create a file name p5.py

Step 2: Declare a variable name uname and initialize it with value “root”

Step 3: read a user name from <STDIN>

Step 4: Test if input user name matched with value in uname.

Step 5:If matched, display message “login is valid “ else display “login is invalid.”

# Activity

- Write a python program

Step 1: create a filename p6.py

Step 2: Read a port number from <STDIN>

Step 3: Test whether input port number range between 501-599

Step 4: If matched, initialize the application name as “Test-App1”.

Step 5: else display message “invalid port number.”

# Activity

- Write a python program –

Step 1: create file name p7.py

Step 2: Read a shell name from <STDIN>

Step 3: If input shell name is bash, initialize profile file name as "bashrc"

Step 4: If input shell name is ksh, initialize profile filename as "kshrc"

Step 5: If input shell name is psh, initialize profile filename as "winprofile"

Step 6:If neither of the shell name matches, Initialize with default shell name as "nologin" and profile file name as "/etc/profile"

Step 7 : Display shell name and shell profile filename

# Python Looping statements

- General loop is used to execute a block of **statements** or code several times until the given condition becomes false.
- We use for **loop** when we know the number of times to **iterate**.
- while – loop (Condition)
- for – loop (Collection)

# Python Looping statements

## while loop

The while loop in Python is used to iterate over a block of code as long as the test expression (condition) is true.

## Syntax:

**while(condition):**

<codeblock>

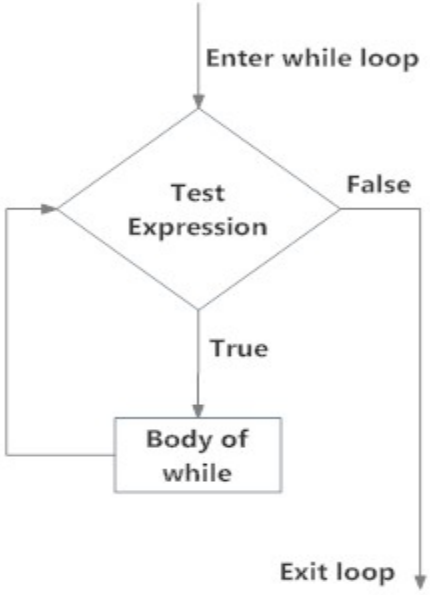



Fig: operation of while loop

# Python Looping statements

3 Points to remember

1. Initialization → **i=0**
2. Condition → while(**i<5**):
3. Increment/Decrement → i=**i+1**

# Python Looping statements

```
>>> i=0
>>>  while(i<3):          # 0 < 3 → True
          print("Hello…{}".format(i))
          i=i+1
```

**Hello…0**

# Python Looping statements

```
>>> i=0
>>> while(i<3): # 1<3 → True
        print("Hello…{}".format(i))
        i=i+1
```

**Hello…0**

**Hello…1**

# Python Looping statements

```
>>> i=0
>>> while(i<3): # 2<3 → True
        print("Hello…{}".format(i))
        i=i+1
```

**Hello…0**

**Hello…1**

**Hello…2**

# Python Looping statements

```
>>> i=0
>>>  while(i<3):  3<3 False → Exit from loop
        print("Hello…{}".format(i))
        i=i+1
Hello…0
Hello…1
Hello…2
>>>
```

# Python Looping statements

- For loops are used for sequential traversal.
- **Syntax:-**

  **for** variable **in** collection**:**

  code block(s)

  **Example:-**

  for v in **“abcd”:**

  print(“Hello..{}”.format(v))

  Hello..’a’

  Hello..’b’

  Hello..’c’

  Hello..’d’

# break ; continue

- break  - exit from loop
- continue -  continue from next element

# Activity

Write a program:

Step 1: create a file name p8.py

Step 2: declare & initialize the pin number (ex: pin=1234)

Step 3: Use while loop to iterate following statement thrice

(i) Read a pin number from <STDIN>

(ii) Compare a input pin with existing pin number

(iii) If both pin numbers are matched , display pin number is matched at count time & exit from loop.

(iv) If all 3 attempts fails, display message " your pin is blocked."

# Activity

Step 1: create a file name: p9.py

Given String

S="123456578"

Step 2: Calculate sum of numbers

Note : use for loop

# Activity

Write a python program

Step 1: create a new file p10.py

Step 2: Modify the below code using while loop

```
s="abcd"
for var in s:
        print(var)
```

Step 3: Display list of characters one by one


# Lesson - 5

# Python - Collections

- List ( list  - [ ] )
- Tuple ( tuple – () )
- Dictionary  ( dict – { } )
- Set (set)

# List

- **Lists** are just like dynamic sized arrays, declared in other languages.
- List are ordered elements
- A single list may contain mixed data types.
- Lists are **mutable**, and hence, they can be altered even after their creation.

# List

- The elements in a list are indexed according to a definite sequence and the indexing of a list is done with **0** being the **first index**.
- Each element in the list has its definite place in the list, which allows duplicating of elements in the list, with each element having its own distinct place and credibility.
- List supports **indexing** and **slicing**

# List - Examples

- Listname=[]  # Creating a list
- DB=[‘oracle’,’sql’,’plsql’,’mysql’,’sqlite3’]
- Emp=[‘arun’,’sales’,133,1323.23,True]

# Example

```
L=['D1', 10, 3.45,True,None ]
    # 0      1     2       3       4   <== index
    #  -5  -4    -3      -2     -1<== index
type(L)     =>  <class 'list')
type(L[0]) => <class 'str'>
len(L) => 5
```

# List – Index and Slicing

Files=[‘p1.c’ ,’p2.java’,’p3.cpp’,’p4.py’,’repo.log’]

# 0 1 2 3 4 ← index

Files[1] # ‘p2.java’

Files[1:4] # [ ‘p2.java’, ’p3.cpp’, ’p4.py’ ]

Files[:2] # [‘p1.c’,’p2.java’]

# Membership operators

```
# "searchString" in inputList ->True/False

fnames=["p1.log","p2.log","p3.log","test.log"]

if("p3.log" in fnames):
   print("Yes file p3.log is exists")
else:
   print("Sorry file is not exists")
```

# List methods

- Listname.append(Value)

  (or)

- Listname.insert(index,Value)
- # we can add new data to existing list

- Listname.pop(Index)
- # we can delete nth data from existing list

- Listname[index]=updated_value
- # we can modify existing nth data from list

# Example

```
L=[]
print(len(L))
 # Listname.append(Value)  =>None
L.append("p1.log")
L.append(100)
L.append(3.45)
L.append(True)
print(L)  => ['p1.log',100,3.45,True]
```




# List methods

- Listname.pop()
- Listname.insert(Index,Value)
- Listname.index(Value)
- Listname.count(value)

# Activity

Q1. write a python program

Step 1 : Create a file name : p11.py

Step 2: create an empty list

Step 3: display size of list

Step 4: use while loop  5 times

i) To read a hostname from <STDIN>

ii) To add a input hostname to existing list

Step 5: using for loop, display list of elements

Step 6: display size of the list

# Activity

Q2. write a python program

Given List

DBs=[‘oracle’,’sql’,’mysql’,’plsql’]

Step 1: create a file name : **p12.py**

Step 2: read a database name from <STDIN>

Step 3: test input database name is existing or not

Step 4: if input DB name exists, using index(), display index number

Step 5: If input DB does not exist, add the input DB name to the existing list.

Step 6: display list line by line using for loop

# Activity

Q3. write a python program

Given List

Step 1: create a filename p13.py

LB=[‘0.13’,’14.4’,’1.34’,’3.24’,’2.44’]

Step 2: Calculate sum of load balance

# Tuple

- Tuple – Collection of elements like list

- The difference between the two is that we cannot change the elements of a tuple once it is assigned whereas we can change the elements of a list.
  i.e., tuple is immutable

- type(( )) Vs type([])

# Tuple

Syntax:-
tuplename=(list of elements)

**tname=(10,20.45,”data”,True)**
**len(tname) => 4**

# Tuple

- A tuple can also be created without using parentheses. This is known as tuple packing.
- V1=10
- type(V1) => <class ‘int’>

- **V2=10,**

  type(V2) => <class ‘tuple’>


# Iterating Through a list/tuple

```
F=['p1.log','p2.log','p3.log']
for var in F:
    print(var)
```

```
p1.log
p2.log
p3.log
```

```
servers=("unix","linux","aix")
for var in servers:
    print(var)
```

```
unix
linux
aix
```

# Tuple operations

- tuple supports indexing and slicing
- tuple supports membership **in not in** operators.
- We typecast list to tuple vice versa

  tuple(input_List)

  list(input_tuple)

# Deleting a Tuple

- we cannot change the elements in a tuple. It means that we cannot delete or remove items from a tuple.
- Deleting a tuple entirely, however, is possible using the function del().
- **del(tuple_name)**

# Tuple usages in python

- Since tuples are immutable, iterating through a tuple is faster than with list. So there is a slight performance boost.

