./

Learning Report – Advanced Python Programming



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Ver. Rel. No.** | **Release Date** | **Prepared. By** | **Reviewed By** | **To be approved By** | **Remarks/Revision Details** |
| 1 | 08-02-21 | Sneha Anand |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

**Document History**

Table of Contents

[1. FUNCTIONS 5](#_Toc67470870)

[**Syntax:** 6](#_Toc67470871)

[**1.1** **Examples for python functions:** 6](#_Toc67470872)

[2. CLASSES AND OBJECTS IN PYTHON: 9](#_Toc67470873)

[**2.1** **CREATING CLASSES IN PYTHON:** 9](#_Toc67470874)

[**2.2 CREATING AN INSTANCE OF THE CLASS:** 9](#_Toc67470875)

[3 .LISTS: 11](#_Toc67470876)

[4.TUPLES: 12](#_Toc67470877)

[A tuple is a collection of objects which ordered and immutable. Tuples are sequences, just like lists. The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets. 12](#_Toc67470878)

[5. DICTIONARY: 13](#_Toc67470879)

[**Creating the dictionary** 13](#_Toc67470880)

[6. SETS: 14](#_Toc67470881)

[Creating a set 14](#_Toc67470882)

[Example 1: Using curly braces 14](#_Toc67470883)

[Example 2: Using set() method 15](#_Toc67470884)

[7. INHERITANCE IN PYTHON: 17](#_Toc67470885)

[Syntax: 17](#_Toc67470886)

[Syntax 18](#_Toc67470887)

[7.1 Examples for inheritance: 18](#_Toc67470888)

[Syntax 21](#_Toc67470889)

[Example 22](#_Toc67470890)

[8.VARIABLES IN PYTHON 27](#_Toc67470891)

[9.Exceptions in python: 31](#_Toc67470892)

[Common Exceptions 31](#_Toc67470893)

[Example 32](#_Toc67470894)

[**9.1 EXCEPTION HANDLING METHODS IN PYTHON:** 34](#_Toc67470895)

[10.REGULAR EXPRESSIONS: 36](#_Toc67470896)

[11. PYTHON PACKAGE MANAGEMENT 40](#_Toc67470897)

[Command-line interface: 40](#_Toc67470898)

[Using setup.py 41](#_Toc67470899)

[12.THREADS IN PYTHON: 42](#_Toc67470900)

[Starting a New Thread 43](#_Toc67470901)

[Example 43](#_Toc67470902)

[The *Threading* Module 44](#_Toc67470903)

[Creating Thread Using *Threading* Module 45](#_Toc67470904)

[Example 45](#_Toc67470905)

[Synchronizing Threads 47](#_Toc67470906)

[Example 48](#_Toc67470907)

[Multithreaded Priority Queue 50](#_Toc67470908)

[Example 50](#_Toc67470909)

[13 .MAGIC METHODS IN PYTHON: 53](#_Toc67470910)

# FUNCTIONS

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

Python functions are techniques used to combine a set of statements within a program. Functions also let programmers compute a result-value and give parameters that serve as function inputs that may change each time the code runs.

The Function helps to programmer to break the program into the smaller part. It organizes the code very effectively and avoids the repetition of the code. As the program grows, function makes the program more organized.

The advantages of using functions are:

* Avoid repetition of codes.
* Increases program readability.
* Divide a complex problem into simpler ones.
* Reduces chances of error.
* Modifying a program becomes easier by using function.

