PYTHON

Sivaprasad Addepalli

© Copyright 2015 Xilinx

Objectives

Python Basics:

- What is Python ?
- Why we need go for Python ?
- Running a simple Python Script
- Data types and Operators
- Decision Making Statements
- Loops and Functions

© Copyright 2015 Xilinx

Objectives

Advanced Python:

- Functions
- Modules
- Exception Handling
- Files Handling
- Regular Expressions
- Classes and Objects

© Copyright 2015 Xilinx

What is Python ?

- Python is a general-purpose interpreted, interactive, object-oriented and high-level programming language.

- Python was created by Guido van Rossum in the late eighties and early nineties.

- Like Perl, Python source code is also now available under the GNU General Public License (GPL).

© Copyright 2015 Xilinx

Why Python

- It is open source.
- One of the biggest upsides to Python is that it's perfect for rapid development. It doesn't take long to program something in Python.
- Python is very easy to learn.
- Python is a multi-paradigm programming language, much like C++. It's OOP, functional, procedural, and a couple of others.
- For speed of development, plus a learning new ways of thinking, Python.
- Can concentrate more on programming rather than syntax’s.

© Copyright 2015 Xilinx

Running a Python Program

- Let us write a simple Python program in a script. All python files will have extension **.py**.

```
#!/usr/bin/python
print "Hello Python!";
```

```
$ python test.py
```

<u>Output:</u>

```
Hello Python!
```

Variables

- Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

- Python variables do not have to be explicitly declared to reserve memory space. The declaration happens automatically when you assign a value to a variable. The equal sign (=) is used to assign values to variables.

```
#!/usr/bin/python
counter = 100          # An integer assignment
miles   = 1000.0       # A floating point
name    = "John"       # A string

print counter
print miles
print name
```

© Copyright 2015 Xilinx

Data Types

- **These are python standard data types:**
  - Numbers
  - Strings
  - List
  - Tuple
  - Dictionary

© Copyright 2015 Xilinx

Numbers

Number data types store numeric values

Number objects are created when you assign a value to them. For example:

```
var1 = 1
var2 = 10
```

You can also delete the reference to a number object by using the **del** statement.

```
del var1[,var2[,var3[....,varN]]]]
```

© Copyright 2015 Xilinx

Number Functions

max(x1, x2,...)

acos(x)

sqrt(x)

floor(x)

cmp(x, y)

© Copyright 2015 Xilinx

Cont...

- #!/usr/bin/python
- Import math

- print "max(80, 100, 1000) : ", max(80, 100, 1000)
- print "max(-20, 100, 400) : ", max(-20, 100, 400)
- print "math.ceil(100.72) : ", math.ceil(100.72)
- print "math.sqrt(100) : ", math.sqrt(100)

o/p:

```
1000
400
101
10
```

Strings

Strings are amongst the most popular types in Python. We can create them simply by enclosing characters in quotes.

Python treats single quotes the same as double quotes.

Creating strings is as simple as assigning a value to a variable.

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

© Copyright 2015 Xilinx

Cont...

```
#!/usr/bin/python

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

print "var1[0]: ", var1[0]
print "var2[1:5]: ", var2[1:5]
```

O/P:

```
var1[0]:  H
var2[1:5]:  ytho
```

© Copyright 2015 Xilinx

Cont...

- capitalize()
  Capitalizes first letter of string

- isdigit()
  Returns true if string contains only digits and false otherwise

- lower()
  Converts all uppercase letters in string to lowercase

- lstrip()
  Removes all leading whitespace in string

© Copyright 2015 Xilinx

Cont...

```
#!/usr/bin/python
str = "this is string example....wow!!!";
print "str.capitalize() : ", str.capitalize()
str = "123456";  # Only digit in this string
print str.isdigit();
str = "THIS IS STRING EXAMPLE....WOW!!!";
print str.lower();
str = "     this is string example....wow!!!     ";
print str.lstrip();
str = "88888888this is string example....wow!!!8888888";
print str.lstrip('8');
```

© Copyright 2015 Xilinx

```
str.capitalize() :  This is string example....wow!!!
True
this is string example....wow!!!
this is string example....wow!!!
this is string example....wow!!!8888888
```

© Copyright 2015 Xilinx

Lists

- Lists are similar ot arrays but contain more than 1 data type.
- Elements are separated by comma and enclosed within [ ]
- Index starts with 0
- Values are assigned using [ ] or [:]
- + to Concatenation
- * to Repetition

Example:

```
#!/usr/bin/python
list1
 =[10,20,"abc",1.25,"def",2.25,30]
list2=[ 3.25,"def"];
print list1;
print list1[0];
print list1[ 2:];
print list1 [2:5];
print list2 *3;
print list1 +list2;
```

© Copyright 2015 Xilinx

Cont...

- Some useful list functions
  - cmp(list1,list2)
  - len(list)
  - max(list)
  - min(list)
  - list.append(obj)
  - list.count(obj)
  - list.extend(seq)
  - list.index(obj)
  - list.remove(obj)
  - list.reverse

© Copyright 2015 Xilinx

append vs extend

- Append: appends the object

Example:

```
>>x = [1, 2, 3]
>>x.append([4, 5])
>>x
[1, 2, 3, [4, 5]]
```

- Extend: appends the elements

Example:

```
>>x = [1, 2, 3]
>>x.extend([4, 5])
>>x
[1, 2, 3, 4, 5]
```

© Copyright 2015 Xilinx

Tuples

- These are read only lists.
- They cannot be updated.
- They are enclosed in parenthesis ie () and elements are separated by comas.

Example:

```
#!/usr/bin/python
tuple1=(10,20,"abc","def",1.25,2.2
 5);
tuple2=("xyz",30);
print tuple1;
print tuple1[0:3];
print tuple1 [2:];
print tuple2 *3;
tuple[3]=5;       #invalid syntax
print tuple1 +tuple2;
```

© Copyright 2015 Xilinx

List vs Tuple

- Key difference is tuples are immutable
- Tuples are often used in place of simple structures

Example:

```
>>value = 4, 5          # packing values
>>x, y = value          # unpacking values
>>x
4
```

- return multiple values from a function

Example:

```
def f(in_str):
   out_str = in_str.upper()
   return True, out_str # Creates tuple automatically
succeeded, b = f("a") # Automatic tuple unpacking
```

© Copyright 2015 Xilinx

Dictionary

- Like a hash table.
  ie:contains key-value pairs.
- Enclosed within { }.
- Accessed and assigned using [ ].
- **Example:**

```
#!/usr/bin/python:
dict1={'name' : 'xyz' , 'id' :1234, 'work' : 'tcs'}
print dict1['id'];                # prints 1234
print dict1;                      # complete dictinary
print dict1.keys();          #  all  the keys
print dict1.values();      #  all the values
```

© Copyright 2015 Xilinx

Cont...