- Tuples that contain immutable elements can be used as a key for a dictionary. With lists, this is not possible.

- If you have data that doesn't change, implementing it as tuple will guarantee that it remains write-protected.

- tuple() type of structures used in functions

# Activity

Predict the error message

Q1. T=(1,2.3,’D1’)
T[1]=“data1”

Q2.  T=(10,20,30,40,50)
print(T[-6])

Q3. T=(1,2)
del(T[1])

# Activity

Q1. Write  a python program

Step 1: create a filename p14.py

Given tuple

**Products=(“P1”,”P2”,”P3”,”P4”,”P5”)**

Step 2:  display the list of products except **P2** and **P3**

Note :use for loop statement

# Activity

Write a python program

Step 1: create a filename p15.py

Given Tuple :

**EMP=('101,leo,sales,1000','102,paul,prod,2000','103,raj,HR,3000' )**

Step 2: use for loop along with split() to get the following expected result.

Expected result:-

Emp name is leo working department is sales

Emp name is paul working department is prod

Emp name is raj working department is HR

---

Sum of Emp's cost is: 6000

---




# Lesson - 6

# Dictionary

- Python dictionary is an unordered collection of items.
- Syntax:-

dict_name={“key1”:Value,”Key2”:Value,..”Kn”:”Vn”}

Data – {‘Key’ : Value }

app={“port”:80,”service”:”apache2”}

| Key | Value |
|---|---|
| port | 80 |
| service | apache2 |

# dict- operations

# Accessing Elements from Dictionary
dict_name[‘Key’] → Value / KeyError

# adding newdata to existing dictionary
dict_name[‘NewKey’]=Value

# modifying existing dictionary element
dict_name[‘ExistingKey’]=Updated_value

# deleting nth element
del(dict_name[‘Key’])

# Activity

Step 1 - Open a python shell

Step 2 - create an empty dict (ex: d={})

Step 3 - read a hostname from <STDIN>

read an IPAddress from <STDIN>

Step 4 - add a input details to existing dict

with hostname as a key and IPAddress as its value

Step 5 - display dictionary and it's size

# dict- methods

dict.get(Key)

dict.setdefault(Key,Value)

dict.pop("key")

dict.keys()

# Activity

Write a python program

Step 1 : create a file name: p16.py

Step 2 : create an empty dict

Step 3 : use looping statements – 5times

i) Read a hostname from <STDIN>

ii) Read a IP-Address from <STDIN>

iii) Add a input details to existing dict

iv) with hostname as a key and IP address as it's value

Step 4 : display Key/ value details to monitor

# Dictionary Membership Test

- We can test if a key is in a dictionary or not using the keyword **in**.
- Note that the membership test is only for the keys and not for the values.

- "key" in input_dictionary → True/False

# Activity

Write a python program – modify p16.py file

Step 1: create a new file p17.py

Step 2: Use membership operator to test whether the input hostname already exists or not.

Step 3: if it's exists already, display pop up message "**Sorry your input hostname is exist**".

# Iterating Through a Dictionary

We can iterate through each key in a dictionary using a for loop.

```
d={“K1”:”V1”,”K2”:”V2”,”K3”:”V3”}
for var in d:
    print(var)
```

K1

K2

K3

# Activity

Write a python program

Step 1 : create a new file p18.py with existing code of p17.py

Step 2: Using for loop – display key /value details

i.e., hostname  and IP-Address details

# set

- A **set is an unordered collection of items**. Every element is unique (no duplicates) and must be immutable (which cannot be changed)
- Sets can be used to perform mathematical set operations like **union, intersection, symmetric difference** etc.

# How to create a set?

- A set is created by placing all the items (elements) inside curly braces {}, separated by comma or by using the built-in function **set().**

- **var={1,2,3,4,"Data1","Data2"}**
- **>>> type(var)**
- **<class 'set'>**

- **>>> v1=set()**
- **>>> type(v1)**
- **<class 'set'>**

# Empty set

- To make a set without any elements we use the **set()** function without any argument.

```
var={}
type(var)
<class 'dict'>  ← dictionary
v1=set()

type(v1)
<class 'set'>
len(v1)
0
```

- Every element is **unique** (no duplicates) and must be **immutable** (which cannot be changed).
- >>> v2={10,20,30,10,20,"DATA1","data1","DATA1"}
- >>> print(v2)
- {10, 'DATA1', 'data1', 20, 30} **# there is no duplicate element**

- We cannot access or change an element of set using indexing or slicing. set does not support it.

- v2={10,20,30}

type(v2)

<class 'set'>

>>> v2[0] ←

- Traceback (most recent call last):
- File "<stdin>", line 1, in <module>
- TypeError: 'set' object does not support indexing

>>> print(v2)

{10, 20, 30}

# How to change a set in Python?

- set are unordered, indexing have no meaning.
- We cannot access or change an element of set using indexing or slicing.
- We can add single element using the add() method and multiple elements using the update() method.
- The update() method can take tuples, lists, strings or other sets as its argument.

# add() vs update()

```
>>> v3={"Data1","Data2","Data3"}
>>>
>>> len(v3)
3
>>> print(v3)
{'Data1', 'Data2', 'Data3'}
>>>
>>>
>>> v3.add("Data4")
>>> v3
{'Data1', 'Data2', 'Data4', 'Data3'}
>>>
>>> v3.add("Data4")
>>> v3
{'Data1', 'Data2', 'Data4', 'Data3'}
>>>
>>> # avoiding duplicate entry
...
```

```
>>> v4={"Text1","Text2"}
>>>
>>>
v4.update(["Text3\n","Text4\n","Text5\n"])
>>> v4
{'Text1', 'Text2', 'Text3\n', 'Text5\n', 'Text4\n'}
>>>
>>>
v4.update(("Text3\n","Text4\n","Text6\n"))
>>>
>>> v4
{'Text1', 'Text2', 'Text6\n', 'Text3\n', 'Text5\n',
'Text4\n'}
>>>
>>> len(v4)
6
```

# How to remove elements from a set?

- A particular item can be removed from set using methods, discard() and remove().
- while using discard() if the item does not exist in the set, it remains unchanged.
- But remove() will raise an error in such condition.

# remove() vs discard()

```
>>> v3={"Data1","Data2","Data3"}
>>>
>>> len(v3)
3
>>> print(v3)
{'Data1', 'Data2', 'Data3'}
>>>
>>>
>>> v3.remove("Data3")
>>> v3
{'Data1', 'Data2'}
>>>
>>> v3.remove("Data7")
>>> traceback (most recent call last):
 File "<stdin>", line 1, in <module>
eyError: 'data7'
```

```
>>> v4={"Text1","Text2"}
>>>
>>> v4.discard(“Text2”)
>>> v4
{'Text1‘}
>>>
>>> v4.discard(“Text5”)
>>>
>>> v4
{'Text1’}
```

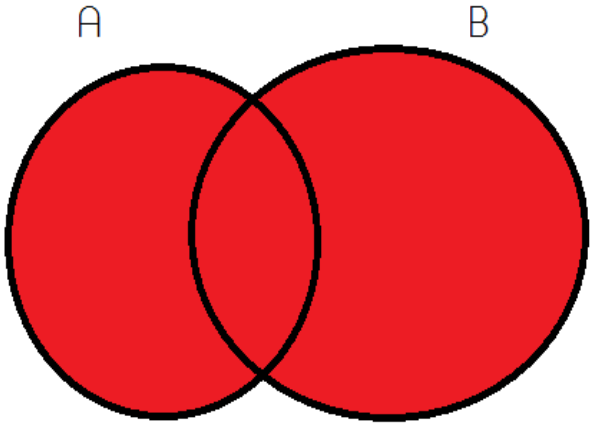
# Python Set Operations

- Sets can be used to carry out mathematical set operations like **union, intersection, difference and symmetric difference**.
- We can do this with operators or methods.