Rules to define a function in Python:

* A Function block begins with the keyword **def** followed by the function name and parentheses.
* Any input parameters or arguments should be placed within these parentheses. The parameters can also be defined inside these parentheses.
* The first statement of a function can be an optional statement - the documentation string of the function or *docstring*.
* The code block within every function starts with a colon (:) and is indented.
* The statement return (expression) exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

## **Syntax:**

def <function\_name>([arguments]):

statements

return ()

## **Examples for python functions:**

def display(x):

print("in display:",x);

print("start");

display(10);

print("end");

'''

'''

#addtion function

def add(a,b):

#print("result=",(a+b));

a=a+10;

b=b+20;

return(a+b)

print("start");

i=20;

j=30;

s=add(i,j);

print("result:",s);

print("i=",i);

print("j=",j);

print("end");

'''

'''

def add(a,b):

#print("result=",(a+b));

a=a+10;

b=b+20;

return(a+b);

def test(l):

l.append(11);

l.append(22);

l1=[1,2,3,4,5];

print(l1);

test(l1); # here l1 adress is passed(the real one,not duplicate);same thing for tuple,string,etc

print(l1);

# to send copy->pass by value test(l1

'''

'''

def inc(a,b):

#print("result=",(a+b));

a=a+10;

b=b+20;

return(a,b);

i=10;

j=20;

print("before");

print("i:",i);

print("j:",j);

result=inc(i,j);

print(result); # here result is tuple

i,j=inc(i,j);

print("after");

print("i:",i);

print("j:",j);

'''

#default arguments

'''

def inc(a,b=20):

#print("result=",(a+b));

a=a+10;

b=b+20;

return(a,b);

i=10;

j=20;

i,j=inc(a=i,b=j); #keyword evaluvators

print(i,j);

'''

'''

#variable number of arguments

def sum(\*args):

s=0;

for ele in args:

s=s+ele;

return(s);

print(sum(10,20));

print(sum(10,20,30));

print(sum(10,20,30,40));

'''

#variable number of arguments with name

def sum(\*\*d):

s=0;

for ele in d.values():

s=s+ele;

return(s);

print(sum(a=10,b=20));

print(sum(x=10,y=20,z=30));

print(sum(a=10,b=20,x=30,y=40));

# CLASSES AND OBJECTS IN PYTHON:

A class is a blueprint for creating objects (a particular data structure), providing initial values for state (member variables or attributes), and implementations of behavior (member functions or methods).

The class comes into existence when it instantiated.

An Object is an instance of a Class. A class is like a blueprint while an instance is a copy of the class with actual values.

## **CREATING CLASSES IN PYTHON:**

In Python, a class can be created by using the keyword class, followed by the class name. The syntax to create a class is given below.

**Syntax**

class Class\_Name:

# Statement-1

.

.

.

# Statement-N

## **2.2 CREATING AN INSTANCE OF THE CLASS:**

When an object of a class is created, the class is said to be instantiated. All the instances share the attributes and the behavior of the class. But the values of those attributes, i.e., the state are unique for each object. A single class may have any number of instances.

The syntax to create the instance of the class is given below.

<object\_name> = <class\_name>(<arguments>)

The following example creates the instance of the class Employee defined in the above example.

**Example**

1. class Employee:
2. 'Common base class for all employees'
3. empCount = 0
4. def \_\_init\_\_(self, name, salary):
5. self.name = name
6. self.salary = salary
7. Employee.empCount += 1
9. def displayCount(self):
10. print "Total Employee %d" % Employee.empCount
11. def displayEmployee(self):
12. print "Name: ", self.name,“ Salary: ", self.salary
13. "This would create first object of Employee class"
14. emp1 = Employee ("Zara", 2000)
15. "This would create second object of Employee class"
16. emp2 = Employee ("Manni", 5000)
17. emp1.displayEmployee()
18. emp2.displayEmployee()
19. print ("Total Employee %d:" % Employee.empCount)

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

Name: Zara, Salary: 2000

Name: Manni, Salary: 5000

Total Employee: 2

# 3.LISTS:

A list is a data structure in Python that is a mutable, or changeable, ordered sequence of elements. Each element or value that is inside of a list is called an item. Just as strings are defined as characters between quotes, lists are defined by having values between square brackets [ ] .