- Some useful build in functions

– dict.items()	list all the keys and values of dict
– dict.keys()	list all the keys of dict
– dict.values()	list all the values of dict
– dict.update(dict2)	append dict2 to dict
– dict.clear()	clears the values of a dict
– del dict	removes the dict

Example:

```
>>> dict1 = {'1':'a'}
>>> dict2 = {'2':'b'}
>>> dict1.update(dict2)   # dict1 = {'1': 'a', '2': 'b'}
>>> dict1.items()           #  [('1', 'a'), ('2', 'b')]
```

© Copyright 2015 Xilinx

Cont...

Adding of two dictionaries into a new dict using for loop

Example:

```
#!/usr/bin/python
from collections import defaultdict
dict1={'1':'a', '2':'b', '3':'c'}
dict2={'4':'d', '3':'e'}
f_dict = defaultdict(list)
for d in (dict2, dict1):
  for key,value in d.iteritems():
      f_dict[key].extend(value)
print f_dict
```

OUTPUT:

```
defaultdict(<type 'list'>, {'1': ['a'], '3': ['e', 'c'], '2': ['b'], '4': ['d']})
```

© Copyright 2015 Xilinx

Operators

- **Types of operators:**
  - Arithmetic Operators
  - Comparision (ie Relational) Operators
  - Assignment Operators
  - Logical Operators
  - Bitwise Operators
  - Membership Operators
  - Identity Operators

# increment(++) and decrement(--) operators are not in python

© Copyright 2015 Xilinx

Arithmetic operators

Operator	Description
+ Addition	Adds values on either side of the operator
- Subtraction	Subtracts right hand operand from left hand operand
* Multiplication	Multiplies values on either side of the operator
/ Division	Divides left hand operand by right hand operand
% Modulus	Divides left hand operand by right hand operand and returns remainder
** Exponent	Performs exponential (power) calculation on operators a**b
// Floor Division	The division of operands where the result is the quotient in which the digits after the decimal point are removed.

© Copyright 2015 Xilinx

Comparison Operators

Operator	Description
==	Checks if the value of two operands are equal or not, if yes then condition becomes true
!=	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true
<>	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true. (This is similar to != operator).
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.

© Copyright 2015 Xilinx

Bitwise opeartors

Operator	Description
&	Binary AND Operator copies a bit to the result if it exists in both operands.
|	Binary OR Operator copies a bit if it exists in eather operand.
^	Binary XOR Operator copies the bit if it is set in one operand but not both.
~	Binary Ones Complement Operator is unary and has the efect of 'flipping' bits.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand

© Copyright 2015 Xilinx

Logical operators

Operator	Description
and	Logical AND operator. If both the operands are true then then condition becomes true.
or	Logical OR Operator. If any of the two operands are non zero then then condition becomes true.
not	Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then logical NOT operator will make false.

© Copyright 2015 Xilinx

Cont..

```
#!/usr/bin/python
a = 10
b = 20
if a and b:
    print "a and b are true"
else:
    print "Either a is not true or b is not true"
if a or b:
   print "Either a is true or b is true or both are true"
else:
   print "Neither a is true nor b is true"
if not a:
   print "a is not true"
else:
   print "Line 5 - a and b are true"
```

© Copyright 2015 Xilinx

Membership operators

Operator	Description
in	Evaluates to true if it finds a variable in the specified sequence and false otherwise.

Example:

```
#!/usr/bin/python
list = [1, 2, 3, 4, 5]
if  “4” in list:
   print "4 is available in the given list"
else:
   Print "4 is not available in the given list"
```

Cont..

Operator	Description
not in	Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.

Example:

```
#!/usr/bin/python
list = [1, 2, 3, 4, 5]
if  “4” not in list:
   print "4 is not available in the given list"
else:
   Print "4 is available in the given list"
```

© Copyright 2015 Xilinx

Identity operators

Operator	Description
is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.

Example:

```
#!/usr/bin/python
a = 10
b = 10
if  a is b :
    print "a and b have same identity"
else:
    print "a and b do not have same identity"
```

© Copyright 2015 Xilinx

Cont...

Operator	Description
is not	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise

Example:

```
#!/usr/bin/python
a = 10
b = 20
if  a is not b :
    print "a and b do not have same identity"
else:
    print "a and b have same identity"
```

© Copyright 2015 Xilinx

Decision making

Statement	**Description**
if	An **if statement** consists of a boolean expression followed by one or more statements.
If else	An **if statement** can be followed by an optional **else statement**, which executes when the boolean expression is false.
nested if	You can use one **if** or **else if** statement inside another **if** or **else if** statement(s).

Any **nonzero/non-null** value ------ assumed as true.

Zero or null --------------------- assumed as false.

