Python Programming

PBD 1806

Python

- Python is a **general-purpose interpreted, interactive, object-oriented, and high-level programming language**. It was created by Guido van Rossum during 1985- 1990
- Its not after Python snake. But after from TV comedy show ‘Monty Python’s Flying Circus’

Overview

- **Python is Interpreted** − Python is processed at runtime by the interpreter. You do not need to compile your program before executing it.
- **Python is Interactive** − You can be at a Python prompt and interact with the interpreter directly to write your programs.
- **Python is Object-Oriented** − Python supports Object-Oriented style or technique of programming that encapsulates code within objects.
- **Python is a Beginner's Language** − Python is a great language for the beginner-level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games

History

- Python 1.0 was released in November 1994. In 2000, Python 2.0 was released. Python 2.7.11 is the latest edition of Python 2.
- Meanwhile, Python 3.0 was released in 2008. Python 3 is not backward compatible with Python 2. The emphasis in Python 3 had been on the removal of duplicate programming constructs and modules.
- Find out the current version ………

Features

- **Easy-to-learn** − Python has few keywords, simple structure, and a clearly defined syntax. This allows a student to pick up the language quickly.
- **Easy-to-read** − Python code is more clearly defined and visible to the eyes.
- **Easy-to-maintain** − Python's source code is fairly easy-to-maintain.
- **A broad standard library** − Python's bulk of the library is very portable and cross-platform compatible on UNIX, Windows, and Macintosh.
- **Interactive Mode** − Python has support for an interactive mode which allows interactive testing and debugging of snippets of code.

....

- **Portable** − Python can run on a wide variety of hardware platforms and has the same interface on all platforms.
- **Extendable** − You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient.
- **Databases** − Python provides interfaces to all major commercial databases.
- **GUI Programming** − Python supports GUI applications that can be created and ported to many system calls, libraries and windows systems, such as Windows MFC, Macintosh, and the X Window system of Unix.
- **Scalable** − Python provides a better structure and support for large programs than shell scripting.

………..

- Apart from the above-mentioned features, Python has a big list of good features. A, few are listed below −
  - It supports functional and structured programming methods as well as OOP.
  - It can be used as a scripting language or can be compiled to byte-code for building large applications.
  - It provides very high-level dynamic data types and supports dynamic type checking.
  - It supports automatic garbage collection.
  - It can be easily integrated with C, C++, COM, ActiveX, CORBA, and Java

Environment setup

- Python 3 is available for Windows, Mac OS and most of the flavors of Linux operating system.
- **Getting Python for Windows platform**
    - Binaries of latest version of Python 3 are available on https://www.python.org/downloads/windows/
    - Download the required Python release for your system(either Win32 or Win64)
    - Execute the downloaded file, follow the instructions (set environment path variable)
- **Python Official Website** − https://www.python.org/
- You can download Python documentation from the following site. The documentation is available in HTML, PDF and PostScript formats.
    - **Python Documentation Website** − www.python.org/doc/

Running python

- There are three different ways to start Python −

1. **Interactive Interpreter**
    - You can start Python from Unix, DOS, or any other system that provides you a command-line interpreter or shell window.
    - Enter python on the command line.
        - C:>python
    - Start coding right away in the interactive interpreter.

………

- Here is the list of all the available command line options

-d	provide debug output
-O	generate optimized bytecode (resulting in .pyo files)
-v	verbose output (detailed trace on import statements)
file	run Python script from given file

…….

2. **Script from the Command-line**

   A Python script can be executed at the command line by invoking the interpreter on your application.

   C:>python script.py

Note − Be sure the file permission mode should apply to allows execution

…….

3. **Integrated Development Environment**

   You can run Python from a Graphical User Interface (GUI) environment as well, if you have a GUI application on your system that supports Python.

Python Building Blocks

Identifiers

- An Identifier is a name used to identify a variable, function, class or other object. An identifier starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores and digits (0 to 9).
- Python does not allow special characters such as @, $, and % within identifiers. Python is a case sensitive programming language. **HELLO and hello both are different.**

……

- Rules for naming conventions for identifiers −
  - Class names start with an uppercase letter. All other identifiers start with a lowercase letter.
  - Starting an identifier with a single leading underscore indicates that the identifier is private.
  - Starting an identifier with two leading underscores indicates a strong private identifier.
  - If the identifier also ends with two trailing underscores, the identifier is a language-defined special name.

Reserved (Key) words.

- You cannot use them as constants or variables or any other identifier names. All the Python keywords contain lowercase letters only.

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

Line Indentation

- Python does not use braces({}) to indicate blocks of code for class and function definitions or flow control. Blocks of code are denoted by line indentation.
- The number of spaces in the indentation is variable, but all statements within the block must be indented the same amount.
- For example

```
if True:
     print ("True")
else:
    print ("False")
```

Multiline statements

- Single statement in Python typically ends with a new line. The line continuation character (\) is used to denote that the line is continuing.
- For example

```
total = item_one + \
        item_two + \
        item_three
```

Quotation

- Python accepts single ('), double (") and triple (''' or """) quotes to denote string literal.
- The triple quotes are used to represent the string across multiple lines.
- For example, all the following are legal −

  word = 'word'

  sentence = "This is a sentence."

  paragraph = """This is a paragraph. It is made up of multiple lines and sentences."""

Comments

- hash sign (#) is used at the beginning of a sentence. Python interpreter ignores them.
- A line containing only whitespace, possibly with a comment, is known as a blank line and Python totally ignores it.
- Python does not have multiple-line commenting feature. You have to comment each line individually as follows
- Example

```
#!/usr/bin/python3
# First comment
print ("Hello, Python!") # second comment
```

- This produces output −

```
Hello, Python!
```

end=“ “

- The print() function inserts a new line at the end, by default.
- In Python 3, "end =' '" appends space instead of newline.

  print(x, end=" ") # Appends a space instead of a newline in Python 3

Strings

- Strings in Python are arrays of bytes representing unicode characters.
- However, Python does not have a character data type, a single character is simply a string with a length of 1. Square brackets can be used to access individual elements of the string.
- Example
  - a = "Hello, World!"
    print(a[1])   =>   e
  - b = "Hello, World!"
    print(b[2:5])   =>   llo
- The strip() method removes any whitespace from the beginning or the end:
  - a = " Hello, World! "
    print(a.strip())   =>   "Hello, World!"
- The len() method returns the length of a string:
  - a = "Hello, World!"
    print(len(a))   =>   12
- The lower() method returns the string in lower case:
  - a = "Hello, World!"
    print(a.lower())

Strings

- The upper() method returns the string in upper case:

  -
    ```
    a = "Hello, World!"
    print(a.upper())
    ```
- The replace() method replaces a string with another string: or a character

  -
    ```
    a = "Hello, World!"
    print(a.replace("H", "J"))
    ```
- The split() method splits the string into substrings if it finds instances of the separator:

  -
    ```
    a = "Hello, World!"
    print(a.split(",")) # returns ['Hello', ' World!']
    ```

Format with Strings

- We cannot combine strings and numbers like this

  -
    ```
    age = 36
    txt = "My name is John, I am " + age
    print(txt)
    ```
- The format() takes arguments formats them and places the in the string where the placeholders {} are

  -
    ```
    age = 36
    txt = "My name is John, and I am {}"
    print(txt.format(age))
    ```
- The format() method takes unlimited number of arguments, and are placed into the respective placeholders:

  -
    ```
    quantity = 3
    itemno = 567
    price = 49.95
    myorder = "I want {} pieces of item {} for {} dollars."
    print(myorder.format(quantity, itemno, price))
    ```

Format with Strings

- You can use index numbers {0} to be sure the arguments are placed in the correct placeholders:

  -
    ```
    quantity = 3
    itemno = 567
    price = 49.95
    myorder = "I want to pay {2} dollars for {0} pieces of item {1}."
    print(myorder.format(quantity, itemno, price))
    ```

Input statements.

- Allows the user to enter input values.