**Example**

list1 = ['physics', 'chemistry', 1997, 2000]

list2 = [1, 2, 3, 4, 5 ]

list3 = ["a", "b", "c", "d"]

The important characteristics of Python lists are as follows:

* Lists are ordered.
* Lists can contain any arbitrary objects.
* List elements can be accessed by index.
* Lists can be nested to arbitrary depth.
* Lists are mutable.
* Lists are dynamic.

Python offers the following list functions:

* sort(): Sorts the list in ascending order.
* type(list): It returns the class type of an object.
* append(): Adds a single element to a list.
* extend(): Adds multiple elements to a list.
* index(): Returns the first appearance of the specified value.
* max(list): It returns an item from the list with max value.
* min(list): It returns an item from the list with min value.
* len(list): It gives the total length of the list.
* list(seq): Converts a tuple into a list.
* cmp(list1, list2): It compares elements of both lists list1 and list2.

# 4.TUPLES:

A tuple is a collection of objects which ordered and immutable. Tuples are sequences, just like lists. The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets.

**Example**

tup1 = ('physics', 'chemistry', 1997, 2000);

tup2 = (1, 2, 3, 4, 5 );

tup3 = "a", "b", "c", "d";

The empty tuple is written as two parentheses containing nothing −

tup1 = ();

To write a tuple containing a single value you have to include a comma, even though there is only one value −

tup1 = (50,);

Python has two built-in methods that you can use on tuples:

* count() : Returns the number of times a specified value occurs in a tuple
* index() : Searches the tuple for a specified value and returns the position of

where it was found

# 5.DICTIONARY:

Dictionary is an unordered collection of data values, used to store data values like a map, which unlike other Data Types that hold only single value as an element. Dictionary holds key: value pair. Key value is provided in the dictionary to make it more optimized.

* Keys must be a single element
* Value can be any type such as list, tuple, integer, etc.

In other words, we can say that a dictionary is the collection of key-value pairs where the value can be any Python object. In contrast, the keys are the immutable Python object, i.e., Numbers, string, or tuple.

## Creating the dictionary

The dictionary can be created by using multiple key-value pairs enclosed with the curly brackets {}, and each key is separated from its value by the colon (:).

**EXAMPLE:**

Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}

print(type(Employee))

print("printing Employee data .... ")

print(Employee)

In the above dictionary **Dict** : The keys **Name** and **Age** are the string that is an immutable object.

# 6.SETS:

A Python set is the collection of the unordered items. Each element in the set must be unique, immutable, and the sets remove the duplicate elements. Sets are mutable which means we can modify it after its creation.

## Creating a set

The set can be created by enclosing the comma-separated immutable items with the curly braces {}. Python also provides the set() method, which can be used to create the set by the passed sequence.

### Example 1: Using curly braces

1. Days = {"Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"}
2. print(Days)
3. print("looping through the set elements ... ")
4. for i in Days:
5. print(i)

**Output:**

{'Friday', 'Tuesday', 'Monday', 'Saturday', 'Thursday', 'Sunday', 'Wednesday'}

looping through the set elements ...

Friday

Tuesday

Monday

Saturday

Thursday

Sunday

Wednesday

### Example 2: Using set() method

1. Days = set(["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"])
2. print(Days)
3. print("looping through the set elements ... ")
4. for i in Days:
5. print(i)

**Output:**

{'Friday', 'Wednesday', 'Thursday', 'Saturday', 'Monday', 'Tuesday', 'Sunday'}

looping through the set elements ...

Friday

Wednesday

Thursday

Saturday

Monday

Tuesday

Sunday

# 7. INHERITANCE IN PYTHON:

Inheritance is the most important aspect of object-oriented programming, which simulates the real-world concept of inheritance. It specifies that the child object acquires all the properties and behaviors of the parent object.

By using inheritance, we can create a class which uses all the properties and behavior of another class. The new class is known as a derived class or child class, and the one whose properties are acquired is known as a base class or parent class.

It provides the re-usability of the code.

Inheritance is an important aspect of the object-oriented paradigm. Inheritance provides code reusability to the program because we can use an existing class to create a new class instead of creating it from scratch.

