Python Basic Concepts and Programming

Module 1
3rd Semester MCA

Python programming language

- Python is an example of a high-level language as C, C++, Java.
- Also low-level languages, i.e, "machine lang" or "assembly lang."

- There are multiple IDEs (Integrated Development Environment) available for working with Python. Some of them are PyCharm, LiClipse, IDLE, Jupyter, Spyder etc.

- The basic Python can be downloaded from the link: https://www.python.org/downloads/

- Python has set of libraries for various purposes like large-scale data processing, predictive analytics, scientific computing etc.

- Based on one's need, the required packages can be downloaded. Free open source : Anaconda

https://anaconda.org/anaconda/python

Interpreter and compiler

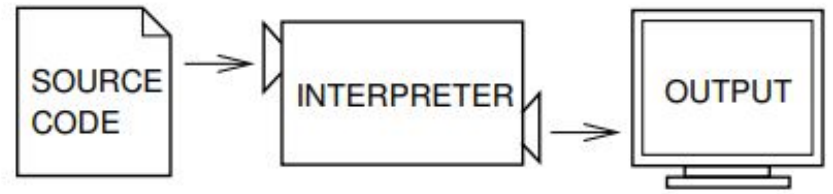


Figure 1.1: An interpreter processes the program a little at a time, alternately reading lines and performing computations.

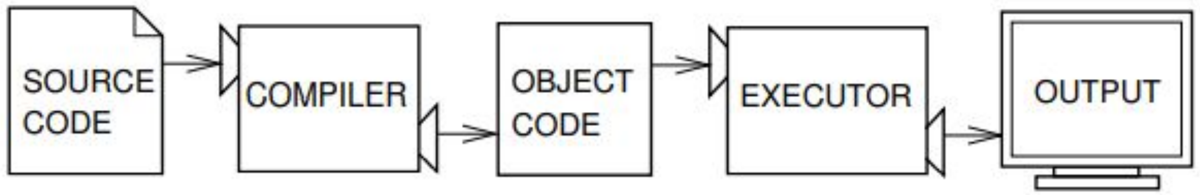


Figure 1.2: A compiler translates source code into object code, which is run by a hardware executor.

- Two kinds of programs process high-level languages into low-level languages: interpreters and compilers
- A compiler reads the program.
- Translates it completely before the program starts running.
- High-level program is called the source code. Translated program is called the object code or the executable.

Interactive mode and Script mode.

- There are two ways to use the interpreter:
- In **interactive** mode, you type Python programs and the interpreter displays the result:

```
>>> 1 + 1
2
>>> print(4)
4
>>> message = 'And now for something completely different'
>>> n = 17
>>> pi = 3.1415926535897931
```

- Alternatively, you can store code in a file and use the interpreter to execute the contents of the file, which is called a **script**. By convention, Python scripts have names that end with .py.

SOME FUNDAMENTALS

Whitespace is significant in Python. Where other languages may use {} or (), Python uses indentation to denote code blocks.

Comments:

- Single-line comments denoted by #.
- Multi-line comments begin and end with three "".
- Typically, multi-line comments are meant for documentation.
- Comments should express information that cannot be exp

PYTHON TYPING

- Python is a strongly, dynamically typed language.
- Strong Typing
  - Obviously, Python isn't performing static type checking, but it does prevent mixing operations between mismatched types.
  - Explicit conversions are required in order to mix types.
  - Example: 2 + "four" not going to fly
- Dynamic Typing
  - All type checking is done at runtime.
  - No need to declare a variable or give it a type before use.

Let's start by looking at Python's built-in data types.

NUMERIC TYPES

The subtypes are int, long, float and complex.
Their respective constructors are int(), long(), float(), and complex().
All numeric types, except complex, support the typical numeric operations you'd expect to find (a list is available here). Mixed arithmetic is supported, with the "narrower" type widened to that of the other. The same rule is used for mixed comparisons.

* Numeric
    int: equivalent to C's long int in 2.x but unlimited in 3.x.
    float: equivalent to C's doubles.
    long: unlimited in 2.x and unavailable in 3.x.
    complex: complex numbers..

Supported operations include constructors (i.e. int(3)), arithmetic, negation, modulus, absolute value, exponentiation, etc.

SEQUENCE DATA TYPES

- There are seven sequence subtypes: strings, Unicode strings, lists, tuples, bytearrays, buffers, and xrange objects.
- All data types support arrays of objects but with varying limitations.
- The most commonly used sequence data types are strings, lists, and tuples. The xrange data type finds common use in the construction of enumeration-controlled loops. The others are used less commonly.

range() – takes more memory as it keeps the entire lists of elements in memory. xrange functionality is implemented range() in 3.x

xrange() – takes less memory as it keeps only one element at a time in memory. Exist only in 2.x

SEQUENCE TYPES: UNICODE STRINGS

Unicode strings can be used to store
and manipulate Unicode data.
As simple as creating a normal string (just put a 'u' on it!).
Use Unicode-Escape encoding for special
characters.
Also has a raw mode, use 'ur' as a prefix.
To translate to a regular string, use the
encode() method.
To translate from a regular string to
Unicode, use the unicode() function.

```
myunicodestr1 = u"Hi Class!"
myunicodestr2 =
"Hi\u0020Class!“
print myunicodestrl, myunicodestr2
newunicode =u'\xe4\xf6\xfc'
print newunicode
newstr = newunicode.encode('utf-8')
print newstr
print unicode (newstr, 'utf-8')
```

SEQUENCE TYPES: LISTS

Lists are an incredibly useful compound data type.

Lists can be initialized by the constructor, or with a bracket structure containing or more elements.

Lists are mutable - it is possible to change their contents. They contain the additional mutable operations.

Lists are nestable.

Feel free to create lists of lists of lists

```
mylist = [42, 'apple', ufunicode apple', 5234656]
print mylist
mylist [2] = 'banana'
print mylist
mylist [3] = [['iteml', 'item2 ['item3', 'item4']]
print mylist
mylist.sort ()
print mylist
print mylist.pop()
mynewlist = [x*2 for x in range (0,5)]
print mynewlist
```

```
Output:
(42. 'apple', u'unicode apple, 5234656)
(42, 'apple', banana, 5234656)
(42, apple, banana', [[item1", item21, [item, tem
[42, [[item1", "item2] [item3, item4] apple banana)
```

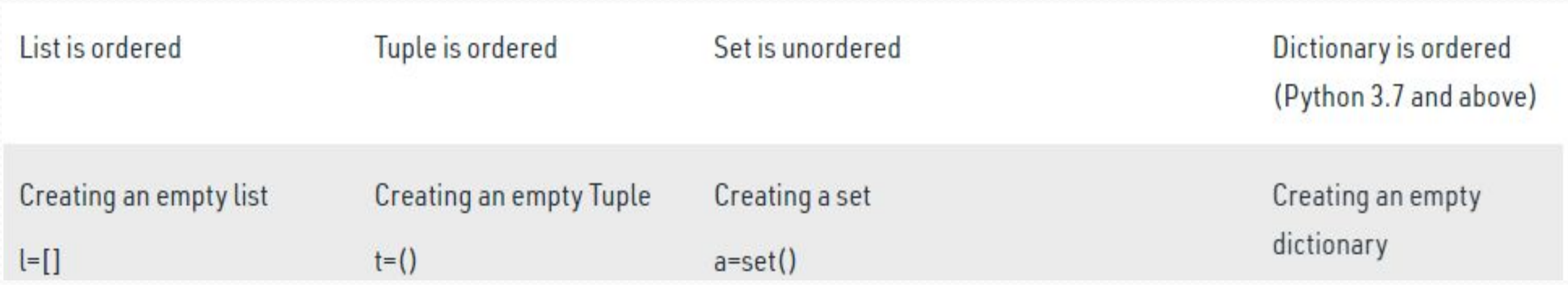
SEQUENCE TYPES: STRINGS

- Created by simply enclosing characters in either single-or double-quotes.
- It's enough to simply assign the string to a variable.
- Strings are immutable.
- There are a tremendous amount of built-in string-methods (listed here)

mystring "Hi, I'm a string!“

List , Tuple, Set, Dictionary

List	**Tuple**	**Set**	**Dictionary**
List is a non-homogeneous data structure that stores the elements in single row and multiple rows and columns	Tuple is also a non-homogeneous data structure that stores single row and multiple rows and columns	Set data structure is also non-homogeneous data structure but stores in single row	Dictionary is also a non-homogeneous data structure which stores key value pairs
List can be represented by [ ]	Tuple can be represented by ( )	Set can be represented by { }	Dictionary can be represented by { }
List allows duplicate elements	Tuple allows duplicate elements	Set will not allow duplicate elements	Set will not allow duplicate elements and dictionary doesn’t allow duplicate keys.
List can use nested among all	Tuple can use nested among all	Set can use nested among all	Dictionary can use nested among all
Example: [1, 2, 3, 4, 5]	Example: (1, 2, 3, 4, 5)	Example: {1, 2, 3, 4, 5}	Example: {1, 2, 3, 4, 5}
List can be created using **list()** function	Tuple can be created using **tuple()** function.	Set can be created using **set()** function	Dictionary can be created using **dict()** function.
List is mutable i.e we can make any changes in list.	Tuple is immutable i.e we can not make any changes in tuple	Set is mutable i.e we can make any changes in set. But elements are not duplicated.	Dictionary is mutable. But Keys are not duplicated.



