Chapter 1 : Basics of Python

Data Types in Python:-

1. Python Numbers
2. Python List
3. Python Tuple
4. Python Strings
5. Python Set
6. Python Dictionary
7. Conversion between data types

1. Python Numbers

Integers, floating point numbers and complex numbers falls under Python numbers category. They are defined as int, float and complex class in Python.We can use the type() function to know which class a variable or a value belongs to and the isinstance() function to check if an object belongs to a particular class.

```
a = 5
print(a, "is of type", type(a))

a = 2.0
print(a, "is of type", type(a))

a = 1+2j
print(a, "is complex number?", isinstance(1+2j,complex))
```

OUTPUT-

```
5 is of type <class 'int'>
2.0 is of type <class 'float'>
(1+2j) is complex number? True
```

2.Python List

List is an ordered sequence of items. It is one of the most used data type in Python and is very flexible. All the items in a list do not need to be of the same type.Declaring a list is pretty straight forward. Items separated by commas are enclosed within brackets [ ].

```
 a = [1, 2.2, 'python']
```

We can use the slicing operator [ ] to extract an item or a range of items from a list. Index starts from 0 in Python.

```
a = [5,10,15,20,25,30,35,40]

# a[2] = 15

print("a[2] = ", a[2])

# a[0:3] = [5, 10, 15]

print("a[0:3] = ", a[0:3])
```

OUTPUT-

```
# a[5:] = [30, 35, 40]

print("a[5:] = ", a[5:])

a[2] =  15

a[0:3] =  [5, 10, 15]

a[5:] =  [30, 35, 40]
```

3. Python Tuple

Tuple is an ordered sequence of items same as list.The only difference is that tuples are immutable. Tuples once created cannot be modified.

Tuples are used to write-protect data and are usually faster than list as it cannot change dynamically.

It is defined within parentheses () where items are separated by commas.

```
 t = (5,'program', 1+3j)
```

We can use the slicing operator [] to extract items but we cannot change its value.

```
t = (5,'program', 1+3j)

# t[1] = 'program'

print("t[1] = ", t[1])

# t[0:3] = (5, 'program', (1+3j))

print("t[0:3] = ", t[0:3])

# Generates error

# Tuples are immutable

t[0] = 10
```

OUTPUT-

```
t[1] =  program

t[0:3] =  (5, 'program', (1+3j))
```

We get an error-'tuple' object does not support item assignment for t[0]=10

4. Python Strings

String is sequence of Unicode characters. We can use single quotes or double quotes to represent strings. Multi-line strings can be denoted using triple quotes, ''' or """.

```
 s = "This is a string"
s = '''a multiline
```

Like list and tuple, slicing operator [ ] can be used with string. Strings are immutable.

```
s = 'Hello world!'
```

```
# s[4] = 'o'

print("s[4] = ", s[4])

# s[6:11] = 'world'

print("s[6:11] = ", s[6:11])

# Generates error

# Strings are immutable in Python

s[5] ='d'
```

OUTPUT-

```
s[4] =  o

s[6:11] =  world
```

NOTE- for s[5]='d' we get an error ('str' object does not support item assignment)

5. Python Set

Set is an unordered collection of unique items. Set is defined by values separated by comma inside braces { }. Items in a set are not ordered.

```
a = {5,2,3,1,4}

# printing set variable

print("a = ", a)

# data type of variable a

print(type(a))
```

OUTPUT-

```
a =  {1, 2, 3, 4, 5}
```

```
<class 'set'>
```

We can perform set operations like union, intersection on two sets. Set have unique values. They eliminate duplicates.

```
 a = {1,2,2,3,3,3}
 a
{1, 2, 3}
```

NOTE-Since, set are unordered collection, indexing has no meaning. Hence the slicing operator [ ] does not work.

```
a = {1,2,3}
a[1]
```

We get an error-'set' object does not support indexing

6. Python Dictionary

Dictionary is an unordered collection of key-value pairs.It is generally used when we have a huge amount of data. Dictionaries are optimized for retrieving data. We must know the key to retrieve the value.In Python, dictionaries are defined within braces {} with each item being a pair in the form key:value. Key and value can be of any type.

```
d = {1:'value','key':2}
 type(d)
<class 'dict'>
```

```
d = {1:'value','key':2}

print(type(d))

print("d[1] = ", d[1]);

print("d['key'] = ", d['key']);

# Generates error

print("d[2] = ", d[2]);
```

OUTPUT-

```
<class 'dict'>
```

```
d[1] =  value

d['key'] =  2
```

We get an error for print("d[2] = ", d[2]);

7. Conversion between data types

We can convert between different data types by using different type conversion functions like int(), float(), str() etc.

```
 float(5)
5.0
```

Conversion from float to int will truncate the value (make it closer to zero).

```
 int(10.6)
10
int(-10.6)
-10
```

Conversion to and from string must contain compatible values.

```
 float('2.5')
2.5
 str(25)
'25'
 int('1p')
Traceback (most recent call last):
  File "<string>", line 301, in runcode
  File "<interactive input>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '1p'
```

We can even convert one sequence to another.

```
set([1,2,3])
{1, 2, 3}
 tuple({5,6,7})
(5, 6, 7)
 list('hello')
['h', 'e', 'l', 'l', 'o']
```

To convert to dictionary, each element must be a pair

```
dict([[1,2],[3,4]])
{1: 2, 3: 4}
dict([(3,26),(4,44)])
{3: 26, 4: 44}
```

Chapter 2 : Control Statements

Python if Statement Syntax

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

Here, the program evaluates the test expression and will execute statement(s) only if the text expression is True.If the text expression is False, the statement(s) is not executed.In Python, the body of the if statement is indicated by the indentation. Body starts with an indentation and the first unindented line marks the end.Python interprets non-zero values as True. None and 0 are interpreted as False.

```
# If the number is positive, we print an appropriate message

num = 3
if num > 0:
    print(num, "is a positive number.")
print("This is always printed.")

num = -1
if num > 0:
    print(num, "is a positive number.")
print("This is also always printed.")
```

OUTPUT-

```
3 is a positive number
This is always printed
This is also always printed.
```

In the above example, num > 0 is the test expression.

The body of if is executed only if this evaluates to True.

When variable num is equal to 3, test expression is true and body inside body of if is executed.

If variable num is equal to -1, test expression is false and body inside body of if is skipped.

The print() statement falls outside of the if block (unindented). Hence, it is executed regardless of the test expression.

Python if...else Statement

Syntax of if...else

```
if test expression:
    Body of if
else:
    Body of else
```

The if..else statement evaluates test expression and will execute body of if only when test condition is True.If the condition is False, body of else is executed. Indentation is used to separate the blocks.

```
# Program checks if the number is positive or negative

# And displays an appropriate message

num = 3

# Try these two variations as well.

# num = -5

# num = 0

if num >= 0:

    print("Positive or Zero")

else:

    print("Negative number")
```

OUTPUT-Positive or Zero

In the above example, when num is equal to 3, the test expression is true and body of if is executed and body of else is skipped.If num is equal to -5, the test expression is false and body of else is executed and body of if is skipped.If num is equal to 0, the test expression is true and body of if is executed and body of else is skipped.

Python if...elif...else Statement

Syntax of if...elif...else

```
if test expression:
    Body of if
elif test expression:
    Body of elif
else:
    Body of else
```

The elif is short for else if. It allows us to check for multiple expressions.If the condition for if is False, it checks the condition of the next elif block and so on.If all the conditions are False, body of else is executed.Only one block among the several if...elif...else blocks is executed according to the condition.The if block can have only one else block. But it can have multiple elif blocks.

```
# In this program,

# we check if the number is positive or

# negative or zero and

# display an appropriate message

num = 3.4

# Try these two variations as well:

# num = 0

# num = -4.5

if num > 0:

    print("Positive number")

elif num == 0:

    print("Zero")
```

```
else:

    print("Negative number")
```

OUTPUT-

```
Positive number
```

When variable num is positive, Positive number is printed.

If num is equal to 0, Zero is printed.

If num is negative, Negative number is printed

Python Nested if statements

We can have a if...elif...else statement inside another if...elif...else statement. This is called nesting in computer programming.Any number of these statements can be nested inside one another. Indentation is the only way to figure out the level of nesting. This can get confusing, so must be avoided if we can.

Python Nested if Example

```
# In this program, we input a number
# check if the number is positive or
# negative or zero and display
# an appropriate message
# This time we use nested if

num = float(input("Enter a number: "))
if num >= 0:
    if num == 0:
        print("Zero")
    else:
        print("Positive number")
else:
    print("Negative number")
```

Output 1

```
Enter a number: 5
Positive number
```

Output 2

```
Enter a number: -1
Negative number
```

Output 3

```
Enter a number: 0
Zero
```

For Loop

Syntax of for Loop

```
for val in sequence:
        Body of for
```

Here, val is the variable that takes the value of the item inside the sequence on each iteration.Loop continues until we reach the last item in the sequence. The body of for loop is separated from the rest of the code using indentation.

```
# Program to find the sum of all numbers stored in a list

# List of numbers

numbers = [6, 5, 3, 8, 4, 2, 5, 4, 11]

# variable to store the sum

sum = 0

# iterate over the list

for val in numbers:
        sum = sum+val

# Output: The sum is 48

print("The sum is", sum)
```

OUTPUT-The sum is 48

The range() function

We can generate a sequence of numbers using range() function. range(10) will generate numbers from 0 to 9 (10 numbers).We can also define the start, stop and step size as range(start,stop,step size). step size defaults to 1 if not provided.This function does not store all the values in memory, it would be inefficient. So it remembers the start, stop, step size and generates the next number on the go.To force this function to output all the items, we can use the function list().

The following example will clarify this.

```
# Output: range(0, 10)
print(range(10))

# Output: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(list(range(10)))

# Output: [2, 3, 4, 5, 6, 7]
print(list(range(2, 8)))

# Output: [2, 5, 8, 11, 14, 17]
print(list(range(2, 20, 3)))
```

We can use the range() function in for loops to iterate through a sequence of numbers. It can be combined with the len() function to iterate through a sequence using indexing. Here is an example.

```
# Program to iterate through a list using indexing
```

```
genre = ['pop', 'rock', 'jazz']

# iterate over the list using index

for i in range(len(genre)):

        print("I like", genre[i])
```

OUTPUT-

```
I like pop
I like rock
I like jazz
```

for loop with else

A for loop can have an optional else block as well. The else part is executed if the items in the sequence used in for loop exhausts.break statement can be used to stop a for loop. In such case, the else part is ignored.Hence, a for loop's else part runs if no break occurs.Here is an example to illustrate this.

```
digits = [0, 1, 5]

for i in digits:

   print(i)

else:

   print("No items left.")
```

OUTPUT-

```
0
1
5
No items left.
```

Here, the for loop prints items of the list until the loop exhausts. When the for loop exhausts, it executes the block of code in the else and prints No items left.

While Loop

Syntax of while Loop in Python

```
while test_expression:
    Body of while
```

In while loop, test expression is checked first. The body of the loop is entered only if the test_expression evaluates to True. After one iteration, the test expression is checked again. This process continues until the test_expression evaluates to False.In Python, the body of the while loop is determined through indentation.Body starts with indentation and the first unindented line marks the end.Python interprets any non-zero value as True. None and 0 are interpreted as False.

```
# Program to add natural

# numbers upto

# sum = 1+2+3+...+n

# To take input from the user,

# n = int(input("Enter n: "))

n = 10

# initialize sum and counter

sum = 0

i = 1
```

```
while i <= n:

    sum = sum + i

    i = i+1    # update counter

# print the sum

print("The sum is", sum)
```

INPUT-

Enter n: 10

OUTPUT-The sum is 55

while loop with else

Same as that of for loop, we can have an optional else block with while loop as well.The else part is executed if the condition in the while loop evaluates to False. The while loop can be terminated with a break statement.In such case, the else part is ignored. Hence, a while loop's else part runs if no break occurs and the condition is false.Here is an example to illustrate this.

```
# Example to illustrate

# the use of else statement

# with the while loop

counter = 0

while counter < 3:

    print("Inside loop")

    counter = counter + 1
```

```
else:

    print("Inside else")
```

OUTPUT-

```
Inside loop
Inside loop
Inside loop
Inside else
```

Here, we use a counter variable to print the string Inside loop three times.On the fourth iteration, the condition in while becomes False. Hence, the else part is executed.

Looping techniques

The infinite loop

We can create an infinite loop using while statement. If the condition of while loop is always True, we get an infinite loop.

Example #1: Infinite loop using while

```
# An example of infinite loop
# press Ctrl + c to exit from the loop

while True:
   num = int(input("Enter an integer: "))
   print("The double of",num,"is",2 * num)
```

Output

```
Enter an integer: 3
The double of 3 is 6
Enter an integer: 5
The double of 5 is 10
Enter an integer: 6
The double of 6 is 12
```

```
Enter an integer:
Traceback (most recent call last):
```

Loop with condition at the top

This is a normal while loop without break statements. The condition of the while loop is at the top and the loop terminates when this condition is False.