In inheritance, the child class acquires the properties and can access all the data members and functions defined in the parent class. A child class can also provide its specific implementation to the functions of the parent class. In this section of the tutorial, we will discuss inheritance in detail.

In python, a derived class can inherit base class by just mentioning the base in the bracket after the derived class name. Consider the following syntax to inherit a base class into the derived class.

### Syntax:

1. class derived-class(base class):
2. <class-suite>

A class can inherit multiple classes by mentioning all of them inside the bracket. Consider the following syntax.

### Syntax

1. class derive-class(<base class 1>, <base class 2>, ..... <base class n>):
2. <class - suite>

# 9.Exceptions in python:

An exception can be defined as an unusual condition in a program resulting in the interruption in the flow of the program.

Whenever an exception occurs, the program stops the execution, and thus the further code is not executed. Therefore, an exception is the run-time errors that are unable to handle to Python script. An exception is a Python object that represents an error

Python provides a way to handle the exception so that the code can be executed without any interruption. If we do not handle the exception, the interpreter doesn't execute all the code that exists after the exception.

Python has many **built-in exceptions** that enable our program to run without interruption and give the output. These exceptions are given below:

## Common Exceptions

Python provides the number of built-in exceptions, but here we are describing the common standard exceptions. A list of common exceptions that can be thrown from a standard Python program is given below.

1. **ZeroDivisionError:** Occurs when a number is divided by zero.
2. **NameError:** It occurs when a name is not found. It may be local or global.
3. **IndentationError:** If incorrect indentation is given.
4. **IOError:** It occurs when Input Output operation fails.
5. **EOFError:** It occurs when the end of the file is reached, and yet operations are being performed.

**The problem without handling exceptions**

As we have already discussed, the exception is an abnormal condition that halts the execution of the program.

Suppose we have two variables **a** and **b**, which take the input from the user and perform the division of these values. What if the user entered the zero as the denominator? It will interrupt the program execution and through a **ZeroDivision** exception. Let's see the following example.

### Example

1. a = int(input("Enter a:"))
2. b = int(input("Enter b:"))
3. c = a/b
4. print("a/b = %d" %c)
6. #other code:
7. print("Hi I am other part of the program")

**Output:**

Enter a:10

Enter b:0

Traceback (most recent call last):

File "exception-test.py", line 3, in <module>

c = a/b;

ZeroDivisionError: division by zero

The above program is syntactically correct, but it through the error because of unusual input. That kind of programming may not be suitable or recommended for the projects because these projects are required uninterrupted execution. That's why an exception-handling plays an essential role in handling these unexpected exceptions. We can handle these exceptions in the following way.

## **9.1 EXCEPTION HANDLING METHODS IN PYTHON:**

# exception handling

'''

print("before");

i=10;

j=0;

#here divsion by zero is not possible so it will give exception

k= i/j;

print("k value:",k);

print("after");

'''

'''

output=.

Traceback (most recent call last):

File "11\_exception\_handle.py", line 6, in <module>

k= i/j;

ZeroDivisionError: division by zero

'''

'''

1)name error

2)index error

3)zerodivision error

etc

'''

#syntax

'''

try:

statement(s)

except ExceptionType:

handle here

'''

i=10;

j=0; #put j=1; and run

l= [1,2,3,4];

try:

print("before");

i=i+x; # name error->no expection for name

l[10]=100; #index error

k= i/j; #ZeroDivisionError

#print("k value:",k);

except ZeroDivisionError as ex:

print("ZeroDivisionError:",ex.args);

except IndexError as ex:

print("IndexError",ex.args);

except Exception as ex:

print("Main Exception",ex.args);

else:

print("in else:no exception");#print if no expection

finally:

print("program end"); #print after all ends

# 10.REGULAR EXPRESSIONS:

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

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

We would cover two important functions, which would be used to handle regular expressions. But a small thing first: There are various characters, which would have special meaning when they are used in regular expression. To avoid any confusion while dealing with regular expressions, we would use Raw Strings as **r'expression'**.