```
input("\n\nPress the enter the name.")
x=input("\n\nPress the enter the name.")
Print(x)
```

"\n\n" is used to create two new lines before the user enters his name.

- The semicolon (;) allows multiple statements on a single line

```
import sys; x = 'foo'; sys.stdout.write(x + '\n')
```

Variables

- Variables are reserved memory locations to store values. It means that when you create a variable, you reserve some space in the memory.
- Based on the data type of a variable, the interpreter allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to the variables, you can store integers, decimals or characters in these variables.
- . The equal sign (=) is used to assign values to variables.

```
counter = 100 # An integer assignment
miles = 1000.0 # A floating point
name = "John" # A string
print (counter)
print (miles)
print (name)
```

Multiple assignment

- Python allows you to assign a single value to several variables simultaneously.
- For example −

  a = b = c = 1

  Here, an integer object is created with the value 1, and all the three variables are assigned to the same memory location. You can also assign multiple objects to multiple variables. For example −

  a, b, c = 1, 2, "john"

  Here, two integer objects with values 1 and 2 are assigned to the variables a and b respectively, and one string object with the value "john" is assigned to the variable c.

Standard Data Types.

- Python has six standard data types –
  - Numbers
  - String
  - List
  - Tuple
  - Dictionary
  - Sets

Numbers

- Number data types store numeric values.
- Number objects are created when you assign a value to them. For example −

  var1 = 1

  var2 = 10
- Python supports three different numerical types −
  - int (signed integers)
  - float (floating point real values)
- All integers in Python3 are represented as long integers. Hence, there is no separate number type as long.

Strings

- Strings in Python are identified as a contiguous set of characters represented in the quotation marks.
- Python allows either pair of single or double quotes. Subsets of strings can be taken using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the string and working their way from -1 to the end.
- The plus (+) sign is the string concatenation operator and the asterisk (*) is the repetition operator. For example

…….

str = ‘Hi Everyone'

str1=“ Good morning{

- print (str) # Prints complete string
- print (str[0]) # Prints first character of the string
- print (str[2:5]) # Prints characters starting from 3rd to 5th
- print (str[2:]) # Prints string starting from 3rd character
- print (str * 2) # Prints string two times
- print (str + "TEST") # Prints concatenated string
- Print(str+str1) # prints concatenated string

Output

- Hi Everyone
- H
- Ev
- Everyone
- Hi EveryoneHi Everyone
- Hi EveryoneTEST
- Hi EveryoneGood Morning

Lists

- A list contains items separated by commas and enclosed within square brackets ([]). To some extent, lists are similar to arrays in C. One of the differences between them is that all the items belonging to a list can be of different data type.
- The values stored in a list can be accessed using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the list and working their way to end -1.

Lists

- For example -

list = ['abcd', 786 , 2.23, 'john', 70.2]

tinylist = [123, 'john']

  - print (list)   # Prints complete list
  - print (list[0])   # Prints first element of the list
  - print (list[1:3])   # Prints elements starting from 2nd till 3rd
  - print (list[2:])   # Prints elements starting from 3rd element
  - print (tinylist * 2) # Prints list two times
  - print (list + tinylist) # Prints concatenated lists

- Output-

```
['abcd', 786, 2.23, 'john', 70.2]
abcd
[786, 2.23]
[2.23, 'john', 70.2]
[123, 'john', 123, 'john']
['abcd', 786, 2.23, 'john', 70.2, 123, 'john']
```

Tuples

- A tuple is another sequence data type that is similar to the list. A tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parenthesis.
- The main difference between lists and tuples are − Lists are enclosed in brackets ([ ]) and their elements and size can be changed, while tuples are enclosed in parentheses (( )) and cannot be updated. Tuples can be thought of as **read-only** lists.
  - tuple = ('abcd', 786 , 2.23, 'john', 70.2)
  - list = ['abcd', 786 , 2.23, 'john', 70.2]
  - tuple[2] = 1000   # Invalid
  - list[2] = 1000   # Valid

Dictionaries

- Python's dictionaries consist of key-value pairs.
- A dictionary key can be almost any Python type, but are usually numbers or strings. Values, on the other hand, can be any arbitrary Python object.
- Dictionaries are enclosed by curly braces ({ }) and values can be assigned and accessed using square braces ([]).
- In Dictionaries all the elements are unordered. Hence no indexing for accessing.

```
dict = {}
dict['one'] = "This is one"
dict[2]     = "This is two"
tinydict = {'name': 'john','code':6734, 'dept': 'sales'}
print (dict['one'])       # Prints value for 'one' key
print (dict[2])          # Prints value for 2 key
print (tinydict)          # Prints complete dictionary
print (tinydict['name'])  # Prints the value for the key 'name'
print (tinydict.keys())   # Prints all the keys
print (tinydict.values()) # Prints all the values
```

```
This is one
This is two
{'name': 'john', 'code': 6734, 'dept': 'sales'}
john
dict_keys(['name', 'code', 'dept'])
dict_values(['john', 6734, 'sales'])
```

<u>Nested Dictionary</u>

- mydict = {

  'Apple': {'American':'16', 'Mexican':10, 'Chinese':5},
  'Grapes':{'Arabian':'25','Indian':'20'}

  s }

 print(mydict)

operators

- Python language supports the following types of operators −
  - Arithmetic Operators {+,-,*,/,%,//} ‘// ‘ gives decimal answer.
  - Comparison (Relational) Operators {<, <=, >, >=, ==, !=} returns True/False
  - Assignment Operators
  - Membership Operators {in , not in}
  - Logical operators {AND, OR, NOT}

Assignment Operators

=	c=10
+=	c+=1
-=	c-=2
*=	c*=3
/=	c/=4
%=	c%=5
**=	c**=2 {Exponent}
//=	c//=3

Membership operators

```
a = 10
b = 20
list = [1, 2, 3, 4, 5 ]

if ( a in list ):
  print ("Line 1 - a is available in the given list")
else:
  print ("Line 1 - a is not available in the given list")