List is ordered	Tuple is ordered	Set is unordered	Dictionary is ordered (Python 3.7 and above)
Creating an empty list l=[]	Creating an empty Tuple t=()	Creating a set a=set()	Creating an empty dictionary

```
T=[]          t=()          b=set(a)          d={}
```

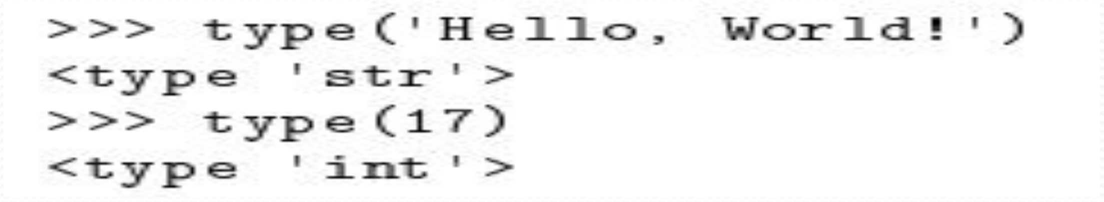
Variables, Expressions and Statements

- A variable is a named place in the memory.
- Stores data.
- Data is used using the variable "name".
- An assignment statement creates new variables.
- The type of it will be decided by the value assigned to it.

Values and types :

- A value is basic things, like a letter or a number.

  i.e, 1, 2, and 'Hello, World!'.
- These values belong to different types:
  - 2 is an integer,
  - 'Hello, World!' is a string,



```
>>> type('Hello, World!')
<type 'str'>
>>> type(17)
<type 'int'>
```

Cont.

- **Rules to follow when naming the variables**.
  - Variable names can contain letters, numbers, and the underscore.
  - Variable names cannot contain spaces and other special characters.
  - Variable names cannot start with a number.
  - Case matters—for instance, temp and Temp are different.
  - Keywords cannot be used as a variable name.
- Valid variable names are: Spam, eggs, spam23 ,_speed
- Underscore character (_) can appear in a name. Ex: *my_first_variable* .
- As Python is case-sensitive, variable name *sample* is different from *SAMPLE* .
- invalid variable

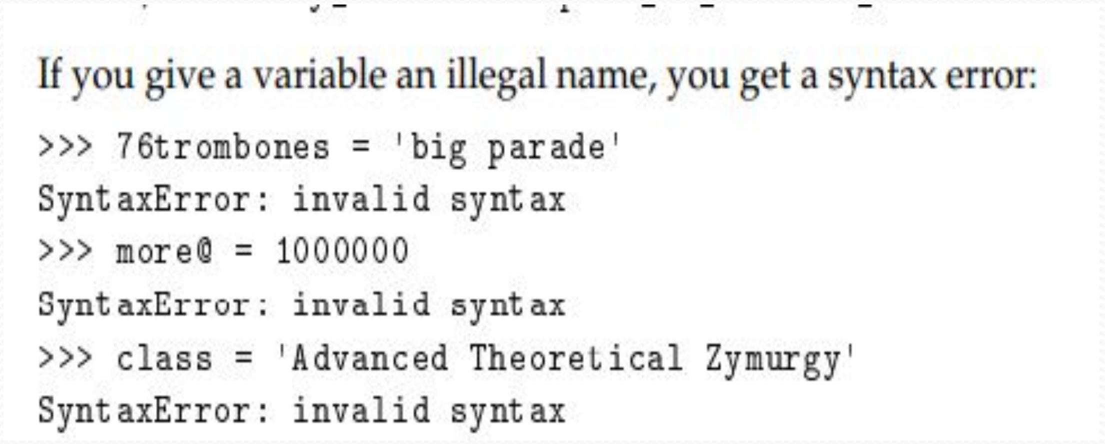
  >>> 76trombones = 'big parade‘ >>> more@ = 1000000 >>> class = 'Advanced'

Variables

```
>>> message = 'And now for something completely different'
>>> n = 17
>>> pi = 3.1415926535897932
```

The type of a variable is the type of the value it refers to.

```
>>> type(message)
<type 'str'>
>>> type(n)
<type 'int'>
>>> type(pi)
<type 'float'>
```



If you give a variable an illegal name, you get a syntax error:

```
>>> 76trombones = 'big parade'
SyntaxError: invalid syntax
>>> more@ = 1000000
SyntaxError: invalid syntax
>>> class = 'Advanced Theoretical Zymurgy'
SyntaxError: invalid syntax
```

Python 2 has 31 keywords:

and
del
from
not
while
as
elif
global
or
In Python 3, exec is no longer a keyword, but nonlocal is
with
assert
else

```
import datetime
   x = datetime.datetime.now()
   print(x)
****************************

from datetime import time
   x = time(hour=15)
   print(x)
****************************

import calendar as c
print(c.month_name[1])
****************************

def myfunction():
     global x
     x = "hello“
****************************

   x = -1
   if x < 0:
```

- imports : specific module
- from : It imports specific parts of a module
- assert : used when debugging code
  - We can write a message to be written if the code returns False
- pass :
  - placeholder for future code.
  - When the pass statement is executed,
  - Nothing happens,
  - Avoid getting an error when empty code is not allowed.
  - Empty code is not allowed in:
    - loops,
    - function definitions,
    - class definitions,
    - or in if statements.

Statements and Expressions

- A statement is a unit of code that the Python interpreter can execute.
- *assignment statement*
  - It consists of an expression on the right-hand side and a variable to store the result
  - multiple assignments : a, b, c = 2, “B”, 3.5
- ***print statement***
  - print is a function.
  - Takes string or variable as a argument to display.

```
>>> x=5      #assignment statement
>>> x=5+3    #assignment statement
>>> print(x) #printing statement
```

  - Optional arguments with print statement:
    - ***sep :*** separator
    - ***end :*** advance to the next line

```
print("a","b","c","d",sep=";")
a;b;c;d
```

Expressions

- combination of values, variables, and operators
- legal expressions

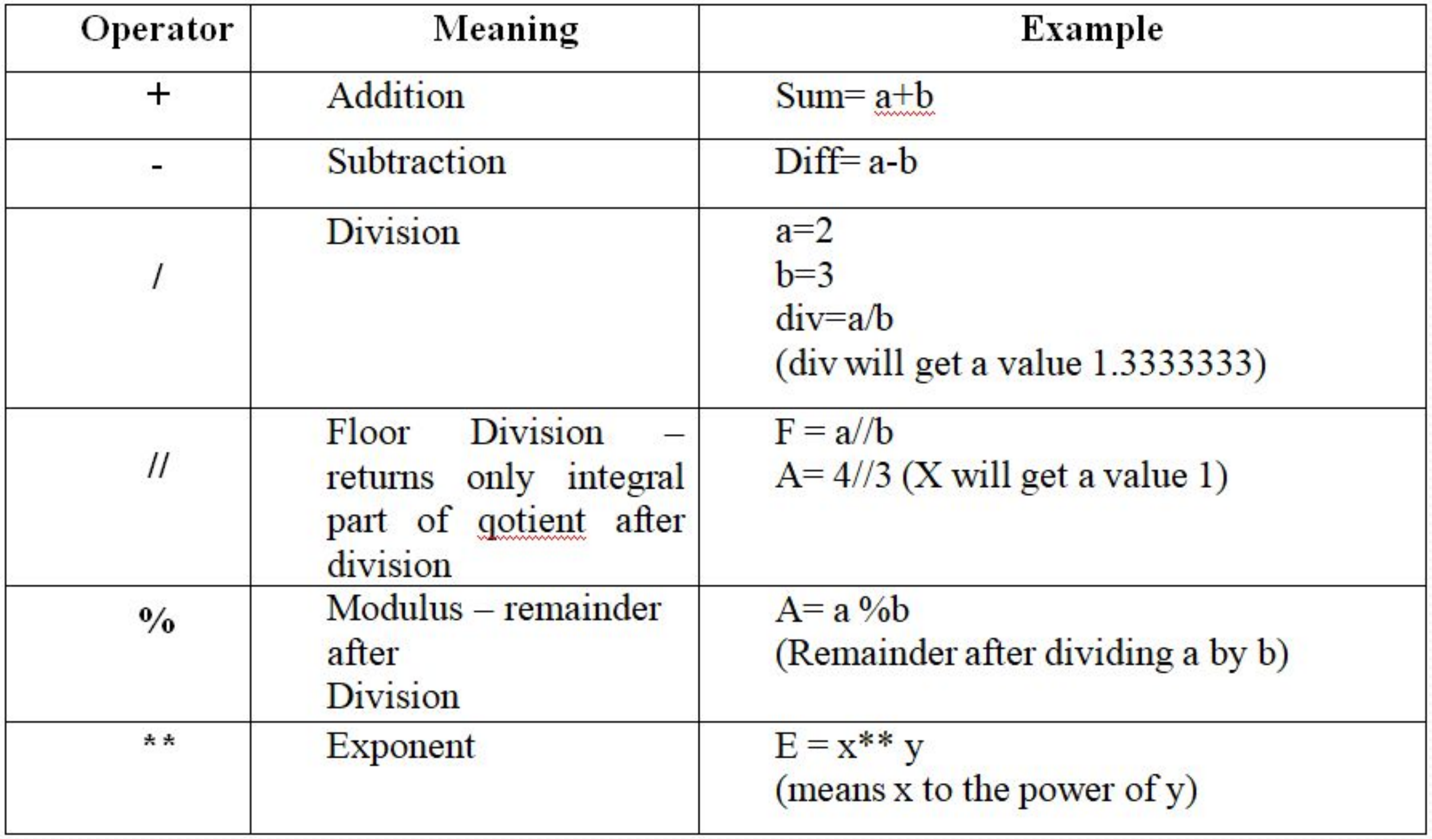
```
>>> x=5
>>> x+1
6
```

```
print("A“)
print("B")
print("C", end=" ")
print("E")

O/P:
A
B
C E
```

Operators, Precedence and Associativity

- It represent computations like addition and multiplication.
- The values the operator is applied to are called operands.