A = {1, 2, 3, 4, 5}

B = {4, 5, 6, 7, 8}

# Set Union



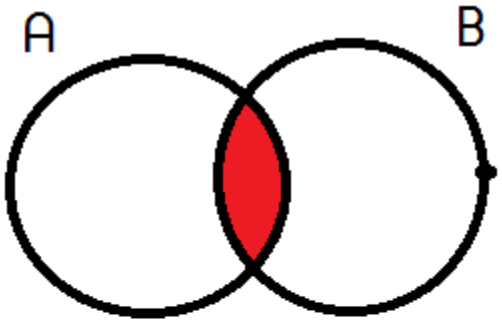

A | B

A.union(B)

B.union(A)

```
>>> A={1,2,3,4,5}
>>>
>>> B={1,2,3,6,7,8,9}
>>>
>>> A.union(B)
{1, 2, 3, 4, 5, 6, 7, 8, 9}
>>>
>>> B.union(A)
{1, 2, 3, 4, 5, 6, 7, 8, 9}
>>>
>>> A|B
{1, 2, 3, 4, 5, 6, 7, 8, 9}
>>>
>>> B|A
{1, 2, 3, 4, 5, 6, 7, 8, 9}
>>>
```

# Set Intersection




A.intersection(B)
B.intersection(A)

A & B
B &A

```
>>> A={1,2,3,4,5}
>>> B={1,2,3,6,7,8,9}
>>>
>>> A&B
{1, 2, 3}
>>>
>>> B&A
{1, 2, 3}
>>>
>>> A.intersection(B)
{1, 2, 3}
>>>
>>> B.intersection(A)
{1, 2, 3}
```

# Activity

Q1. open a python shell

C={“p1.c”,”p2.c”,”p3.java”,”Demo”}

D={“p1.java”,”p1.c”,”p3.java”,”p2.c”,”Demo”,”D1”}

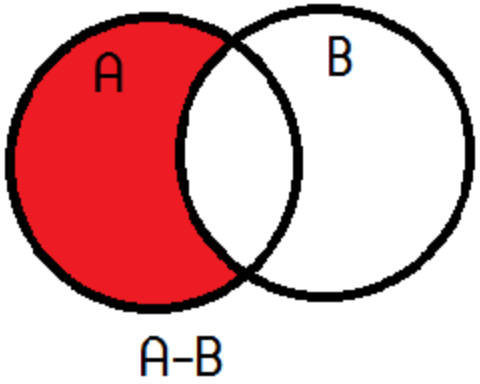
A) Filter common files from the above two sets

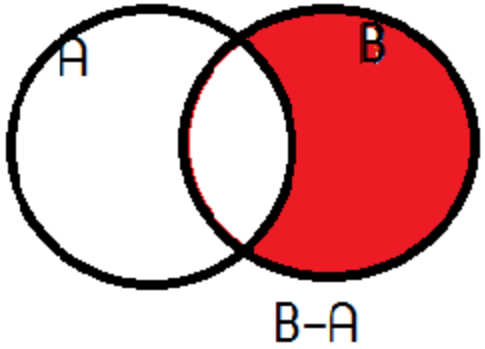
B) Combine both sets into single set and omit duplicate elements.

C) type cast to list

# Set Difference

- Difference of A and B (A - B) is a set of elements that are only in A but not in B. Similarly, B - A is a set of element in B but not in A.



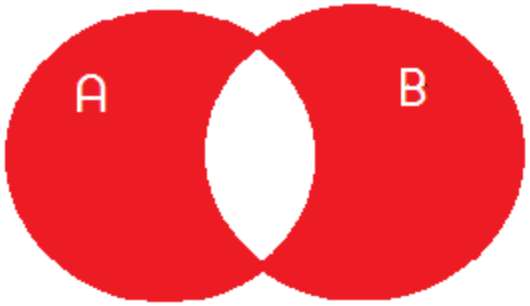
# set difference

```
>>> A={1,2,3,4,5}
>>> B={1,2,3,6,7,8,9}
>>>
>>> A-B
{4, 5}
>>> B-A
{8, 9, 6, 7}
>>>
>>> A.difference(B)
{4, 5}
>>>
>>> B.difference(A)
{8, 9, 6, 7}
>>>
```

# Set Symmetric Difference

- Symmetric Difference of A and B is a set of elements in both A and B except those that are common in both.
- Symmetric difference is performed using ^ operator.
- Same can be accomplished using the method **symmetric_difference()**.

# Set Symmetric Difference




- >>> A={1,2,3,4,5}
- >>>
- >>> B={1,2,3,6,7,8,9}
- >>>
- **>>> A^B**
- {4, 5, 6, 7, 8, 9}
- >>>
- **>>> B^A**
- {4, 5, 6, 7, 8, 9}
- >>>
- **>>> A.symmetric_difference(B)**
- {4, 5, 6, 7, 8, 9}
- >>>
- **>>> B.symmetric_difference(A)**
- {4, 5, 6, 7, 8, 9}
- >>>

# Activity

Predict the result of below set operations

S1={‘data1’,’data2’,’data3’}

S2={‘data2’,’data3’,’data4’,’data5’}

```
print(S1-S2)
print(S2-S1)
print(S1 ^ S2)
```



# Lesson - 7

# File Handling

input()

print()

python™

COMPUTER STORAGE

DEVICES

# File categories

- Reading data from <FILE> → Python → display to monitor
- Python → Create / Write data to FILE
- Reading data from <FILE> → Python → Create/Write data to another FILE

# File – read operation

Open a file => fileobject=open(inputfile,mode)

|

Read content fileobject.read() / fileobject.readlines()

|

Close a file => fileobject.close()

# File – create/write operation

Open a file => fileobject=open(result_file,"w")

# w – write ; a –append

|

Read content fileobject.write( "InputString\n")

|

Close a file => fileobject.close()

# File – read/write operation

```
FH=Open(“inputFile”,”r”)
WH=open(“resultFile”,”w”)
S=FH.read()
WH.write(S+”\n”)
FH.close()
WH.close()
```

# Activity

Write a python program

Step 1: Create a filename  p19.py

Step 2: Read an existing TEXT file from your disk

Step 3: Display file content line by line.

# Activity

Write a python program

Step 1 : create a filename p20.py

Step 2 : create a new emp.csv file under D:\\

Step 3 : write any 5 sample csv content to emp.csv file

Step 4 : close the file

# Activity

Write a python program

i) create a filename p21.py to demonstrate the **cp** command

Syntax:

cp oldfile  newfile

# with statement in python

```
with open("inputFile","r")  as fileobject:
        fileobject.read() / fileobject.readlines()
```

```
with open("resultFile","w") as fileobject:
        fileobject.write("Single String\n")
```

Note : fileobject.close()  note required

# Examples

```
with open("D:\\test.log") as FH:
        s=FH.read()
        print(s)
```

Reading data from <FILE>

```
with open("D:\\result.log“,”w”) as WH:
        WH.write("data1\n")
        WH.write("data2\n")
        WH.write("data3\n")
```

Create/writing data to FILE

```
with open("D:\\test.log") as FH:
        with open("D:\\r1.log","w") as WH:
                for var in FH.readlines():
                        WH.write(var)
```

Read/write operation

# Activity

Write a python program

Step 1 : Create a filename p22.py

Step 2 : Use **with** statement to modify p20.py and p21.py program

# Activity

Write a python program

Step 1 : create a filename p23.py

Given List

**Net=['interface=eth0','bootproto=dhcp','onboot=none']**

Step 2 : create a new file called property.txt

Step 3 : iterate a given list one by one

Step 4 : write list element into property file

Note : use with statement

# Activity

**Write a python program**

Step 1 : create a filename: p24.py

Step 2 : create an empty dict

Step 3 : read a existing property.txt file (read line by line)

Split each line into multiple values

Key = value

Add the split data to existing dictionary

Step 4: use for loop – display key/value details

Step 5: modify following operation

*Onboot -> yes ; bootproto -> static ;*

*Add new IP-address ex: IPADDR=10.20.30.40*

Step 6: display key/value details (Step 4 )

Step 7: create a new property file( p1.txt) and write updated dictionary details in same format.


# Lesson - 8




# What is a function in Python?

- In Python, function is a **group of statements** that perform a specific task.

- Functions help break our program into smaller and modular chunks.

# Syntax of Function

- **def function_name(parameters):**
  **"""docstring"""**
  **statement(s)**

- Keyword **def** marks the start of function header.
- A function name to uniquely identify it.
- Function naming follows the same rules of writing identifiers in Python.
- Parameters (arguments) through which we pass values to a function.
- A colon (:) to mark the end of function header.
- Optional documentation string (docstring) to describe what the function does.
- One or more valid python statements that make up the function body. Statements must have same indentation level.
- An optional return statement to return a value from the function.

# How to call a function in python?

- Once we have defined a function, we can call it from another function, program or even the Python prompt.

