UNIT 3

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.

  A ***constructor*** is a special kind of method that Python calls when it instantiates an object using the definitions found in your class. Python relies on the constructor to perform tasks such as ***initializing*** (assigning values to) any instance variables that the object will need when it starts. Constructors can also verify that there are enough resources for the object and perform any other start-up task you can think of.

The name of a constructor is always the same, **__init__().** The constructor can accept arguments when necessary to create the object. When you create a class without a constructor, Python automatically creates a default constructor for you that doesn't do anything. Every class must have a constructor, even if it simply relies on the default constructor.

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
```

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

```
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 obejcts
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.

User-defined compound types

A class in essence defines a new **data type** . We have been using several of Python's built-in types throughout this book (Integers, Reals, Strings, Lists, Dictionaries etc), but we can also define new types if we wish to. Defining a new type in Python is very easy:

class CLASSNAME:
<statement 1>
<statement 2>
...

<statement n>

Class definitions can appear anywhere in a program, but they are usually near the beginning (after the import statements). The syntax rules for a class definition are the same as for other compound statements. There is a header which begins with the keyword, class, followed by the name of the class, and ending with a colon. In most cases, the statements inside a class definition should be function definitions. Functions defined inside a class have a special name — they are called **methods** of the class.

Although not required by Python, <statement 1> should be a docstring describing the class and <statement 2> should bean **initialization** method. The init method is a special method that is called just after a variable of the class type is **constructed** or **instantiated**. Variables of the class type are also called **instances** or **objects** of that type.

We are now ready to create our own user-defined type: the Point. Consider the concept of a mathematical point. In two dimensions, a point is two numbers (coordinates) that are treated collectively as a single object. In mathematical notation, points are often written in parentheses with a comma separating the coordinates. For example, (0; 0) represents the origin, and (x; y) represents the point x units to the right and y units up from the origin. A natural way to represent a point in Python is with two numeric values.

The question, then, is how to group these two values into a compound object. The quick and dirty solution is to use a list or tuple, and for some applications that might be the best choice. An alternative is to define a new user-defined compound type, also called a **class** . This approach involves a bit more effort, but it has advantages that will become apparent soon. Here's a simple definition of Point which we can put into a file called Point.py:

```
class Point:
"""A class to represent a two-dimensional point"""
def __init__(self):
self.x = 0
self.y = 0
```

The init Method and self

You will see from the class definition above the special method called init . This method is called when a Point object is instantiated. To instantiate a Point object, we call a function named (you guessed it)

Point:

```
>>> type(Point)
<type 'classobj'>
>>> p = Point()
>>> type(p)
<type 'instance'>
```

The function Point() is called the class **constructor**. The variable p is assigned a reference to a new Point object. When calling the function Point (), Python performs some magic behind the scenes and calls the init method from within the class constructor.

The init method has a parameter called self. Every class method must have one parameter and that parameter should be called self. The self parameter is a reference to the object on which the method is being called

Attributes

The init method above contains the definition of two object variables x and y. Object variables are
also called **attributes** of the object. Once a Point object has been instantiated, it is now valid to refer to its
attributes:

```
>>> p = Point()
>>> print p.x, p.y
0 0
>>> p.x = 3
>>> p.y = 4
>>> print p.x, p.y
3 4
```

This syntax is similar to the syntax for selecting a variable from a module, such as math.pi or string.uppercase. Both modules and instances create their own namespaces, and the syntax for accessing names contained in each, called **attributes**, is the same. In this case the attribute we are selecting is a data item from an instance. The following state diagram shows the result of these assignments:

The variable p refers to a Point object, which contains two attributes. Each attribute refers to a number.

The expression p.x means, Go to the object p refers to and get the value of x. The purpose of dot notation
is to identify which variable you are referring to unambiguously. You can use dot notation as part of any
expression, so the following statements are legal:

```
print '(%d, %d)' % (p.x, p.y)
distanceSquared = p.x * p.x + p.y * p.y
```

The first line outputs (3, 4); the second line calculates the value 25.