#regular expressions

#string processiong

import re; #module for regular expressions

#syntax-> re.match("pattern","string")

print(re.match("the","the man"));

#output

'''

<re.Match object; span=(0, 3), match='the'>

'''

print(re.match("the","i am the man"));

#output

'''

None

'''

#search

print(re.search("the","i am the man"));

#output

'''

<re.Match object; span=(5, 8), match='the'>

'''

# symbols

'''

a-z A-Z

0-9

special charaters

1) . -> Mathches any single charater in the string

'''

print(re.match(".....","Hello")); #checking 8 charaters are there or not

'''

2) ? -> preceeding character can repeat 0 or 1 time

'''

print(re.match("He?llo","Hello")); #e can repeat zero 0r 1 times

print(re.match("He?llo","Heello"));

'''

3) + -> preceeding character can repeat 1 or more time

'''

print(re.match("He+llo","Hello"));

print(re.match("He+llo","Hllo"));

'''

4) \* -> preceeding character can repeat 0 or more time

'''

print(re.match("He\*llo","Hello"));

'''

[ ] -> matches any single character in the list

'''

print(re.match("[abc]","abc123def")); #either a or b or c

print(re.match("[abc]+","abc123def")); #either a or b or c

print(re.match("[a-z]+","abcdef123"));

print(re.match("[a-z1-24]+","abcdef123567def"));

'''

output

<re.Match object; span=(0, 8), match='abcdef12'>

'''

print(re.match("[a-z0-9]+","abcdef123567def")); #matched everything

'''

[ ^] -> does matches any single character in the list

'''

print(re.match("[^0-9]+","abcdef123567def"));

'''

output

<re.Match object; span=(0, 6), match='abcdef'>

'''

'''

^ -> strating with same character

'''

print(re.match("^H","HELLO"));

'''

$ -> ending with same character

'''

print(re.match("....O$","HELLO"));

'''

{N,M } -> minimun to maximum

'''

print(re.match("[a-z]{2,5}$","abcdef"));

'''

( ) -> match in group

'''

print(re.match("(ab)?c","abc"));

'''

| -> one or more conditions(or);

'''

print(re.match("[a-z]+ | [0-9] +","abc123def456"));

# 11. PYTHON PACKAGE MANAGEMENT

**Python Package Manager** is a Python utility intended to simplify the tasks of locating, installing, upgrading and removing Python packages. It can determine if the most recent version of a software package is installed on a system, and can install or upgrade that package from a local or remote host.

**pip** is a package-management system written in Python used to install and manage software packages. It connects to an online repository of public and paid-for private packages, called the Python Package Index.

Most distributions of Python come with pip preinstalled. Python 2.7.9 and later (on the python2 series), and Python 3.4 and later include pip (pip3 for Python 3) by default. Python 2.7 (and 3.5) support was dropped with the next release, pip 21, in January 2021.

## Command-line interface:

An output of pip install virtualenv

One major advantage of pip is the ease of its command-line interface which makes installing Python software packages as easy as issuing a command:

pip install some-package-name

Users can also easily remove the package:

pip uninstall some-package-name

Most importantly, **pip** has a feature to manage full lists of packages and corresponding version numbers, possible through a "requirements" file. This permits the efficient re-creation of an entire group of packages in a separate environment (e.g. another computer) or virtual environment This can be achieved with a properly formatted file and the following command, where requirements.txt is the name of the file:

pip install -r requirements.txt

To install some package for a specific python version, **pip** provides the following command, where ${version} is replaced by 2, 3, 3.4, etc.:

pip${version} install some-package-name

### Using setup.py

Pip provides a way to install user-defined projects locally with the use of setup.py file. This method requires the python project to have the following file structure:

example\_project/

├── exampleproject/ Python package with source code.

| ├── \_\_init\_\_.py Make the folder a package.

|  └── example.py Example module.