- To call a function we simply type the function name with appropriate parameters.
- function_name()

# Example

```
>>> def display():
...     print("Hello I am display block")
...
>>> type(display)
<class 'function'>
>>>
>>> display
<function display at 0x02287108>
>>> display()
Hello I am display block
>>>
```

File : p1.py

```
import   os

print("List of files")
os.system("ls")

print("process details:-")
os.system("ps")

print("List of files")
os.system("ls")

print("process details:-")
os.system("ps")
....
print("List of files")
os.system("ls")
```

File : p2.py

0x5432

```
import   os

def  f1():
        print("List of files")
        os.system("ls")
```

0x1234

```
def  f2():
        print("Process details:-")
        os.system("ps")
```

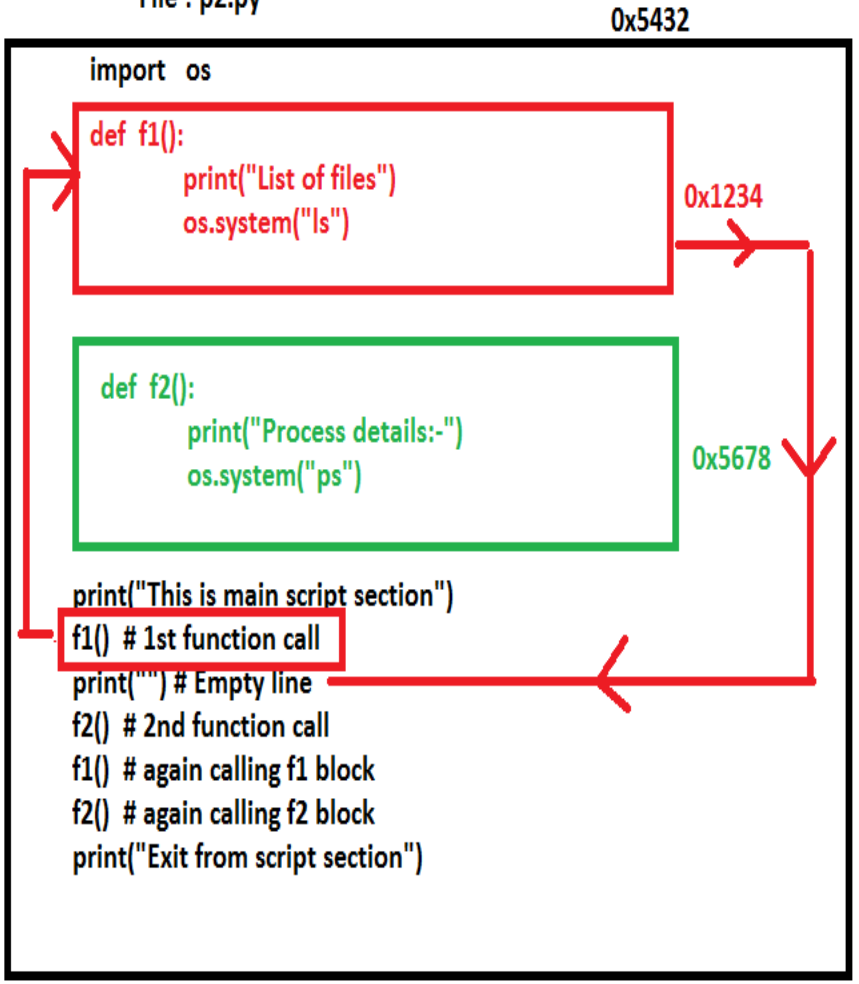
0x5678

```
print("This is main script section")
f1()  # 1st function call
print("") # Empty line
f2()  # 2nd function call
f1()  # again calling f1 block
f2()  # again calling f2 block
print("Exit from script section")
```

File : p1.py

```
import   os

print("List of files")
os.system("ls")

print("process details:-")
os.system("ps")

print("List of files")
os.system("ls")

print("process details:-")
os.system("ps")
....
print("List of files")
os.system("ls")
```

File : p2.py



File: P2.py

0x5432

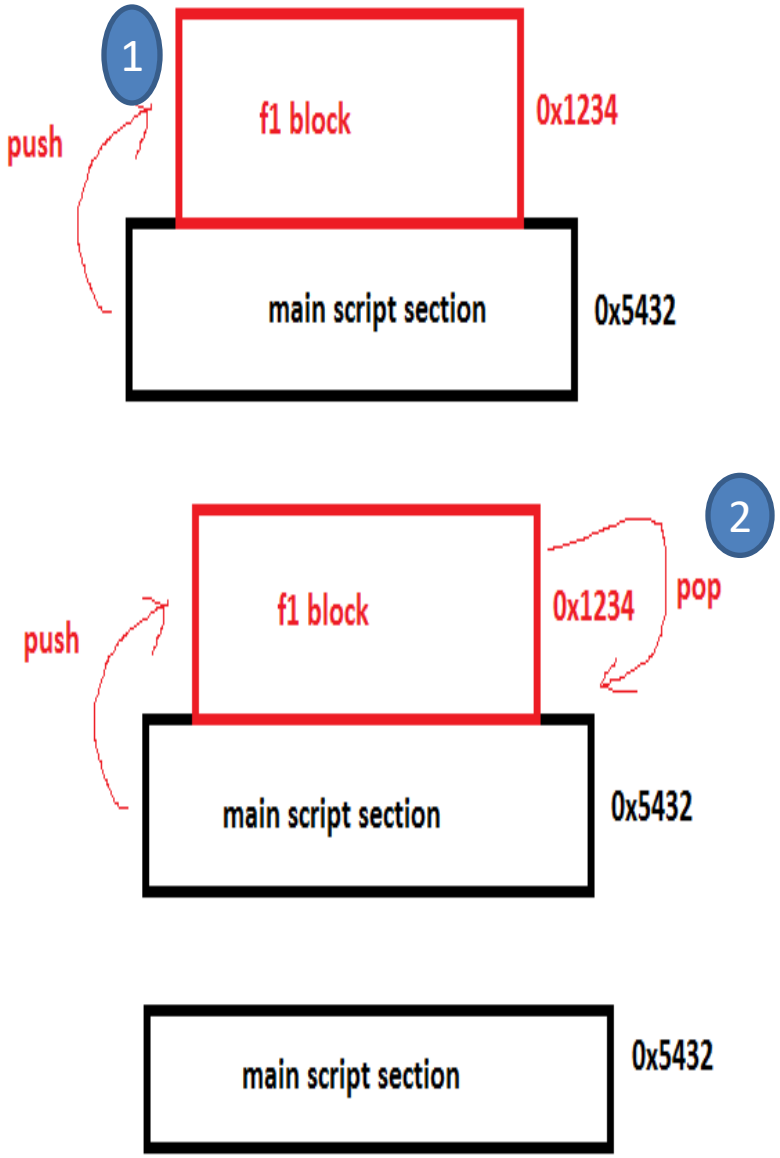
```
import os

def f1():
        print("List of files")
        os.system("ls")

def f2():
        print("Process details:-")
        os.system("ps")

print("This is main script section")
f1() # 1st function call
print("") # Empty line
f2() # 2nd function call
f1() # again calling f1 block
f2() # again calling f2 block
print("Exit from script section")
```

0x1234

0x5678

1

2



# Function call with arguments

- In function call, we can pass any type of values as arguments.

**Syntax :-**

```
def   function_name(arg1,arg2,arg3):
            Code block

function_name(Value1,Value2,Value3)
                    # function call with arguments
```

# Function call with arguments

```
>>> def   f1(a1,a2):
...          print("Function call with arguments")
...          print(type(a1))
...          print(type(a2))
...          print("Exit from function")
```

```
>>> f1(10,2.45)
Function call with arguments
<class 'int'>
<class 'float'>
Exit from function
>>>
>>> f1("abc",[])
Function call with arguments
<class 'str'>
<class 'list'>
Exit from function
```

```
>>> f1((),{})
Function call with arguments
<class 'tuple'>
<class 'dict'>
Exit from function
>>>
>>> f1({"S1","S2"},[])
Function call with arguments
<class 'set'>
<class 'list'>
Exit from function
>>>
```

# Function call with arguments

- We can call a function by using the following types of formal arguments-

- Required arguments **def f1(a1,a2,…an)**
- Default arguments **def f2(variable=value)**
- Variable-length arguments **def f3(*args)**
- Keyword arguments **def f4(**kwargs)**

# Function call with arguments

- **Required Arguments**
- Required arguments are the arguments passed to a function in correct positional order.
- Here, the number of arguments in the function call should match exactly with the function definition.