Operator	Meaning	Example
+	Addition	Sum= a+b
-	Subtraction	Diff= a-b

/	Division	a=2
		b=3
		div=a/b
		(div will get a value 1.3333333)

//	Floor Division – returns only integral part of qotient after division	F = a//b A= 4//3 (X will get a value 1)
%	Modulus – remainder after Division	A= a %b (Remainder after dividing a by b)
**	Exponent	E = x** y (means x to the power of y)

- *Relational* or *Comparison* Operators:
  - Less than, Greater than etc, between two operands.
  - It return a Boolean value either True or False.
- *Assignment* Operators:
  - Assignment operator =, is used for assigning values to variables.

statements	Compound statement
x=x+y	x+=y
y=y//2	y//=2

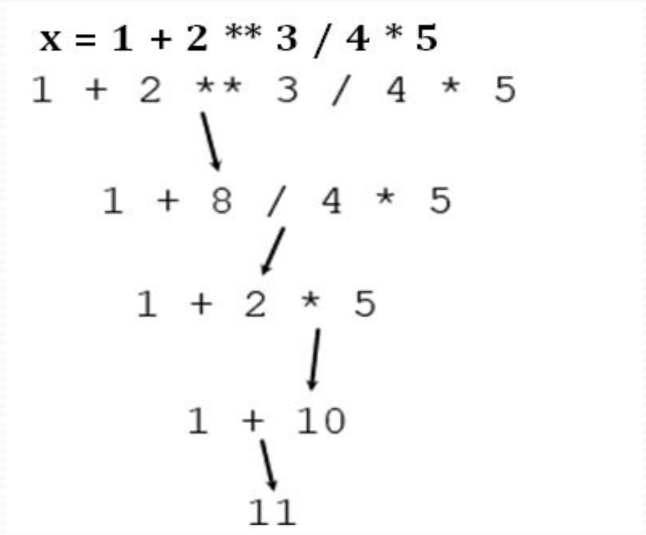
- *Logical* Operators:
  - *and, or, not* are used for comparing or negating the logical values
  - **And:**

    Ex1 :>>> x=5
    ```
    >>> x>0 and x<10
    True
    ```
    Ex2: >>> x= -5
    ```
    >>> x>0 and x<10
    False
    ```
  - **Or :**
    ```
    >>> n=2
    >>>n%2==0r n%3==0
    True
    ```
  - **Not:**
    ```
    >>> x=5
    >>> x> 0 and x<10
    True
    >>> not x
    False
    ```

Precedence and Associativity (Order of operations)

- Operators are special symbols that represent computations like addition and multiplication.
- The values the operator is applied to are called operands.
- The operators +, -, *, / and ** perform addition, subtraction, multiplication, division and exponentiation, as in the following examples:

```
20+32
hour-1
hour*60+minute
minute/60
5**2
(5+9)*(15-7)
```



Order of operations: PEMDAS easy to remember rules:

- **Parentheses** have the highest precedence and can be used to force an expression to evaluate in the order you want
- **Exponentiation** has the next highest precedence
- **Multiplication** and **Division** have the same precedence, which is higher than **Addition** and **Subtraction**,
- Operators with the same precedence are evaluated from left to right (except exponentiation).

Data Types, Indentation, Comments

- Basic data types of Python are
  - Numbers
  - Boolean
  - Strings
  - list
  - tuple
  - dictionary
  - None

- **Number**:
  - Integers, floating point numbers and complex numbers
  - They are defined as int, float and complex class in Python
- **Boolean**
  - conditional statements
  - Boolean value is, either True or False
- **Strings**
  - A sequence of one or more characters
  - Include letters, numbers, and other types of characters.
  - Multiline strings
    - denoted using triple quotes, ''' or " " "

```
>>> s = 'This is single quote string'
>>> s = "This is double quote string"
>>> s = '''This
              is Multiline
                        string'''
```

- **Lists**
  - All the items (elements) inside square brackets [ ]
  - Separated by commas
- **Tuple**
  - An ordered collection of Python objects.
  - Immutable , can't be modified after it's created
  - we can represent tuples using parentheses ( )

- **Dictionary**
  - An unordered collection of data values.
  - consists of key-value pair
  - Key-value is provided within the dictionary to form it more optimized.
  - Each key-value pair is separated by a colon(:),whereas each key's separated by a 'comma'.

    Example: Dict1 = {1 : 'Hello' , 2 : 5.5, 3 : 'World' }
- **None**
  - None is another special data type in Python.
  - None is frequently used to represent the absence of a value.
  - For example, >>> money = None

- **Indentation**
- Programs get structured through indentation
  - In Python it is a requirement and not a matter of style.
  - Any statements written under another statement with the same indentation is interpreted to belong to the same code block
  - Nested statements need to be indented further to the right.
- **Comments**
  - Ex1. #This is a single-line comment
  - Ex2. ''' This is a multiline
  - comment '''

Reading Input

- Built-in function called *input*
- Gets input from the keyboard
- \n at the end of the prompt represents a newline

```
>>> inp = input()
Welcome to world of python
>>> print(inp)
Welcome to world of python
>>>x=input('Please enter some text:\n')
Please enter some text:
Roopa
>>> print(x)
Roopa
```

- user to type an integer,
- try to convert the return value to int using the int() function

```
>>> x=int(input('enter number\n'))
enter number
12
```

Print Output

- **Format operator**
  - The following example uses “%d” to format an integer, “%g” to format a floating point number, and “%s” to format a string:

```
camels = 42
'%d' % camels
 'I have spotted %d camels.' % camels'I have spotted 42 camels.'
```

- **Format function :** string formatting methods string formatting methods
  - positional_argument :
    - It can be integers, floating point numeric constants, strings, characters and even variables.
  - keyword_argument :
    - A variable storing some value, which is passed as parameter.
- Positional arguments are placed in order :
  - Can Reverse the index nnumbers
  - If arguments not specified

```
>>>print("{0}    college{1}  department".format("SDIT",“MCA"))
SDIT college MCA department
>>>print("{ }     college{ }  department".format("SDIT",“MCA"))
SDIT college MCA department
>>>print(“MCA  department    {0} ‘A’ .format(“1", college="SDIT"))
MCA department  1’A’ section SDIT
```

Formatted String Literals

```
import math
print(f'The value of pi is approximately {math.pi:.3f}.')

bugs = 'roaches'
count = 13
area = 'living room'
print(f'Debugging {bugs=} {count=} {area=}')

#Use "+" to always indicate if the number is positive or negative:
txt = "The temperature is between {:+} and {:+} degrees celsius."
print(txt.format(-3, 7))

#Use "_" to add a underscore character as a thousand separator:
txt = "The universe is {:_} years old."
print(txt.format(13800000000))
```

https://www.w3schools.com/python/ref_string_format.asp

String format() Method

print('We are the {} who say "{}!"'.format('knights', 'Ni'))

Positional argument :

print('{0} and {1}'.format('spam', 'eggs'))

print('{1} and {0}'.format('spam', 'eggs'))

print('{} and {}'.format('spam', 'eggs'))

Keyword argument:

print('This {food} is {adjective}.'.format(food='spam', adjective='absolutely horrible'))

Positional and keyword arguments

print('The story of {0}, {1}, and {other}.'.format('Bill', 'Manfred',other='Georg'))

Type Conversions, The type( ) Function and Is Operator

- **Type Conversions**
  - int can convert floating-point values to integers,

  but it doesn't round off; it chops off the fraction part.
  - float converts integers and strings to floating-point numbers.
  - str converts its argument to a string.
- **The type( ) Function**
  - *type* function is called to know the datatype of the value.
  - The expression in parenthesis is called the argument of the function.
  - The argument is a value or variable that we are passing into the function as input to the function.

```
>>> type(33)
<class 'int'>
```