└── README.md README with info of the project.

Within this structure, user can add setup.py to the root of the project (i.e. example\_project for above structure) with the following content:

from setuptools import setup, find\_packages

setup (

name ='example', # Name of the package. This will be used, when the project is imported as a package.

Version ='0.1.0',

Packages =find\_packages(include=['exampleproject', 'exampleproject.\*']) # Pip will automatically install the dependences provided here.

)

After this, pip can install this custom project by running the following command, from the project root directory:

pip install -e .

# 12.THREADS IN PYTHON:

Running several threads is similar to running several different programs concurrently, but with the following benefits −

* Multiple threads within a process share the same data space with the main thread and can therefore share information or communicate with each other more easily than if they were separate processes.
* Threads sometimes called light-weight processes and they do not require much memory overhead; they are cheaper than processes.

A thread has a beginning, an execution sequence, and a conclusion. It has an instruction pointer that keeps track of where within its context it is currently running.

* It can be pre-empted (interrupted)
* It can temporarily be put on hold (also known as sleeping) while other threads are running - this is called yielding.

## Starting a New Thread

To spawn another thread, you need to call following method available in *thread* module −

thread.start\_new\_thread ( function, args[, kwargs] )

This method call enables a fast and efficient way to create new threads in both Linux and Windows.

The method call returns immediately and the child thread starts and calls function with the passed list of *args*. When function returns, the thread terminates.

Here, *args* is a tuple of arguments; use an empty tuple to call function without passing any arguments. *kwargs* is an optional dictionary of keyword arguments.

### Example

#!/usr/bin/python

import thread

import time

# Define a function for the thread

def print\_time( threadName, delay):

count = 0

while count < 5:

time.sleep(delay)

count += 1

print "%s: %s" % ( threadName, time.ctime(time.time()) )

# Create two threads as follows

try:

thread.start\_new\_thread( print\_time, ("Thread-1", 2, ) )

thread.start\_new\_thread( print\_time, ("Thread-2", 4, ) )

except:

print "Error: unable to start thread"

while 1:

pass

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

Thread-1: Thu Jan 22 15:42:17 2009

Thread-1: Thu Jan 22 15:42:19 2009

Thread-2: Thu Jan 22 15:42:19 2009

Thread-1: Thu Jan 22 15:42:21 2009

Thread-2: Thu Jan 22 15:42:23 2009

Thread-1: Thu Jan 22 15:42:23 2009

Thread-1: Thu Jan 22 15:42:25 2009

Thread-2: Thu Jan 22 15:42:27 2009

Thread-2: Thu Jan 22 15:42:31 2009

Thread-2: Thu Jan 22 15:42:35 2009

Although it is very effective for low-level threading, but the *thread* module is very limited compared to the newer threading module.

## The *Threading* Module

The newer threading module included with Python 2.4 provides much more powerful, high-level support for threads than the thread module discussed in the previous section.

The *threading* module exposes all the methods of the *thread* module and provides some additional methods −

* **threading.activeCount()** − Returns the number of thread objects that are active.
* **threading.currentThread()** − Returns the number of thread objects in the caller's thread control.
* **threading.enumerate()** − Returns a list of all thread objects that are currently active.

In addition to the methods, the threading module has the *Thread* class that implements threading. The methods provided by the *Thread* class are as follows −

* **run()** − The run() method is the entry point for a thread.
* **start()** − The start() method starts a thread by calling the run method.
* **join([time])** − The join() waits for threads to terminate.
* **isAlive()** − The isAlive() method checks whether a thread is still executing.
* **getName()** − The getName() method returns the name of a thread.
* **setName()** − The setName() method sets the name of a thread.

## Creating Thread Using *Threading* Module

To implement a new thread using the threading module, you have to do the following −

* Define a new subclass of the *Thread* class.
* Override the *\_\_init\_\_(self [,args])* method to add additional arguments.
* Then, override the run(self [,args]) method to implement what the thread should do when started.