**Example**

```
def  f1(a1,a2):
        print("a1 value:{}\ta2 value:{}".format(a1,a2))

f1(10,1.334) # function call with 2 arguments(int,float)
a1 value:10     a2 value:1.334

f1("AB",["D1","D2","D3"]) # function call with 2 arguments(str,list)
a1 value:AB     a2 value:['D1', 'D2', 'D3']
```

# Function call with arguments

- **Default Arguments**
- A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument.

- def function_name(var=value):
    code block

# Function call with arguments

```
>>> def f2 (a1=10,a2=2.46):
...     print(a1,a2)
...
>>> f2() # empty args
10  2.46
>>> f2("AB") # single args
AB 2.46
>>> f2("AB","SAB")
AB SAB
```

```
>>> def f3(user="root",port=22):
...     print(user,port)
...
>>> f3() # empty args
root 22
>>> f3("userA") # single args
userA 22
>>> f3("userA",120)
userA 120
```


```
def   display (user,passwd, ip="127.0.0.1",port=22):
          print("Login name:{}".format(user))
          print("Password:{}".format(passwd))
          print("IP-Address:{}".format(ip))
          print("PORT Number:{}".format(port))

display("userA","Welcome") # required arguments
Login name:userA
Password:Welcome
IP-Address:127.0.0.1
PORT Number:22

display("userA","Welcome","10.20.30.40") # required arguments and default args
Login name:userA
Password:Welcome
IP-Address:10.20.30.40
PORT Number:22

display("userA","Welcome","10.20.30.40",1240) # required arguments and default args
Login name:userA
Password:Welcome
IP-Address:10.20.30.40
PORT Number:1240
```

# Function call with arguments

- **Variable-length Arguments**
- You may need to process a function for more arguments than you specified while defining the function.
- These arguments are called variable-length arguments and are not named in the function definition, unlike required and default arguments.
- def f1(*args):

    code block

```
>>> def    f1(*a1):  # variable length arguments
...             print(type(a1))
...             print(a1)
...
>>>
>>> f1()   # call with empty argument
<class 'tuple'>
()
>>> f1(10,2.34,"data")  # call with args
<class 'tuple'>
(10, 2.34, 'data')
>>>
>>> f1(10,2.34,"data",["D1","D2","D3"])  # call with args
<class 'tuple'>
(10, 2.34, 'data', ['D1', 'D2', 'D3'])
>>>
```

```
>>> def   f1(a1,a2=100,*a3):  # required args,defaultargs,variablelength args
...            print("A1:{}".format(a1)) # required args
...            print("A2:{}".format(a2)) # default value
...            print("A3:{}".format(a3)) # variable length args-tuple
>>> f1("ab")
A1:ab
A2:100
A3:()
>>> f1("ab","Test")
A1:ab
A2:Test
A3:()
>>> f1("ab","Test","report1","report2","report3")
A1:ab
A2:Test
A3:('report1', 'report2', 'report3')
```

# Function call with arguments

- **Keyword Arguments**
- Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.
- This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters.

# Function call with arguments

Keyword Arguments

```
def  f1(**kwargs):
        code block

f1(variable=value)
```

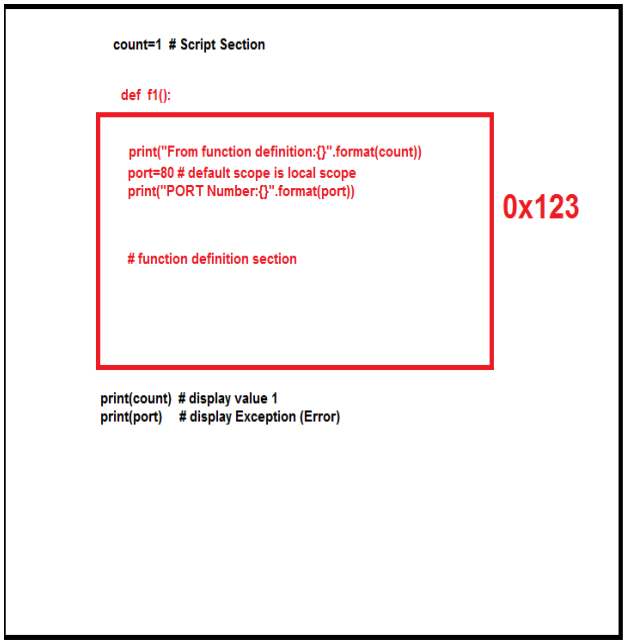
```
>>> def f1(**a1): # keyword arguments
...         print(type(a1))
...         print(a1)
>>>
>>> f1() # empty argument
<class 'dict'>
{}
>>>
>>> f1(name="root",db="mysql",user="root") # keyword arguments
<class 'dict'>
{'name': 'root', 'db': 'mysql', 'user': 'root'}
>>>
>>> def f1(**kwargs):
...          for v in kwargs.keys():
...                print("{}\t{}".format(v,kwargs[v]))

>>> f1(name="root",db="mysql",user="root") # keyword arguments
name    root
db      mysql
user    root
```

```
>>> def display(a1,a2=100,*a3,**a4):
...         print(a1) # required argument
...         print(a2) # default argument
...         print(a3) # variable length args
...         print(a4) # keyword argument
...
>>> display("AB") # required argument
AB
100
()
{}
>>> display("AB","TEST1") # required and default argument
AB
TEST1
()
{}
>>> display("AB","TEST1","TEST2","TEST3","TEST4")
AB
TEST1
('TEST2', 'TEST3', 'TEST4')
{}
```

```
display ("AB","Test1","Test2","Test3","Test4",user="root",passwd="Welcome",port=80)
AB
Test1
('Test2', 'Test3', 'Test4')
{'user': 'root', 'passwd': 'Welcome', 'port': 80}
>>>
```

# Scope

```
count=1 # Script section
def f1():
    print("From function definition:{}".format(count))
    port=80 # default scope is local scope
    print("PORT Number:{}".format(port))

f1() # function call
From function definition:1
PORT Number:80
>>> port # variable port is not defined in script section
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'port' is not defined
>>>
```



# global

- **global** keyword is a keyword that allows a user to modify a variable outside of the current scope.
- **global** keyword is used inside a function only when we want to do assignments or when we want to change a variable.

# Rules of global keyword

- If a variable is assigned a value anywhere within the function's body, it's assumed to be a local unless explicitly declared as global.
- Variables that are only referenced inside a function are implicitly global.
- We Use global keyword to use a global variable inside a function.
- There is no need to use global keyword outside a function.

```
>>> def f1():
...     global port
...     port=80
...
>>> f1() # function call
>>> print(port) # global value
80
>>>
```

# return

- A return statement is used to end the execution of the function call and “returns” the result (value of the expression following the return keyword) to the caller.
- **return** statement can not be used outside the function.
- In python default  return value is **None**.

```
>>> def f1():
...     print("Hello")
>>> rv=f1()
Hello
>>> rv == None
True
>>>
```

# python supports all types of return values

```
>>> def f1():
...     return "abc"  # string
...
>>> f1()
'abc‘

>>> def f2():
...     return 1.355 # float
...
>>> f2()
1.355

>>> def f3():
...     return True # boolean
...
>>> f3()
True

>>> def f4():
...     return ["D1","D2","D3"] # list
...
>>> f4()
['D1', 'D2', 'D3']
```

```
>>> def f5():
...     return ("T1","T2") # tuple
...
>>> f5()
('T1', 'T2')

>>> def f6():
...     return {"K1":"V1","K2":"V2"} # dict
...
>>> f6()
{'K1': 'V1', 'K2': 'V2'}
>>>
>>> def f7():
...     return {"K1","K2",12,3,4,5.45} # set
...
>>> f7()
{3, 4, 'K2', 12, 'K1', 5.45}
>>>
```

# Returning Multiple Values

- In Python, we can return multiple values from a function.
- In python function returns more than one value means the default type will be tuple(immutable).

```
>>> def f1():
...     return 10
...
>>> type(f1())
<class 'int'>
>>>
>>> def f1():
...     return 10,   # more than one value,separated by ,(comma)
...
>>> type(f1())
<class 'tuple'>
>>>
```



```
>>> def f1():
...         return 10,3.45,"ab",["D1","D2"],("T1","T2"),{"K1":"V"}
...
>>> type(f1())
<class 'tuple'>

>>> f1()
(10, 3.45, 'ab', ['D1', 'D2'], ('T1', 'T2'), {'K1': 'V'})
```

# Activity - identify the errors

**Q1. def f1(a1,a2):**

```
    print("Hello")
f1(10,20,None)
```

**Q2. def f2(a1,a2,a3=0,a4):**

```
    print("Hello")
f2(100,200,300,400)
```

**Q3. def f3(a1,a2,a3=0):**

```
    print("Hello")
f3(10)
```

**Q4. def f4(a1,a2=0,*a3,*a4):**

```
    print("Hello")
f4(10)
```

**Q5. def f5(**a2,*a3):**

```
    print("Hello")
f5()
```




# Activity

Write a python program

Step 1: create a filename p25.py file by modifying p24.py

Step 2: Convert each step into separate function

Note: Declare local variable inside the function and return the processed value


# Lesson - 9

# Python Modules

- Python module is  existing python source file
- Filename extension must be .py
- A module can also include runnable code.
- Reusability




# Module basics

- Each file in Python is considered a module.
- Everything within the file is encapsulated within a namespace (which is the name of the file)
- To access code in another module (file), import that file, and then access the functions or data of that module by prefixing with the name of the module, followed by a period.