if ( b not in list ):
  print ("Line 2 - b is not available in the given list")
else:
  print ("Line 2 - b is available in the given list")
```

Logical Operators

- It is used to combine conditional expressions
- AND - returns true, if both are true.
- OR - Returns true, if any one is true
- NOT- Negate the output, True-False; False-True

Operator precedence

1—Exponent

2---Complement, unary plus and minus

3-- *,/, %, //

4-- +, -

5. Relational operators

6. Assignment operators.

Data type conversions

- To convert between types, you use the type-names as a function.
- There are several built-in functions to perform conversion from one data type to another. These functions return a new object representing the converted value

Converts x to a floating-point number.

str(x)

Converts object x to a string representation.

tuple(s)

Converts s to a tuple.

list(s)

Converts s to a list.

dict(d)

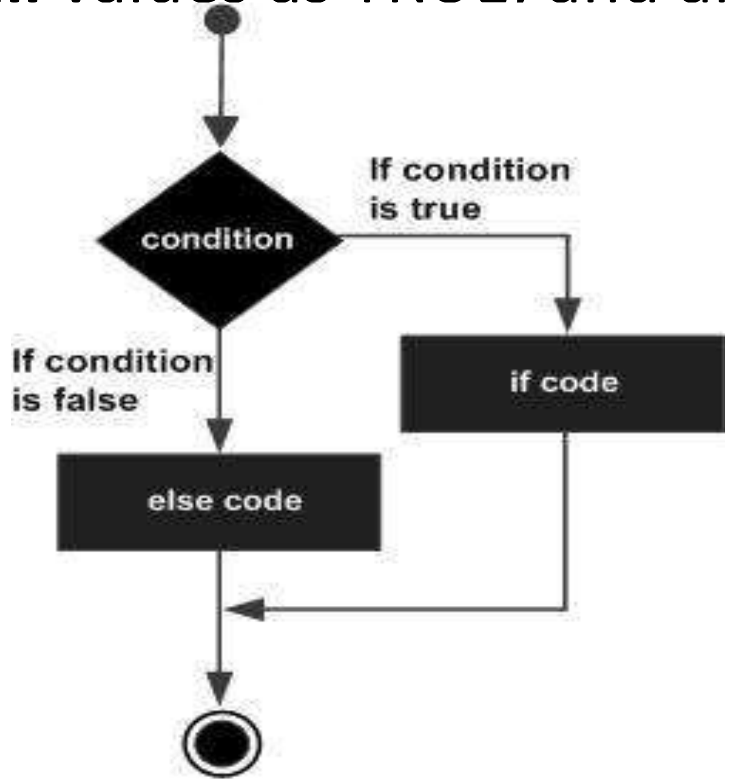
Creates a dictionary. d must be a sequence of (key,value) tuples.

chr(x)

Converts an integer to a character.

Decision Making

- Decision structures evaluate multiple expressions, which produce TRUE or FALSE as the results. You need to determine which action to select and which statements to execute if the outcome is TRUE or FALSE otherwise.
- Python assumes any **non-zero** and **non-null** values as TRUE. and any **zero** or **null values** as FALSE value.



Cont..

1

if

An IF statement consists of a boolean expression followed by one or more statements.

2

if...else

An IF statement can be followed by an optional ELSE statement, which executes when the boolean expression is FALSE.

3

nested if

You can use one IF or ELSE IF statement inside another IF or ELSE IF statement(s).

IF statement

- The **if** statement contains a logical expression using which the data is compared and a decision is made based on the result of the comparison.
- **Syntax**

  if expression: statement(s)
- If the boolean expression evaluates to TRUE, then the block of statement(s) inside the if statement is executed.
- In Python, statements in a block are uniformly indented after the : symbol. If boolean expression evaluates to FALSE, then the first set of code after the end of block is executed.

Example

```
var1 = 100
if var1==100:
   print (“var1")
   print (var1)

var2 = 0
if var2==0:
   print (“var2")
   print (var2)
```

IF…ELSE statement

- An **else** statement can be combined with an **if** statement. An **else** statement contains a block of code that executes if the conditional expression in the if statement resolves to 0 or a FALSE value.
- The else statement is an optional statement and there could be at the most only one **else** statement following **if**.
- **Syntax**

```
if expression:
    statement(s)
else:
    statement(s)
```

Example

Qst.. Discount is calculated on the input amount. Rate of discount is 5%, if the amount is less than 1000, and 10% if it is above 10000.

Solution:

```
amount = int(input("Enter amount: "))

if amount<1000:
   discount = amount*0.05
   print ("Discount",discount)
else:
   discount = amount*0.10
   print ("Discount",discount)

print ("Net payable:",amount-discount)
```

elif statement

- The **elif** statement allows you to check multiple expressions for TRUE and execute a block of code as soon as one of the conditions evaluates to TRUE.
- Similar to the **else**, the **elif** statement is optional. However, unlike **else**, for which there can be at the most one statement, there can be an arbitrary number of **elif** statements following an **if**.
- Python does not support **switch…case** statements as in other languages, but we can use if..elif...statements to represent switch case.

syntax

- **syntax**

```
if expression1:
        statement(s)
elif expression2:
        statement(s)
elif expression3:
        statement(s)
else:
        statement(s)
```

Example

```
amount = int(input("Enter amount: "))

if amount<1000:
   discount = amount*0.05
   print ("Discount",discount)
elif amount<5000:
   discount = amount*0.10
   print ("Discount",discount)
else:
   discount = amount*0.15
   print ("Discount",discount)

print ("Net payable:",amount-discount)
```

Nested IF statement

- When you want to check for another condition after a condition becomes to true. Then you can use the nested **if** block.
- In a nested **if** construct, you can have an **if...elif...else** construct inside another **if...elif...else** construct.

Syntax

- **Syntax**

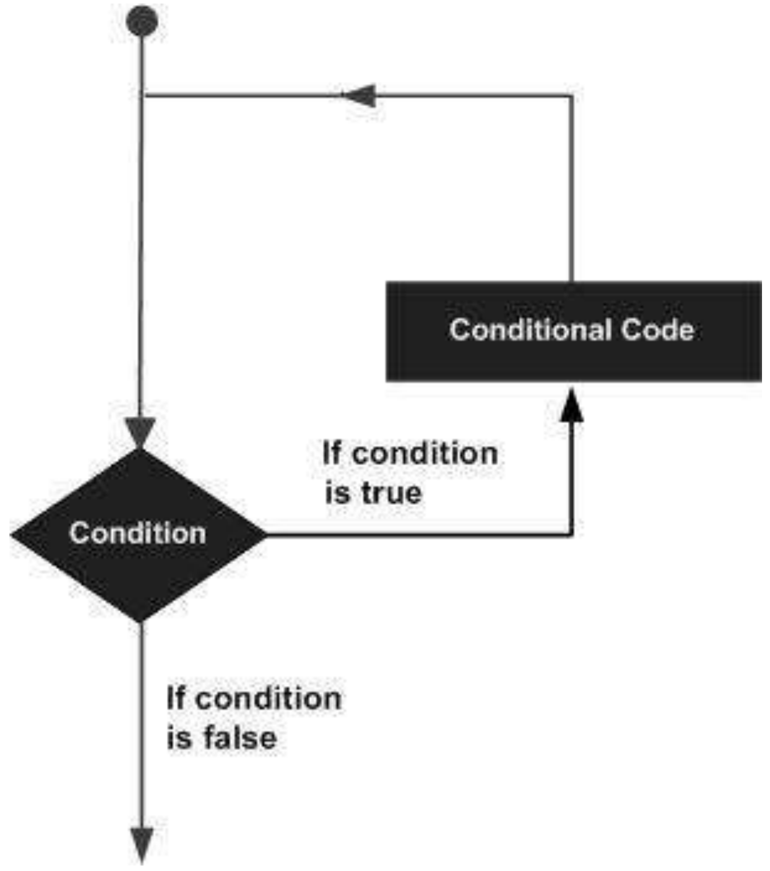
```
if expression1:
   statement(s)
   if expression2:
      statement(s)
   elif expression3:
      statement(s)
   else
      statement(s)
elif expression4:
   statement(s)
else:
   statement(s)
```

Example