Example #2: Loop with condition at the top

```
# Program to illustrate a loop with condition at the top

# Try different numbers

n = 10

# Uncomment to get user input

#n = int(input("Enter n: "))

# initialize sum and counter

sum = 0

i = 1

while i <= n:

   sum = sum + i

   i = i+1    # update counter

# print the sum

print("The sum is",sum)
```

OUTPUT-The sum is 55

Loop with condition in the middle

This kind of loop can be implemented using an infinite loop along with a conditional break in between the body of the loop.

Example #3: Loop with condition in the middle

```
# Program to illustrate a loop with condition in the middle.
# Take input from the user untill a vowel is entered

vowels = "aeiouAEIOU"

# infinite loop
while True:
  v = input("Enter a vowel: ")
  # condition in the middle
  if v in vowels:
      break
  print("That is not a vowel. Try again!")

print("Thank you!")
```

OUTPUT-

```
Enter a vowel: r
That is not a vowel. Try again!
Enter a vowel: 6
That is not a vowel. Try again!
Enter a vowel: ,
That is not a vowel. Try again!
Enter a vowel: u
Thank you!
```

Loop with condition at the bottom

This kind of loop ensures that the body of the loop is executed at least once. It can be implemented using an infinite loop along with a conditional break at the end. This is similar to the do...while loop in C.

Example #4: Loop with condition at the bottom

```
# Python program to illustrate a loop with condition at the bottom
# Roll a dice until user chooses to exit

# import random module
import random

while True:
  input("Press enter to roll the dice")

  # get a number between 1 to 6
  num = random.randint(1,6)
  print("You got",num)
  option = input("Roll again?(y/n) ")

  # condition
  if option == 'n':
      break
```

OUTPUT-

```
Press enter to roll the dice
You got 1
Roll again?(y/n) y
Press enter to roll the dice
You got 5
Roll again?(y/n) n
```

CHAPTER 3: FUNCTIONS

Syntax of Function

```
def function_name(parameters):
        """docstring"""
        statement(s)
```

a function definition which consists of following components.

1. Keyword def marks the start of function header.
2. A function name to uniquely identify it. Function naming follows the same rules of writing identifiers in Python.
3. Parameters (arguments) through which we pass values to a function. They are optional.
4. A colon (:) to mark the end of function header.
5. Optional documentation string (docstring) to describe what the function does.
6. One or more valid python statements that make up the function body. Statements must have same indentation level (usually 4 spaces).
7. An optional return statement to return a value from the function.

```
def greet(name):
        """This function greets to
        the person passed in as
        parameter"""
        print("Hello, " + name + ". Good morning!")
```

How to call a function in python?

Once we have defined a function, we can call it from another function, program or even the Python prompt. To call a function we simply type the function name with appropriate parameters.

```
>>> greet('Paul')
Hello, Paul. Good morning!
```

Types of Functions

Basically, we can divide functions into the following two types:

1. Built-in functions - Functions that are built into Python.
2. User-defined functions - Functions defined by the users themselves.

Method	Description
Python abs()	returns absolute value of a number
Python all()	returns true when all elements in iterable is true
Python any()	Checks if any Element of an Iterable is True
Python ascii()	Returns String Containing Printable Representation
Python bin()	converts integer to binary string
Python bool()	Converts a Value to Boolean
Python bytearray()	returns array of given byte size
Python bytes()	returns immutable bytes object
Python callable()	Checks if the Object is Callable
Python chr()	Returns a Character (a string) from an Integer
Python classmethod()	returns class method for given function
Python compile()	Returns a Python code object
Python complex()	Creates a Complex Number
Python delattr()	Deletes Attribute From the Object
Python dict()	Creates a Dictionary

Python dir()	Tries to Return Attributes of Object
Python divmod()	Returns a Tuple of Quotient and Remainder
Python enumerate()	Returns an Enumerate Object
Python eval()	Runs Python Code Within Program
Python exec()	Executes Dynamically Created Program
Python filter()	constructs iterator from elements which are true
Python float()	returns floating point number from number, string
Python format()	returns formatted representation of a value
Python frozenset()	returns immutable frozenset object
Python getattr()	returns value of named attribute of an object
Python globals()	returns dictionary of current global symbol table
Python hasattr()	returns whether object has named attribute
Python hash()	returns hash value of an object
Python help()	Invokes the built-in Help System
Python hex()	Converts to Integer to Hexadecimal
Python id()	Returns Identify of an Object
Python input()	reads and returns a line of string
Python int()	returns integer from a number or string
Python isinstance()	Checks if a Object is an Instance of Class
Python issubclass()	Checks if a Object is Subclass of a Class

Python iter()	returns iterator for an object
Python len()	Returns Length of an Object
Python list() Function	creates list in Python
Python locals()	returns dictionary of current local symbol table
Python map()	Applies Function and Returns a List
Python max()	returns largest element
Python memoryview()	returns memory view of an argument
Python min()	returns smallest element
Python next()	Retrieves Next Element from Iterator
Python object()	Creates a Featureless Object
Python oct()	converts integer to octal
Python open()	Returns a File object
Python ord()	returns Unicode code point for Unicode character
Python pow()	returns x to the power of y
Python print()	Prints the Given Object
Python property()	returns a property attribute
Python range()	return sequence of integers between start and stop
Python repr()	returns printable representation of an object
Python reversed()	returns reversed iterator of a sequence

Python round()	rounds a floating point number to n digits places.
Python set()	returns a Python set
Python setattr()	sets value of an attribute of object
Python slice()	creates a slice object specified by range()
Python sorted()	returns sorted list from a given iterable
Python staticmethod()	creates static method from a function
Python str()	returns informal representation of an object
Python sum()	Add items of an Iterable
Python super()	Allow you to Refer Parent Class by super
Python tuple() Function	Creates a Tuple
Python type()	Returns Type of an Object
Python vars()	Returns __dict__ attribute of a class
Python zip()	Returns an Iterator of Tuples
Python __import__()	Advanced Function Called by import

2. User Defined Functions

Functions that we define ourselves to do certain specific task are referred as user-defined functions.

```
# Program to illustrate
# the use of user-defined functions

def add_numbers(x,y):
   sum = x + y
   return sum
```

```
num1 = 5
num2 = 6

print("The sum is", add_numbers(num1, num2))
```

OUTPUT-

```
Enter a number: 2.4
Enter another number: 6.5
The sum is 8.9
```

Python Recursive Function

We know that in Python, a function can call other functions. It is even possible for the function to call itself. These type of construct are termed as recursive functions.Following is an example of recursive function to find the factorial of an integer.Factorial of a number is the product of all the integers from 1 to that number. For example, the factorial of 6 (denoted as 6!) is 1*2*3*4*5*6 = 720.

```
# An example of a recursive function to
# find the factorial of a number

def calc_factorial(x):
    """This is a recursive function
    to find the factorial of an integer"""

    if x == 1:
        return 1
    else:
        return (x * calc_factorial(x-1))

num = 4
print("The factorial of", num, "is", calc_factorial(num))
```

OUTPUT-

```
The factorial of 4 is 24
```

In the above example, calc_factorial() is a recursive functions as it calls itself.

When we call this function with a positive integer, it will recursively call itself by decreasing the number.Each function call multiples the number with the factorial of number 1 until the number is equal to one. This recursive call can be explained in the following steps.

```
calc_factorial(4)              # 1st call with 4
4 * calc_factorial(3)          # 2nd call with 3
4 * 3 * calc_factorial(2)      # 3rd call with 2
4 * 3 * 2 * calc_factorial(1)  # 4th call with 1
4 * 3 * 2 * 1                  # return from 4th call as number=1
4 * 3 * 2                      # return from 3rd call
4 * 6                          # return from 2nd call
24                             # return from 1st call
```

Our recursion ends when the number reduces to 1. This is called the base condition.Every recursive function must have a base condition that stops the recursion or else the function calls itself infinitely.

Advantages of Recursion

1. Recursive functions make the code look clean and elegant.
2. A complex task can be broken down into simpler sub-problems using recursion.
3. Sequence generation is easier with recursion than using some nested iteration.

Disadvantages of Recursion

1. Sometimes the logic behind recursion is hard to follow through.
2. Recursive calls are expensive (inefficient) as they take up a lot of memory and time.
3. Recursive functions are hard to debug.

What are lambda functions in Python?

In Python, anonymous function is a function that is defined without a name.While normal functions are defined using the def keyword, in Python anonymous functions are defined using the lambda keyword.Hence, anonymous functions are also called lambda functions.

Syntax of Lambda Function in python

```
lambda arguments: expression
```

Lambda functions can have any number of arguments but only one expression. The expression is evaluated and returned. Lambda functions can be used wherever function objects are required.

```
# Program to show the use of lambda functions

double = lambda x: x * 2

# Output: 10
print(double(5))
```

OUTPUT-10
In the above program, lambda x: x * 2 is the lambda function. Here x is the argument and x * 2 is the expression that gets evaluated and returned.This function has no name. It returns a function object which is assigned to the identifier double. We can now call it as a normal function. The statement

```
double = lambda x: x * 2
```

is nearly the same as

```
def double(x):
  return x * 2
```

Use of Lambda Function in python

We use lambda functions when we require a nameless function for a short period of time.In Python, we generally use it as an argument to a higher-order function (a function that takes in other functions as arguments). Lambda functions are used along with built-in functions like filter(), map() etc.

Example use with filter()

The filter() function in Python takes in a function and a list as arguments.

The function is called with all the items in the list and a new list is returned which contains items for which the function evaluates to True.Here is an example use of filter() function to filter out only even numbers from a list.

```
# Program to filter out only the even items from a list

my_list = [1, 5, 4, 6, 8, 11, 3, 12]

new_list = list(filter(lambda x: (x%2 == 0) , my_list))

# Output: [4, 6, 8, 12]
print(new_list)
```

OUTPUT-[4, 6, 8, 12]

Example use with map()

The map() function in Python takes in a function and a list.The function is called with all the items in the list and a new list is returned which contains items returned by that function for each item.Here is an example use of map() function to double all the items in a list.

```
# Program to double each item in a list using map()

my_list = [1, 5, 4, 6, 8, 11, 3, 12]

new_list = list(map(lambda x: x * 2 , my_list))

# Output: [2, 10, 8, 12, 16, 22, 6, 24]

print(new_list)
```

OUTPUT-[2, 10, 8, 12, 16, 22, 6, 24]

CHAPTER 4: Object Orientation and Inheritance

Overview of OOP Terminology

- Class − A user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.
- Class variable − A variable that is shared by all instances of a class. Class variables are defined within a class but outside any of the class's methods. Class variables are not used as frequently as instance variables are.
- Data member − A class variable or instance variable that holds data associated with a class and its objects.
- Function overloading − The assignment of more than one behavior to a particular function. The operation performed varies by the types of objects or arguments involved.
- Instance variable − A variable that is defined inside a method and belongs only to the current instance of a class.
- Inheritance − The transfer of the characteristics of a class to other classes that are derived from it.
- Instance − An individual object of a certain class. An object obj that belongs to a class Circle, for example, is an instance of the class Circle.
- Instantiation − The creation of an instance of a class.
- Method − A special kind of function that is defined in a class definition.
- Object − A unique instance of a data structure that's defined by its class. An object comprises both data members (class variables and instance variables) and methods.

- Operator overloading − The assignment of more than one function to a particular operator.

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 members, data attributes and functions.

Example

Following is the example of a simple Python class −

```
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
```

- The variable *empCount* is a class variable whose value is 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 do not need to include it when you call the methods.

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)
```

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
```

Now, putting all the concepts together −

```
#!/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

"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
```

OUTPUT-

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

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.
```

Instead of using the normal statements to access attributes, you can use the 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'
```

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.

For the above class let us try to access all these attributes −

```
#!/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

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',

'__init__': <function __init__ at 0xb7c846bc>}
```

Destroying Objects (Garbage Collection)

Python deletes unneeded objects (built-in types or class instances) automatically to free the 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.

An object's reference count increases when it is assigned a new name or placed in a container (list, tuple, or dictionary). The object's reference count decreases when it's deleted with *del*, its reference is reassigned, or its reference goes out of scope. When an object's reference count reaches zero, Python collects it automatically.

```
a = 40      # Create object <40>
b = a       # Increase ref. count  of <40>
c = [b]     # Increase ref. count  of <40>

del a       # Decrease ref. count  of <40>
b = 100     # Decrease ref. count  of <40>
c[0] = -1   # Decrease ref. count  of <40>
```

You normally will not notice when the garbage collector destroys an orphaned instance and reclaims its space. But a class can implement the special method *__del__()*, called a destructor, that is invoked when the instance is about to be destroyed. This method might be used to clean up any non memory resources used by an instance.

Example

This __del__() destructor prints the class name of an instance that is about to be destroyed −

```
#!/usr/bin/python

class Point:
   def __init__( self, x=0, y=0):
      self.x = x
      self.y = y
   def __del__(self):
      class_name = self.__class__.__name__
      print class_name, "destroyed"

pt1 = Point()
pt2 = pt1
pt3 = pt1
print id(pt1), id(pt2), id(pt3) # prints the ids of the objects
del pt1
del pt2
del pt3
```

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

```
3083401324 3083401324 3083401324
Point destroyed
```

Note − Ideally, you should define your classes in separate file, then you should import them in your main program file using *import* statement.