File : ab.py

==========

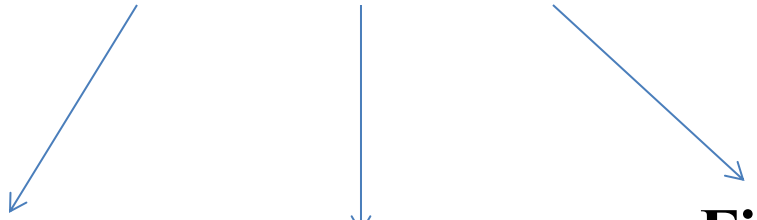
port=80

service="httpd"

def fx():

    return 10

==========



**File : p1.py**

```
import  ab
print (ab.port)
```

**File:  p2.py**

```
import ab
print(ab.port)
print(ab.service)
```

**File: p3.py**

```
import ab
rv=ab.fx()
print("rv={}".format(rv))
```

# What import does

An import statement does three things:

- Finds the file for the given module
- Compiles it to byte code
- Runs the module's code to build any objects (top-level code, e.g., variable initialization)

- env variable  PYTHONPATH

# Activity

Step 1: Create a new file p26.py by modifying p25.py file

Step 2: remove all function calls.

Step 3: open a python shell  and import p26.py file into current working shell

Use help() – understand module docs

# Python standard library

- import sys
- sys.version
- sys.path
- sys.modules
- sys.argv
- sys.exit()
- sys.stdin
- sys.stdout

- help(sys)

- import os
- os.system("command")
- os.system("dir")
- os.system("ps –e|grep bash")
- os.popen("dir").read()
- os.popen("dir").readlines()
- os.listdir(".")
- os.mkdir("dirName")
- os.chdir("dirName")
- help(os)

# Activity

Step 1: Open a python shell

Step 2: Import os module (import  os)

Step 3: Display following information

- Display working directory
- Display list of files under current directory and count the total no.of files under current directory
- Display your running python shell process ID(PID)

# Activity

Write a python program

Create a new file p27.py

*File :pa.py*

```
ip1=int(input("Enter a IP1 value:"))
ip2=int(input("Enter a IP2 value:"))
total=ip1+ip2
Print("Sum of ip1 and ip2 value:{}".format(total))
```

Modify the above code with command line arguments.

# Python standard library

- import math
- import pprint
- import json
- import re
- import time
- import cProfile

- More standard module refer this URL

The Python Standard Library — Python 3.9.5 documentation

# import vs from ... import

- import brings in a whole module; you need to qualify the names by the module name (e.g., sys.argv)
- "import **from**" copies names from the module into the current module; no need to qualify them (note: these are copies, not links, to the original names)

```
from module_x import junk
junk()  # not module_x.junk()
from module_x import * # gets all top-level
                       # names from module_x
```

# Module Packages

- When using import, we can give a directory path instead of a simple name. A directory of Python code is known as a "package":
  **import dir1.dir2.module**
  **or**
  **from dir1.dir2.module import x**
  will look for a file dir1/dir2/module.py
- Note: dir1 must be within one of the directories in the PYTHONPATH
- Note: dir1 and dir2 must be simple names, not using platform-specific syntax (e.g., no C:\)



# Python pip




# What is Pip?

- **pip** is a tool for installing and managing Python packages.
- pip can be install on various operation systems: Linux, Mac, Windows, etc

# How to install <module> in winx?

- **C:\Users\User>python -m pip install fabric**
- Requirement already satisfied: fabric in c:\users\user\appdata\local\programs\p
- site-packages (2.4.0)

# Install Pip on MacOS

- Install pip on MacOS,
  using **easy_install** command and
  upgrade pip to the latest version:
- sudo **easy_install** pip
- sudo **pip install --upgrade** pip

# get-pip.py

- For mac os , easy_install has been deprecated.
- First of all download the **get-pip** file
- **curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py**
- **python get-pip.py # run this file to install pip**

# Install Pip in Ubuntu

- Install pip in Ubuntu, using apt-get package manager:
- sudo apt-get update
- sudo apt-get install python-pip
- sudo pip install --upgrade pip

# Install Pip in CentOS

- Install pip in CentOS from EPEL repository, using yum package manager:
- sudo yum update
- sudo yum install epel-release
- sudo yum install python-pip
- # CentOS-7 and higher
- sudo pip install --upgrade pip
- # CentOS-6 (the last stable version of PIP that is c ompatible with Python 2.6)
- sudo pip install pip==9.0.3

# To list all modules

- pydoc modules
- >>> help('modules')
- # print all names exported by the module
  print(dir(module))




# Lesson - 10

# Python Errors & Exceptions

1. Syntax errors
2. Logical errors (Exceptions)

# Python Logical Errors (Exceptions)

- Errors that occur at runtime (after passing the syntax test) are called e**xceptions** or logical errors.
- We can view all the built-in exceptions using the built-in local() function as follows:
- **print(dir(locals()['__builtins__']))**

# Exceptions in Python

- Python has many built-in exceptions that are raised when your program encounters an error

- When these exceptions occur, the Python interpreter stops the current process and passes it to the calling process until it is handled. If not handled, the program will crash.

# Exception block

```
try:
    code block
except Exception as eobj:
    Handle Exception
else:
        There is no Exception
finally:
        Always running
```

# Example

- var=100
- print(VAR)
- print("List of files:-")
- for v in os.listdir("."):
  print(v)

- print("Exit from script")

```
try:
        var=100
        print(VAR)
except Exception as eobj:
        print(eobj)

print("List of files:-")
for v in os.listdir("."):
        print(v)

print("Exit from script")
```

# Raising Exceptions in Python

- In Python programming, exceptions are raised when errors occur at runtime.

- We can also manually raise exceptions using the raise keyword.

- >>> try:
  ... n=input("Enter a login name:")
  ... if n != "root":
  ... **raise** NameError ("Sorry your login name is not matched")
  ... except Exception as eobj:
  ... print(eobj)
  ...
  Enter a login name:asfdsad
  Sorry your login name is not matched

# Activity

Write a python program

create a new file p28.py

Handle the exceptions in the following cases

*Case 1:*

```
port=8080
print(PORT)
```

*Case 2:*

```
F=Open("invalid file")
F.readlines()
F.close()
```

*Case 3:*

```
import openpyxl
Module Not Found
```




# Lesson - 11

# Functional Style programming

- Functional programming decomposes a problem into a set of functions.
- Ideally, functions only take inputs and produce outputs, and don’t have any internal state that affects the output produced for a given input.
- Well-known functional languages include the ML family.
- Every function's output must only depend on its input.

# Functional Vs OOPs

- Functional programming can be considered the opposite of object-oriented programming.
- Objects are little capsules containing some internal state along with a collection of method calls that let you modify this state, and programs consist of making the right set of state changes.
- Functional programming wants to avoid state changes as much as possible and works with data flowing between functions.

# List comprehension

- List comprehension offers a shorter syntax when you want to create a new list based on the values of an existing list.

- newlist = [*expression* for *item* in *iterable* ]

```
L1=[ ]  # empty list
for var in range(5):
        r=var+100
        L1.append(r)
print(L1)
```

**[ 100, 101, 102, 103, 104 ]**

Vs

```
L2=[ var+100 for var in range(5)]
print(L2)
```

**[100,102,102,103,104]**

# List comprehension with conditional statements

```
L1=[]
for var in range(15):
    if var >10:
        r=var+100
        L1.append(r)
    else:
        r=var+500
        L1.append(r)
L1
```

**[500, 501, 502, 503, 504, 505, 506, 507, 508, 509, 510, 111, 112, 113, 114]**

```
[var+100 if var >10 else var+500 for var in range(15)]
```

# List comprehension with string methods

```
s='welcome'
[var.upper() for var in s]
```

```
['W', 'E', 'L', 'C', 'O', 'M', 'E']
```

# Activity

Modify the following code into list comprehension style

```
L=[]
F=open("D:\\emp.csv")
for var in F.readlines():
    var=var.strip()
    s=var.upper()
    L.append(s)
```

# Defining an Anonymous Function With lambda

- **def** functionName() – named function
- **lambda** – unnamed function
- **λ**
- The term *lambda* comes from **lambda calculus**, a formal system of mathematical logic for expressing computation based on function abstraction and application.