Once you have created the new *Thread* subclass, you can create an instance of it and then start a new thread by invoking the *start()*, which in turn calls *run()* method.

### Example

#!/usr/bin/python

import threading

import time

exitFlag = 0

class myThread (threading.Thread):

def \_\_init\_\_(self, threadID, name, counter):

threading.Thread.\_\_init\_\_(self)

self.threadID = threadID

self.name = name

self.counter = counter

def run(self):

print "Starting " + self.name

print\_time(self.name, 5, self.counter)

print "Exiting " + self.name

def print\_time(threadName, counter, delay):

while counter:

if exitFlag:

threadName.exit()

time.sleep(delay)

print "%s: %s" % (threadName, time.ctime(time.time()))

counter -= 1

# Create new threads

thread1 = myThread(1, "Thread-1", 1)

thread2 = myThread(2, "Thread-2", 2)

# Start new Threads

thread1.start()

thread2.start()

print "Exiting Main Thread"

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

Starting Thread-1

Starting Thread-2

Exiting Main Thread

Thread-1: Thu Mar 21 09:10:03 2013

Thread-1: Thu Mar 21 09:10:04 2013

Thread-2: Thu Mar 21 09:10:04 2013

Thread-1: Thu Mar 21 09:10:05 2013

Thread-1: Thu Mar 21 09:10:06 2013

Thread-2: Thu Mar 21 09:10:06 2013

Thread-1: Thu Mar 21 09:10:07 2013

Exiting Thread-1

Thread-2: Thu Mar 21 09:10:08 2013

Thread-2: Thu Mar 21 09:10:10 2013

Thread-2: Thu Mar 21 09:10:12 2013

Exiting Thread-2

## Synchronizing Threads

The threading module provided with Python includes a simple-to-implement locking mechanism that allows you to synchronize threads. A new lock is created by calling the *Lock()* method, which returns the new lock.

The *acquire(blocking)* method of the new lock object is used to force threads to run synchronously. The optional *blocking* parameter enables you to control whether the thread waits to acquire the lock.

If *blocking* is set to 0, the thread returns immediately with a 0 value if the lock cannot be acquired and with a 1 if the lock was acquired. If blocking is set to 1, the thread blocks and wait for the lock to be released.

The *release()* method of the new lock object is used to release the lock when it is no longer required.

### Example

#!/usr/bin/python

import threading

import time

class myThread (threading.Thread):

def \_\_init\_\_(self, threadID, name, counter):

threading.Thread.\_\_init\_\_(self)

self.threadID = threadID

self.name = name

self.counter = counter

def run(self):

print "Starting " + self.name

# Get lock to synchronize threads

threadLock.acquire()

print\_time(self.name, self.counter, 3)

# Free lock to release next thread

threadLock.release()

def print\_time(threadName, delay, counter):

while counter:

time.sleep(delay)

print "%s: %s" % (threadName, time.ctime(time.time()))

counter -= 1

threadLock = threading.Lock()

threads = []

# Create new threads

thread1 = myThread(1, "Thread-1", 1)

thread2 = myThread(2, "Thread-2", 2)

# Start new Threads

thread1.start()

thread2.start()

# Add threads to thread list

threads.append(thread1)

threads.append(thread2)

# Wait for all threads to complete

for t in threads:

t.join()

print "Exiting Main Thread"

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

Starting Thread-1

Starting Thread-2

Thread-1: Thu Mar 21 09:11:28 2013

Thread-1: Thu Mar 21 09:11:29 2013

Thread-1: Thu Mar 21 09:11:30 2013

Thread-2: Thu Mar 21 09:11:32 2013

Thread-2: Thu Mar 21 09:11:34 2013

Thread-2: Thu Mar 21 09:11:36 2013

Exiting Main Thread

## Multithreaded Priority Queue

The *Queue* module allows you to create a new queue object that can hold a specific number of items. There are following methods to control the Queue −

* **get()** − The get() removes and returns an item from the queue.
* **put()** − The put adds item to a queue.
* **qsize()** − The qsize() returns the number of items that are currently in the queue.
* **empty()** − The empty( ) returns True if queue is empty; otherwise, False.
* **full()** − the full() returns True if queue is full; otherwise, False.