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.

Syntax

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

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

Example

```
#!/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

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
```

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

```
Calling child constructor

Calling child method

Calling parent method

Parent attribute : 200
```

Similar way, you can drive a class from multiple parent classes as follows −

```
class A:        # define your class A

.....


class B:         # define your class B

.....
```

```
class C(A, B):   # subclass of A and B
.....
```

You can use issubclass() or isinstance() functions to check a relationships of two classes and instances.

- The issubclass(sub, sup) boolean function returns true if the given subclass sub is indeed a subclass of the superclass sup.
- The isinstance(obj, Class) boolean function returns true if *obj* is an instance of class *Class* or is an instance of a subclass of Class

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.

Example

```
#!/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
```

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

```
Calling child method
```

Base Overloading Methods

Following table lists some generic functionality that you can override in your own classes −

Sr.No.	Method, Description & Sample Call
1	__init__ (self [,args...]) Constructor (with any optional arguments) Sample Call : *obj = className(args)*
2	__del__(self) Destructor, deletes an object Sample Call : *del obj*
3	__repr__(self) Evaluable string representation Sample Call : *repr(obj)*
4	__str__(self) Printable string representation

	Sample Call : *str(obj)*
5	__cmp__ (self, x) Object comparison Sample Call : *cmp(obj, x)*

Overloading Operators

Suppose you have created a Vector class to represent two-dimensional vectors, what happens when you use the plus operator to add them? Most likely Python will yell at you.

You could, however, define the *__add__* method in your class to perform vector addition and then the plus operator would behave as per expectation −

Example

```
#!/usr/bin/python

class Vector:
   def __init__(self, a, b):
      self.a = a
      self.b = b

   def __str__(self):
      return 'Vector (%d, %d)' % (self.a, self.b)

   def __add__(self,other):
      return Vector(self.a + other.a, self.b + other.b)

v1 = Vector(2,10)
v2 = Vector(5,-2)
print v1 + v2
```

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

```
Vector(7,8)
```

Data Hiding

An object's attributes may or may not be visible outside the class definition. You need to name attributes with a double underscore prefix, and those attributes then are not be directly visible to outsiders.

Example

```
#!/usr/bin/python

class JustCounter:
  __secretCount = 0

  def count(self):
    self.__secretCount += 1
    print self.__secretCount

counter = JustCounter()
counter.count()
counter.count()
print counter.__secretCount
```

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

```
1

2

Traceback (most recent call last):

  File "test.py", line 12, in <module>

    print counter.__secretCount

AttributeError: JustCounter instance has no attribute '__secretCount'
```

Python protects those members by internally changing the name to include the class name. You can access such attributes as *object._className__attrName*. If you would replace your last line as following, then it works for you −

```
.........................
print counter._JustCounter__secretCount
```

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

```
1
2
2
```

CHAPTER 5: Exception Handling

List of Standard Exceptions −

Sr.No.	Exception Name & Description
1	Exception Base class for all exceptions
2	StopIteration Raised when the next() method of an iterator does not point to any object.
3	SystemExit Raised by the sys.exit() function.
4	StandardError Base class for all built-in exceptions except StopIteration and SystemExit.
5	ArithmeticError Base class for all errors that occur for numeric calculation.

6	OverflowError Raised when a calculation exceeds maximum limit for a numeric type.
7	FloatingPointError Raised when a floating point calculation fails.
8	ZeroDivisionError Raised when division or modulo by zero takes place for all numeric types.
9	AssertionError Raised in case of failure of the Assert statement.
10	AttributeError Raised in case of failure of attribute reference or assignment.
11	EOFError

	Raised when there is no input from either the raw_input() or input() function and the end of file is reached.
12	ImportError Raised when an import statement fails.
13	KeyboardInterrupt Raised when the user interrupts program execution, usually by pressing Ctrl+c.
14	LookupError Base class for all lookup errors.
15	IndexError Raised when an index is not found in a sequence.
16	KeyError Raised when the specified key is not found in the dictionary.

17	NameError Raised when an identifier is not found in the local or global namespace.
18	UnboundLocalError Raised when trying to access a local variable in a function or method but no value has been assigned to it.
19	EnvironmentError Base class for all exceptions that occur outside the Python environment.
20	IOError Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist.
21	IOError Raised for operating system-related errors.
22	SyntaxError

	Raised when there is an error in Python syntax.
23	IndentationError Raised when indentation is not specified properly.
24	SystemError Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit.
25	SystemExit Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit.
26	TypeError Raised when an operation or function is attempted that is invalid for the specified data type.
27	ValueError

	Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.
28	RuntimeError Raised when a generated error does not fall into any category.
29	NotImplementedError Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented.

Assertions in Python

An assertion is a sanity-check that you can turn on or turn off when you are done with your testing of the program.The easiest way to think of an assertion is to liken it to a raise-if statement (or to be more accurate, a raise-if-not statement). An expression is tested, and if the result comes up false, an exception is raised.Assertions are carried out by the assert statement, the newest keyword to Python, introduced in version 1.5.Programmers often place assertions at the start of a function to check for valid input, and after a function call to check for valid output.

The *assert* Statement

When it encounters an assert statement, Python evaluates the accompanying expression, which is hopefully true. If the expression is false, Python raises an *AssertionError* exception.

The syntax for assert is −

```
assert Expression[, Arguments]
```

If the assertion fails, Python uses Argument Expression as the argument for the AssertionError. AssertionError exceptions can be caught and handled like any other exception using the try-except statement, but if not handled, they will terminate the program and produce a traceback.

Example

Here is a function that converts a temperature from degrees Kelvin to degrees Fahrenheit. Since zero degrees Kelvin is as cold as it gets, the function bails out if it sees a negative temperature −

```
#!/usr/bin/python
def KelvinToFahrenheit(Temperature):
   assert (Temperature >= 0),"Colder than absolute zero!"
   return ((Temperature-273)*1.8)+32
print KelvinToFahrenheit(273)
print int(KelvinToFahrenheit(505.78))
print KelvinToFahrenheit(-5)
```

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

```
32.0
451
Traceback (most recent call last):
File "test.py", line 9, in <module>
print KelvinToFahrenheit(-5)
File "test.py", line 4, in KelvinToFahrenheit
assert (Temperature >= 0),"Colder than absolute zero!"
AssertionError: Colder than absolute zero!
```

What is Exception?

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 cannot 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 terminates and quits.

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.

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.
```

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.

Example

This example opens a file, writes content in the, file and comes out gracefully because there is no problem at all −

```
#!/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 produces the following result −

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

Example

This example tries to open a file where you do not have write permission, 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 produces the following result −

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

The *except* Clause with No Exceptions

You can also use the except statement with no exceptions defined as follows −

```
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.

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.
```

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.
   ......................
```

You cannot use *else* clause as well along with a finally clause.

Example

```
#!/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
```

Same example can be written more cleanly as follows −

```
#!/usr/bin/python

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

When an exception is thrown in the *try* block, the execution immediately passes to the *finally* block. After all the statements in the *finally* block are executed, the exception is raised again and is handled in the *except*statements if present in the next higher layer of the *try-except* statement.

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...
```

If you write 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 receives 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.

Example

Following is an example for a single exception −

```
#!/usr/bin/python

# Define a function here.
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 produces the following result −

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

Raising an Exception

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

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.

Example

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 the except clause as follows −

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

User-Defined Exceptions

Python also allows you to create your own exceptions by deriving classes from the standard built-in exceptions.

Here is an example related to *RuntimeError*. Here, a class is created that is subclassed from *RuntimeError*. This is useful when you need to display more specific information when an exception is caught.In the try block, the user-defined exception is raised and caught in the except block. The variable e is used to create an instance of the class *Networkerror*.

```
class Networkerror(RuntimeError):
  def __init__(self, arg):
    self.args = arg
```

So once you defined above class, you can raise the exception as follows −

```
try:
  raise Networkerror("Bad hostname")
except Networkerror,e:
  print e.args
```

CHAPTER 6:File Analysis

Hashing Algorithms

The most used algorithms to hash a file are MD5 and SHA-1. They are used because they are fast and they provide a good way to identify different files. The hash function only uses the contents of the file, not the name. Getting the same hash of two separating files means that there is a high probability the contents of the files are identical, even though they have different names.

MD5 File Hash in Python

The code is made to work with Python 2.7 and higher (including Python 3.x).

```
import hashlib
hasher = hashlib.md5()
with open('myfile.jpg', 'rb') as afile:
    buf = afile.read()
    hasher.update(buf)
print(hasher.hexdigest())
```

The code above calculates the MD5 digest of the file. The file is opened in rb mode, which means that you are going to read the file in binary mode. This is because the MD5 function needs to read the file as a sequence of bytes. This will make sure that you can hash any type of file, not only text files.It is important to notice the read function. When it is called with no arguments, like in this case, it will read all the contents of the file and load them into memory. This is dangerous if you are not sure of the file's size. A better version will be:

MD5 Hash for Large Files in Python

```
import hashlib
BLOCKSIZE = 65536
hasher = hashlib.md5()
with open('anotherfile.txt', 'rb') as afile:
    buf = afile.read(BLOCKSIZE)
    while len(buf) > 0:
        hasher.update(buf)
        buf = afile.read(BLOCKSIZE)
print(hasher.hexdigest())
```

If you need to use another algorithm just change the md5 call to another supported function, e.g. SHA1:

SHA1 File Hash in Python

```
import hashlib
BLOCKSIZE = 65536
hasher = hashlib.sha1()
with open('anotherfile.txt', 'rb') as afile:
    buf = afile.read(BLOCKSIZE)
    while len(buf) > 0:
        hasher.update(buf)
        buf = afile.read(BLOCKSIZE)
print(hasher.hexdigest())
```

shutil.copyfileobj(*fsrc*, *fdst*[, *length*])

Copy the contents of the file-like object *fsrc* to the file-like object *fdst*. The integer *length*, if given, is the buffer size. In particular, a negative *length*value means to copy the data without looping over the source data in chunks; by default the data is read in chunks to avoid uncontrolled memory consumption. Note that if the current file position of the *fsrc* object is not 0, only the contents from the current file position to the end of the file will be copied.

shutil.copyfile(*src*, *dst*)

Copy the contents (no metadata) of the file named *src* to a file named *dst*. *dst* must be the complete target file name; look at shutil.copy() for a copy that accepts a target directory path. If *src* and *dst* are the same files, Error is raised. The destination location must be writable; otherwise, an IOErrorexception will be raised. If *dst* already exists, it will be replaced. Special files such as character or block devices and pipes cannot be copied with this function. *src* and *dst* are path names given as strings.

shutil.copymode(*src*, *dst*)

Copy the permission bits from *src* to *dst*. The file contents, owner, and group are unaffected. *src* and *dst* are path names given as strings.

shutil.copystat(*src*, *dst*)

Copy the permission bits, last access time, last modification time, and flags from *src* to *dst*. The file contents, owner, and group are unaffected. *src*and *dst* are path names given as strings.

shutil.copy(*src*, *dst*)

Copy the file *src* to the file or directory *dst*. If *dst* is a directory, a file with the same basename as *src* is created (or overwritten) in the directory specified. Permission bits are copied. *src* and *dst* are path names given as strings.

shutil.copy2(*src*, *dst*)

Similar to shutil.copy(), but metadata is copied as well – in fact, this is just shutil.copy() followed by copystat(). This is similar to the Unix command cp -p.

shutil.ignore_patterns(**patterns*)

This factory function creates a function that can be used as a callable for copytree()'s *ignore* argument, ignoring files and directories that match one of the glob-style *patterns* provided. See the example below.

New in version 2.6.

shutil.copytree(*src*, *dst*, *symlinks=False*, *ignore=None*)

Recursively copy an entire directory tree rooted at *src*. The destination directory, named by *dst*, must not already exist; it will be created as well as missing parent directories. Permissions and times of directories are copied with copystat(), individual files are copied using shutil.copy2().

If *symlinks* is true, symbolic links in the source tree are represented as symbolic links in the new tree, but the metadata of the original links is NOT copied; if false or omitted, the contents and metadata of the linked files are copied to the new tree.

If *ignore* is given, it must be a callable that will receive as its arguments the directory being visited by copytree(), and a list of its contents, as returned by os.listdir(). Since copytree() is called recursively, the *ignore* callable will be called once for each directory that is copied. The callable must return a sequence of directory and file names relative to the current directory (i.e. a subset of the items in its second argument); these names will then be ignored in the copy process. ignore_patterns() can be used to create such a callable that ignores names based on glob-style patterns.

If exception(s) occur, an Error is raised with a list of reasons.

The source code for this should be considered an example rather than the ultimate tool.

Changed in version 2.3: Error is raised if any exceptions occur during copying, rather than printing a message.