© Copyright 2015 Xilinx

IF

Syntax:

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

Example:

```
#!/usr/bin/python
var1 = 100
var2 = 0
if var1:
  print "var1 value is:",var1
if var2:
  print "var2 value is:" , var2
print  "Good bye!"
```

OUTPUT:

```
var1 value is 100
Good bye
```

© Copyright 2015 Xilinx

IF ELSE

Syntax:

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

Example:

```
 #!/usr/bin/python
var1 = 100
if var1:
   print "var1 value is:”,var1
else:
   print "var1 is not defined"
```

OUTPUT:

```
Var1 value is:100
```

© Copyright 2015 Xilinx

IF ELIF ELSE

Syntax:

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

Example:

```
 #!/usr/bin/python
Var1,var2 = 0,100
if var1:
    print "var1 value is:”,var1
elif var1:
    print "var2 value is:”,var2
else:
    print "var1 and var2 are not
  defined"
```

OUTPUT:

```
var2 value is:100
```

© Copyright 2015 Xilinx

Nested IF

Syntax:

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

Example:

```
 #!/usr/bin/python
Var1,var2 = 100,200
if var1:
     print "var1 value is:”,var1
     if var2:
        print "var2 value is:”,var2
     else:
        print "var2 is not defined"
else:
      print "var1 is not defined"
```

OUTPUT:

```
var1 value is:100
var2 value is:200
```

© Copyright 2015 Xilinx

Loops

While:

```
while expression:
    statement(s)


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

Example:

```
#!/usr/bin/python
count = 0
while count < 9:
     count = count + 1
else:
     print “count is not less than 9”
print 'The count is:', count
print "Good bye!"
```

OUTPUT:

```
count is not less than 9
The count is:8
Good bye!
```

© Copyright 2015 Xilinx

Cont...

For:

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

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

Example:

```
#!/usr/bin/python
 for letter in 'TCSHYD':
        print 'Current Letter :', letter
 print "Good bye!"
```

OUTPUT:

```
Current Letter :T
Current Letter :C
Current Letter :S
Current Letter :H
Current Letter :Y
Current Letter :D
Good bye!
```

© Copyright 2015 Xilinx

Loop control statements

Break:

```
#!/usr/bin/python
for letter in 'Python':
    if letter == 'h':
    break ;
    print 'Current Letter :', letter ;
print "over";
```

OUTPUT:

```
Current Letter :P
Current Letter :y
Current Letter :t
over
```

Continue:

```
#!/usr/bin/python
for letter in 'Python':
    if letter == 'h':
     continue
     print 'Current Letter :', letter ;
print "over";
```

OUTPUT:

```
Current Letter : P
Current Letter : y
Current Letter : t
Current Letter : o
Current Letter : n
over
```

© Copyright 2015 Xilinx

FUNCTIONS

© Copyright 2015 Xilinx

Functions

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

© Copyright 2015 Xilinx

Functions

<u>Defining a Function:</u>

Here are simple rules to define a function in Python.

Function blocks begin with the keyword **def** followed by the function name and parentheses (( )).

Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.

The code block within every function starts with a colon (:) and is indented.

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.

© Copyright 2015 Xilinx

Functions

Syntax:

```
def functionname( parameters ):
    “documentation”
     function_suite
     return [expression]
```

Example:

```
def sample( str ):
    “This is my first function”
    print str
    return
```

© Copyright 2015 Xilinx

Functions

Calling a Function:

Defining a function only gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code.

```
#!/usr/bin/python
# Function definition is here
def sample( str ):
   “This is my first function”
   print str;
   return;

# Now you can call printme function
sample("I'm first call to user defined function!");
sample("Again second call to the same function");
docstring=sample.__doc__
```

© Copyright 2015 Xilinx

Functions

When the python script is executed, it produces the following result:

```
I'm first call to user defined function!
Again second call to the same function
This is my first function
```

© Copyright 2015 Xilinx

Functions

Pass by reference:

All parameters (arguments) in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function.

© Copyright 2015 Xilinx

Functions

```
#!/usr/bin/python

def changeme( mylist ):
   mylist.append([1,2,3,4]);
   print "Values inside the function: ", mylist
   return

mylist = [10,20,30];
changeme( mylist );
print "Values outside the function: ", mylist
```

Output:

```
Values inside the function:  [10, 20, 30, [1, 2, 3, 4]]
Values outside the function:  [10, 20, 30, [1, 2, 3, 4]]
```

© Copyright 2015 Xilinx

Functions

Function Arguments:

You can call a function by using the following types of formal arguments:

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

© Copyright 2015 Xilinx

Functions

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.

```
#!/usr/bin/python
def printme( str ):
   print str;
   return;

# Now you can call printme function
printme();
```

© Copyright 2015 Xilinx

Functions

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

© Copyright 2015 Xilinx

Functions

Keyword arguments:

Keyword arguments are related to the function calls. When you use Keyword arguments in a function call, the caller identifies the arguments by the parameter name.

This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters.

```
#!/usr/bin/python
def printme( str ):
print str;
  return;
printme( str = "My string");
```

© Copyright 2015 Xilinx

Functions

```
#!/usr/bin/python

# Function definition is here
def printinfo( name, age ):
   print "Name: ", name;
   print "Age ", age;
   return;

# Now you can call printinfo function
printinfo( age=50, name="miki" );
```

© Copyright 2015 Xilinx

Functions

<u>Default arguments:</u>

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

```
#!/usr/bin/python
def printinfo( name, age = 35 ):
   print "Name: ", name;
   print "Age ", age;
   return;

# Now you can call printinfo function
printinfo( age=50, name="miki" );
printinfo( name="miki" );
```

© Copyright 2015 Xilinx

Functions

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

© Copyright 2015 Xilinx

Functions

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

```
def functionname([formal_args,] *var_args_tuple ):
    function_suite
    return [expression]
```

© Copyright 2015 Xilinx

Functions

```
#!/usr/bin/python

# Function definition is here
def printinfo( arg1, *vartuple ):
   print "Output is: "
   print arg1
   for var in vartuple:
      print var
   return;

# Now you can call printinfo function
printinfo( 10 );
printinfo( 70, 60, 50 );
```

© Copyright 2015 Xilinx

Functions

When the above code is executed, it produces the following result:

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

© Copyright 2015 Xilinx

Functions

The return Statement:

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.

```
#!/usr/bin/python

def sum( arg1, arg2 ):
   total = arg1 + arg2
   print "Inside the function : ", total
   return total;

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

© Copyright 2015 Xilinx

Functions

When the above code is executed, it produces the following result:

```
Inside the function :  30
Outside the function :  30
```

For returning more than one value, we need to use comma (,).

© Copyright 2015 Xilinx

MODULES

© Copyright 2015 Xilinx

MODULES

A module is a file containing Python definitions and statements. The file name is the module name with the suffix .py appended.

As the program gets longer, you may want to split it into several files for easier maintenance. You may also want to use a handy function that you've written in several programs without copying its definition into each program.