```
num = int(input("enter number"))
if num%2 == 0:
   if num%3 == 0:
      print ("Divisible by 3 and 2")
   else:
      print ("divisible by 2 not divisible by 3")
else:
   if num%3 == 0:
      print ("divisible by 3 not divisible by 2")
   else:
      print  ("not Divisible by 2 not divisible by 3")
```

Loops

- statements are executed sequentially – The first statement in a function is executed first, followed by the second, and so on. Loops are used when you need to execute a block of code several number of times.



Cont

1

while Repeats a statement or a set of statements while a given condition is TRUE. It tests the condition before executing the loop body.

2

for Executes a sequence of statements multiple times and controls the code that manages the loop variable.

3

nested You can use one or more loop inside any another while, or for loop.

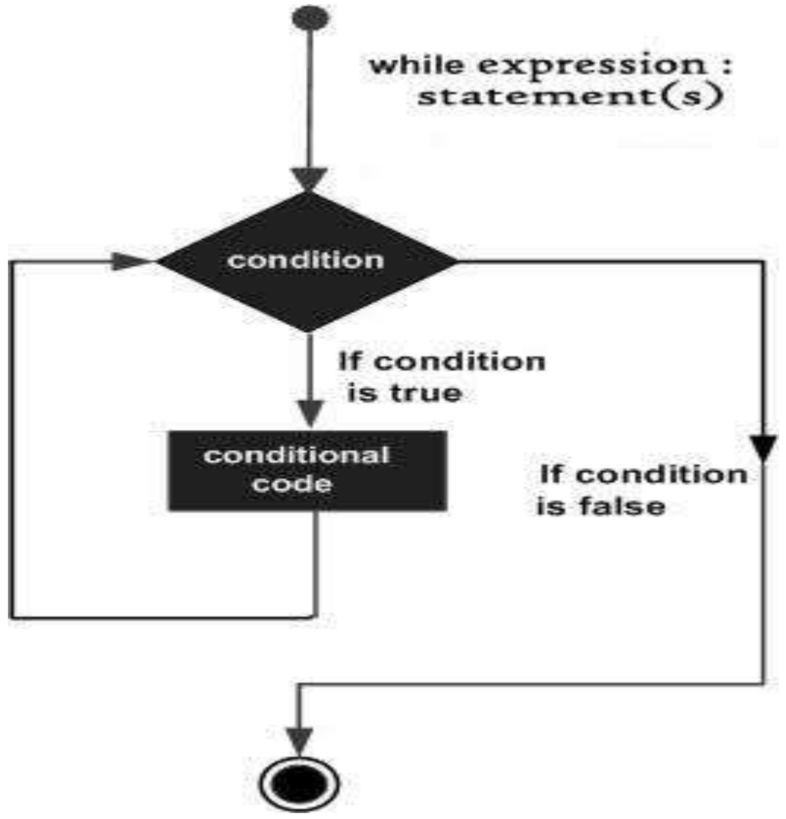
while loop

- A **while** loop statement in Python repeatedly executes a target statement/s as long as a given condition is true.
- **Syntax**

```
while expression:
        statement(s)
```

- The **condition** may be any expression, and true for any non-zero value. The loop iterates while the condition is true.
- When the condition becomes false, program control passes to the line immediately following the loop.
- In Python, all the statements indented by the same number of character spaces in a programming construct are considered to be part of a single block of code. Indentation as its method of grouping statements.

Flow chart


while expression :
statement(s)
condition
If condition is true
conditional code
If condition is false

Example

```
count = 0
while (count < 9):
   print ('The count is:', count)
   count = count + 1
```

Infinite loop

A loop becomes infinite if a condition never becomes FALSE. A loop that never ends. You must be cautious.

Example:

```
var = 1
while var == 1 :  # This constructs an infinite loop
   num = int(input("Enter a number  :"))
   print ("You entered: ", num)
```

- The above example goes in an infinite loop and you need to use CTRL+C to exit the program.

else with loops

- Python supports having an **else** statement associated with a loop statement.
  - If the **else** statement is used with a **for** loop, the **else** statement is executed when the loop gets terminated
  - If the **else** statement is used with a **while** loop, the **else** statement is executed when the condition becomes false.

Example

```
count = 0
while count <= 5:
   print (count, " is  less than 5")
   count = count + 1
else:
   print (count, " is not less than 5")
```

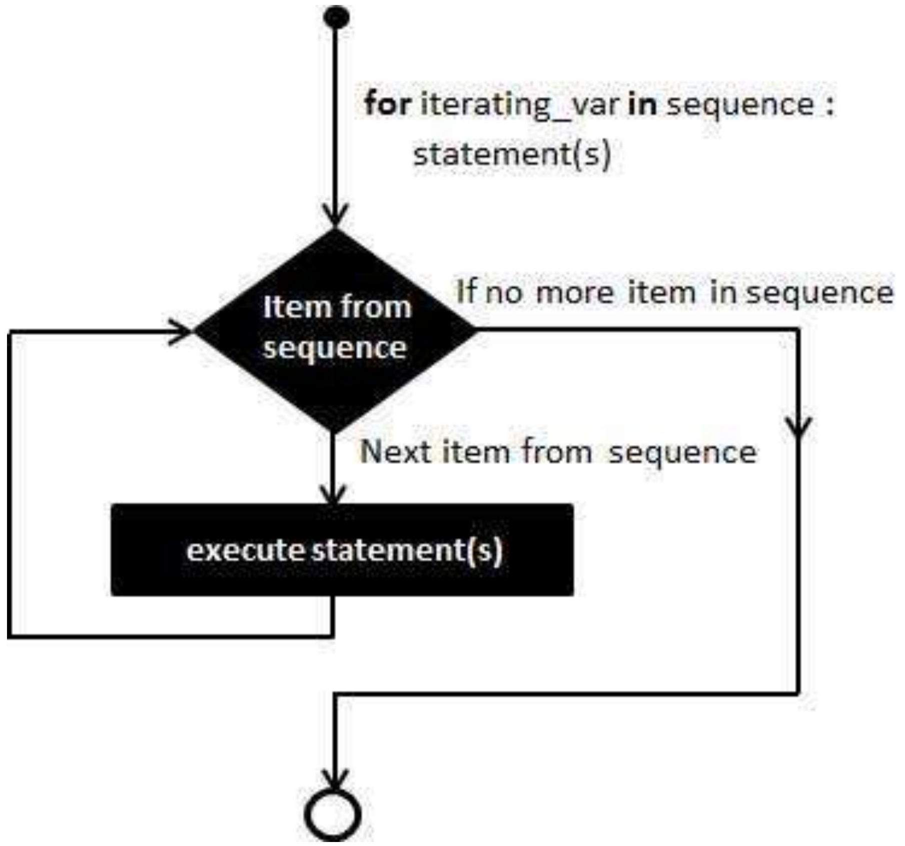
for loop

- The **for** statement in Python has the ability to iterate over the items of any sequence, such as a list or a string.
- **Syntax**

```
for iterating_var in sequence:
        statements(s)
```

- If a sequence contains an expression list, it is evaluated first. Then, the first item in the sequence is assigned to the iterating variable *iterating_var*.
- Next, the statements block is executed. Each item in the list is assigned to *iterating_var*, and the statement(s) block is executed until the entire sequence is completed.

Flow chart


for iterating_var in sequence :
statement(s)
Item from sequence
If no more item in sequence
Next item from sequence
execute statement(s)

Sequence-- range()

- The built-in function range() generates an iterator of arithmetic sequences.
- **Example**

```
>>> range(5)
      range(0, 5)
>>> list(range(5))
       [0, 1, 2, 3, 4]
```

```
>>> for var in list(range(5)):
                print (var)
```

- **Output**

```
0
1
2
3
4
```

Sequence (string, list)

```
for letter in 'Python':    # string
   print ('Letter :', letter)
print()
fruits = ['banana', 'apple',  'mango']
for fruit in fruits:          # List
   print ('fruit :', fruit)
---
```

Using index for iterating

iterating through each item by index offset of the sequence.

- **Example**

```
fruits = ['banana', 'apple',  'mango']
for index in range(len(fruits)):
   print (‘fruit :', fruits[index])
****
```

the len() built-in function, which provides the total number of elements in the list and the range() built-in function to give us the actual sequence to iterate over.

else with for loop

- **else** block is executed only if for loops terminates normally (and not by forceful exit by break statement)
- Example

```
numbers = [11,33,55,67,78,54,6733,44,22]

for num in numbers:
  if num%2 == 0:
    print ('even number')
    break
else:
  print ('the list doesnot contain even number')
```

Nested loops

- Python allows the usage of one loop inside another loop.
- **Syntax**

1.
```
for iterating_var in sequence:
    for iterating_var in sequence:
        statements(s)
    statements(s)
```

2.
```
while expression:
    while expression:
        statement(s)
    statement(s)
```

example