Changed in version 2.5: Create intermediate directories needed to create *dst*, rather than raising an error. Copy permissions and times of directories using copystat().

Changed in version 2.6: Added the *ignore* argument to be able to influence what is being copied.

shutil.rmtree(*path*[, *ignore_errors*[, *onerror*]])

Delete an entire directory tree; *path* must point to a directory (but not a symbolic link to a directory). If *ignore_errors* is true, errors resulting from failed removals will be ignored; if false or omitted, such errors are handled by calling a handler specified by *onerror* or, if that is omitted, they raise an exception.

If *onerror* is provided, it must be a callable that accepts three parameters: *function*, *path*, and *excinfo*. The first parameter, *function*, is the function which raised the exception; it will be os.path.islink(), os.listdir(), os.remove() or os.rmdir(). The second parameter, *path*, will be the path name passed to *function*. The third parameter, *excinfo*, will be the exception information return by sys.exc_info(). Exceptions raised by *onerror* will not be caught.

Changed in version 2.6: Explicitly check for *path* being a symbolic link and raise OSError in that case.

shutil.move(*src*, *dst*)

Recursively move a file or directory (*src*) to another location (*dst*).

If the destination is an existing directory, then *src* is moved inside that directory. If the destination already exists but is not a directory, it may be overwritten depending on os.rename() semantics.

If the destination is on the current filesystem, then os.rename() is used. Otherwise, *src* is copied (using shutil.copy2()) to *dst* and then removed.

exception shutil.Error

This exception collects exceptions that are raised during a multi-file operation. For copytree(), the exception argument is a list of 3-tuples (*srcname*, *dstname*, *exception*).

CHAPTER 7: File Handling

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. 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).
- 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).

Here is a list of the different modes of opening a file −

Sr.No.	Modes & Description
1	r

	Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
2	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.
3	r+ Opens a file for both reading and writing. The file pointer placed at the beginning of the file.
4	rb+ Opens a file for both reading and writing in binary format. The file pointer placed at the beginning of the file.
5	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.

6	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.
7	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.
8	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.
9	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.

10	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.
11	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.
12	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.

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

Sr.No.	Attribute & Description

1	file.closed Returns true if file is closed, false otherwise.
2	file.mode Returns access mode with which file was opened.
3	file.name Returns name of the file.
4	file.softspace Returns false if space explicitly required with print, true otherwise.

Example

```
#!/usr/bin/python

# 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
print "Softspace flag : ", fo.softspace
```

This produces the following result −

```
Name of the file:  foo.txt
Closed or not :  False
Opening mode :  wb
Softspace flag :  0
```

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();
```

Example

```
#!/usr/bin/python

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

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

This produces the following result −

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

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);
```

Here, passed parameter is the content to be written into the opened file.

Example

```
#!/usr/bin/python

# Open a file
fo = open("foo.txt", "wb")
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!!
```

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. apart from text data.

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.

Example

Let's take a file *foo.txt*, which we 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 produces the following result −

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

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, it means use the beginning of the file as the reference position and 1 means use the current position as the reference position and if it is set to 2 then the end of the file would be taken as the reference position.

Example

Let us take a file *foo.txt*, which we created above.

```
#!/usr/bin/python

# Open a file
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 opend file
fo.close()
```

This produces the following result −

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

Renaming and Deleting Files

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.

Syntax

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

Example

Following is the example to rename an existing file *test1.txt* −

```
#!/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)
```

Example

Following is the example to delete an existing file *test2.txt* −

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

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

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")
```

Example

Following is the example to create a directory *test* in the current directory −

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

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

The *chdir()* Method

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

Following is the example to go into "/home/newdir" directory −

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

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

The *getcwd()* Method

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

Syntax

```
os.getcwd()
```

Example

Following is the example to give current directory −

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

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

The *rmdir()* Method

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

Following is the example to remove "/tmp/test" directory. It is required to give fully qualified name of the directory, otherwise it would search for that directory in the current directory.

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

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

File & Directory Related Methods

There are three important sources, which provide a wide range of utility methods to handle and manipulate files & directories on Windows and Unix operating systems. They are as follows −

- File Object Methods: The *file* object provides functions to manipulate files.
- OS Object Methods: This provides methods to process files as well as directories.

CHAPTER 8: GUI

Tkinter Programming

Tkinter is the standard GUI library for Python. Python when combined with Tkinter provides a fast and easy way to create GUI applications. Tkinter provides a powerful object-oriented interface to the Tk GUI toolkit.

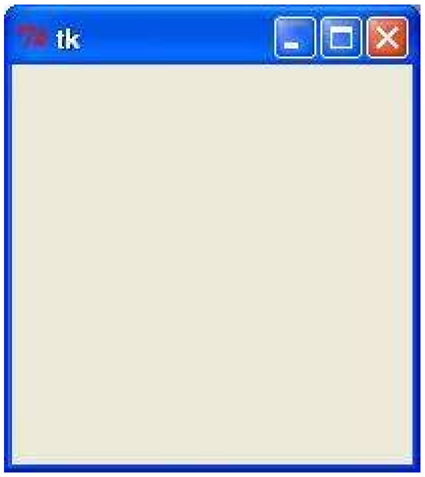
Creating a GUI application using Tkinter is an easy task. All you need to do is perform the following steps −

- Import the *Tkinter* module.
- Create the GUI application main window.
- Add one or more of the above-mentioned widgets to the GUI application.
- Enter the main event loop to take action against each event triggered by the user.

Example

```
#!/usr/bin/python

import tkinter
top = tkinter.Tk()
# Code to add widgets will go here...
top.mainloop()
```

This would create a following window −



Tkinter Widgets

Tkinter provides various controls, such as buttons, labels and text boxes used in a GUI application. These controls are commonly called widgets.

There are currently 15 types of widgets in Tkinter. We present these widgets as well as a brief description in the following table −

Sr.No.	Operator & Description
1	Button The Button widget is used to display buttons in your application.
2	Canvas The Canvas widget is used to draw shapes, such as lines, ovals, polygons and rectangles, in your application.
3	Checkbutton The Checkbutton widget is used to display a number of options as checkboxes. The user can select multiple options at a time.
4	Entry

	The Entry widget is used to display a single-line text field for accepting values from a user.
5	Frame The Frame widget is used as a container widget to organize other widgets.
6	Label The Label widget is used to provide a single-line caption for other widgets. It can also contain images.
7	Listbox The Listbox widget is used to provide a list of options to a user.
8	Menubutton The Menubutton widget is used to display menus in your application.
9	Menu The Menu widget is used to provide various commands to a user. These commands are contained inside Menubutton.

10	Message The Message widget is used to display multiline text fields for accepting values from a user.
11	Radiobutton The Radiobutton widget is used to display a number of options as radio buttons. The user can select only one option at a time.
12	Scale The Scale widget is used to provide a slider widget.
13	Scrollbar The Scrollbar widget is used to add scrolling capability to various widgets, such as list boxes.
14	Text The Text widget is used to display text in multiple lines.
15	Toplevel The Toplevel widget is used to provide a separate window container.

16	Spinbox The Spinbox widget is a variant of the standard Tkinter Entry widget, which can be used to select from a fixed number of values.
17	PanedWindow A PanedWindow is a container widget that may contain any number of panes, arranged horizontally or vertically.
18	LabelFrame A labelframe is a simple container widget. Its primary purpose is to act as a spacer or container for complex window layouts.
19	tkMessageBox This module is used to display message boxes in your applications.

Let us study these widgets in detail −

Standard attributes

Let us take a look at how some of their common attributes.such as sizes, colors and fonts are specified.

- Dimensions
- Colors
- Fonts

- Anchors
- Relief styles
- Bitmaps
- Cursors

Let us study them briefly −

Geometry Management

All Tkinter widgets have access to specific geometry management methods, which have the purpose of organizing widgets throughout the parent widget area. Tkinter exposes the following geometry manager classes: pack, grid, and place.

- The *pack()* Method − This geometry manager organizes widgets in blocks before placing them in the parent widget.
- The *grid()* Method − This geometry manager organizes widgets in a table-like structure in the parent widget.
- The *place()* Method − This geometry manager organizes widgets by placing them in a specific position in the parent widget.

CHAPTER 9: Data Structures

Creation of Linked list

A linked list is created by using the node class we studied in the last chapter. We create a Node object and create another class to use this ode object. We pass the appropriate values through the node object to point the to the next data elements. The below program creates the linked list with three data elements. In the next section we will see how to traverse the linked list.

```
class Node:
   def __init__(self, dataval=None):
      self.dataval = dataval
      self.nextval = None

class SLinkedList:
   def __init__(self):
      self.headval = None

list1 = SLinkedList()
list1.headval = Node("Mon")
e2 = Node("Tue")
e3 = Node("Wed")
# Link first Node to second node
list1.headval.nextval = e2

# Link second Node to third node
e2.nextval = e3
```

Traversing a Linked List

Singly linked lists can be traversed in only forwrad direction starting form the first data element. We simply print the value of the next data element by assgining the pointer of the next node to the current data element.

```
class Node:
   def __init__(self, dataval=None):
        self.dataval = dataval
        self.nextval = None

class SLinkedList:
    def __init__(self):
        self.headval = None

    def listprint(self):
        printval = self.headval
        while printval is not None:
            print (printval.dataval)
            printval = printval.nextval

list = SLinkedList()
list.headval = Node("Mon")
e2 = Node("Tue")
e3 = Node("Wed")

# Link first Node to second node
list.headval.nextval = e2

# Link second Node to third node
e2.nextval = e3

list.listprint()
```

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

```
Mon
Tue
Wed
```

Insertion in a Linked List

Inserting element in the linked list involves assigning the pointers from the existing nodes to the newly inserted node. Depending on whether the new data element is getting inserted at the beginning or at the middle or at the end of the linked list, we have the below scenarios.

Inserting at the Beginning of the Linked List.This involves pointing the next pointer of the new data node to the current head of the linked list. So the current head of the linked list becomes the second data element and the new node becomes the head of the linked list.

```
class Node:
   def __init__(self, dataval=None):
      self.dataval = dataval
      self.nextval = None

class SLinkedList:
   def __init__(self):
      self.headval = None

# Print the linked list
   def listprint(self):
      printval = self.headval
      while printval is not None:
         print (printval.dataval)
         printval = printval.nextval
   def AtBegining(self,newdata):
      NewNode = Node(newdata)

# Update the new nodes next val to existing node
      NewNode.nextval = self.headval
      self.headval = NewNode

list = SLinkedList()
list.headval = Node("Mon")
e2 = Node("Tue")
e3 = Node("Wed")

list.headval.nextval = e2
e2.nextval = e3

list.AtBegining("Sun")

list.listprint()
```

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

```
Sun
Mon
Tue
Wed
```

Inserting at the End of the Linked List

This involves pointing the next pointer of the the current last node of the linked list to the new data node. So the current last node of the linked list becomes the second last data node and the new node becomes the last node of the linked list.

```
class Node:
   def __init__(self, dataval=None):
      self.dataval = dataval
      self.nextval = None

class SLinkedList:
   def __init__(self):
      self.headval = None

# Function to add newnode
   def AtEnd(self, newdata):
      NewNode = Node(newdata)
      if self.headval is None:
         self.headval = NewNode
         return
      laste = self.headval
      while(laste.nextval):
         laste = laste.nextval
      laste.nextval=NewNode

# Print the linked list
   def listprint(self):
      printval = self.headval
      while printval is not None:
         print (printval.dataval)
         printval = printval.nextval
```

```
list = SLinkedList()
list.headval = Node("Mon")
e2 = Node("Tue")
e3 = Node("Wed")

list.headval.nextval = e2
e2.nextval = e3

list.AtEnd("Thu")

list.listprint()
```

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

```
Mon
Tue
Wed
Thu
```

Inserting in between two Data Nodes

This involves chaging the pointer of a specific node to point to the new node. That is possible by passing in both the new node and the existing node after which the new node will be inserted. So we define an additional class which will change the next pointer of the new node to the next pointer of middle node. Then assign the new node to next pointer of the middle node.

```
class Node:
   def __init__(self, dataval=None):
      self.dataval = dataval
      self.nextval = None

class SLinkedList:
   def __init__(self):
      self.headval = None

# Function to add node
   def Inbetween(self,middle_node,newdata):
      if middle_node is None:
          print("The mentioned node is absent")
          return

      NewNode = Node(newdata)
      NewNode.nextval = middle_node.nextval
      middle_node.nextval = NewNode

# Print the linked list
   def listprint(self):
      printval = self.headval
      while printval is not None:
          print (printval.dataval)
          printval = printval.nextval


list = SLinkedList()
list.headval = Node("Mon")
e2 = Node("Tue")
e3 = Node("Thu")

list.headval.nextval = e2
e2.nextval = e3

list.Inbetween(list.headval.nextval,"Fri")

list.listprint()
```