- **Is Operator**
  - If we run these assignment statements:

```
>>> a = 'banana'
>>> b = 'banana'
>>> a is b
True
```

- When two variables are referring to same object, they are called as ***identical*** objects.
- When two variables are referring to different objects, but contain a same value, they are known as ***equivalent*** **objects.**

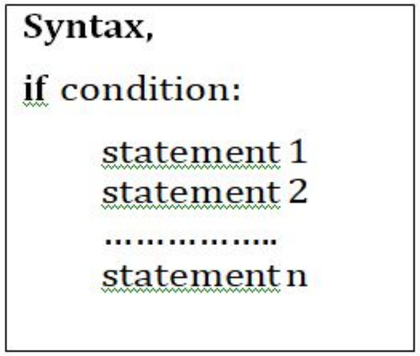
```
>>>s1=input("Enter a string:")
>>>s2= input("Enter a string:")
>>>s1 is s2          #check s1 and s2 are identical
False
>>>s1 == s2          #check s1 and s2 are equivalent
True
```

Here s1 and s2 are equivalent, but not identical

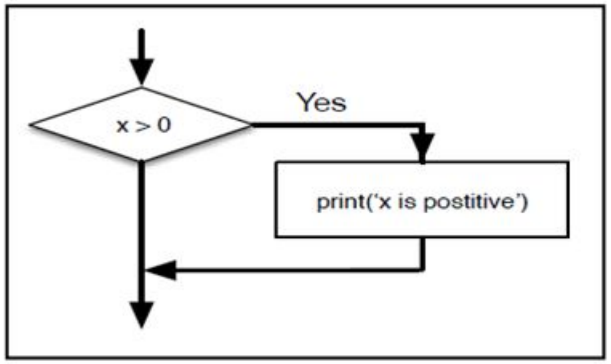
```
x = str(3)    # x will be '3'
y = int(3)    # y will be 3
z = float(3)  # z will be 3.0
```

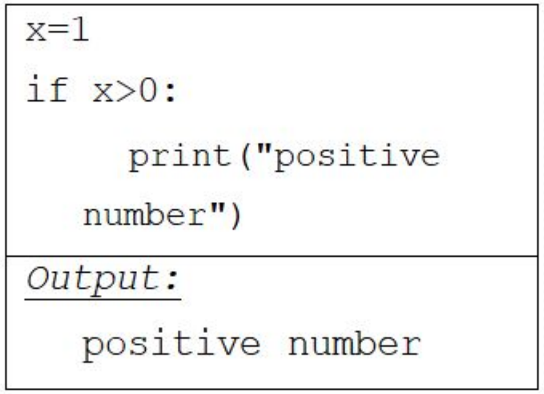
Control Flow Statements

The if Decision Control Flow Statement



```
Syntax,
if condition:
        statement 1
        statement 2
        ………………
        statement n
```





```
x=1
if x>0:
    print("positive
  number")
```

Output:

```
  positive number
```

if…else Decision Control Flow Statement (alternative execution)

Syntax:

```
if condition :
        statements
else :
      statements
```

Example:

```
x=6
if  x%2 == 0:
  print('x is even')
else :
        print('x      is
      odd')
```

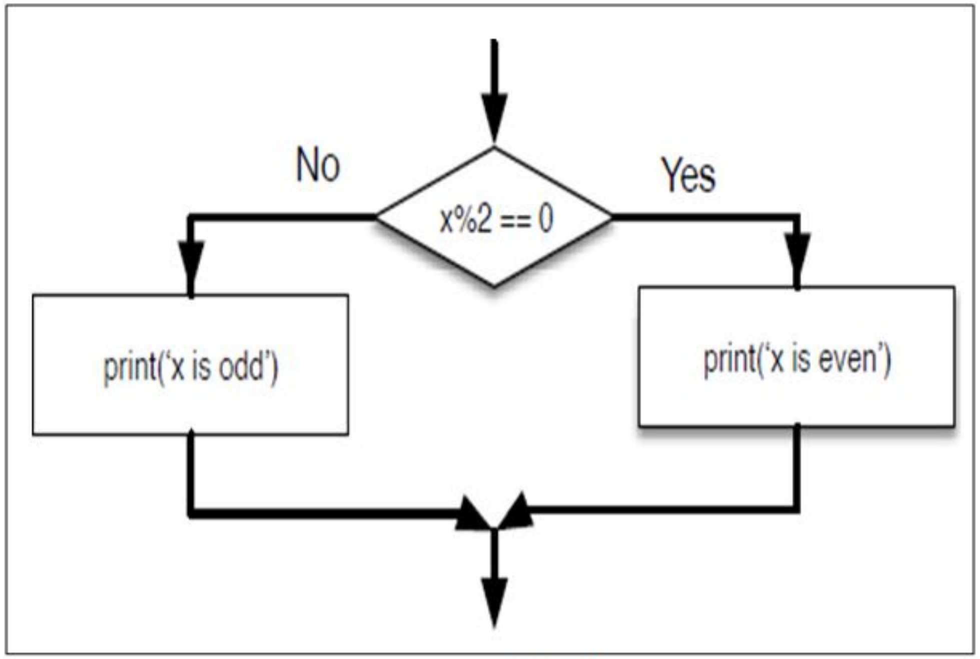


Figure : if-Then-Else Logic

The if…elif…else Decision Control Statement (*Chained conditionals*)

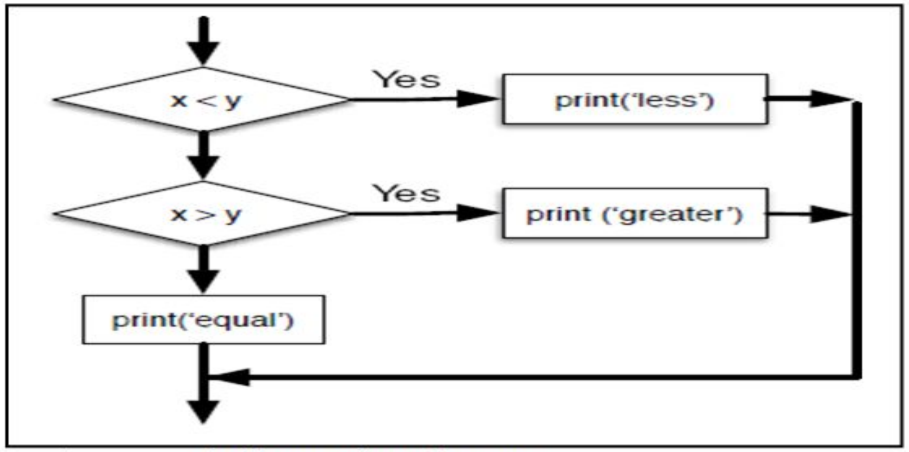


Figure : If-Then-ElseIf Logic

Syntax

```
if condition1:
        Statement
elif condition2:
        Statement
....................
elif condition_n:
        Statement
else:
       Statement
```

Example

```
x=1
y=6
if x < y:
      print('x is less than y')
elif x > y:
      print('x is greater than y')
else:
      print('x and y are equal')

Output:
 X is less than y
```

Nested if Statement

Example

```
x=3 y=4
if x == y:
print('x and y are equal') else:
if x < y:
print('x is less than y') else:
print('x is greater than y')
```

Output:

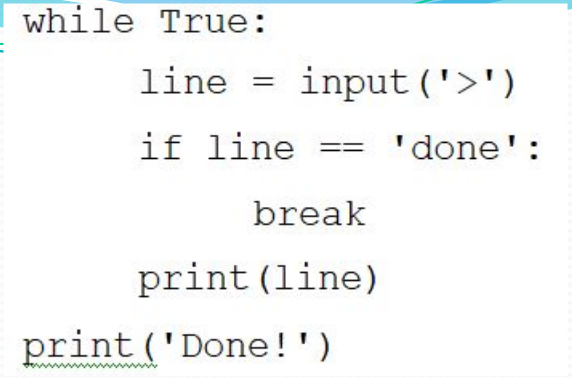
```
x is less than y
```

ITERATION

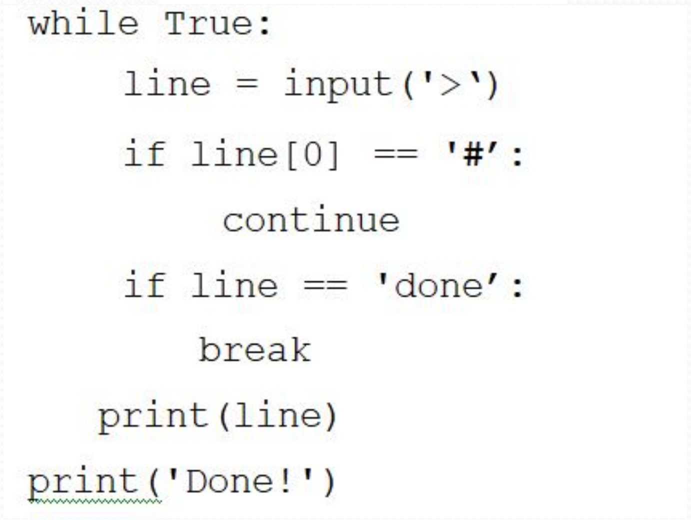
- **The while statement**
  - The syntax of while loop :

```
while condition:
      block_of_statements
statements after the loop
```