```
for i in range(1,11):            1  {1,2,3,….10}
   for j in range(1,11):         2   {1,2,3,….10}
      k=i*j                      10  {1,2,3,…10}
      print (k, end=' ')
   print()
```

The print() function inner loop has **end=' '** which appends a space instead of default newline. Hence, the numbers will appear in one row.

Loop control statements

The **Loop control statements** change the execution from its normal sequence

- **break** Terminates the loop statement and transfers execution to the statement immediately following the loop. execution at the next statement is taken. Can use with **while and for.** Break in inner loop stops the execution of the innermost loop and follows next line.
- **continue** Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating. Can use with **while and for**
- **pass** The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute.

break example

```
for letter in 'Python':     # First Example
   if letter == 'h':
      break
   print (‘Letter :', letter)       {P y t }

var = 10                    # Second Example
while var > 0:
   print ('value :', var)       {10,9,8,7,6}
   var = var -1
   if var == 5:
      break
```

Example break

```
no = int(input('any number: '))
numbers = [11,33,55,39,55,75,37,21,23,41,13]

for num in numbers:
   if num == no:
      print ('number found in list')
      break
else:
   print ('number not found in list')
```

continue

- The **continue** statement in Python returns the control to the beginning of the current loop. When encountered, the loop starts next iteration without executing the remaining statements in the current iteration.

```
for letter in 'Python':     # First Example
    if letter == 'h':
        continue
    print ('Letter :', letter)              {P y t o n}

var = 10                    # Second Example
while var > 0:
    var = var -1
    if var == 5:
        continue
    print ('value :', var)                  {9,8,7,6,4,3,2,1,0}
```

pass

- The **pass** statement is a *null* operation; nothing happens when it executes.

```
for letter in 'Python':
   if letter == 'h':
      pass
      print ('block')
   print (‘Letter :', letter)     { P y t block h o n}
```

Mathematical functions

- abs(x) The absolute value of x: the (positive) distance between x and zero.
- ceil(x) The ceiling of x: the smallest integer not less than x.
- cmp(x, y)

-1 if x < y, 0 if x == y, or 1 if x > y. Deprecated in Python 3.

- exp(x) The exponential of x: e^x
- fabs(x) The absolute value of x.
- floor(x) The floor of x: the largest integer not greater than x.
- log(x) The natural logarithm of x, for x > 0.
- log10(x) The base-10 logarithm of x for x > 0.
- max(x1, x2,...) The largest of its arguments: the value closest to positive infinity
- min(x1, x2,...) The smallest of its arguments: the value closest to negative infinity.
- pow(x, y) The value of x**y.
- round(x [,n]) x rounded to n digits from the decimal point. Python rounds away from zero as a tie-breaker: round(0.5) is 1.0 and round(-0.5) is -1.0.
- sqrt(x) The square root of x for x > 0.

Random number function

- choice(seq) A random item from a list, tuple, or string.

Ex: random.choice(range(100), random.choice([1, 2, 3, 5, 9], random.choice('Hello World')

- randrange ([start,] stop [,step]) A randomly selected element from range(start, stop, step).

Ex:random.randrange(1, 100, 2), random.randrange(100)

- random() A random float r, such that 0 is less than or equal to r and r is less than 1

Ex:random.random()

- seed([x]) Sets the integer starting value used in generating random numbers. Call this function before calling any other random module function. Returns None.
- shuffle(lst) Randomizes the items of a list in place. Returns None.

String

- Strings can be created simply by enclosing characters in quotes. Python treats single quotes the same as double quotes.

```
var1 = 'Hello World!'
var2 = "Python Programming“
```

- Python does not support a character type; these are treated as strings of length one.
- To access substrings, use the square brackets for slicing along with the index or indices to obtain your substring.

```
print ("var1[0]: ", var1[0])      →H
print ("var2[1:5]: ", var2[1:5])  →ytho
```

- You can "update" an existing string by (re)assigning a variable to another string.

String built in functions

- capitalize()

Capitalizes first letter of string

```
str = "this is string example....wow!!!"
print ("str.capitalize() : ", str.capitalize())
```

str.capitalize() : This is string example....wow!!!

- center(width, fillchar)

Returns a string padded with fillchar with the original string centered to a total of width columns. Default filler is a space

```
str = "this is string example....wow!!!"
print ("str.center(40, 'a') : ", str.center(40, 'a'))
```

str.center(40, 'a') : aaaathis is string example....wow!!!aaaa

- count(str, beg = 0,end = len(string))

Counts how many times str occurs in string or in a substring of string if starting index beg and ending index end are given.

```
str = "this is string example....wow!!!"
sub = 'i'
print ("str.count('i') : ", str.count(sub))
sub = 'exam'
print ("str.count('exam', 10, 40) : ", str.count(sub,10,40))
```

str.count('i') : 3

str.count('exam', 4, 40) : 1

Cont..

- endswith(suffix, beg = 0, end = len(string))

Determines if string or a substring of string (if starting index beg and ending index end are given) ends with suffix; returns true if so and false otherwise.

Example

```
Str='this is string example....wow!!!'
suffix='!!'
print (Str.endswith(suffix))
print (Str.endswith(suffix,20))
suffix='exam'
print (Str.endswith(suffix))
print (Str.endswith(suffix, 0, 19))
```

```
True
True
False
True
```

cont

swapcase() Inverts case for all letters in string.

```
str = "this is string example....wow!!!"
print (str.swapcase())
str = "This Is String Example....WOW!!!"
print (str.swapcase())
```

THIS IS STRING EXAMPLE....WOW!!!

tHIS iS sTRING eXAMPLE....wow!!!

upper() Converts lowercase letters in string to uppercase

```
str = "this is string example....wow!!!"
print ("str.upper : ",str.upper())
```

str.upper : THIS IS STRING EXAMPLE....WOW!!!

isdecimal() Returns true if a unicode string contains only decimal characters and false otherwise

```
str = "this2020“
print (str.isdecimal())
str = "23443434"
print (str.isdecimal())
```

False

True

Cont.

- find(str, beg = 0 end = len(string))

Determine if str occurs in string or in a substring of string if starting index beg and ending index end are given returns index if found and -1 otherwise.

```
str1 = "this is string example....wow!!!"
str2 = "exam";
print (str1.find(str2))
print (str1.find(str2, 10))
print (str1.find(str2, 40))
```

```
15
15
-1
```

- index(str, beg = 0, end = len(string))

Same as find(), but raises an exception if str not found.

```
15
15
Traceback (most recent call last): File "test.py", line 7, in print (str1.index(str2, 40)) ValueError: substring not found shell returned 1
```

cont

- isalnum()

Returns true if string has at least 1 character and all characters are alphanumeric and false otherwise.

- isalpha()

Returns true if string has at least 1 character and all characters are alphabetic and false otherwise.

- isdigit()

Returns true if string contains only digits and false otherwise.

- islower()

Returns true if string has at least 1 cased character and all cased characters are in lowercase and false otherwise.

- isnumeric()

Returns true if a unicode string contains only numeric characters and false otherwise.

- isspace()

Returns true if string contains only whitespace characters and false otherwise.

- istitle()

Returns true if string is properly "titlecased" and false otherwise.

- isupper()

Returns true if string has at least one cased character and all cased characters are in uppercase and false otherwise.

Cont..

len(string)

Returns the length of the string

```
str = "this is string example....wow!!!"
print ("Length of the string: ", len(str))
```

Length of the string: 32

lower()

Converts all uppercase letters in string to lowercase.

```
str = "THIS IS STRING EXAMPLE....WOW!!!"
print (str.lower())
```

this is string example....wow!!!

lstrip()

Removes all leading whitespace in string.