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

```
Mon
Tue
Fri
Thu
```

Removing an Item form a Liked List

We can remove an existing node using the key for that node. In the below program we locate the previous node of the node which is to be deleted. Then point the next pointer of this node to the next node of the node to be deleted.

```
class Node:
    def __init__(self, data=None):
        self.data = data
        self.next = None

class SLinkedList:
    def __init__(self):
        self.head = None

    def Atbegining(self, data_in):
        NewNode = Node(data_in)
        NewNode.next = self.head
        self.head = NewNode

# Function to remove node
    def RemoveNode(self, Removekey):

        HeadVal = self.head

        if (HeadVal is not None):
            if (HeadVal.data == Removekey):
                self.head = HeadVal.next
                HeadVal = None
                return

        while (HeadVal is not None):
            if HeadVal.data == Removekey:
                break
            prev = HeadVal
            HeadVal = HeadVal.next

        if (HeadVal == None):
            return

        prev.next = HeadVal.next

        HeadVal = None

    def LListprint(self):
        printval = self.head
        while (printval):
            print(printval.data),
            printval = printval.next


llist = SLinkedList()
llist.Atbegining("Mon")
llist.Atbegining("Tue")
llist.Atbegining("Wed")
llist.Atbegining("Thu")
llist.RemoveNode("Tue")
llist.LListprint()
```

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

```
Thu
Wed
Mon
```

PUSH into a Stack

```
class Stack:

    def __init__(self):
        self.stack = []

    def add(self, dataval):
# Use list append method to add element
        if dataval not in self.stack:
            self.stack.append(dataval)
            return True
        else:
            return False
# Use peek to look at the top of the stack

    def peek(self):
                return self.stack[0]
```

```
AStack = Stack()
AStack.add("Mon")
AStack.add("Tue")
AStack.peek()
print(AStack.peek())
AStack.add("Wed")
AStack.add("Thu")
print(AStack.peek())
```

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

```
Mon
Mon
```

POP from a Stack

As we know we can remove only the too most data element from the stack, we implement a python program which does that. The remove function in the following program returns the top most element. we check the top element by calculating the size of the stack first and then use the in-built pop() method to find out the top most element.

```
class Stack:

   def __init__(self):
      self.stack = []

   def add(self, dataval):
# Use list append method to add element
      if dataval not in self.stack:
         self.stack.append(dataval)
         return True
      else:
         return False

# Use list pop method to remove element
   def remove(self):
      if len(self.stack) <= 0:
         return ("No element in the Stack")
      else:
            return self.stack.pop()

AStack = Stack()
AStack.add("Mon")
AStack.add("Tue")
print(AStack.remove())
AStack.add("Wed")
AStack.add("Thu")
print(AStack.remove())
```

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

```
Tue
Thu
```

Adding Elements to a Queue

In the below example we create a queue class where we implement the First-in-First-Out method. We use the in-built insert method for adding data elements.

```
class Queue:

  def __init__(self):
      self.queue = list()

  def addtoq(self,dataval):
# Insert method to add element
      if dataval not in self.queue:
          self.queue.insert(0,dataval)
          return True
      return False

  def size(self):
      return len(self.queue)

TheQueue = Queue()
TheQueue.addtoq("Mon")
TheQueue.addtoq("Tue")
TheQueue.addtoq("Wed")
print(TheQueue.size())
```

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

```
3
```

Removing Element from a Queue

In the below example we create a queue class where we insert the data and then remove the data using the in-built pop method.

.

```
class Queue:

  def __init__(self):
      self.queue = list()

  def addtoq(self,dataval):
# Insert method to add element
      if dataval not in self.queue:
          self.queue.insert(0,dataval)
          return True
      return False
# Pop method to remove element
  def removefromq(self):
      if len(self.queue)>0:
          return self.queue.pop()
      return ("No elements in Queue!")

TheQueue = Queue()
TheQueue.addtoq("Mon")
TheQueue.addtoq("Tue")
TheQueue.addtoq("Wed")
print(TheQueue.removefromq())
print(TheQueue.removefromq())
```

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

```
Mon
Tue
```

Tree represents the nodes connected by edges. It is a non-linear data structure. It has the following properties.

- One node is marked as Root node.
- Every node other than the root is associated with one parent node.
- Each node can have an arbiatry number of chid node.

We create a tree data structure in python by using the concept os node discussed earlier. We designate one node as root node and then add more nodes as child nodes. Below is program to create the root node.

Create Root

We just create a Node class and add assign a value to the node. This becomes tree with only a root node.

```
class Node:

   def __init__(self, data):

      self.left = None
      self.right = None
      self.data = data


   def PrintTree(self):
      print(self.data)

root = Node(10)

root.PrintTree()
```

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

```
10
```

Inserting into a Tree

To insert into a tree we use the same node class created above and add a insert class to it. The insert class compares the value of the node to the parent node and decides to add it as a left node or a right node. Finally the PrintTree class is used to print the tree.

```
class Node:

    def __init__(self, data):

        self.left = None
        self.right = None
        self.data = data

    def insert(self, data):
# Compare the new value with the parent node
        if self.data:
            if data < self.data:
                if self.left is None:
                    self.left = Node(data)
                else:
                    self.left.insert(data)
            elif data > self.data:
                if self.right is None:
                    self.right = Node(data)
                else:
                    self.right.insert(data)
        else:
            self.data = data

# Print the tree
    def PrintTree(self):
        if self.left:
            self.left.PrintTree()
        print( self.data),
        if self.right:
            self.right.PrintTree()

# Use the insert method to add nodes
root = Node(12)
root.insert(6)
root.insert(14)
root.insert(3)

root.PrintTree()
```

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

```
3 6 12 14
```

Travesring a Tree

The tree can be traversed by deciding on a sequence to visit each node. As we can clearly see we can start at a node then visit the left sub-tree first and right sub-tree next. Or we can also visit the right sub-tree first and left sub-tree next. Accordingly there are different names for these tree traversal methods. We study them in detail in the chapter implementing the tree traversal algorithms here.

Search for a value in a B-tree

Searching for a value in a tree involves comparing the incoming value with the value exiting nodes. Here also we traverse the nodes from left to right and then finally with the parent. If the searched for value does not match any of the exitign value, then we return not found message else the found message is returned.

```
class Node:

    def __init__(self, data):

        self.left = None
        self.right = None
        self.data = data

# Insert method to create nodes
    def insert(self, data):

        if self.data:
            if data < self.data:
                if self.left is None:
                    self.left = Node(data)
                else:
                    self.left.insert(data)
            elif data > self.data:
                if self.right is None:
                    self.right = Node(data)
                else:
                    self.right.insert(data)
        else:
            self.data = data
# findval method to compare the value with nodes
    def findval(self, lkpval):
        if lkpval < self.data:
            if self.left is None:
                return str(lkpval)+" Not Found"
            return self.left.findval(lkpval)
        elif lkpval > self.data:
            if self.right is None:
                return str(lkpval)+" Not Found"
            return self.right.findval(lkpval)
        else:
            print(str(self.data) + ' is found')
# Print the tree
    def PrintTree(self):
        if self.left:
            self.left.PrintTree()
        print( self.data),
        if self.right:
            self.right.PrintTree()


root = Node(12)
root.insert(6)
root.insert(14)
root.insert(3)
print(root.findval(7))
print(root.findval(14))
```

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

```
7 Not Found
14 is found
```

CHAPTER 10: DATABASE PROGRAMMING

What is MySQLdb?

MySQLdb is an interface for connecting to a MySQL database server from Python. It implements the Python Database API v2.0 and is built on top of the MySQL C API.

How do I Install MySQLdb?

Before proceeding, you make sure you have MySQLdb installed on your machine. Just type the following in your Python script and execute it −

```
#!/usr/bin/python

import MySQLdb
```

If it produces the following result, then it means MySQLdb module is not installed −

```
Traceback (most recent call last):
  File "test.py", line 3, in <module>
    import MySQLdb
ImportError: No module named MySQLdb
```

To install MySQLdb module, use the following command −

For Ubuntu, use the following command -

```
$ sudo apt-get install python-pip python-dev libmysqlclient-dev
```

For Fedora, use the following command -

```
$ sudo dnf install python python-devel mysql-devel redhat-rpm-config gcc
```

For Python command prompt, use the following command -

```
pip install MySQL-python
```

Note − Make sure you have root privilege to install above module.

Database Connection

Before connecting to a MySQL database, make sure of the followings −

- You have created a database TESTDB.
- You have created a table EMPLOYEE in TESTDB.
- This table has fields FIRST_NAME, LAST_NAME, AGE, SEX and INCOME.
- User ID "testuser" and password "test123" are set to access TESTDB.
- Python module MySQLdb is installed properly on your machine.
- You have gone through MySQL tutorial to understand MySQL Basics.

Example

Following is the example of connecting with MySQL database "TESTDB"

```
#!/usr/bin/python

import MySQLdb

# Open database connection
db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )

# prepare a cursor object using cursor() method
cursor = db.cursor()

# execute SQL query using execute() method.
cursor.execute("SELECT VERSION()")

# Fetch a single row using fetchone() method.
data = cursor.fetchone()
print "Database version : %s " % data

# disconnect from server
db.close()
```

While running this script, it is producing the following result in my Linux machine.

```
Database version : 5.0.45
```

If a connection is established with the datasource, then a Connection Object is returned and saved into db for further use, otherwise db is set to None. Next, db object is used to create a cursor object, which in turn is used to execute SQL queries. Finally, before coming out, it ensures that database connection is closed and resources are released.

Creating Database Table

Once a database connection is established, we are ready to create tables or records into the database tables using execute method of the created cursor.

Example

Let us create Database table EMPLOYEE −

```
#!/usr/bin/python

import MySQLdb

# Open database connection
db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )

# prepare a cursor object using cursor() method
cursor = db.cursor()

# Drop table if it already exist using execute() method.
cursor.execute("DROP TABLE IF EXISTS EMPLOYEE")

# Create table as per requirement
sql = """CREATE TABLE EMPLOYEE (
        FIRST_NAME  CHAR(20) NOT NULL,
        LAST_NAME  CHAR(20),
        AGE INT,
        SEX CHAR(1),
        INCOME FLOAT )"""

cursor.execute(sql)

# disconnect from server
db.close()
```

INSERT Operation

It is required when you want to create your records into a database table.

Example

The following example, executes SQL *INSERT* statement to create a record into EMPLOYEE table −

```
#!/usr/bin/python

import MySQLdb

# Open database connection
db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )

# prepare a cursor object using cursor() method
cursor = db.cursor()

# Prepare SQL query to INSERT a record into the database.
sql = """INSERT INTO EMPLOYEE(FIRST_NAME,
         LAST_NAME, AGE, SEX, INCOME)
         VALUES ('Mac', 'Mohan', 20, 'M', 2000)"""
try:
   # Execute the SQL command
   cursor.execute(sql)
   # Commit your changes in the database
   db.commit()
except:
   # Rollback in case there is any error
   db.rollback()

# disconnect from server
db.close()
```

Above example can be written as follows to create SQL queries dynamically −

```
#!/usr/bin/python

import MySQLdb
```

```
# Open database connection
db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )

# prepare a cursor object using cursor() method
cursor = db.cursor()

# Prepare SQL query to INSERT a record into the database.
sql = "INSERT INTO EMPLOYEE(FIRST_NAME, \
       LAST_NAME, AGE, SEX, INCOME) \
       VALUES ('%s', '%s', '%d', '%c', '%d' )" % \
       ('Mac', 'Mohan', 20, 'M', 2000)
try:
   # Execute the SQL command
   cursor.execute(sql)
   # Commit your changes in the database
   db.commit()
except:
   # Rollback in case there is any error
   db.rollback()

# disconnect from server
db.close()
```

Example

Following code segment is another form of execution where you can pass parameters directly −

```
..................................

user_id = "test123"

password = "password"


con.execute('insert into Login values("%s", "%s")' % \

             (user_id, password))

..................................
```

READ Operation

READ Operation on any database means to fetch some useful information from the database.

Once our database connection is established, you are ready to make a query into this database. You can use either fetchone() method to fetch single record or fetchall() method to fetech multiple values from a database table.

- fetchone() − It fetches the next row of a query result set. A result set is an object that is returned when a cursor object is used to query a table.
- fetchall() − It fetches all the rows in a result set. If some rows have already been extracted from the result set, then it retrieves the remaining rows from the result set.
- rowcount − This is a read-only attribute and returns the number of rows that were affected by an execute() method.

Example

The following procedure queries all the records from EMPLOYEE table having salary more than 1000 −

```
#!/usr/bin/python

import MySQLdb

# Open database connection
db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )

# prepare a cursor object using cursor() method
cursor = db.cursor()

sql = "SELECT * FROM EMPLOYEE \
      WHERE INCOME > '%d'" % (1000)
try:
   # Execute the SQL command
   cursor.execute(sql)
   # Fetch all the rows in a list of lists.
   results = cursor.fetchall()
   for row in results:
      fname = row[0]
      lname = row[1]
      age = row[2]
      sex = row[3]
      income = row[4]
      # Now print fetched result
      print "fname=%s,lname=%s,age=%d,sex=%s,income=%d" % \
             (fname, lname, age, sex, income )
except:
   print "Error: unable to fecth data"

# disconnect from server
db.close()
```

This will produce the following result −

```
fname=Mac, lname=Mohan, age=20, sex=M, income=2000
```

Update Operation

UPDATE Operation on any database means to update one or more records, which are already available in the database.