EX:
```
i = 1
while i<=5:
      print(i) i= i+1
print(" printing is done")
```



```
while True:
        line = input('>')
        if line == 'done':
                break
        print(line)
print('Done!')
```



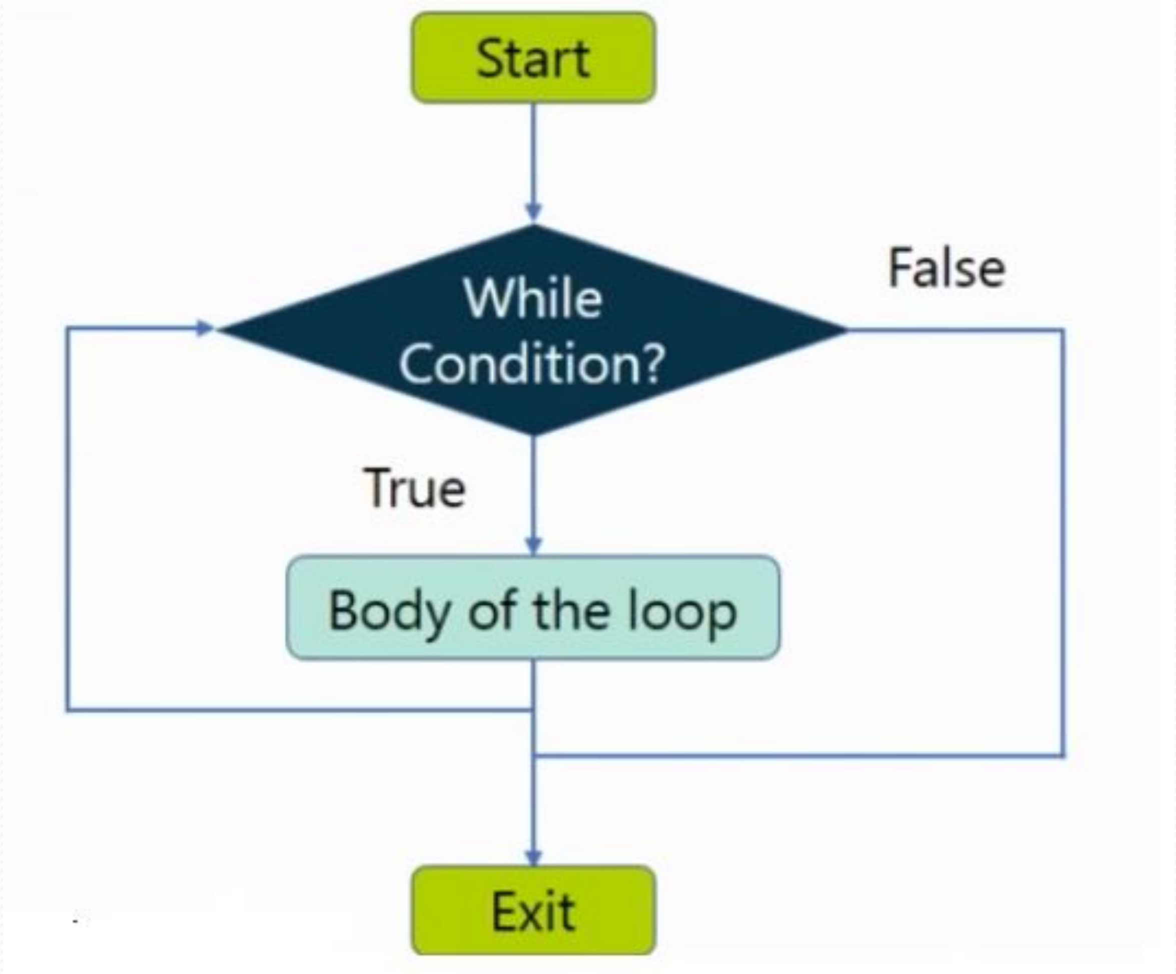
```
while True:
       line = input('>')
       if line[0] == '#':
            continue
       if line == 'done':
            break
     print(line)
print('Done!')
```

Infinite loops with *break*

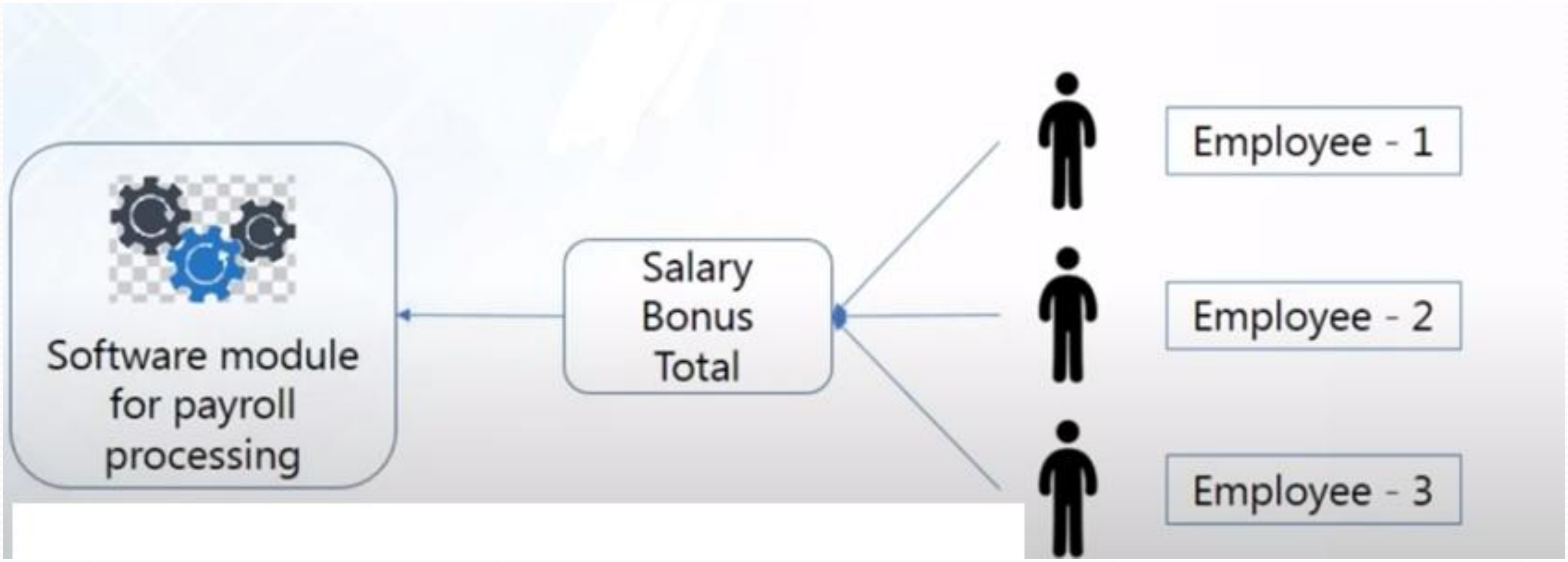
- The *break* statement can be used to break out of a *for* or *while* loop before the loop is finished.