```
str = “      this is string example....wow!!! “
print (str.lstrip())
 str = "*****this is string example....wow!!!*****"
print (str.lstrip('*'))
```

this is string example....wow!!!

this is string example....wow!!!***

cont

replace(old, new [, max])

Replaces all occurrences of old in string with new or at most max occurrences if max given.

```
str = "this is string example....wow!!! this is really string“
print (str.replace("is", "was"))
print (str.replace("is", "was", 3))
```

```
thwas was string example....wow!!! thwas was really string
thwas was string example....wow!!! thwas is really string
```

rstrip()

Removes all trailing whitespace of string.

```
str = “        this is string example....wow!!!            "
print (str.rstrip())
str = "*****this is string example....wow!!!*****"
print (str.rstrip('*'))
```

```
        this is string example....wow!!!
*****this is string example....wow!!!
```

Lists

- In a list is that the items in a list need not be of the same type.
- Creating a list is as simple as putting different comma-separated values between square brackets.
- For example −

```
list1 = ['physics', 'chemistry', 1997, 2000]
list2 = [1, 2, 3, 4, 5 ]
list3 = ["a", "b", "c", "d"]
```

- Similar to string indices, list indices start at 0

```
 print ("list1[0]: ", list1[0])
print ("list2[1:5]: ", list2[1:5])
```

```
list1[0]: physics
list2[1:5]: [2, 3, 4, 5]
```

Updating/deleting

```
list1[2] = 2001
print (list1)


del list1[2]
print (list1)
```

Length

```
len([1, 2, 3])   ->   3
```

concatenation

```
[1, 2, 3] + [4, 5, 6]    ->        [1, 2, 3, 4, 5, 6]
```

Iteration

```
for x in [1,2,3] : print (x,end = ' ')
1 2 3
```

List functions

max(list) Returns max element in the list

```
list1, list2 = ['C++','Java', 'Python'], [456, 700, 200]
print ("Max value element : ", max(list1))
print ("Max value element : ", max(list2))
```

Max value element : Python

Max value element : 700

min(list) Returns min element in the list.

```
print ("min value element : ", min(list1))
print ("min value element : ", min(list2))
```

min value element : C++

min value element : 200

list(seq) Converts a tuple into list.

```
aTuple = (123, 'C++', 'Java', 'Python')
list1 = list(aTuple)
print ("List elements : ", list1)
str="Hello World"
list2=list(str)
print ("List elements : ", list2)
```

List elements : [123, 'C++', 'Java', 'Python']

List elements : ['H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd']

list.sort()

Sorts elements of list

```
list1 = ['physics', 'Biology', 'chemistry', 'maths']
list1.sort()
print ("list now : ", list1)
```

list now : ['Biology', 'chemistry', 'maths', 'physics']

list.append(obj) Appends element to list at the end

```
list1 = ['C++', 'Java', 'Python']
list1.append('C#')
print ("updated list : ", list1)
updated list : ['C++', 'Java', 'Python', 'C#']
```

list.count(obj)

Returns count of how many times element occurs in list

```
aList = [123, 'xyz', 'zara', 'abc', 123]
print ("Count for 123 : ", aList.count(123))
print ("Count for zara : ", aList.count('zara'))
```

Count for 123 : 2

Count for zara : 1

list.extend(seq) Appends the contents of other seq to list

list1 = ['physics', 'chemistry', 'maths']

list2=list(range(5)) #creates list of numbers between 0-4
list1.extend(list2)

print ('Extended List :', list1)

Extended List : ['physics', 'chemistry', 'maths', 0, 1, 2, 3, 4]

list.index(obj)

Returns the lowest index in list that element appears

list1 = ['physics', 'chemistry', 'maths']

print ('Index of chemistry', list1.index('chemistry'))

print ('Index of C#', list1.index('C#'))   {absent----ValaueError}

Index of chemistry 1

**Traceback (most recent call last): File "test.py", line 3, in <module>
print ('Index of C#', list1.index('C#')) ValueError: 'C#' is not in list**

list.insert(index, obj) Inserts object element into list at offset index

```
list1 = ['physics', 'chemistry', 'maths']
list1.insert(1, 'Biology')
print ('Final list : ', list1)
```

Final list : ['physics', 'Biology', 'chemistry', 'maths']

list.pop(obj = list[-1]) Removes and returns last object or obj from list

```
list1 = ['physics', 'Biology', 'chemistry', 'maths']
list1.pop()
print ("list now : ", list1)
list1.pop(1)
print ("list now : ", list1)
```

```
list now : ['physics', 'Biology', 'chemistry']
list now : ['physics', 'chemistry']
```

list.remove(obj) Removes object obj from list

```
list1 = ['physics', 'Biology', 'chemistry', 'maths']
list1.remove('Biology')
print ("list now : ", list1)
```

list now : ['physics', 'chemistry', 'maths']

list.reverse()

Reverses objects of list in place

```
list1 = ['physics', 'Biology', 'chemistry', 'maths']
list1.reverse()
print ("list now : ", list1)
```

l**ist now : ['maths', 'chemistry', 'Biology', 'physics']**

Tuples

- Tuples are sequences, just like lists. The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets

```
tup1=(“physics”,”chemistry”,1997,2000)
tup2=(1,2,3,4,5,6)
tup3=“a”,”b”,”c”,”d”,”e”
print "tup1[0]: ", tup1[0]     ->  physics
 print "tup2[1:5]: ", tup2[1:5]  -> 2,3,4,5
```

# Following action is not valid for tuples
# tup1[0] = 100   {cannot change}

```
tup4 = tup2 + tup3
print tup4
```

(1,2,3,4,5,6,a,b,c,d,e)

Tuple Functions

cmp(tuple1, tuple2)

Compares elements of both tuples.

```
tuple1, tuple2 = (123, 'xyz'), (456, 'abc')
print cmp(tuple1, tuple2)
print cmp(tuple2, tuple1)
tuple3 = tuple2 + (786,);
print cmp(tuple2, tuple3)
```

```
-1
 1
-1
```

len(tuple)

Gives the total length of the tuple.

```
tuple1, tuple2 = (123, 'xyz', 'zara'), (456, 'abc')
print ("First tuple length : ", len(tuple1))
 print ("Second tuple length : ", len(tuple2))
```

```
First tuple length : 3
Second tuple length : 2
```

Tuple functions

max(tuple)
Returns item from the tuple with max value.
	`tuple1, tuple2 = ('maths', 'che', 'phy', 'bio'), (456, 700, 200)`
	`print ("Max value element : ", max(tuple1))`
	`print ("Max value element : ", max(tuple2))`
	
	Max value element : phy
	Max value element : 700

min(tuple)
Returns item from the tuple with min value.
	`tuple1, tuple2 = ('maths', 'che', 'phy', 'bio'), (456, 700, 200)`
	`print ("min value element : ", min(tuple1))`
	`print ("min value element : ", min(tuple2))`
	
	min value element : bio
	min value element : 200

tuple(seq)
Converts a list into tuple.
	`list1= ['maths', 'che', 'phy', 'bio']`
	`tuple1=tuple(list1)`
	`print ("tuple elements : ", tuple1)`
	
	tuple elements : ('maths', 'che', 'phy', 'bio')

Dictionary

Keys are unique within a dictionary while values may not be. The values of a dictionary can be of any type, but the keys must be of an immutable data type such as strings, numbers, or tuples.