The following procedure updates all the records having SEX as 'M'. Here, we increase AGE of all the males by one year.

Example

```
#!/usr/bin/python

import MySQLdb

# Open database connection
db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )

# prepare a cursor object using cursor() method
cursor = db.cursor()

# Prepare SQL query to UPDATE required records
sql = "UPDATE EMPLOYEE SET AGE = AGE + 1
                          WHERE SEX = '%c'" % ('M')
```

```
try:
   # Execute the SQL command
   cursor.execute(sql)
   # Commit your changes in the database
   db.commit()
except:
   # Rollback in case there is any error
   db.rollback()

# disconnect from server
db.close()
```

DELETE Operation

DELETE operation is required when you want to delete some records from your database. Following is the procedure to delete all the records from EMPLOYEE where AGE is more than 20 −

Example

```
#!/usr/bin/python

import MySQLdb

# Open database connection
db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )

# prepare a cursor object using cursor() method
cursor = db.cursor()

# Prepare SQL query to DELETE required records
sql = "DELETE FROM EMPLOYEE WHERE AGE > '%d'" % (20)
try:
   # Execute the SQL command
   cursor.execute(sql)
   # Commit your changes in the database
   db.commit()
except:
   # Rollback in case there is any error
   db.rollback()
```

```
# disconnect from server
db.close()
```

Performing Transactions

Transactions are a mechanism that ensures data consistency. Transactions have the following four properties −

- Atomicity − Either a transaction completes or nothing happens at all.
- Consistency − A transaction must start in a consistent state and leave the system in a consistent state.
- Isolation − Intermediate results of a transaction are not visible outside the current transaction.
- Durability − Once a transaction was committed, the effects are persistent, even after a system failure.

The Python DB API 2.0 provides two methods to either *commit* or *rollback* a transaction.

Example

You already know how to implement transactions. Here is again similar example −

```
# Prepare SQL query to DELETE required records
sql = "DELETE FROM EMPLOYEE WHERE AGE > '%d'" % (20)
try:
   # Execute the SQL command
   cursor.execute(sql)
   # Commit your changes in the database
   db.commit()
except:
   # Rollback in case there is any error
   db.rollback()
```

COMMIT Operation

Commit is the operation, which gives a green signal to database to finalize the changes, and after this operation, no change can be reverted back.

Here is a simple example to call commit method.

```
db.commit()
```

ROLLBACK Operation

If you are not satisfied with one or more of the changes and you want to revert back those changes completely, then use rollback() method.

Here is a simple example to call rollback() method.

```
db.rollback()
```

Disconnecting Database

To disconnect Database connection, use close() method.

```
db.close()
```

If the connection to a database is closed by the user with the close() method, any outstanding transactions are rolled back by the DB. However, instead of depending on any of DB lower level implementation details, your application would be better off calling commit or rollback explicitly.

Handling Errors

There are many sources of errors. A few examples are a syntax error in an executed SQL statement, a connection failure, or calling the fetch method for an already canceled or finished statement handle.

The DB API defines a number of errors that must exist in each database module. The following table lists these exceptions.

Sr.No.	Exception & Description

1	Warning Used for non-fatal issues. Must subclass StandardError.
2	Error Base class for errors. Must subclass StandardError.
3	InterfaceError Used for errors in the database module, not the database itself. Must subclass Error.
4	DatabaseError Used for errors in the database. Must subclass Error.
5	DataError Subclass of DatabaseError that refers to errors in the data.
6	OperationalError

	Subclass of DatabaseError that refers to errors such as the loss of a connection to the database. These errors are generally outside of the control of the Python scripter.
7	IntegrityError Subclass of DatabaseError for situations that would damage the relational integrity, such as uniqueness constraints or foreign keys.
8	InternalError Subclass of DatabaseError that refers to errors internal to the database module, such as a cursor no longer being active.
9	ProgrammingError Subclass of DatabaseError that refers to errors such as a bad table name and other things that can safely be blamed on you.
10	NotSupportedError Subclass of DatabaseError that refers to trying to call unsupported functionality.

Your Python scripts should handle these errors, but before using any of the above exceptions, make sure your MySQLdb has support for that exception. You can get more information about them by reading the DB API 2.0 specification.

CHAPTER 11: SOCKET PROGRAMMING

What is Sockets?

Sockets are the endpoints of a bidirectional communications channel. Sockets may communicate within a process, between processes on the same machine, or between processes on different continents.

Sockets may be implemented over a number of different channel types: Unix domain sockets, TCP, UDP, and so on. The *socket* library provides specific classes for handling the common transports as well as a generic interface for handling the rest.

Sockets have their own vocabulary −

Sr.No.	Term & Description
1	Domain The family of protocols that is used as the transport mechanism. These values are constants such as AF_INET, PF_INET, PF_UNIX, PF_X25, and so on.
2	type The type of communications between the two endpoints, typically SOCK_STREAM for connection-oriented protocols and SOCK_DGRAM for connectionless protocols.

3	protocol Typically zero, this may be used to identify a variant of a protocol within a domain and type.
4	hostname The identifier of a network interface − ● A string, which can be a host name, a dotted-quad address, or an IPV6 address in colon (and possibly dot) notation ● A string "<broadcast>", which specifies an INADDR_BROADCAST address. ● A zero-length string, which specifies INADDR_ANY, or ● An Integer, interpreted as a binary address in host byte order.
5	port Each server listens for clients calling on one or more ports. A port may be a Fixnum port number, a string containing a port number, or the name of a service.

The *socket* Module

To create a socket, you must use the *socket.socket()* function available in *socket* module, which has the general syntax −

s = socket.socket (socket_family, socket_type, protocol=0)

Here is the description of the parameters −

- socket_family − This is either AF_UNIX or AF_INET, as explained earlier.
- socket_type − This is either SOCK_STREAM or SOCK_DGRAM.
- protocol − This is usually left out, defaulting to 0.

Once you have *socket* object, then you can use required functions to create your client or server program. Following is the list of functions required −

Server Socket Methods

Sr.No.	Method & Description
1	s.bind() This method binds address (hostname, port number pair) to socket.
2	s.listen() This method sets up and start TCP listener.
3	s.accept() This passively accept TCP client connection, waiting until connection arrives (blocking).

Client Socket Methods

Sr.No.	Method & Description
1	s.connect() This method actively initiates TCP server connection.

General Socket Methods

Sr.No.	Method & Description
1	s.recv() This method receives TCP message
2	s.send() This method transmits TCP message
3	s.recvfrom() This method receives UDP message
4	s.sendto()

	This method transmits UDP message
5	s.close() This method closes socket
6	socket.gethostname() Returns the hostname.

A Simple Server

To write Internet servers, we use the socket function available in socket module to create a socket object. A socket object is then used to call other functions to setup a socket server.

Now call bind(hostname, port) function to specify a *port* for your service on the given host.

Next, call the *accept* method of the returned object. This method waits until a client connects to the port you specified, and then returns a *connection* object that represents the connection to that client.

```
#!/usr/bin/python           # This is server.py file

import socket               # Import socket module

s = socket.socket()         # Create a socket object
host = socket.gethostname() # Get local machine name
port = 12345                # Reserve a port for your service.
s.bind((host, port))        # Bind to the port

s.listen(5)                 # Now wait for client connection.
while True:
   c, addr = s.accept()     # Establish connection with client.
   print 'Got connection from', addr
   c.send('Thank you for connecting')
   c.close()                # Close the connection
```

A Simple Client

Let us write a very simple client program which opens a connection to a given port 12345 and given host. This is very simple to create a socket client using Python's *socket* module function.

The socket.connect(hosname, port) opens a TCP connection to *hostname*on the *port*. Once you have a socket open, you can read from it like any IO object. When done, remember to close it, as you would close a file.

The following code is a very simple client that connects to a given host and port, reads any available data from the socket, and then exits −

```
#!/usr/bin/python           # This is client.py file

import socket               # Import socket module

s = socket.socket()         # Create a socket object
host = socket.gethostname() # Get local machine name
port = 12345                # Reserve a port for your service.

s.connect((host, port))
print s.recv(1024)
s.close                     # Close the socket when done
```

Now run this server.py in background and then run above client.py to see the result.

```
# Following would start a server in background.
$ python server.py &

# Once server is started run client as follows:
$ python client.py
```

This would produce following result −

```
Got connection from ('127.0.0.1', 48437)
Thank you for connecting
```

Python Internet modules

A list of some important modules in Python Network/Internet programming.

Protocol	Common function	Port No	Python module
HTTP	Web pages	80	httplib, urllib, xmlrpclib
NNTP	Usenet news	119	nntplib
FTP	File transfers	20	ftplib, urllib
SMTP	Sending email	25	smtplib
POP3	Fetching email	110	poplib
IMAP4	Fetching email	143	imaplib
Telnet	Command lines	23	telnetlib
Gopher	Document transfers	70	gopherlib, urllib

Please check all the libraries mentioned above to work with FTP, SMTP, POP, and IMAP protocols.

CHAPTER 12: DJANGO

Create a Project

Whether you are on Windows or Linux, just get a terminal or a cmd prompt and navigate to the place you want your project to be created, then use this code −

```
$ django-admin startproject myproject
```

This will create a "myproject" folder with the following structure −

```
myproject/
  manage.py
  myproject/
    __init__.py
    settings.py
    urls.py
    wsgi.py
```

The Project Structure

The “myproject” folder is just your project container, it actually contains two elements −

- manage.py − This file is kind of your project local django-admin for interacting with your project via command line (start the development server, sync db...). To get a full list of command accessible via manage.py you can use the code −

```
$ python manage.py help
```

- The “myproject” subfolder − This folder is the actual python package of your project. It contains four files −

- __init__.py − Just for python, treat this folder as package.
- settings.py − As the name indicates, your project settings.
- urls.py − All links of your project and the function to call. A kind of ToC of your project.
- wsgi.py − If you need to deploy your project over WSGI.

Setting Up Your Project

Your project is set up in the subfolder myproject/settings.py. Following are some important options you might need to set −

```
DEBUG = True
```

This option lets you set if your project is in debug mode or not. Debug mode lets you get more information about your project's error. Never set it to 'True' for a live project. However, this has to be set to 'True' if you want the Django light server to serve static files. Do it only in the development mode.

```
DATABASES = {
  'default': {
    'ENGINE': 'django.db.backends.sqlite3',
    'NAME': 'database.sql',
    'USER': '',
    'PASSWORD': '',
    'HOST': '',
    'PORT': '',
  }
}
```

Database is set in the 'Database' dictionary. The example above is for SQLite engine. As stated earlier, Django also supports −

- MySQL (django.db.backends.mysql)
- PostGreSQL (django.db.backends.postgresql_psycopg2)

- Oracle (django.db.backends.oracle) and NoSQL DB
- MongoDB (django_mongodb_engine)

Before setting any new engine, make sure you have the correct db driver installed.

You can also set others options like: TIME_ZONE, LANGUAGE_CODE, TEMPLATE…

Now that your project is created and configured make sure it's working −

```
$ python manage.py runserver
```

You will get something like the following on running the above code −

```
Validating models...


0 errors found
September 03, 2015 - 11:41:50
Django version 1.6.11, using settings 'myproject.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Create an Application

We assume you are in your project folder. In our main “myproject” folder, the same folder then manage.py −

```
$ python manage.py startapp myapp
```

You just created myapp application and like project, Django create a “myapp” folder with the application structure −

```
myapp/
  __init__.py
```

```
admin.py
models.py
tests.py
views.py
```

- __init__.py − Just to make sure python handles this folder as a package.
- admin.py − This file helps you make the app modifiable in the admin interface.
- models.py − This is where all the application models are stored.
- tests.py − This is where your unit tests are.
- views.py − This is where your application views are.

Get the Project to Know About Your Application

At this stage we have our "myapp" application, now we need to register it with our Django project "myproject". To do so, update INSTALLED_APPS tuple in the settings.py file of your project (add your app name) −

```
INSTALLED_APPS = (
   'django.contrib.admin',
   'django.contrib.auth',
   'django.contrib.contenttypes',
   'django.contrib.sessions',
   'django.contrib.messages',
   'django.contrib.staticfiles',
   'myapp',
)
```

Starting the Admin Interface

The Admin interface depends on the django.countrib module. To have it working you need to make sure some modules are imported in the INSTALLED_APPS and MIDDLEWARE_CLASSES tuples of the myproject/settings.py file.

For INSTALLED_APPS make sure you have −

```
INSTALLED_APPS = (
   'django.contrib.admin',
   'django.contrib.auth',
   'django.contrib.contenttypes',
   'django.contrib.sessions',
   'django.contrib.messages',
   'django.contrib.staticfiles',
   'myapp',
)
```

For MIDDLEWARE_CLASSES −