To support this, Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter. Such a file is called a module; definitions from a module can be imported into other modules or into the main module

```
def print_func( par ):
   print "Hello : ", par
   return
```

© Copyright 2015 Xilinx

MODULES

<u>The import Statement:</u>

You can use any Python source file as a module by executing an import statement in some other Python source file. The import has the following

<u>syntax:</u>

```
import module1[, module2[,... moduleN]
```

```
#!/usr/bin/python
import support

# Now you can call defined function that module as follows
support.print_func("Zara")
```

© Copyright 2015 Xilinx

MODULES

The from...import Statement

Python's from statement lets you import specific attributes from a module into the current namespace. The from...import has the following syntax:

```
from modname import name1[, name2[, ... nameN]]
```

For example, to import the function fibonacci from the module fib, use the following statement:

```
from fib import fibonacci
fibonacci
```

This statement does not import the entire module fib into the current namespace; it just introduces the item fibonacci from the module fib into the global symbol table of the importing module.

© Copyright 2015 Xilinx

MODULES

The from...import * Statement:

It is also possible to import all names from a module into the current namespace by using the following import statement:

```
from modname import *
```

This provides an easy way to import all the items from a module into the current namespace; however, this statement should be used sparingly.

© Copyright 2015 Xilinx

MODULES

Locating Modules:

When you import a module, the Python interpreter searches for the module in the following sequences:

The current directory.

If the module isn't found, Python then searches each directory in the shell variable PYTHONPATH.

If all else fails, Python checks the default path. On UNIX, this default path is normally /usr/local/lib/python/.

The module search path is stored in the system module sys as the sys.path variable. The sys.path variable contains the current directory, PYTHONPATH, and the installation-dependent default.

© Copyright 2015 Xilinx

MODULES

The dir( ) Function:

The dir() built-in function returns a sorted list of strings containing the names defined by a module.

The list contains the names of all the modules, variables and functions that are defined in a module. Following is a simple example:

```
#!/usr/bin/python
import math
content = dir(math)
print content;
```

© Copyright 2015 Xilinx

FILES

© Copyright 2015 Xilinx

FILES

Printing to the Screen:

The simplest way to produce output is using the print statement where you can pass zero or more expressions separated by commas. This function converts the expressions you pass into a string and writes the result to standard output as follows:

```
#!/usr/bin/python
print "Python is really a great language,", "isn't it?";
```

This would produce the following result on your standard screen:

```
Python is really a great language, isn't it?
```

© Copyright 2015 Xilinx

FILES

Python provides two built-in functions to read a line of text from standard input, which by default comes from the keyboard. These functions are:

raw_input

input

© Copyright 2015 Xilinx

FILES

The raw_input Function:

The raw_input([prompt]) function reads one line from standard input and returns it as a string (removing the trailing newline).

```
#!/usr/bin/python
str = raw_input("Enter your input: ");
print "Received input is : ", str
```

This would prompt you to enter any string and it would display same string on the screen. When I typed "Hello Python!", its output is like this:

```
Enter your input: Hello Python
Received input is :  Hello Python
```

© Copyright 2015 Xilinx

FILES

The input Function:

The input([prompt]) function is equivalent to raw_input, except that it assumes the input is a valid Python expression and returns the evaluated result to you.

```
#!/usr/bin/python
str = input("Enter your input: ");
print "Received input is : ", str
```

This would produce the following result against the entered input:

```
Enter your input: [x*5 for x in range(2,10,2)]
Recieved input is :  [10, 20, 30, 40]
```

© Copyright 2015 Xilinx

FILES

Opening and Closing Files:

Python provides basic functions and methods necessary to manipulate files by default. You can do your most of the file manipulation using a file object.

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])
```

© Copyright 2015 Xilinx

FILES

Here is paramters' detail:

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. A complete list of possible values is given below in the table. This is optional parameter and the default file access mode is read (r).

© Copyright 2015 Xilinx

FILES

Modes 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 will be at the beginning of the file.

rb+ Opens a file for both reading and writing in binary format. The file pointer will be 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.

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.

© Copyright 2015 Xilinx

FILES

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.

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.

© Copyright 2015 Xilinx

FILES

The file object attributes:

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

Here is a list of all attributes related to file object:

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.

© Copyright 2015 Xilinx

FILES

```
#!/usr/bin/python

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

This would produce the following result:

```
Name of the file:  foo.txt
Closed or not :  False
Opening mode :  w
```

© Copyright 2015 Xilinx

FILES

The close() Method:

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();
```

© Copyright 2015 Xilinx

FILES

```
#!/usr/bin/python

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

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

This would produce the following result:

```
Name of the file:  foo.txt
```

© Copyright 2015 Xilinx

FILES

Reading and Writing Files:

The file object provides a set of access methods to make our lives easier.
We would see how to use read() and write() methods to read and write files.

The write() Method:

The write() method writes any string to an open file. It is important to note that Python strings can have binary data and not just text.

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

Syntax:

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

© Copyright 2015 Xilinx

FILES

```
#!/usr/bin/python

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

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

The above method would create foo.txt file and would write given content in that file and finally it would close that file. If you would open this file, it would have following content.

```
Python is a great language.
Yeah its great!!
```

© Copyright 2015 Xilinx

FILES

The read() Method:

The read() method reads a string from an open file. It is important to note that python strings can have binary data and not just text.

Syntax:

```
fileObject.read([count]);
```

Here, passed parameter is the number of bytes to be read from the Openedfile. 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.

© Copyright 2015 Xilinx

FILES

Let's take a file foo.txt, which we have created above.

```
#!/usr/bin/python

# Open a file
fo = open("foo.txt", "r+")
str = fo.read(10);
print "Read String is : ", str
# Close opend file
fo.close()
```

This would produce the following result:

```
Read String is :  Python is
```

© Copyright 2015 Xilinx

FILES

<u>**Renaming and Deleting Files:**</u>

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:

The rename() method takes two arguments, the current filename and the new filename.

<u>Syntax:</u>

```
os.rename(current_file_name, new_file_name)
```

© Copyright 2015 Xilinx

FILES

```
#!/usr/bin/python
import os

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

The remove() Method:

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)
```

© Copyright 2015 Xilinx

FILES

```
#!/usr/bin/python
import os

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

© Copyright 2015 Xilinx

FILES

Directories in Python:

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.