```
    dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
    dict['Age'] = 8; # update existing entry
    dict['School'] = "DPS School" # Add new entry
    print ("dict['Age']: ", dict['Age'])
    print ("dict['School']: ", dict['School'])
```

Output

```
    dict['Age']: 8
    dict['School']: DPS School
```

```
   del dict['Name'] # remove entry with key 'Name‘
    dict.clear() # remove all entries in dict
    del dict # delete entire dictionary
```

Duplicate entries

```
   dict = {'Name': 'Zara', 'Age': 7, 'Name': 'Manni'}
    print ("dict['Name']: ", dict['Name'])
```

dict['Name']: Manni

Dictionary Functions

dict.copy()

Returns a shallow copy of dictionary dict

dict1 = {'Name': 'Manni', 'Age': 7, 'Class': 'First'}

dict2 = dict1.copy()

print ("New Dictionary : ",dict2)

New dictionary : {'Name': 'Manni', 'Age': 7, 'Class': 'First'}

dict.items()

Returns a list of *dict*'s (key, value) tuple pairs

dict = {'Name': 'Zara', 'Age': 7}

print ("Value : %s" % dict.items())

Value : [('Age', 7), ('Name', 'Zara')]

dict.update(dict2)

Adds dictionary *dict2*'s key-values pairs to *dict*

dict = {'Name': 'Zara', 'Age': 7}

dict2 = {'Sex': 'female' }

dict.update(dict2)

print ("updated dict : ", dict)

updated dict : {'Sex': 'female', 'Age': 7, 'Name': 'Zara'}

Python Functions

Introduction

- A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.
- Python gives you many built-in functions like print(), etc. but you can also create your own functions. These functions are called *user-defined functions.*
- Function blocks begin with the keyword **def** followed by the function name and parentheses ( ) and a colon : and source code are indented.
- Any input parameters or arguments should be placed within these parentheses.
- The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.
- you can execute it by calling it from another function or directly from the Python prompt.

Cont..

- **Example**

```
def print_me( str ):
        print (str)
        return
print_me(“function called once”)
print_me(“function called second”)
```

Pass by reference or value

- All parameters (arguments) in the Python language are passed by reference (object[mutable/immutable]). It means if you change the value of parameters within a function, the change reflects back in the calling function also.

```
# function definition
def changeme( mylist1 ):
   print ("Values inside the function before change: ", mylist)
   mylist[2]=50
   print ("Values inside the function after change: ", mylist)
   return
# Now you can call changeme function
mylist = [10,20,30]
changeme( mylist )
```

Values inside the function before change: [10, 20, 30]

Values inside the function after change: [10, 20, 50]

Function arguments

- You can call a function by using the following types of formal arguments −
  - Required arguments/Named/Positional
  - Keyword arguments
  - Default arguments
  - Variable-length 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.

```
def printme( str ):
        print (str)
        return
printme()
```

```
Traceback (most recent call last):
  File "test.py", line 11, in <module>
        printme();
TypeError: printme() takes exactly 1 argument (0 given)
```

Keyword arguments

- 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

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


printinfo( age = 50, name = "miki" )
```

Name: miki

Age 50

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 printinfo( name, age = 35 ):
   print ("Name: ", name)
   print ("Age ", age)
   return
printinfo( age = 50, name = "miki" )
printinfo( name = "miki" )
```

```
Name: miki
Age 50
Name: miki
Age 35
```

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 functionname([formal_args,] *var_args_tuple ):
        function_suite

        return [expression]
```

- An asterisk (*) is placed before the variable name that holds the values of all non keyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call

Example

```
 def printinfo( arg1, *vartuple ):
        print ("Output is: ")
        print (arg1)
        for var in vartuple:
                print (var)
        return
printinfo( 10 )
printinfo( 70, 60, 50 )
```

o/p

```
Output is: 10
Output is: 70 60 50
```

Positional & Keyword arguments

```
def fun(x, y):
   print(2*x+y)
fun(2,3)

def quadratic(a, b, c):
   if b**2-4*a*c >= 0:
        return -b/(2*a) + math.sqrt(b**2-4*a*c)/(2*a)
   else:
        print("Error: no solution")
print(quadratic(1, -4, 4)) #CALL BY POSITION
print(quadratic(c=4, a=1, b=-4) #CALL BY KEYWORD
```

Positional and Keyword arguments together

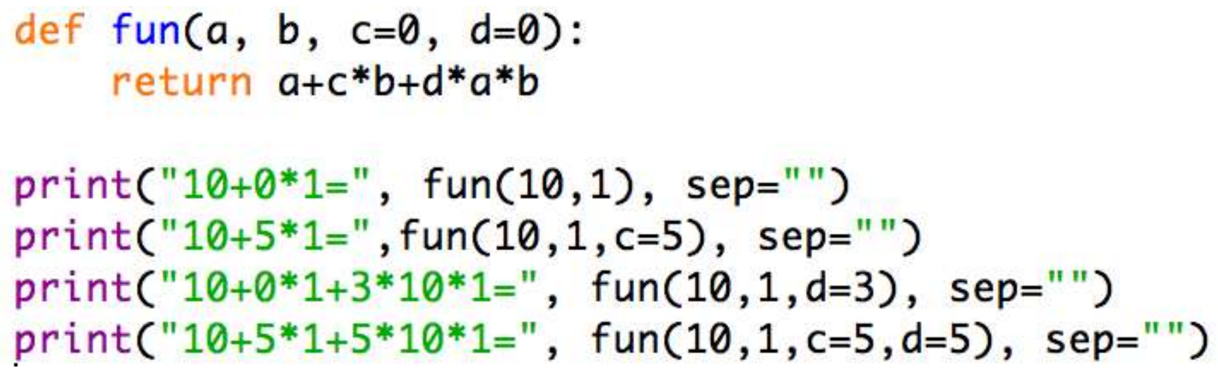
You can force the user of a function to use keywords by introducing an asterisk into the definition of the function:

- All arguments after the asterisk need to be passed by keyword
- The arguments before the asterisk can be positional

```
def function ( posarg1, * , keywarg1 ):
def fun(a, b, *, c):
…
print(fun(2, 3, c=5))
```

Default arguments

default arguments used at last in the arguments list.



```python
def fun(a, b, c=0, d=0):
    return a+c*b+d*a*b

print("10+0*1=", fun(10,1), sep="")
print("10+5*1=",fun(10,1,c=5), sep="")
print("10+0*1+3*10*1=", fun(10,1,d=3), sep="")
print("10+5*1+5*10*1=", fun(10,1,c=5,d=5), sep="")
```

```
10+0*1=10
10+5*1=15
10+0*1+3*10*1=40
10+5*1+5*10*1=65
```

Anonymous functions

- These functions are called anonymous because they are not declared in the standard manner by using the **def** keyword. You can use the **lambda** keyword to create small anonymous functions.
- Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.
- An anonymous function cannot be a direct call to print because lambda requires an expression.
- Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.

Anonymous functions

Lambda expression consists of a keyword lambda

- followed by one or more variables
- followed by a colon
- followed by an expression for the function
- This example implements the function

lambda x : 5*x*x-4*x+3

$x \rightarrow 5x2 - 4x + 3$

Import math

n = lambda x, y: math.sqrt(x*x+y*y)

print(n(2.3, 1.7))

Global and local variable

```
total = 0              # This is global variable.

def sum( arg1, arg2 ):
   “Add both the parameters and return them."
   total = arg1 + arg2;      # Here total is local variable.
   print ("Inside the function local total : ", total)
   return total

sum( 10, 20 )
print ("Outside the function global total : ", total )
```

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

File Handling

Opening a file

- **The open Function**
  - Before you can read or write a file, you have to open it using Python's built-in open() function. This function creates a **file** object, which would be utilized to call other support methods associated with it.
- Syntax

file object = open(file_name [, access_mode][, buffering])

- Here are parameter details −
  - **file_name** − The file_name argument is a string value that contains the name of the file that you want to access.
  - **access_mode** − The access_mode determines the mode in which the file has to be opened, i.e., read, write, append, etc. This is an optional parameter and the default file access mode is read (r).
  - **buffering** − If the buffering value is set to 0, no buffering takes place. If the buffering value is 1, line buffering is performed while accessing a file. If you specify the buffering value as an integer greater than 1, then buffering action is performed with the indicated buffer size. If negative, the buffer size is the system default(default behavior).