# lambda

- lambda <parameter_list>: <expression>

| Component | Meaning |
|---|---|
| lambda | The keyword that introduces a lambda expression |
| <parameter_list> | An optional comma-separated list of parameter names |
| : | Punctuation that separates <parameter_list> from <expression> |
| <expression> | An expression usually involving the names in <parameter_list> |

# lambda expression

- The value of a lambda expression is a callable function like def functionname.

- It takes arguments, as specified by <parameter_list>, and returns a value, as indicated by <expression>

```
def fx(a):
    return a+100
```

Vs

```
lambda a:a+100
<function __main__.<lambda>(a)>
```

**fx(10) => 110**

**fy=lambda a:a+100**

**fy(100) => 110**

# Lambda exp and function call

f1=**lambda a,b:a+b**
f1(10,20) => 30

f2=**lambda a,b:a>b**
f2(100,5) => True

```
def f(a1):
    return a1+100
```

f3=**lambda a:f(a)**
f3(10) => 110

# Activity

Write a python program

Create a new file – p29.py

Modify the below codes into lambda style

```
def fx(a):
        return a+".log"
Files=[]
for var in ['p1','p2','p3','p4','p5']:
        r=fx(var)
        Files.append(r)
```


# Lesson - 12

# functionaltools

- map => map(function,collection)
- filter  => filter(function,collection)
- reduce => reduce(function,collection)

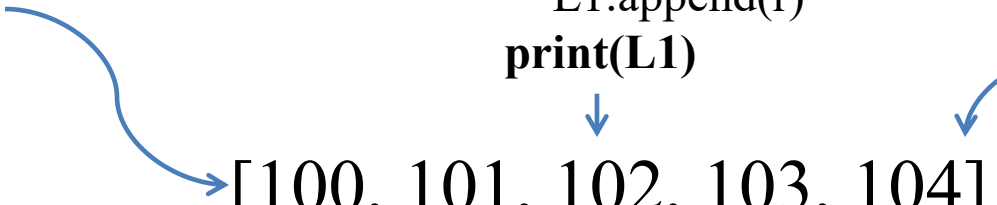
# map()

- **map(<function>,collection)**
- map() returns in iterator that yields the results of applying function <function> to each element of <iterable>.

```
L1=[] # empty list

for var in range(5):
    r=var+100
    L1.append(r)
print(L1)
```

```
L1=[]
def f1(a):
    return a+100

for var in range(5):
    r=f1(var)
    L1.append(r)
print(L1)
```

```
map(f1,range(5))
<map at 0x502cd90>
L1=list(map(f1,range(5)))
print(L1)
```



# Activity

- L1=list(map(f1,range(5)))
- Modify the above code by replacing f1 with lambda

# map

- map() – function supports arithmetic, comparison expression

- list(map(lambda a,b:a+b,[10,20,30,40],[100,200,300,400]))
  [110, 220, 330, 440]

- list(map(lambda a,b:a>b,[120,20,130,450],[100,200,300,400]))
  [True, False, False, True]




# Activity

**Predict the output**

```
def f1(a):
    if a == 'p1':
        return a+".log"
    elif a == 'p2':
        return a+".java"
    elif a == 'p3':
        return a+".py"
    else:
        return a+".txt"

list(map(lambda a:f1(a),['p1','p2','p3','p4','p5']))
```

# filter

- **filter()** allows you to select or filter items from an iterable based on evaluation of the given function.
- **filter(<function>,collection)**
- filter(<function>, <iterable>) applies function <function> to each element of <iterable> and returns an iterator that yields all items for which <function> is True.

# Filter- examples

- filter(lambda a:a>10,range(15))
- <filter at 0x507f250>
- list(filter(lambda a:a>10,range(15)))
- [11, 12, 13, 14]

- fnames=['p1.log','test.java','p1.c','p2.java','p1.java','p2.cpp']
- list(filter(lambda a:a in 'p1.c',fnames)) => **['p1.c']**

- **list(filter(lambda a:a == 'p1.c' or a == 'p1.java' or a == 'test.java',fnames))**

  **['test.java', 'p1.c', 'p1.java']**

# Activity

Write a python program

Step 1: Create a new file p30.py

Step 2: Given Depts list

**Depts=['admin','sales','crm','QA','HR','prod']**

Step 3: Filter following departments from the list sales,QA,prod

Note : use comprehension and filter function

# reduce

- reduce() - **Reducing an Iterable to a Single Value.**
- In python 3.x to use reduce(), you need to import it from a module called functools.

```
L=[10,20,30,40,50]
s=0
for var in L:
    s=s+var
print(s) =>150

from functools import  reduce
print(reduce(lambda s,var:s+var,L)) => 150
```

# Activity

Write a python program

Create a new file : p31.py file

Given **LB=[0.35,2.32,3.23,4.25,0.42]**

Calculate Sum CPU LoadBalance

Test whether the total load balance is above 10.5

if so display warning message **“High cpu utlization”**

Note: use reduce()


# Lesson - 13




# Introduction about python OOPs

- Class
- Object
- Method

# class

- Classes are used to create new user-defined data structures that contain arbitrary information about object.
- We can think class is a blueprint of the object.
- Syntax about class

**class** classname:

members

**class** is a keyword, class name is user defined.

# Class - Examples

1

```
class box:
    '''empty class'''
    pass

print(type(box))
<class 'type'>
```

2

```
class box:
    bname='Box-1'
    bsize=134

# classname.attribute
print(box.bname)
print(box.bsize)

box.bname="Box-2"
box.bsize=450

print(box.bname,box.bsize)
```

class attributes

We can overwrite class attrs




# NameError vs AttributeError

var=100

print(VAR) => **Name Error**

class Box:

    var=100

print(Box.VAR) => **Attribute Error**

# Activity

**Write a python program**

Step 1: Create a file name: p32.py

Step 2: Create a class name Employee

Step 3: Add following employee attribute details to Employee class

*Employee name (ename), Employee ID(eid)*

and initialize them with default values

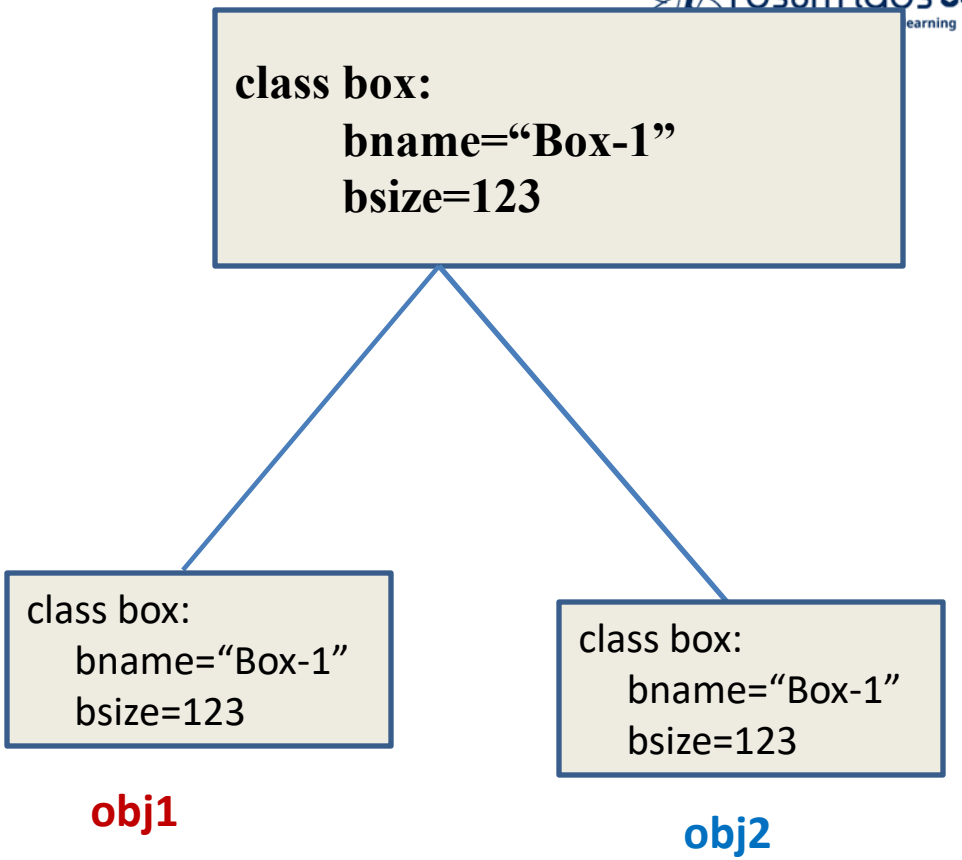
Step 4: Display employee details from outside the class