```
MIDDLEWARE_CLASSES = (
   'django.contrib.sessions.middleware.SessionMiddleware',
   'django.middleware.common.CommonMiddleware',
   'django.middleware.csrf.CsrfViewMiddleware',
   'django.contrib.auth.middleware.AuthenticationMiddleware',
   'django.contrib.messages.middleware.MessageMiddleware',
   'django.middleware.clickjacking.XFrameOptionsMiddleware',
)
```

Before launching your server, to access your Admin Interface, you need to initiate the database −

```
$ python manage.py migrate
```

syncdb will create necessary tables or collections depending on your db type, necessary for the admin interface to run. Even if you don't have a superuser, you will be prompted to create one.

If you already have a superuser or have forgotten it, you can always create one using the following code −

```
$ python manage.py createsuperuser
```

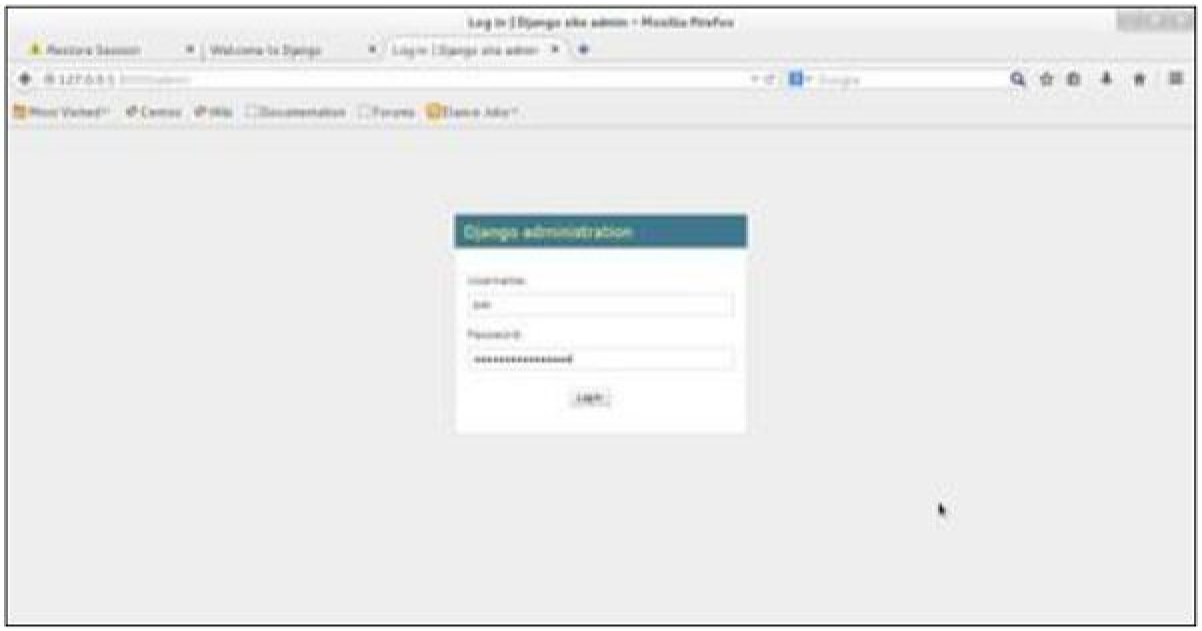
Now to start the Admin Interface, we need to make sure we have configured a URL for our admin interface. Open the myproject/url.py and you should have something like −

```
from django.conf.urls import patterns, include, url

from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns('',
   # Examples:
   # url(r'^$', 'myproject.views.home', name = 'home'),
   # url(r'^blog/', include('blog.urls')),

   url(r'^admin/', include(admin.site.urls)),
)
```

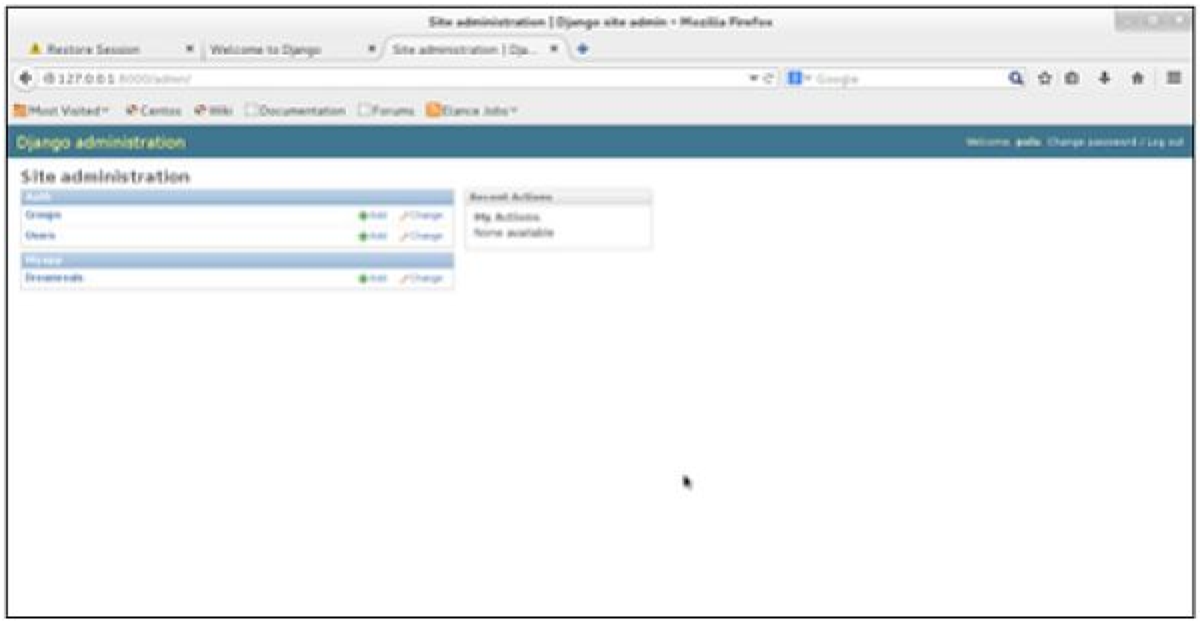
Now just run the server.

```
$ python manage.py runserver
```

And your admin interface is accessible at: http://127.0.0.1:8000/admin/



Once connected with your superuser account, you will see the following screen −



That interface will let you administrate Django groups and users, and all registered models in your app. The interface gives you the ability to do at least the "CRUD" (Create, Read, Update, Delete) operations on your models.

In Django, views have to be created in the app views.py file.

Simple View

We will create a simple view in myapp to say "welcome to my app!"

See the following view −

```
from django.http import HttpResponse

def hello(request):
   text = """<h1>welcome to my app !</h1>"""
   return HttpResponse(text)
```

In this view, we use HttpResponse to render the HTML (as you have probably noticed we have the HTML hard coded in the view). To see this view as a page we just need to map it to a URL (this will be discussed in an upcoming chapter).

We used HttpResponse to render the HTML in the view before. This is not the best way to render pages. Django supports the MVT pattern so to make the precedent view, Django - MVT like, we will need −

A template: myapp/templates/hello.html

And now our view will look like −

```
from django.shortcuts import render

def hello(request):
  return render(request, "myapp/template/hello.html", {})
```

Views can also accept parameters −

```
from django.http import HttpResponse

def hello(request, number):
  text = "<h1>welcome to my app number %s!</h1>"% number
  return HttpResponse(text)
```

When linked to a URL, the page will display the number passed as a parameter. Note that the parameters will be passed via the URL (discussed in the next chapter).

Now that we have a working view as explained in the previous chapters. We want to access that view via a URL. Django has his own way for URL mapping and it's done by editing your project url.py file (myproject/url.py). The url.py file looks like −

```
from django.conf.urls import patterns, include, url
from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns('',
  #Examples
  #url(r'^$', 'myproject.view.home', name = 'home'),
  #url(r'^blog/', include('blog.urls')),
   url(r'^admin', include(admin.site.urls)),
)
```

When a user makes a request for a page on your web app, Django controller takes over to look for the corresponding view via the url.py file, and then return the HTML response or a 404 not found error, if not found. In url.py, the most important thing is the "urlpatterns" tuple. It's where you define the mapping between URLs and views. A mapping is a tuple in URL patterns like −

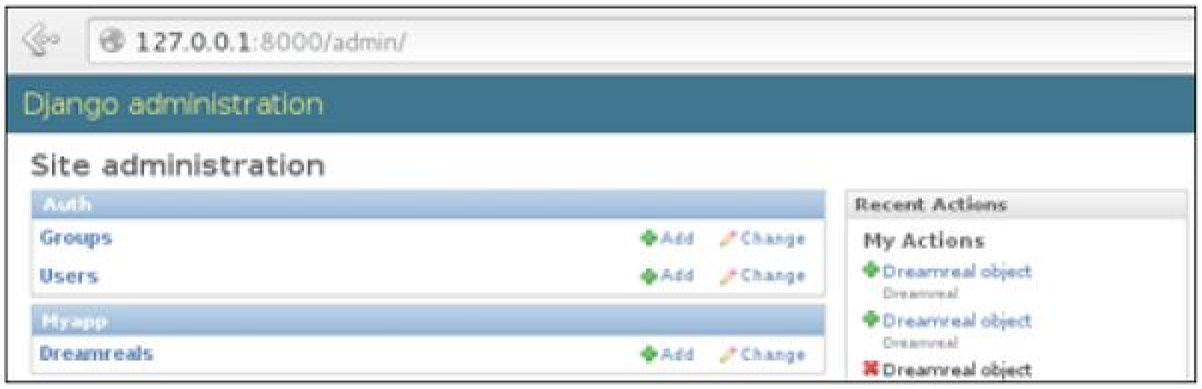
```
from django.conf.urls import patterns, include, url
from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns('',
   #Examples
   #url(r'^$', 'myproject.view.home', name = 'home'),
   #url(r'^blog/', include('blog.urls')),

   url(r'^admin', include(admin.site.urls)),
   url(r'^hello/', 'myapp.views.hello', name = 'hello'),
)
```

The marked line maps the URL "/home" to the hello view created in myapp/view.py file. As you can see above a mapping is composed of three elements −

- The pattern − A regexp matching the URL you want to be resolved and map. Everything that can work with the python 're' module is eligible for the pattern (useful when you want to pass parameters via url).
- The python path to the view − Same as when you are importing a module.
- The name − In order to perform URL reversing, you'll need to use named URL patterns as done in the examples above. Once done, just start the server to access your view via :http://127.0.0.1/hello

Organizing Your URLs

So far, we have created the URLs in “myprojects/url.py” file, however as stated earlier about Django and creating an app, the best point was to be able to reuse applications in different projects. You can easily see what the problem is, if you are saving all your URLs in the “projecturl.py” file. So best practice is to create an “url.py” per application and to include it in our main projects url.py file (we included admin URLs for admin interface before).



How is it Done?

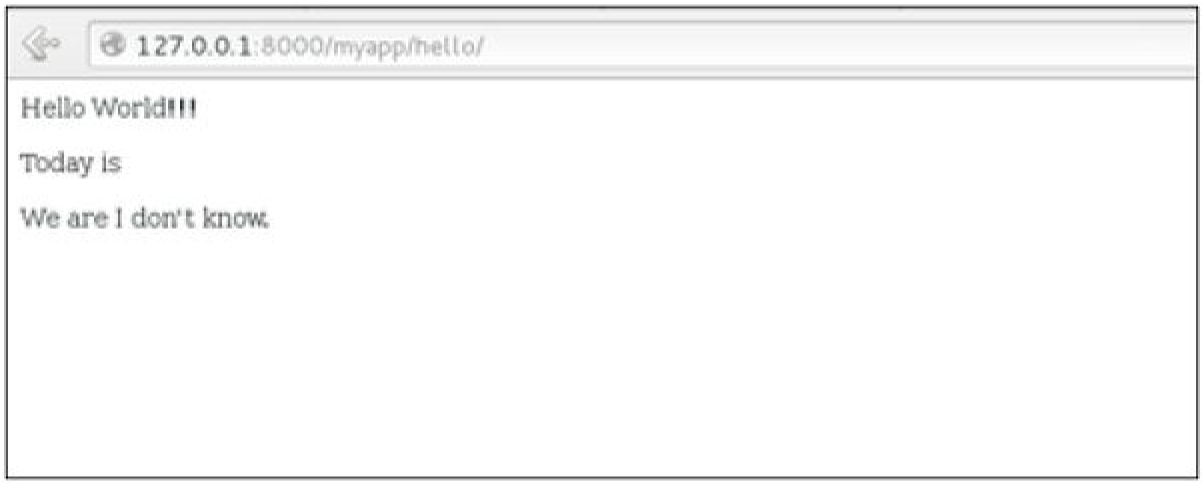
We need to create an url.py file in myapp using the following code −

```
from django.conf.urls import patterns, include, url

urlpatterns = patterns('', url(r'^hello/', 'myapp.views.hello', name = 'hello'),)
```

Then myproject/url.py will change to the following −