The mkdir() Method:

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")
```

© Copyright 2015 Xilinx

FILES

```
#!/usr/bin/python
import os
os.mkdir("test")
os.chdir("newdir")
os.chdir("/home/newdir")
```

© Copyright 2015 Xilinx

FILES

The getcwd() Method:

The getcwd() method displays the current working directory.

```
os.getcwd()
```

```
#!/usr/bin/python
import os
os.getcwd()
# This would  remove "/tmp/test"  directory.
os.rmdir( "/tmp/test"  )
```

© Copyright 2015 Xilinx

EXCEPTIONS

© Copyright 2015 Xilinx

EXCEPTIONS

An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions. In general,when a Python script encounters a situation that it can't cope with, it raises an exception. An exception is a Python object that represents an error.

When a Python script raises an exception, it must either handle the exception immediately otherwise it would terminate and come out.

Handling an exception:

If you have some suspicious code that may raise an exception, you can defend your program by placing the suspicious code in a try: block. After the try: block, include an except: statement, followed by a block of code which handles the problem as elegantly as possible.

© Copyright 2015 Xilinx

EXCEPTIONS

Syntax:

Here is simple syntax of try....except...else blocks:

```
try:
   You do your operations here;
   ......................
except ExceptionI:
   If there is ExceptionI, then execute this block.
except ExceptionII:
   If there is ExceptionII, then execute this block.
   ......................
else:
   If there is no exception then execute this block.
```

© Copyright 2015 Xilinx

EXCEPTIONS

Here are few important points about the above-mentioned syntax:

A single try statement can have multiple except statements. This is useful when the try block contains statements that may throw different types of exceptions.

You can also provide a generic except clause, which handles any exception.

After the except clause(s), you can include an else-clause. The code in the else-block executes if the code in the try: block does not raise an exception.

The else-block is a good place for code that does not need the try: block's protection.

© Copyright 2015 Xilinx

EXCEPTIONS

```
#!/usr/bin/python

try:
   fh = open("testfile", "w")
   fh.write("This is my test file for exception handling!!")
except IOError:
   print "Error: can\'t find file or read data"
else:
   print "Written content in the file successfully"
   fh.close()
```

This will produce the following result:

```
Written content in the file successfully
```

© Copyright 2015 Xilinx

EXCEPTIONS

Here is one more simple example, which tries to open a file where you do not have permission to write in the file, so it raises an exception:

```
#!/usr/bin/python
try:
   fh = open("testfile", "r")
   fh.write("This is my test file for exception handling!!")
except IOError:
   print "Error: can\'t find file or read data"
else:
   print "Written content in the file successfully"
```

This will produce the following result:

```
Error: can't find file or read data
```

© Copyright 2015 Xilinx

EXCEPTIONS

The except clause with no exceptions:

```
try:
   You do your operations here;
   ......................
except:
   If there is any exception, then execute this block.
   ......................
else:
   If there is no exception then execute this block.
```

This kind of a try-except statement catches all the exceptions that occur. Using this kind of try-except statement is not considered a good programming practice though, because it catches all exceptions but does not make the programmer identify the root cause of the problem that may occur.

© Copyright 2015 Xilinx

EXCEPTIONS

The except clause with multiple exceptions:

You can also use the same except statement to handle multiple exceptions as follows:

```
try:
   You do your operations here;
   ......................
except(Exception1[, Exception2[,...ExceptionN]]]):
   If there is any exception from the given exception list,
   then execute this block.
   ......................
else:
   If there is no exception then execute this block.
```

© Copyright 2015 Xilinx

EXCEPTIONS

The try-finally clause:

You can use a finally: block along with a try: block. The finally block is a place to put any code that must execute, whether the try-block raised an exception or not. The syntax of the try-finally statement is this:

```
try:
   You do your operations here;
   ......................
   Due to any exception, this may be skipped.
finally:
   This would always be executed.
   ......................
```

Note that you can provide except clause(s), or a finally clause, but not both. You can not use else clause as well along with a finally clause.

© Copyright 2015 Xilinx

EXCEPTIONS

```
#!/usr/bin/python

try:
   fh = open("testfile", "w")
   fh.write("This is my test file for exception handling!!")
finally:
   print "Error: can\'t find file or read data"
```

If you do not have permission to open the file in writing mode, then this will produce the following result:

```
Error: can't find file or read data
```

© Copyright 2015 Xilinx

EXCEPTIONS

Argument of an Exception:

An exception can have an argument, which is a value that gives additional information about the problem. The contents of the argument vary by exception. You capture an exception's argument by supplying a variable in the except clause as follows:

```
try:
   You do your operations here;
   ......................
except ExceptionType, Argument:
   You can print value of Argument here...
```

© Copyright 2015 Xilinx

EXCEPTIONS

If you are writing the code to handle a single exception, you can have a variable follow the name of the exception in the except statement. If you are trapping multiple exceptions, you can have a variable follow the tuple of the exception.

This variable will receive the value of the exception mostly containing the cause of the exception. The variable can receive a single value or multiple values in the form of a tuple. This tuple usually contains the error string, the error number, and an error location.

© Copyright 2015 Xilinx

EXCEPTIONS

```
#!/usr/bin/python
def temp_convert(var):
   try:
      return int(var)
   except ValueError, Argument:
      print "The argument does not contain numbers\n", Argument

# Call above function here.
temp_convert("xyz");
```

This would produce the following result:

```
The argument does not contain numbers
invalid literal for int() with base 10: 'xyz'
```

© Copyright 2015 Xilinx

EXCEPTIONS

Raising an exceptions:

You can raise exceptions in several ways by using the raise statement. The general syntax for the raise statement.

Syntax:

```
raise [Exception [, args [, traceback]]]
```

Here, Exception is the type of exception (for example, NameError) and argument is a value for the exception argument. The argument is optional; if not supplied, the exception argument is None.

The final argument, traceback, is also optional (and rarely used in practice), and if present, is the traceback object used for the exception.

© Copyright 2015 Xilinx

EXCEPTIONS

An exception can be a string, a class or an object. Most of the exceptions that the Python core raises are classes, with an argument that is an instance of the class. Defining new exceptions is quite easy and can be done as follows:

```
def functionName( level ):
   if level < 1:
      raise "Invalid level!", level
      # The code below to this would not be executed
      # if we raise the exception
```

Note: In order to catch an exception, an "except" clause must refer to the same exception thrown either class object or simple string. For example, to capture above exception, we must write our except clause as follows:

© Copyright 2015 Xilinx

EXCEPTIONS

```
try:
   Business Logic here...
except "Invalid level!":
   Exception handling here...
else:
   Rest of the code here...
```

© Copyright 2015 Xilinx

Regular Expressions

© Copyright 2015 Xilinx

REGULAR EXPRESSIONS

A regular expression is a special sequence of characters that helps you match or find other strings or sets of strings, using a specialized syntax held in a pattern. Regular expressions are widely used in UNIX world.

The module re provides full support for Perl-like regular expressions in Python. The re module raises the exception re.error if an error occurs while compiling or using a regular expression.

The match Function

This function attempts to match RE pattern to string with optional flags.

Here is the syntax for this function:

```
re.match(pattern, string, flags=0)
```

© Copyright 2015 Xilinx

REGULAR EXPRESSIONS

pattern	This is the regular expression to be matched.
string	This is the string, which would be searched to match the pattern at the beginning of string.
flags	You can specify different flags using bitwise OR (|). These are modifiers, which are listed in the table below.

The re.match function returns a match object on success, None on failure. We would use group(num) or groups() function of match object to get matched expression.

© Copyright 2015 Xilinx

REGULAR EXPRESSIONS

group(num=0)	This method returns entire match (or specific subgroup num)
groups()	This method returns all matching subgroups in a tuple (empty if there weren't any)

```
#!/usr/bin/python
import re
line = "Cats are smarter than dogs"
matchObj = re.match( r'(.*) are (.*?) .*', line, re.M|re.I)
if matchObj:
   print "matchObj.group() : ", matchObj.group()
   print "matchObj.group(1) : ", matchObj.group(1)
   print "matchObj.group(2) : ", matchObj.group(2)
else:
   print "No match!!"
```

© Copyright 2015 Xilinx

REGULAR EXPRESSIONS

When the above code is executed, it produces following result:

```
matchObj.group() :  Cats are smarter than dogs
matchObj.group(1) :  Cats
matchObj.group(2) :  smarter
```

© Copyright 2015 Xilinx

REGULAR EXPRESSIONS

The search Function

This function searches for first occurrence of RE pattern within string with optional flags.

Here is the syntax for this function:

```
re.search(pattern, string, flags=0)
```

© Copyright 2015 Xilinx

REGULAR EXPRESSIONS

```
#!/usr/bin/python
import re

line = "Cats are smarter than dogs";

searchObj = re.search( r'(.*) are (.*?) .*', line, re.M|re.I)

if searchObj:
   print "searchObj.group() : ", searchObj.group()
   print "searchObj.group(1) : ", searchObj.group(1)
   print "searchObj.group(2) : ", searchObj.group(2)
else:
   print "Nothing found!!"
```

© Copyright 2015 Xilinx

REGULAR EXPRESSIONS

Matching vs Searching:

Python offers two different primitive operations based on regular expressions: match checks for a match only at the beginning of the string, while search checks for a match anywhere in the string (this is what Perl does by default).

© Copyright 2015 Xilinx

REGULAR EXPRESSIONS

```
#!/usr/bin/python
import re
line = "Cats are smarter than dogs";
matchObj = re.match( r'dogs', line, re.M|re.I)
if matchObj:
   print "match --> matchObj.group() : ", matchObj.group()
else:
   print "No match!!"
searchObj = re.search( r'dogs', line, re.M|re.I)
if searchObj:
   print "search --> searchObj.group() : ", searchObj.group()
else:
   print "Nothing found!!"
```

© Copyright 2015 Xilinx

REGULAR EXPRESSIONS

When the above code is executed, it produces the following result:

```
No match!!
search --> matchObj.group() :  dogs
```

Search and Replace:

Some of the most important re methods that use regular expressions is sub.

Syntax:

```
re.sub(pattern, repl, string, max=0)
```

This method replaces all occurrences of the RE pattern in string with repl, substituting all occurrences unless max provided. This method would return modified string.

© Copyright 2015 Xilinx

REGULAR EXPRESSIONS

```
#!/usr/bin/python
import re
phone = "2004-959-559 # This is Phone Number"
# Delete Python-style comments
num = re.sub(r'#.*$', "", phone)
print "Phone Num : ", num

# Remove anything other than digits
num = re.sub(r'\D', "", phone)
print "Phone Num : ", num
```

When the above code is executed, it produces the following result:

```
Phone Num :  2004-959-559
Phone Num :  2004959559
```

© Copyright 2015 Xilinx

REGULAR EXPRESSIONS

Modifier	Description
re.I	Performs case-insensitive matching.
re.L	Interprets words according to the current locale. This interpretation affects the alphabetic group (\w and \W), as well as word boundary behavior (\b and \B).
re.M	Makes $ match the end of a line (not just the end of the string) and makes ^ match the start of any line (not just the start of the string).
re.S	Makes a period (dot) match any character, including a newline.
re.U	Interprets letters according to the Unicode character set. This flag affects the behavior of \w, \W, \b, \B.
re.X	Permits "cuter" regular expression syntax. It ignores whitespace (except inside a set [] or when escaped by a backslash) and treats unescaped # as a comment marker.

© Copyright 2015 Xilinx

REGULAR EXPRESSIONS

^	Matches beginning of line.
$	Matches end of line.
.	Matches any single character except newline. Using m option allows it to match newline as well.
[...]	Matches any single character in brackets.
[^...]	Matches any single character not in brackets
re*	Matches 0 or more occurrences of preceding expression.
re+	Matches 1 or more occurrence of preceding expression.
re?	Matches 0 or 1 occurrence of preceding expression.
re{ n}	Matches exactly n number of occurrences of preceding expression.
re{ n,}	Matches n or more occurrences of preceding expression.
re{ n, m}	Matches at least n and at most m occurrences of preceding expression.
al b	Matches either a or b.
(re)	Groups regular expressions and remembers matched text.

© Copyright 2015 Xilinx

REGULAR EXPRESSIONS

\w Matches word characters.

\W Matches nonword characters.

\s Matches whitespace. Equivalent to [\t\n\r\f].

\S Matches nonwhitespace.

\d Matches digits. Equivalent to [0-9].

\D Matches nondigits.

\A Matches beginning of string.

\Z Matches end of string. If a newline exists, it matches just before newline.

\z Matches end of string.

\G Matches point where last match finished.

\b Matches word boundaries when outside brackets. Matches backspace (0x08) when inside brackets.

\B Matches nonword boundaries.

\n, \t, etc. Matches newlines, carriage returns, tabs, etc.

\1...\9 Matches nth grouped subexpression.

\10 Matches nth grouped subexpression if it matched already. Otherwise refers to the octal representation of a character code.

© Copyright 2015 Xilinx

REGULAR EXPRESSIONS

[Pp]ython	Match "Python" or "python"
rub[ye]	Match "ruby" or "rube"
[aeiou]	Match any one lowercase vowel
[0-9]	Match any digit; same as [0123456789]
[a-z]	Match any lowercase ASCII letter
[A-Z]	Match any uppercase ASCII letter
[a-zA-Z0-9]	Match any of the above
[^aeiou]	Match anything other than a lowercase vowel
[^0-9]	Match anything other than a digit

© Copyright 2015 Xilinx

REGULAR EXPRESSIONS

.	Match any character except newline
\d	Match a digit: [0-9]
\D	Match a nondigit: [^0-9]
\s	Match a whitespace character: [\t\r\n\f]
\S	Match nonwhitespace: [^ \t\r\n\f]
\w	Match a single word character: [A-Za-z0-9_]
\W	Match a nonword character: [^A-Za-z0-9_]

Repetition Cases:

Example	Description
ruby?	Match "rub" or "ruby": the y is optional
ruby*	Match "rub" plus 0 or more ys
ruby+	Match "rub" plus 1 or more ys
\d{3}	Match exactly 3 digits
\d{3,}	Match 3 or more digits
\d{3,5}	Match 3, 4, or 5 digits

© Copyright 2015 Xilinx

REGULAR EXPRESSIONS

^Python	Match "Python" at the start of a string or internal line
Python$	Match "Python" at the end of a string or line
\APython	Match "Python" at the start of a string
Python\Z	Match "Python" at the end of a string
\bPython\b	Match "Python" at a word boundary
\brub\B	\B is nonword boundary: match "rub" in "rube" and "ruby" but not alone
Python(?=!)	Match "Python", if followed by an exclamation point
Python(?!!)	Match "Python", if not followed by an exclamation point

Special syntax with parentheses:

Example	Description
R(?#comment)	Matches "R". All the rest is a comment
R(?i)uby	Case-insensitive while matching "uby"
R(?i:uby)	Same as above
rub(?:y|le))	Group only without creating \1 backreference

© Copyright 2015 Xilinx

Classes/Objects

© Copyright 2015 Xilinx

Classes/Objects

Python has been an object-oriented language from day one. Because of this, creating and using classes and objects are downright easy.

Creating Classes:

The class statement creates a new class definition. The name of the class immediately follows the keyword class followed by a colon as follows:

```
class ClassName:
   'Optional class documentation string'
   class_suite
```

The class has a documentation string, which can be accessed via ClassName.__doc__.

The class_suite consists of all the component statements defining class

© Copyright 2015 Xilinx

Classes/Objects

```
class Employee:
   'Common base class for all employees'
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print "Total Employee %d" % Employee.empCount

   def displayEmployee(self):
      print "Name : ", self.name,  ", Salary: ", self.salary
```

© Copyright 2015 Xilinx

Classes/Objects

The variable empCount is a class variable whose value would be shared among all instances of a this class. This can be accessed as Employee.empCount from inside the class or outside the class.

The first method __init__() is a special method, which is called class constructor or initialization method that Python calls when you create a new instance of this class.

You declare other class methods like normal functions with the exception that the first argument to each method is self. Python adds the self argument to the list for you; you don't need to include it when you call the methods.

© Copyright 2015 Xilinx

Classes/Objects

Creating instance objects:

To create instances of a class, you call the class using class name and pass in whatever arguments its __init__ method accepts.

```
"This would create first object of Employee class"
emp1 = Employee("Zara", 2000)
"This would create second object of Employee class"
emp2 = Employee("Manni", 5000)
```

© Copyright 2015 Xilinx

Classes/Objects

Accessing attributes:

You access the object's attributes using the dot operator with object. Class variable would be accessed using class name as follows:

```
emp1.displayEmployee()
emp2.displayEmployee()
print "Total Employee %d" % Employee.empCount
```

© Copyright 2015 Xilinx

Classes/Objects

Accessing attributes:

You access the object's attributes using the dot operator with object. Class variable would be accessed using class name as follows:

```
emp1.displayEmployee()
emp2.displayEmployee()
print "Total Employee %d" % Employee.empCount
```

© Copyright 2015 Xilinx

Classes/Objects

```
#!/usr/bin/python
class Employee:
   'Common base class for all employees'
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print "Total Employee %d" % Employee.empCount

   def displayEmployee(self):
      print "Name : ", self.name,  ", Salary: ", self.salary
```

© Copyright 2015 Xilinx

Classes/Objects

```
"This would create first object of Employee class"
emp1 = Employee("Zara", 2000)
"This would create second object of Employee class"
emp2 = Employee("Manni", 5000)
emp1.displayEmployee()
emp2.displayEmployee()
print "Total Employee %d" % Employee.empCount
```

When the above code is executed, it produces the following result:

```
Name :  Zara ,Salary:  2000
Name :  Manni ,Salary:  5000
Total Employee 2
```

© Copyright 2015 Xilinx

Classes/Objects

You can add, remove or modify attributes of classes and objects at any time:

```
emp1.age = 7  # Add an 'age' attribute.
emp1.age = 8  # Modify 'age' attribute.
del emp1.age  # Delete 'age' attribute.
```

© Copyright 2015 Xilinx

Classes/Objects

Instead of using the normal statements to access attributes, you can use following functions:

The getattr(obj, name[, default]) : to access the attribute of object.

The hasattr(obj,name) : to check if an attribute exists or not.

The setattr(obj,name,value) : to set an attribute. If attribute does not exist, then it would be created.

The delattr(obj, name) : to delete an attribute.