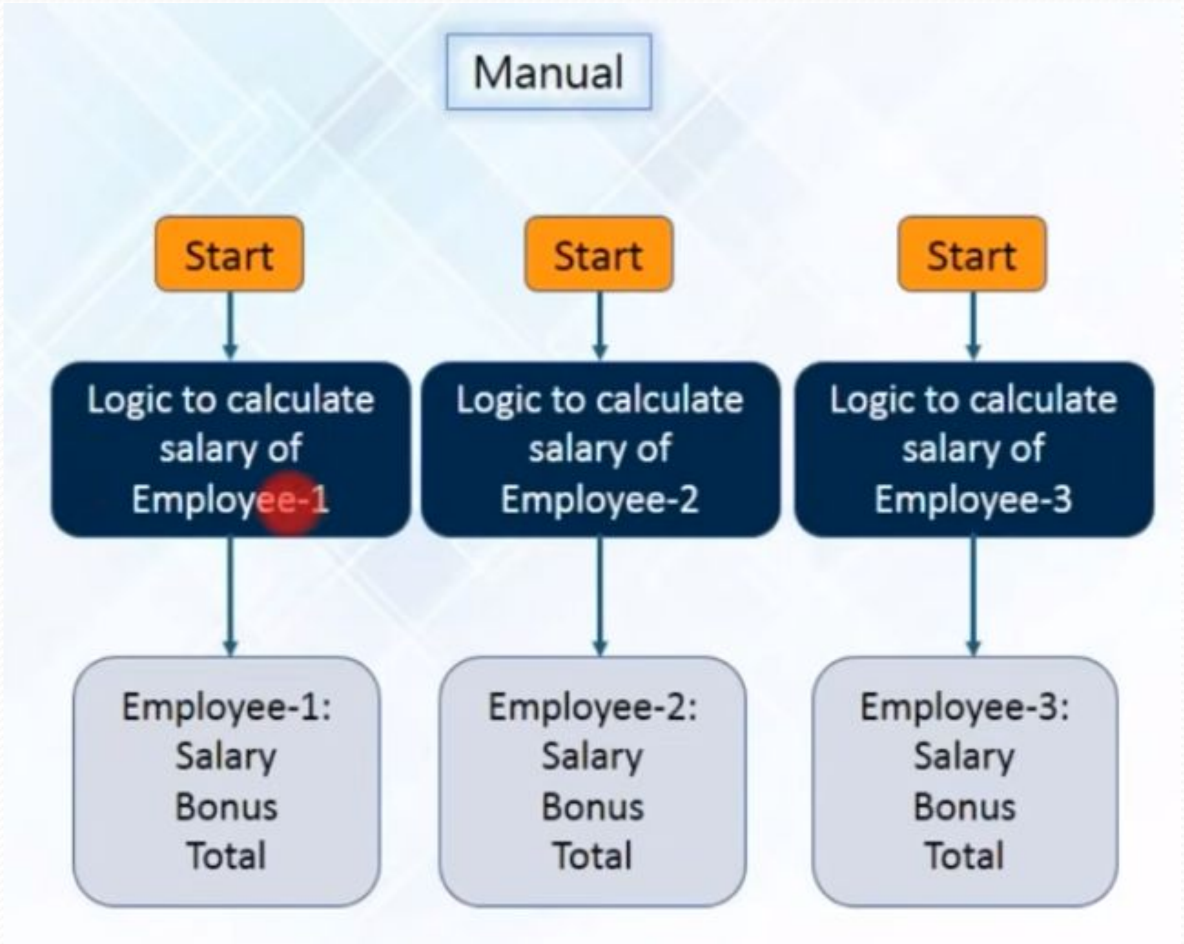
Finishing iterations with *continue*

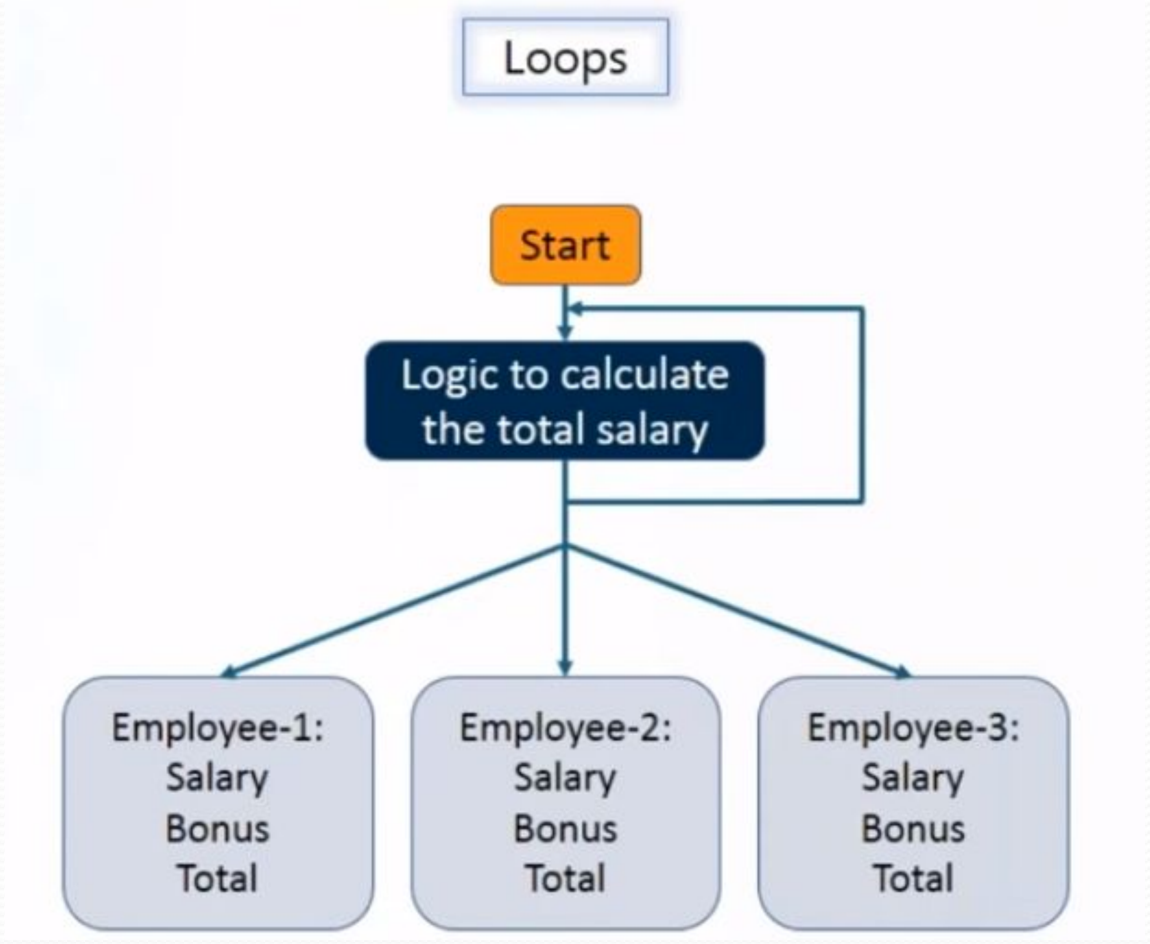
- the continue statement skips to the next iteration without finishing the body of the loop for the current iteration.


Start
While Condition?
False
True
Body of the loop
Exit

Example pay roll




Manual
Start
Start
Start
Logic to calculate salary of Employee-1
Logic to calculate salary of Employee-2
Logic to calculate salary of Employee-3
Employee-1: Salary Bonus Total
Employee-2: Salary Bonus Total
Employee-3: Salary Bonus Total


Loops
Start
Logic to calculate the total salary
Employee-1: Salary Bonus Total
Employee-2: Salary Bonus Total
Employee-3: Salary Bonus Total

Definite loops using for

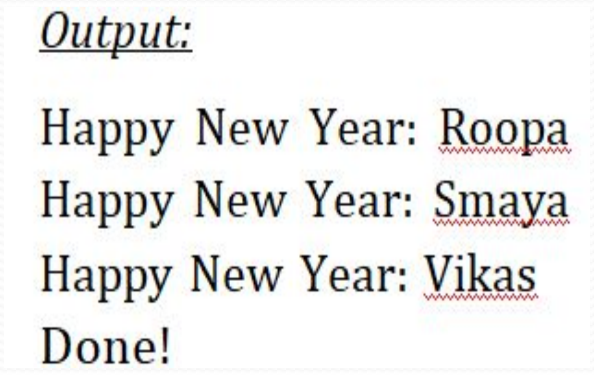
- *for loop with sequence*
- *for loop with range( ) function*

Syntax of ***for* loop with sequence:**

```
for var in list/sequence :
    statements to be repeated
```

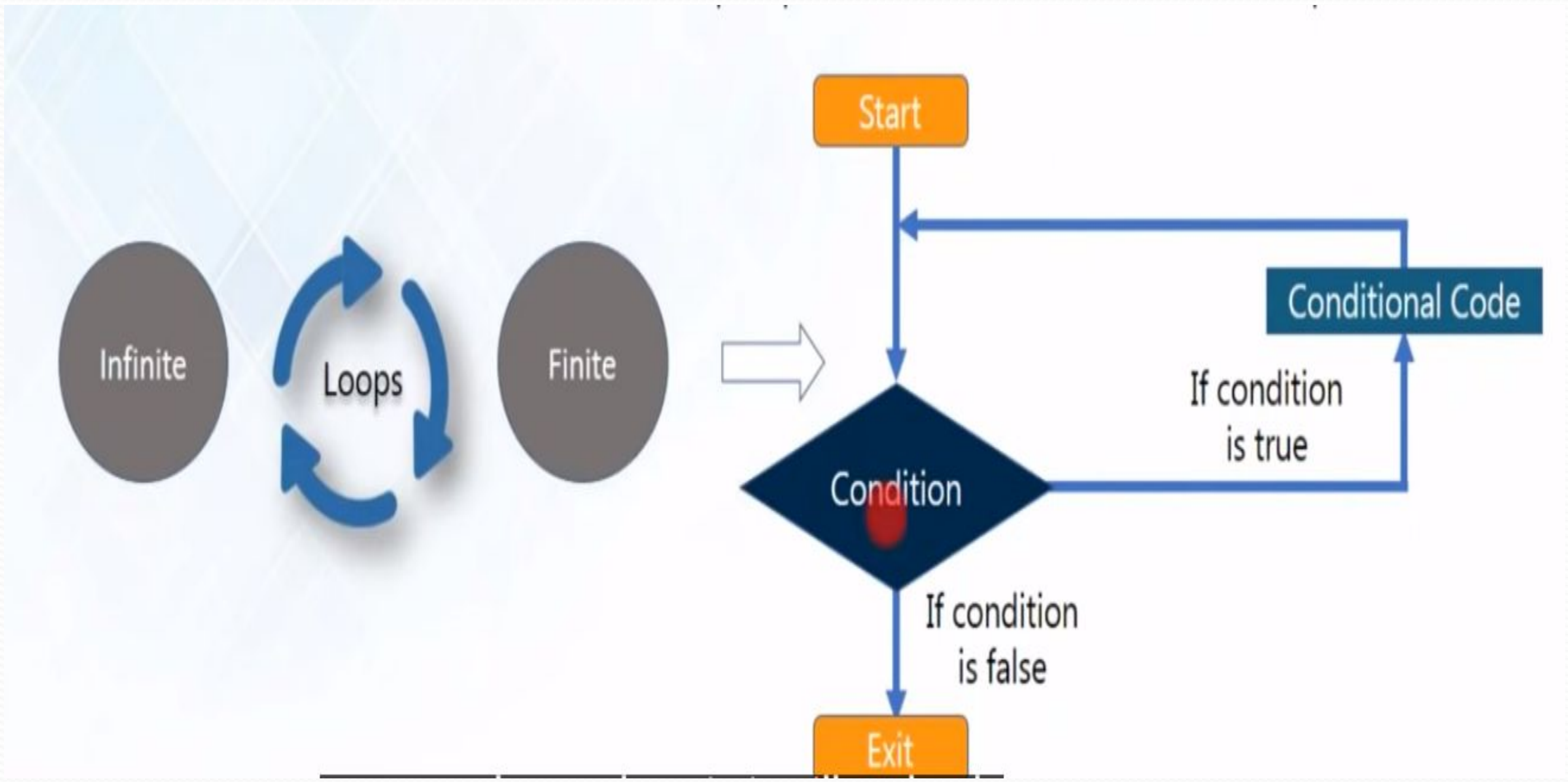
```
friends = ['Roopa', 'Smaya', 'Vikas’]
for name in friends:
        print('Happy New Year:', name)
print('Done!')

for i in "Hello":
    print(i, end=“\t“)
```