```
from django.conf.urls import patterns, include, url
from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns('',
   #Examples
   #url(r'^$', 'myproject.view.home', name = 'home'),
   #url(r'^blog/', include('blog.urls')),

   url(r'^admin', include(admin.site.urls)),
   url(r'^myapp/', include('myapp.urls')),
)
```

We have included all URLs from myapp application. The home.html that was accessed through "/hello" is now "/myapp/hello" which is a better and more understandable structure for the web app.



Hello World!!!

Today is

We are I don't know.

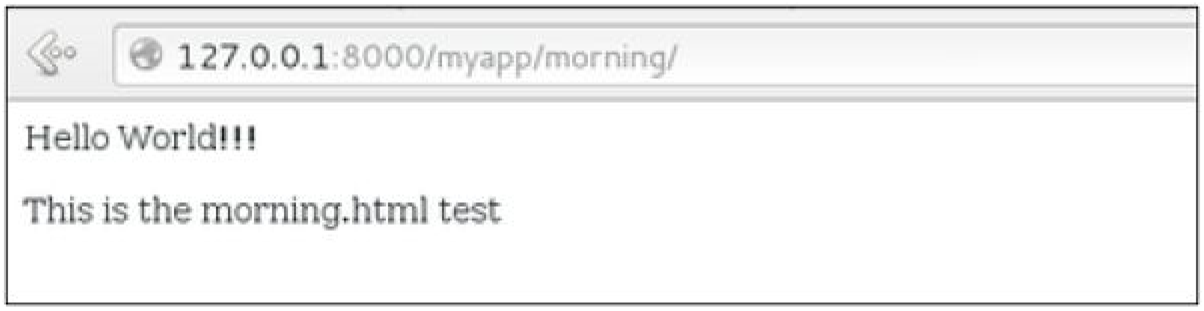
Now let's imagine we have another view in myapp "morning" and we want to map it in myapp/url.py, we will then change our myapp/url.py to −

```
from django.conf.urls import patterns, include, url

urlpatterns = patterns('',
   url(r'^hello/', 'myapp.views.hello', name = 'hello'),
   url(r'^morning/', 'myapp.views.morning', name = 'morning'),
)
```

This can be re-factored to −

```
from django.conf.urls import patterns, include, url

urlpatterns = patterns('myapp.views',
   url(r'^hello/', 'hello', name = 'hello'),
   url(r'^morning/', 'morning', name = 'morning'),)
```

As you can see, we now use the first element of our urlpatterns tuple. This can be useful when you want to change your app name.



Hello World!!!

This is the morning.html test

Sending Parameters to Views

We now know how to map URL, how to organize them, now let us see how to send parameters to views. A classic sample is the article example (you want to access an article via “/articles/article_id”).

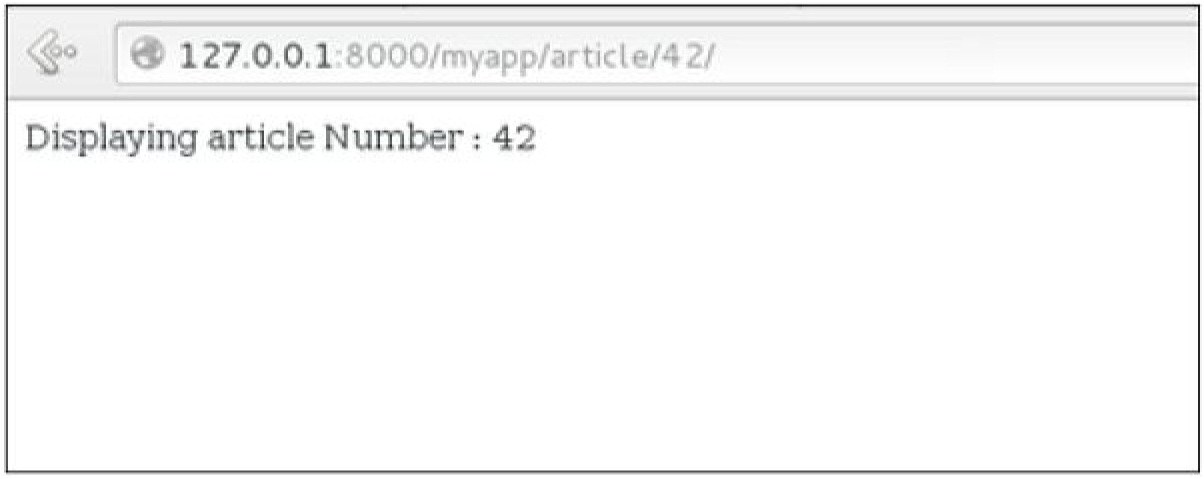
Passing parameters is done by capturing them with the regexp in the URL pattern. If we have a view like the following one in “myapp/view.py”

```
from django.shortcuts import render
from django.http import HttpResponse

def hello(request):
   return render(request, "hello.html", {})

def viewArticle(request, articleId):
   text = "Displaying article Number : %s"%articleId
   return HttpResponse(text)
```

We want to map it in myapp/url.py so we can access it via “/myapp/article/articleId”, we need the following in “myapp/url.py” −

```
from django.conf.urls import patterns, include, url

urlpatterns = patterns('myapp.views',
   url(r'^hello/', 'hello', name = 'hello'),
   url(r'^morning/', 'morning', name = 'morning'),
   url(r'^article/(\d+)/', 'viewArticle', name = 'article'),)
```

When Django will see the url: “/myapp/article/42” it will pass the parameters '42' to the viewArticle view, and in your browser you should get the following result −


127.0.0.1:8000/myapp/article/42/

Displaying article Number : 42

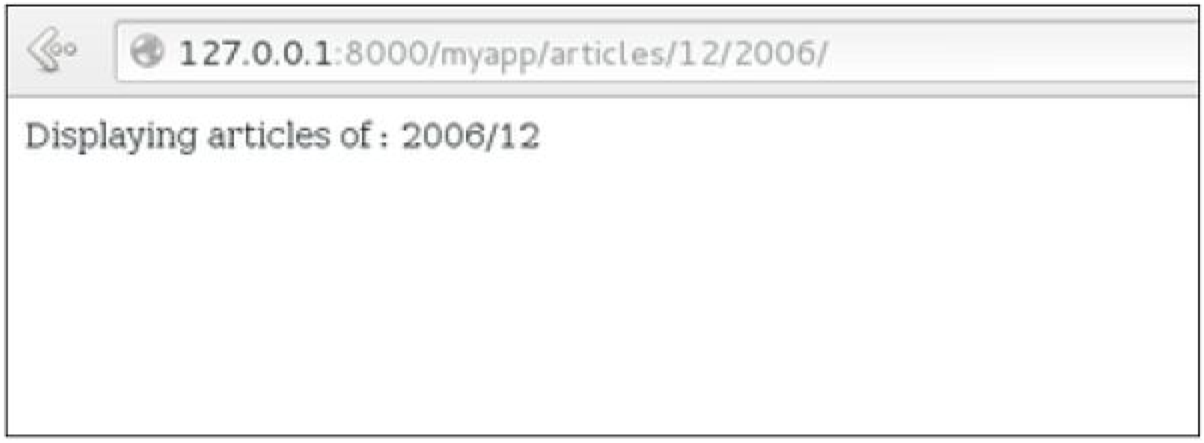
Note that the order of parameters is important here. Suppose we want the list of articles of a month of a year, let's add a viewArticles view. Our view.py becomes −

```
from django.shortcuts import render
from django.http import HttpResponse

def hello(request):
   return render(request, "hello.html", {})

def viewArticle(request, articleId):
   text = "Displaying article Number : %s"%articleId
   return HttpResponse(text)

def viewArticle(request, month, year):
   text = "Displaying articles of : %s/%s"%(year, month)
   return HttpResponse(text)
```

The corresponding url.py file will look like −

```
from django.conf.urls import patterns, include, url

urlpatterns = patterns('myapp.views',
   url(r'^hello/', 'hello', name = 'hello'),
   url(r'^morning/', 'morning', name = 'morning'),
   url(r'^article/(\d+)/', 'viewArticle', name = 'article'),
   url(r'^articles/(\d{2})/(\d{4})', 'viewArticles', name = 'articles'),)
```

Now when you go to “/myapp/articles/12/2006/” you will get 'Displaying articles of: 2006/12' but if you reverse the parameters you won’t get the same result.



```
Displaying articles of : 2006/12
```

To avoid that, it is possible to link a URL parameter to the view parameter. For that, our url.py will become −

```
from django.conf.urls import patterns, include, url

urlpatterns = patterns('myapp.views',
   url(r'^hello/', 'hello', name = 'hello'),
   url(r'^morning/', 'morning', name = 'morning'),
   url(r'^article/(\d+)/', 'viewArticle', name = 'article'),
   url(r'^articles/(?P\d{2})/(?P\d{4})', 'viewArticles', name = 'articles'),)
```

Django makes it possible to separate python and HTML, the python goes in views and HTML goes in templates. To link the two, Django relies on the render function and the Django Template language.

The Render Function

This function takes three parameters −

- Request − The initial request.

- The path to the template − This is the path relative to the TEMPLATE_DIRS option in the project settings.py variables.
- Dictionary of parameters − A dictionary that contains all variables needed in the template. This variable can be created or you can use locals() to pass all local variable declared in the view.

Django Template Language (DTL)

Django's template engine offers a mini-language to define the user-facing layer of the application.

Displaying Variables

A variable looks like this: {{variable}}. The template replaces the variable by the variable sent by the view in the third parameter of the render function. Let's change our hello.html to display today's date −

hello.html

```
<html>

  <body>
    Hello World!!!<p>Today is {{today}}</p>
  </body>

</html>
```

Then our view will change to −

```
def hello(request):
  today = datetime.datetime.now().date()
  return render(request, "hello.html", {"today" : today})
```

We will now get the following output after accessing the URL/myapp/hello −

Hello World!!!

Today is Sept. 11, 2015

As you have probably noticed, if the variable is not a string, Django will use the __str__ method to display it; and with the same principle you can access an object attribute just like you do it in Python. For example: if we wanted to display the date year, my variable would be: {{today.year}}.

Filters

They help you modify variables at display time. Filters structure looks like the following: {{var|filters}}.

Some examples −

- {{string|truncatewords:80}} − This filter will truncate the string, so you will see only the first 80 words.
- {{string|lower}} − Converts the string to lowercase.
- {{string|escape|linebreaks}} − Escapes string contents, then converts line breaks to tags.

You can also set the default for a variable.

Tags

Tags lets you perform the following operations: if condition, for loop, template inheritance and more.

Tag if

Just like in Python you can use if, else and elif in your template −

```
<html>
  <body>

     Hello World!!!<p>Today is {{today}}</p>
     We are
```

```
    {% if today.day == 1 %}

    the first day of month.
    {% elif today == 30 %}

    the last day of month.
    {% else %}

    I don't know.
    {%endif%}

  </body>
</html>
```

In this new template, depending on the date of the day, the template will render a certain value.

Tag for

Just like 'if', we have the 'for' tag, that works exactly like in Python. Let's change our hello view to transmit a list to our template −

```
def hello(request):
  today = datetime.datetime.now().date()

  daysOfWeek = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']
  return render(request, "hello.html", {"today" : today, "days_of_week" : daysOfWeek})
```

The template to display that list using {{ for }} −

```
<html>
  <body>

    Hello World!!!<p>Today is {{today}}</p>
    We are
    {% if today.day == 1 %}

    the first day of month.
    {% elif today == 30 %}

    the last day of month.
    {% else %}
```

```
      I don't know.
      {%endif%}

      <p>
         {% for day in days_of_week %}
         {{day}}
      </p>

      {% endfor %}

   </body>
</html>
```

And we should get something like −

```
Hello World!!!

Today is Sept. 11, 2015

We are I don't know.

Mon

Tue

Wed

Thu

Fri

Sat

Sun
```

Block and Extend Tags

A template system cannot be complete without template inheritance. Meaning when you are designing your templates, you should have a main template with holes that the child's template will fill according to his own need, like a page might need a special css for the selected tab.

Let’s change the hello.html template to inherit from a main_template.html.

main_template.html

```
<html>
  <head>

    <title>
      {% block title %}Page Title{% endblock %}
    </title>

  </head>

  <body>

    {% block content %}
      Body content
    {% endblock %}

  </body>
</html>
```

hello.html

```
{% extends "main_template.html" %}
{% block title %}My Hello Page{% endblock %}
{% block content %}

Hello World!!!<p>Today is {{today}}</p>
We are
{% if today.day == 1 %}

the first day of month.
{% elif today == 30 %}

the last day of month.
{% else %}

I don't know.
{%endif%}
```

```
<p>
   {% for day in days_of_week %}
   {{day}}
</p>

{% endfor %}
{% endblock %}
```

In the above example, on calling /myapp/hello we will still get the same result as before but now we rely on extends and block to refactor our code −

In the main_template.html we define blocks using the tag block. The title block will contain the page title and the content block will have the page main content. In home.html we use extends to inherit from the main_template.html then we fill the block define above (content and title).

Comment Tag

The comment tag helps to define comments into templates, not HTML comments, they won't appear in HTML page. It can be useful for documentation or just commenting a line of code.