```
hasattr(emp1, 'age')    # Returns true if 'age' attribute exists
getattr(emp1, 'age')    # Returns value of 'age' attribute
setattr(emp1, 'age', 8) # Set attribute 'age' at 8
delattr(empl, 'age')    # Delete attribute 'age'
```

© Copyright 2015 Xilinx

Classes/Objects

Built-In Class Attributes:

Every Python class keeps following built-in attributes and they can be accessed using dot operator like any other attribute:

__dict__ : Dictionary containing the class's namespace.

__doc__ : Class documentation string or None if undefined.

__name__: Class name.

__module__: Module name in which the class is defined. This attribute is "__main__" in interactive mode.

__bases__ : A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.

© Copyright 2015 Xilinx

Classes/Objects

```
#!/usr/bin/python
class Employee:
  'Common base class for all employees'
  empCount = 0

  def __init__(self, name, salary):
    self.name = name
    self.salary = salary
    Employee.empCount += 1

  def displayCount(self):
   print "Total Employee %d" % Employee.empCount

  def displayEmployee(self):
    print "Name : ", self.name,  ", Salary: ", self.salary
```

© Copyright 2015 Xilinx

Classes/Objects

```
print "Employee.__doc__:", Employee.__doc__
print "Employee.__name__:", Employee.__name__
print "Employee.__module__:", Employee.__module__
print "Employee.__bases__:", Employee.__bases__
print "Employee.__dict__:", Employee.__dict__
```

When the above code is executed, it produces the following result:

```
Employee.__doc__: Common base class for all employees
Employee.__name__: Employee
Employee.__module__: __main__
Employee.__bases__: ()
Employee.__dict__: {'__module__': '__main__', 'displayCount':
<function displayCount at 0xb7c84994>, 'empCount': 2,
'displayEmployee': <function displayEmployee at 0xb7c8441c>,
'__doc__': 'Common base class for all employees',
```

© Copyright 2015 Xilinx

Classes/Objects

Destroying Objects (Garbage Collection):

Python deletes unneeded objects (built-in types or class instances) automatically to free memory space. The process by which Python periodically reclaims blocks of memory that no longer are in use is termed garbage collection.

Python's garbage collector runs during program execution and is triggered when an object's reference count reaches zero. An object's reference count changes as the number of aliases that point to it changes.

© Copyright 2015 Xilinx

Classes/Objects

Class Inheritance:

Instead of starting from scratch, you can create a class by deriving it from a preexisting class by listing the parent class in parentheses after the new class name.

The child class inherits the attributes of its parent class, and you can use those attributes as if they were defined in the child class. A child class can also override data members and methods from the parent.

Derived classes are declared much like their parent class; however, a list of base classes to inherit from are given after the class name:

```
class SubClassName (ParentClass1[, ParentClass2, ...]):
   'Optional class documentation string'
   class_suite
```

© Copyright 2015 Xilinx

Classes/Objects

```
#!/usr/bin/python

class Parent:        # define parent class
  parentAttr = 100
  def __init__(self):
    print "Calling parent constructor"

  def parentMethod(self):
    print 'Calling parent method'

  def setAttr(self, attr):
    Parent.parentAttr = attr

  def getAttr(self):
    print "Parent attribute :", Parent.parentAttr
```

© Copyright 2015 Xilinx

Classes/Objects

```
class Child(Parent): # define child class
   def __init__(self):
      print "Calling child constructor"

   def childMethod(self):
      print 'Calling child method'

c = Child()          # instance of child
c.childMethod()      # child calls its method
c.parentMethod()     # calls parent's method
c.setAttr(200)       # again call parent's method
c.getAttr()          # again call parent's method
```

© Copyright 2015 Xilinx

Classes/Objects

When the above code is executed, it produces the following result:

```
Calling child constructor
Calling child method
Calling parent method
Parent attribute : 200
```

© Copyright 2015 Xilinx

Classes/Objects

Overriding Methods:

You can always override your parent class methods. One reason for overriding parent's methods is because you may want special or different functionality in your subclass.

```
#!/usr/bin/python
class Parent:        # define parent class
   def myMethod(self):
      print 'Calling parent method'
class Child(Parent): # define child class
   def myMethod(self):
      print 'Calling child method'
c = Child()          # instance of child
c.myMethod()         # child calls overridden method
```

© Copyright 2015 Xilinx

Sending Email

© Copyright 2015 Xilinx

Sending Email

Simple Mail Transfer Protocol (SMTP) is a protocol, which handles sending e-mail and routing e-mail between mail servers.

Python provides smtplib module, which defines an SMTP client session object that can be used to send mail to any Internet machine with an SMTP or ESMTP listener daemon.

Here is a simple syntax to create one SMTP object, which can later be used to send an e-mail:

```
import smtplib

smtpObj = smtplib.SMTP( [host [, port [, local_hostname]]] )
```

© Copyright 2015 Xilinx

Sending Email

Here is the detail of the parameters:

host: This is the host running your SMTP server. You can specifiy IP address of the host or a domain name like tutorialspoint.com. This is optional argument.

port: If you are providing host argument, then you need to specify a port, where SMTP server is listening. Usually this port would be 25.

local_hostname: If your SMTP server is running on your local machine, then you can specify just localhost as of this option.

© Copyright 2015 Xilinx

Sending Email

An SMTP object has an instance method called sendmail, which will typically be used to do the work of mailing a message. It takes three parameters:

The sender - A string with the address of the sender.

The receivers - A list of strings, one for each recipient.

The message - A message as a string formatted as specified in the various RFCs.

© Copyright 2015 Xilinx

Sending Email

```
#!/usr/bin/python
import smtplib
sender = 'from@fromdomain.com'
receivers = ['to@todomain.com']
message = """From: From Person <from@fromdomain.com>
To: To Person <to@todomain.com>
Subject: SMTP e-mail test
This is a test e-mail message.
"""
try:
   smtpObj = smtplib.SMTP('localhost')
   smtpObj.sendmail(sender, receivers, message)
   print "Successfully sent email"
except SMTPException:
   print "Error: unable to send email"
```

© Copyright 2015 Xilinx

THANK YOU

© Copyright 2015 Xilinx