Output:

```
Happy New Year: Roopa
Happy New Year: Smaya
Happy New Year: Vikas
Done!
```


Infinite
Loops
Finite
Start
Conditional Code
If condition is true
Condition
If condition is false
Exit

Syntax of ***for* loop with range( ) function:**

for variable in range(start, end, steps):
        statements to be repeated

```
for i in range(5):
    print(i, end= “\t”)                0 1 2 3 4
for i in range(3,0,-1):
   print(i)                    3
print('Blast off!!')                2
                         1
                         Blast off!!
```

Ex:

```
words=[“John”,”Sam”,”Sham”]
for w in words:
   print(w) # for len of words print(len(w))
OR
for w in words.items:
   print(w)

range() Method returns length of list elements
for i in range(len(words)):
   print(i, words[i]) #To display words use print(words[i])
```

Ex. Cont.

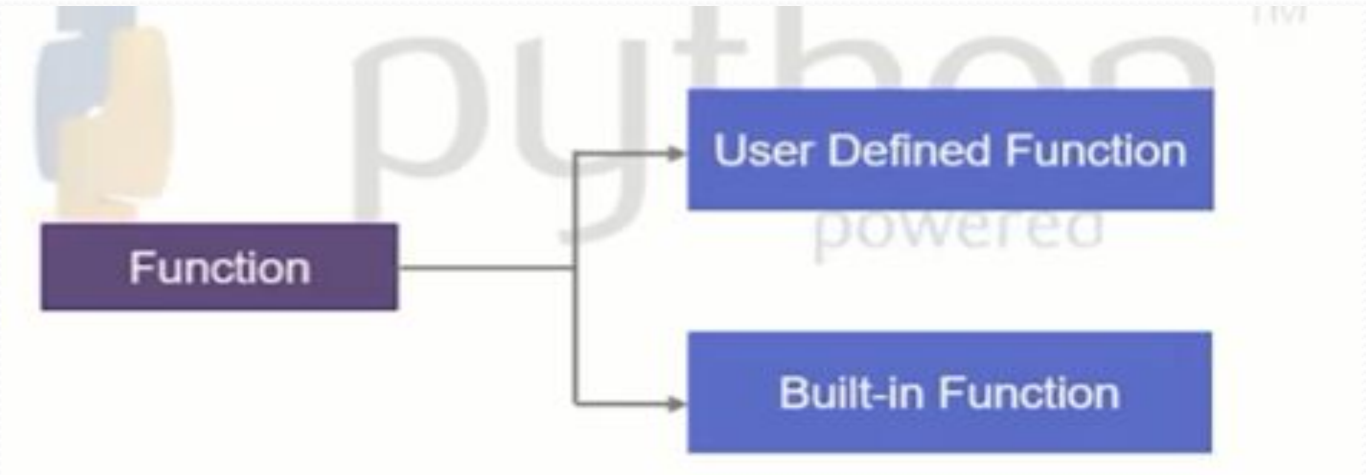
```
for i in reversed(range(1, 10, 2)):
... print(i)

basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
for i in sorted(basket):
... print(i)

basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
for f in sorted(set(basket)):
... print(f)
```

What are functions?

A function is a block of organized, reusable sets of instructions that is used to perform some related actions



Syntax :

- def function_name(parameters):

  statement(s)

Functions in python

- **Function definition:**

  every function should start with “def” keyword

  - Name of the function
  - Parameters/arguments(optional)
  - Function name should end with :
  - Return (empty or value)
  - Multi value return can done(using tuples)
  - Call by Reference Parameters is implemented
  - It behaves like any other object such as an int or a list.

Comparision of function usage

```
public int add(int a, int b)
{
int sum=0;
sum = a+b;
return sum;
}

def add(a,b):
   sum=0
   sum=a+b
   return sum
```

Function call

- Function name
- Arguments and parameters

Functions

- A sequence of instructions intended to perform a specific independent task is known as a *function*.
- You can pass data to be processed, as parameters to the function. Some functions can return data as a result.
- In Python, all functions are treated as objects, so it is more flexible compared to other high- level languages.
- In this section, we will discuss various types of built-in functions, user-defined functions, applications/uses of functions etc.

Ex:

```
def print_greeting():
    print “Hello!”
    print “How are you today?”

print_greeting() #Calling the function
```

Ex:

```
def hello_func(name,somelist):
   print “Hello,”,name,”!\n”
   name = “Caitlin”
   somelist[0]=3
   return 1, 2
myname=“Ben”
mylist=[1,2]
a,b= hello_func(myname,mylist)
print myname,mylist
print a,b
```

Ex:

- If the python interpreter is running the module as the main program, it sets the special variable
- i.e __name__ = “__main__”. This allows flexibility in writing your module.
- __name__ is another built ins, has two underscores on either side!

```
if __name__ == “__main__”
   print (even_fib())
```

Function types

- Built in functions

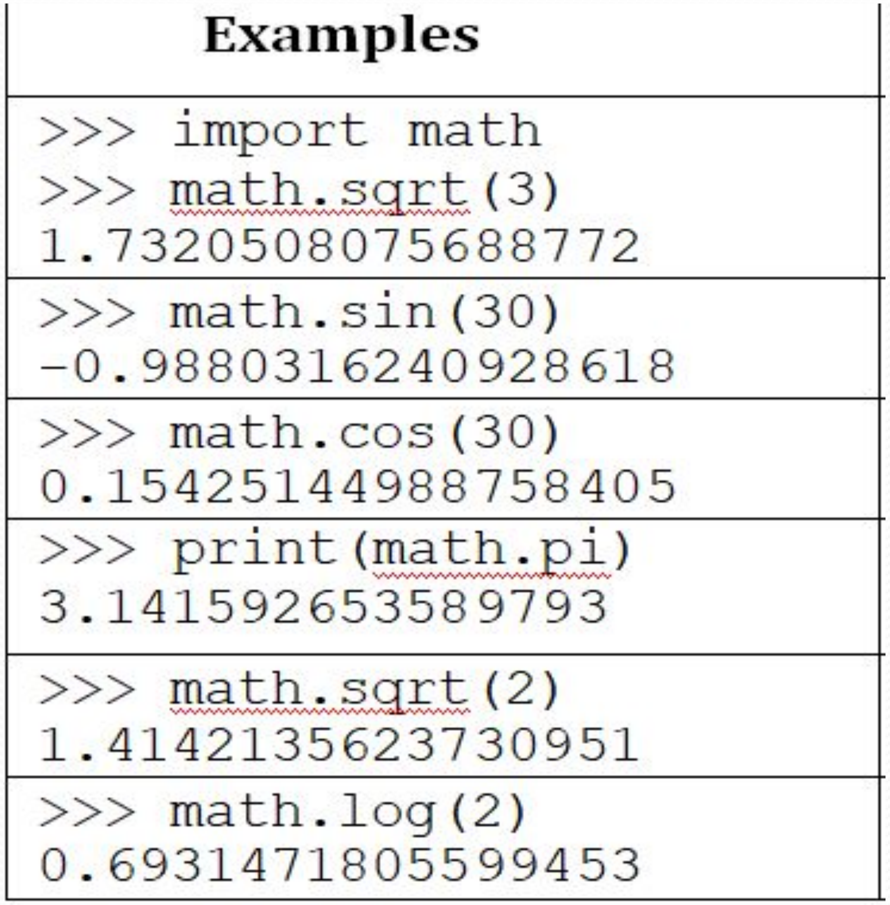
Built-in- functions	Syntax/Examples	Comments
max( )	>>>max('hello world') 'w'	display character having maximum ASCII code
min( )	>>>min('hello world') ' '	display least character having minimum ASCII code
len( )	>>>len('hello world') 11	display length of string

round( )	>>>round(3.8)
	4
	>>>round(3.3)
	3
	>>>round(3.5)
	4
	>>>round(3.141592653, 2)
	3.14
		round the value with single argument.
		