### Example

#!/usr/bin/python

import Queue

import threading

import time

exitFlag = 0

class myThread (threading.Thread):

def \_\_init\_\_(self, threadID, name, q):

threading.Thread.\_\_init\_\_(self)

self.threadID = threadID

self.name = name

self.q = q

def run(self):

print "Starting " + self.name

process\_data(self.name, self.q)

print "Exiting " + self.name

def process\_data(threadName, q):

while not exitFlag:

queueLock.acquire()

if not workQueue.empty():

data = q.get()

queueLock.release()

print "%s processing %s" % (threadName, data)

else:

queueLock.release()

time.sleep(1)

threadList = ["Thread-1", "Thread-2", "Thread-3"]

nameList = ["One", "Two", "Three", "Four", "Five"]

queueLock = threading.Lock()

workQueue = Queue.Queue(10)

threads = []

threadID = 1

# Create new threads

for tName in threadList:

thread = myThread(threadID, tName, workQueue)

thread.start()

threads.append(thread)

threadID += 1

# Fill the queue

queueLock.acquire()

for word in nameList:

workQueue.put(word)

queueLock.release()

# Wait for queue to empty

while not workQueue.empty():

pass

# Notify threads it's time to exit

exitFlag = 1

# Wait for all threads to complete

for t in threads:

t.join()

print "Exiting Main Thread"

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

Starting Thread-1

Starting Thread-2

Starting Thread-3

Thread-1 processing One

Thread-2 processing Two

Thread-3 processing Three

Thread-1 processing Four

Thread-2 processing Five

Exiting Thread-3

Exiting Thread-1

Exiting Thread-2

Exiting Main Thread

# 13 .MAGIC METHODS IN PYTHON:

Python magic method is defined as the special method which adds "magic" to a class. It starts and ends with double underscores, for example, \_init\_ or \_str\_.

The built-in classes define many magic methods. The dir() function can be used to see the number of magic methods inherited by a class. It has two prefixes, and suffix underscores in the method name.

It is most frequently used to define the overloaded behaviors of predefined operators.

**\_\_init\_\_**

The \_init\_ method is called after the instance of the class has been created but before it returned to the caller. It is invoked without any call, when an instance of the class is created like constructors in other programming languages such as C++, Java, C#, PHP, etc. These methods are also known as initialize and are called after \_new\_. Its where you should initialize the instance variables.

**\_\_str\_\_**

This function computes "informal" or a nicely printable string representation of an object and must return a string object.

**\_\_repr\_\_**

This function is called by the repr() built-in function to compute the "official" string representation of an object and returns a machine-readable representation of a type. The goal of the \_repr\_ is to be unambiguous.

**\_\_len\_\_**

This function should return the length of an object.

**\_\_call\_\_**

We can make an object callable by adding the \_call\_ magic method, and it is another method that is not needed quite as often is \_call\_.

If defined in a class, then that class can be called. But if it was a function, instance itself rather than modifying.

**\_\_del\_\_**

Just as \_init\_, which is a constructor method, \_del\_ is like a destructor. If you have opened a file in \_init \_, then \_del\_ can close it.

**\_\_bytes\_\_**

It offers to compute a byte-string representation of an object and should return a string object.

**\_\_ge\_\_**

This method gets invoked when >= operator is used and returns True or False.

**\_\_neg\_\_**

This function gets called for the unary operator.

**\_\_ipow\_\_**

This function gets called on the exponents with arguments. e.g. a\*\*=b.

**\_\_le\_\_**

This function gets called on comparison using <= operator.

**\_nonzero\_**

This function returns the Boolean value of the object. It gets invoked when the bool (self) function is called.