File modes and description

r

Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.

rb

Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode.

r+

Opens a file for both reading and writing. The file pointer placed at the beginning of the file.

rb+

Opens a file for both reading and writing in binary format. The file pointer placed at the beginning of the file.

w

Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.

Continue…

wb

Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.

w+

Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.

wb+

Opens a file for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.

a

Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.

Contrinue….

ab

Opens a file for appending in binary format. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.

a+

Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.

ab+

Opens a file for both appending and reading in binary format. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.

The file object attributes

Once a file is opened and you have one *file* object, you can get various information related to that file.

Attribute & Description

file.closed

Returns true if file is closed, false otherwise.

file.mode

Returns access mode with which file was opened.

file.name

Returns name of the file.

Opening a file   example

```
# Open a file
fo = open(“foo.txt", "wb")
print ("Name of the file: ", fo.name)
print ("Closed or not : ", fo.closed)
print ("Opening mode : ", fo.mode)
fo.close()
```

Closing a file

- The close() method of a file object flushes any unwritten information and closes the file object, after which no more writing can be done.
- Python automatically closes a file when the reference object of a file is reassigned to another file. It is a good practice to use the close() method to close a file.
- Syntax

  fileObject.close();

```
# Open a file
fo = open("foo.txt", "wb")
print ("Name of the file: ", fo.name)

# Close opened file
fo.close()
```

Reading

- The **read()** method reads a string from an open file.
- Syntax

  fileObject.read([count]);
- Here, passed parameter is the number of bytes to be read from the opened file. This method starts reading from the beginning of the file and if *count* is missing, then it tries to read as much as possible, maybe until the end of file.

```
# Open a file
fo = open("foo.txt", "r+")
print ("Name of the file: ", fo.name)

line = fo.read(10)
print ("Read Line: %s" % (line))

# Close opened file
fo.close()
```

Example

Assuming that 'foo.txt' file contains following text:

```
This is 1st line
This is 2nd line
This is 3rd line
This is 4th line
This is 5th line
```

o/p

```
Name of the file:  foo.txt
Read Line: This is 1s
```

Method readline()

- Syntax

  file.readline([size])

- Reads one entire line from the file. A trailing newline character is kept in the string.

```
fo = open("foo.txt", "r+")
print ("Name of the file: ", fo.name)

line = fo.readline()
print ("Read Line: %s" % (line))

line = fo.readline(5)
print ("Read Line: %s" % (line))

fo.close()
```

o/p

```
Name of the file:  foo.txt
Read Line: This is 1st line


Read Line: This
```

Method file.readlines([sizehint])

- Syntax:

  file.readlines([sizehint])

- Reads until EOF using readline() and return a list containing the lines. If the optional sizehint argument is present, instead of reading up to EOF, whole lines totalling approximately sizehint bytes (possibly after rounding up to an internal buffer size) are read.

```
line = fo.readlines()
print ("Read Line: %s" % (line))
line = fo.readlines(2)
print ("Read Line: %s" % (line))
```

output

```
Read Line: ['This is 1st line\n', 'This is 2nd line\n', 'This is 3rd line\n', 'This is
4th line\n', 'This is 5th line\n']
Read Line: []
```

writing

The write() Method

The **write()** method writes any string to an open file.

The write() method does not add a newline character ('\n') to the end of the string −

Syntax

```
fileObject.write(string);
```

Example

```
# Open a file
fo = open("foo.txt", "w")
fo.write( "Python is a great language.\nYeah its great!!\n")

# Close opened file
fo.close()
```

Methos writelines()

- Syntax

  file.writelines(sequence)

- Writes a sequence of strings to the file. The sequence can be any iterable object producing strings, typically a list of strings.

Example

```
fo = open("abc.txt", "r+")
print ("Name of the file: ", fo.name)

seq = ["This is 6th line\n", "This is 7th line"]
# Write sequence of lines at the end of the file.
fo.seek(0, 2)
line = fo.writelines( seq )
fo.seek(0,0)
for index in range(7):
    line = next(fo)
    print ("Line No %d - %s" % (index, line))
fo.close()
```

Output

```
Name of the file:  foo.txt
Line No 0 - This is 1st line

Line No 1 - This is 2nd line

Line No 2 - This is 3rd line

Line No 3 - This is 4th line

Line No 4 - This is 5th line

Line No 5 - This is 6th lineThis is 6th line

Line No 6 - This is 7th line
```

Method next()

- Syntax

  next(file)

- Returns the next line from the file each time it is being called.

##Assuming that 'foo.txt' contains following lines

C++

Java

Python

Perl

PHP

Example next()

```
fo = open("foo.txt", "r")
print ("Name of the file: ", fo.name)

for index in range(5):
   line = next(fo)
   print ("Line No %d - %s" % (index, line))

# Close opened file
fo.close()
```

Output

```
Name of the file:  foo.txt
Line No 0 - C++


Line No 1 - Java


Line No 2 - Python


Line No 3 - Perl


Line No 4 - PHP
```

File positions

- The *tell()* method tells you the current position within the file; in other words, the next read or write will occur at that many bytes from the beginning of the file.
- The *seek(offset[, from])* method changes the current file position. The **offset** argument indicates the number of bytes to be moved. The **from** argument specifies the reference position from where the bytes are to be moved.
- If *from* is set to 0, the beginning of the file is used as the reference position. If it is set to 1, the current position is used as the reference position. If it is set to 2 then the end of the file would be taken as the reference position.

File positions example

```
fo = open(“foo.txt", "r+")
str = fo.read(10)
print ("Read String is : ", str)

# Check current position
position = fo.tell()
print ("Current file position : ", position)

# Reposition pointer at the beginning once again
position = fo.seek(0, 0)
str = fo.read(10)
print ("Again read String is : ", str)

# Close opened file
fo.close()
```

o/p

```
Read String is :  Python is
Current file position :  10
Again read String is :  Python is
```

Renaming

- Python **os** module provides methods that help you perform file-processing operations, such as renaming and deleting files.
- To use this module, you need to import it first and then you can call any related functions.
- The rename() method takes two arguments, the current filename and the new filename.
- Syntax

  os.rename(current_file_name, new_file_name)

```
import os
# Rename a file from test1.txt to test2.txt
os.rename( "test1.txt", "test2.txt" )
```

Remove/Delete

- You can use the remove() method to delete files by supplying the name of the file to be deleted as the argument.
- Syntax

os.remove(file_name)

```
import os

# Delete file test2.txt
os.remove("text2.txt")
```

Method mkdir()

All files are contained within various directories, and Python has no problem handling these too. The **os** module has several methods that help you create, remove, and change directories.

- You can use the mkdir() method of the os module to create directories in the current directory. You need to supply an argument to this method, which contains the name of the directory to be created.
- Syntax

```
os.mkdir("newdir")
```

- **Example**

```
import os
# Create a directory "test"
os.mkdir("test")
```

Method- chdir() , getcwd()

- You can use the chdir() method to change the current directory. The chdir() method takes an argument, which is the name of the directory that you want to make the current directory.
- Syntax

  os.chdir("newdir")
- Example

```
import os
# Changing a directory to "/home/newdir"
os.chdir("/home/newdir")
```

- **The getcwd() method** displays the current working directory.
- Syntax

  os.getcwd()
- **Example:**

```
import os

# This would give location of the current directory
os.getcwd()
```

Method rmdir()

- The rmdir() method deletes the directory, which is passed as an argument in the method.
- Before removing a directory, all the contents in it should be removed.
- Syntax

  os.rmdir('dirname')

- Example

```
import os
# This would  remove "/tmp/test"  directory.
os.rmdir( "/tmp/test"  )
```