		round the value with 2 arguments.

pow()	>>>pow(2,4)	2^4
abs()	>>>abs(-3.2)	returns the absolute value of object

Commonly Used Modules

Math functions



Examples

`>>> import math`
`>>> math.sqrt(3)`
`1.7320508075688772`

`>>> math.sin(30)`
`-0.9880316240928618`

`>>> math.cos(30)`
`0.15425144988758405`

`>>> print(math.pi)`
`3.141592653589793`

`>>> math.sqrt(2)`
`1.4142135623730951`

`>>> math.log(2)`
`0.6931471805599453`

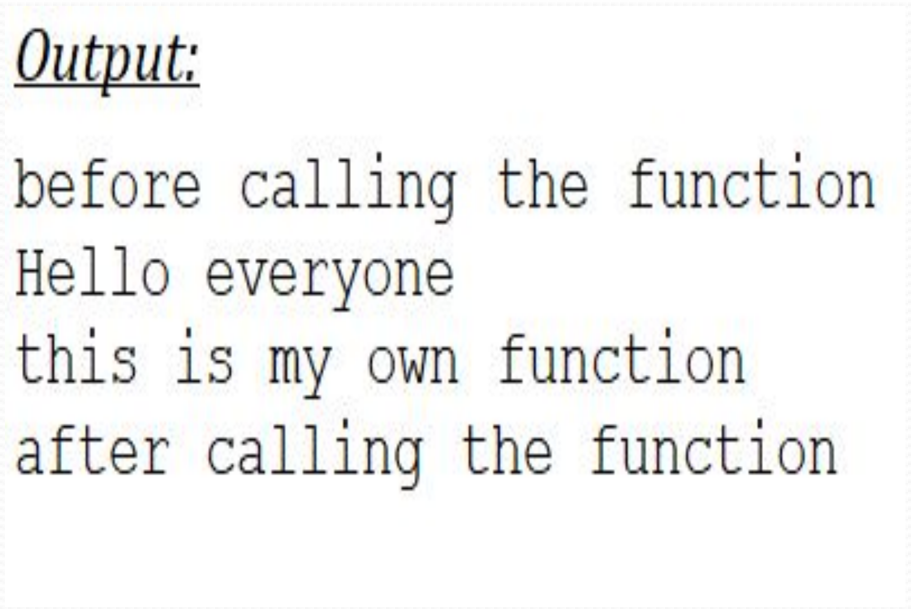
Function Definition and Calling the Function

- Python facilitates programmer to define his/her own functions.
- The function written once can be used wherever and whenever required.
- The syntax of user-defined function would be:

```
def fname(arg_list):
    statement_1
    statement_2
    ...............
    statement_n
    return value
```

```
def myfun():
    print("Hello everyone")
    print("this is my own function")

print("before calling the function")
myfun()      #fuction call
print("after calling the function")
```



Output:

```
before calling the function
Hello everyone
this is my own function
after calling the function
```

Function calls

- A function is a named sequence of instructions for performing a task.
- Whenever we want to do that task, a function is called by its name

```
>>> type(33)
<class 'int'>
```

The return Statement and void Function

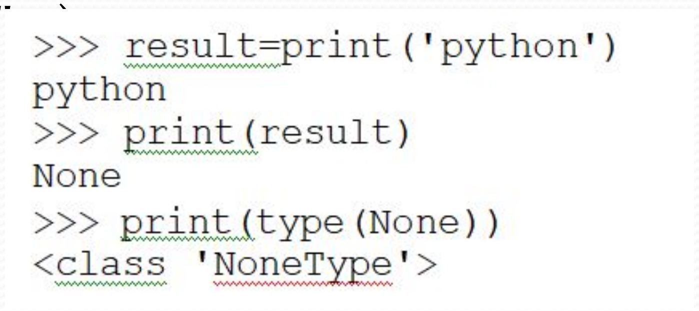
- A function that performs some task, but do not return any value to the calling function is known as ***void*** function
- The function which returns some result to the calling function after performing a task is known as ***fruitful*** function.

```
def addition(a,b):   #function definition
    sum=a + b
    return sum

x=addition(3, 3)      #function call
    print("addition of 2 numbers:"
```

<u>Output:</u>

addition of 2 numbers:6



```
>>> result=print('python')
python
>>> print(result)
None
>>> print(type(None))
<class 'NoneType'>
```

Conti.

- Fruitful functions:

  a functions returns a value.

- Void functions:

  a function that always returns None.

Types of arguments in function

- Required arguments
- Keyword arguments
- Default arguments
- Variable length arguments

Required arguments

- Same and order
- Example:

```
#example for required arguments
def display(num1,num2):
        print(num1,num2)

display(100,200)
```

Keyword arguments

- Oder or position is not required
- Initialization done base of keywords
- Example:

```
#expample for keyword arguement

def display(num1,num2):
    print(num1,num2)

display(num2=10, num1=20)
```

Default arguments

- Number of argument need not be same
- Some of arguments will be consider as default
- Example:

```
#example for default arguments

def display(name, course="MCA"):
    print(name)
    print(course)

display(name="robert", course="BCA")
display(name="rishab")
```

Ex:

```
def print_message():
   print(“Hi, how are you\n”)
   print(“Have a nice day\n”)

def repeat_message():
   print_message()
   print_message()

repeat_message()
```

```
# Python code to demonstrate  call by value
string = "Geeks"

def test(string):
    string = "GeeksforGeeks"
    print("Inside Function:", string)

# Driver's code
test(string)
print("Outside Function:", string)

# Python code to demonstrate     call by reference
 def add_more(list):
    list.append(50)
    print("Inside Function", list)

# Driver's code
mylist = [10,20,30,40]

add_more(mylist)
print("Outside Function:", mylist)
```

```
def swap(s1, s2):
    tmp = s1
    s1 = s2
    s2 = tmp
    return s1,s2

s1 = 1
s2 = 2
print("Before Swap :", s1,s2)
s1,s2=swapo(s1, s2)
print("After Swap :", s1,s2)
```

- **Default Parameters**
  - parameters optional and use default values
  - If user does not want to provide values for some arg. This is done with the help of default argument values
  - Parameters by appending to the parameter name in the function definition the assignment operator (=) followed by the default value.(Constant value)

```
def say(message, times=1):
    print(message * times)
 say('Hello')
say('World', 5)
```

- **Keyword Arguments**
  - Some functions with many parameters
  - To specify only some of them, then you can give values for such parameters by naming them.

```
def func(a, b=5, c=10):
    print('a is', a, 'and b is', b, 'and c is', c)

func(3, 7)
func(25, c=24)
func(c=50, a=100)
```

Global vs. Local variables

- Local variables can be accessed only inside the function in which they are declared.
- Global variables can be accessed throughout the program body by all functions.

```
total = 0 # This is global variable. # Function definition is here
def sum( arg1, arg2 ):
    total = arg1 + arg2 # Here total is local variable.
    print("Inside the function local total : ", total)
    return total
sum( 10, 20 ); # Now you can call sum function
print("Outside the function global total : ", total)
```

Output :

```
Inside the function local total : 30
Outside the function global total : 0
```

*args and **kwargs, Command Line Arguments

- pass a variable number of arguments to the calling function
- *__args__ -allows you to pass a non-keyworded variable length tuple
- **__kwargs__ - you to pass keyworded, variable length dictionary

```
def total(a=5, *numbers, **phonebook):
    print('a', a)   #iterate through all the items in tuple
    for single_item in numbers:
        print('single_item', single_item) #iterate through all the items in
    dictionary
    for first_part, second_part in phonebook.items():
        print(first_part,second_part)

total(10,1,2,3,Jack=1123,John=2231,Inge=1560)
```

Cont.

- **Command Line Arguments**
    - The **sys** module provides a global variable named *argv*

```
C:\Code>python sqrtcmdline.py 2 5
from sys import argv
for n in range(int(argv[1]), int(argv[2]) + 1):
    print(n, sqrt(n))
```

Assignment Questions

- Define python? Write a short note on data types in Python.
- Explain the syntax of the conditional statements and for loop with an example
- Explain the fruitful and void functions? Give examples. Create a python program for calculator application using functions.
- Define function. Explain the built-in functions and commonly used functions
- Differentiate between local and global variables with suitable examples.