# Object

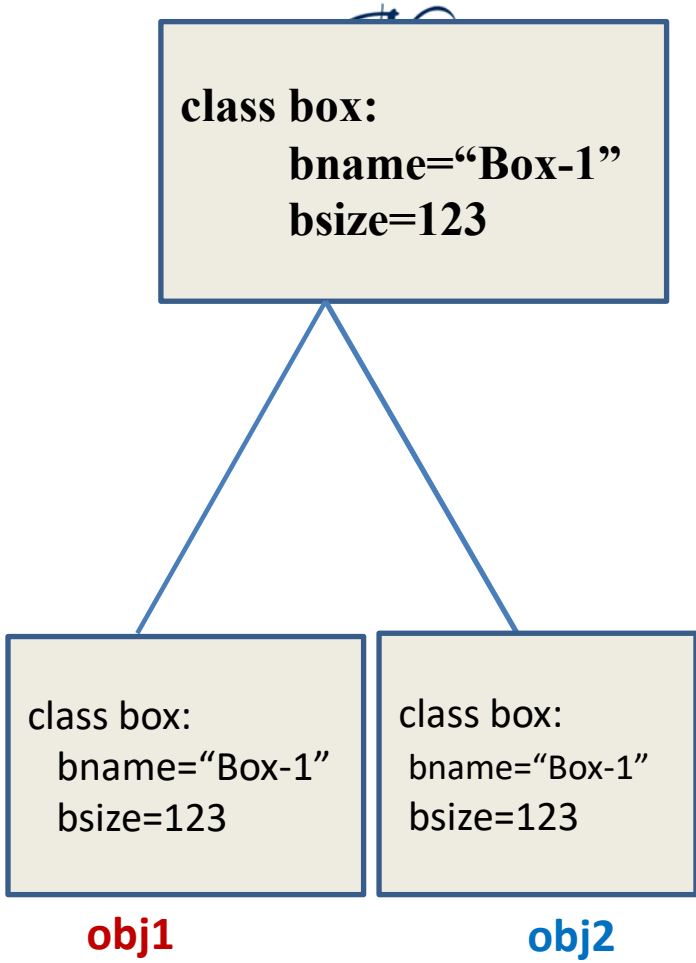
- While the class is the blueprint, an *instance* is a copy of the class with *actual* values, literally an object belonging to a specific class.
- An object (instance) is an instantiation of a class.
- From single class we can create more than one object.

```
class box:
    bname='Box-1'
    bsize=123

obj1=box()

obj1
<__main__.box at 0x507f700>

obj2=box()
obj2
<__main__.box at 0x507ffd0>
```

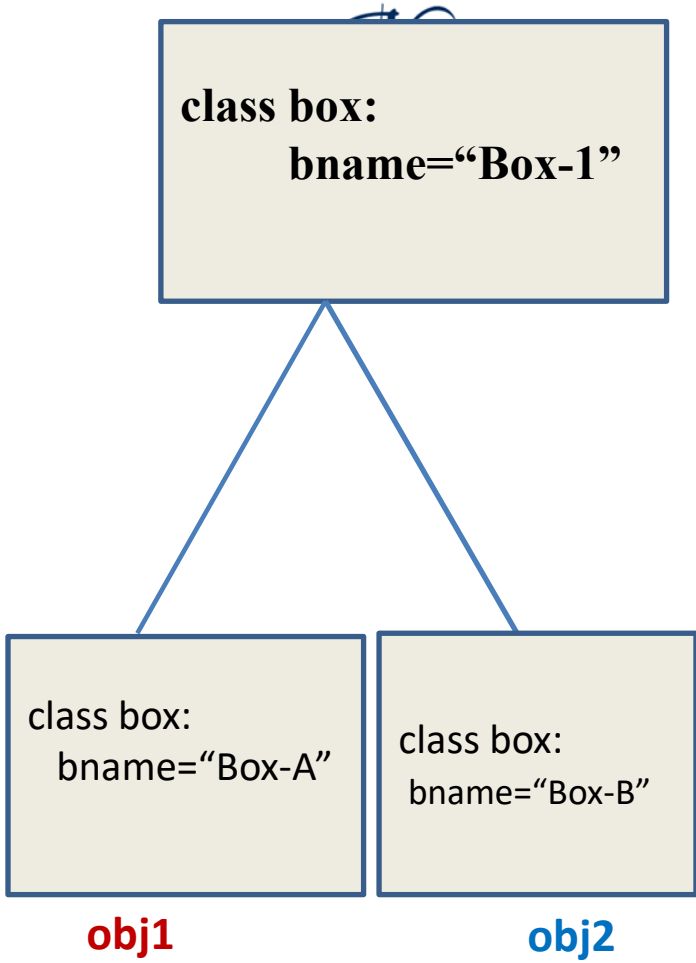


```
class box:
    bname='Box-1'
    bsize=123
```

```
obj1=box()
print(obj1.bname,obj1.bsize)

obj2=box()
print(obj2.bname,obj2.bsize)
```



```
class box:
    bname='Box-1'

obj1=box()
obj2=box()

obj1.bname="Box-A"
obj2.bname="Box-B"

print (obj1.bname, obj2.bname)
```



# Predict the output

```
class server:
    sname="default-Sever"


obj1=server()
obj1.sname="Unix"
obj2=server()
obj2.sname="Linux"
print(obj1.sname,obj2.sname) #(A)
```

*obj1.sname="sunos"*

*obj2.sname="aix"*

*print(obj1.sname,obj2.sname)* ***#(B)***

# Activity

Write a python program

Step 1: Create a new file – p33.py by modifying p32.py

Step 2: Dynamically create multiple objects and initialize them with values

Step 3:  Display each employee details (each object)




# Lesson - 14

# Methods

- Methods are functions defined inside the body of a class.
- They are used to define the behaviors of an object.

```
def f1():
    print("Hello")
print(type(f1))  => <class 'function'>


class Cname:
    def f1():
        print("Hello")
 obj=Cname()
 print(type(obj.f1))   => <class 'method'>
```

# TypeError

```
def fx():
    print("Hello")

fx()
fx(10) # TypeError: fx() takes 0 positional arguments but 1 was given

class  Cname:
     def fx():
          print("Hello")

obj=Cname()

obj.fx()  # TypeError: fx() takes 0 positional arguments but 1 was given
```

```
class cname:
    def method1(self):
        print("MethodCall")

obj1=cname()
obj2=cname()
obj3=cname()
obj1.method1() # method1(obj1)
obj2.method1() # method1(obj2)
obj3.method1() # method1(obj3)
```

```
class box:
    bname="defaultName"
    def f1(self,a1):
        self.bname=a1
        print("This is initialized block")
    def f2(self):
        print("Box Name:{}".format(self.bname))
```

```
obj1=box()
obj1.f1("Box-1")
obj1.f2()

obj2=box()
obj2.f1("Box-2")
obj2.f2()
```

# Activity

Write a python program

Step 1: create a new file p34.py by modifying p33.py

Step 2: Create a 3 methods:

getdata() – To initialize employee details

display() – To display employee details

update() – To update employee working department

# Private member

**Class attribute - starts with double underscore__**

```
class One:
        fname="p1.log"          # public variable
      __passwd="welcome"        # user defined private variable

obj=One()
obj.fname   => p1.log
obj.__passwd  => Attribute Error
One.__passwd => Attribute Error
```

# How to access private member?

Class attribute - starts with double underscore__

```
class One:
        fname="p1.log"          # public variable
      __passwd="welcome"        # user defined private variable
       def f1(self):
              print("file name:{}".format(self.fname))
              print("Password:{}".format(self.__passwd))

obj=One()
obj.f1()
```

# Activity

Write a python program

Step 1: create a new file p35.py by modifying p34.py file

Step 2: replace the existing class attribute as private variables


# Lesson - 15

# CASE STUDIES



# Thank you