Methods

Init is an example of a class function or **method**. We can add as many methods as we like to a class, and this is one of the reasons OOP is so powerful. It's quite a different way of thinking bout programming. Rather than having functions that act on any old data, in OOP, the data carries the required functionality around with it. Let's add a few methods to our Point class:

Object
Real world objects shares 2 main characteristics, *state* and *behavior*. Human have state (name, age) and behavior (running, sleeping). Car have state (current speed, current gear) and state (applying brake, changing gear). Software objects are conceptually similar to real-world objects: they too consist of state and related behavior. An object stores its state in *fields* and exposes its behavior through *methods*.

Class
Class is a "template" / "*blueprint*" that is used to create objects. Basically, a class will consists of field, static field, method, static method and constructor. Field is used to hold the state of the class (eg: name of Student object). Method is used to represent the behavior of the class (eg: how a Student object going to stand-up). Constructor is used to create a new Instance of the Class.

Instance
An instance is a unique copy of a Class that representing an Object. When a new instance of a class is created, the JVM will allocate a room of memory for that class instance.

Instances as arguments

You can pass an instance as an argument in the usual way. For example:
def printPoint(p):
print '(' + str(p.x) + ', ' + str(p.y) + ')'
printPoint takes a point as an argument and displays it in the standard format.

If you call printPoint(blank), the output is (3.0, 4.0).

Instances as return values

Functions and methods can also return instances. For example, we may want to calculate the centre of the

rectangle. We can do that by adding a centre method to our class definition:

```
from Point import *
class Rectangle:
def __init__(self):
self.corner = Point()
self.width = 100
self.height = 100
def centre(self):
c = Point()
c.x = self.corner.x + self.width/2.0
c.y = self.corner.y + self.height/2.0
return c
box = Rectangle()
print box.corner.x, box.corner.y, box.width, box.height
```

c = box.centre()
print c.x, c.y
If we execute this program, we get:
0 0 100 100
50.0 50.0

Objects are mutable

We can change the state of an object by making an assignment to one of its attributes. For example, to
change the size of a rectangle without changing its position, we could modify the values of width and
height:
box.width = box.width + 50
box.height = box.height + 100

Copying

Aliasing can make a program difficult to read because changes made in one place might have unexpected
effects in another place. It is hard to keep track of all the variables that might refer to a given object.
Copying an object is often an alternative to aliasing. The copy module contains a function called copy that
can duplicate any object:

```
>>> import copy
>>> p1 = Point()
>>> p1.x = 3
>>> p1.y = 4
>>> p2 = copy.copy(p1)
>>> p1 == p2
False
>>> p1.equals(p2)
```

True the same point, but they contain the same data. To copy a simple object like a Point, which doesn't contain

any embedded objects, copy is sufficient. This is called **shallow copying**. For something like a Rectangle,
which contains a reference to a Point, copy doesn't do quite the right thing. It copies the reference to the
Point object, so both the old Rectangle and the new one refer to a single Point.
If we create a box, b1, in the usual way and then make a copy, b2, using copy, the resulting state diagram
looks like this:
y

x 0
0
100
width
height
corner
width
height
corner
b1 b2
100
100 100
This is almost certainly not what we want. Can you think why?
Fortunately, the copy module contains a method named deepcopy that copies not only the object but also
any embedded objects. You will not be surprised to learn that this operation is called a **deep copy** .
>>> b2 = copy.deepcopy(b1)
Now b1 and b2 are completely separate objects.

Classes and functions

Time

As another example of a user-defined type, we'll define a class called Time that records the time of day. The class definition looks like this:
class Time:
pass
We can create a new Time object and assign attributes for hours, minutes, and
seconds:
time = Time()
time.hours = 11
time.minutes = 59
time.seconds = 30
The state diagram for the Time object looks like this:
Time
Hours 11
Minutes 59
Seconds 30

As an exercise, write a function printTime *that takes a* Time *object*

As a second exercise, write a boolean function after *that takes two* Time *objects,* t1 *and* t2*, as arguments, and returns* True *if* t1 *follows* t2 *chronologically and* False *otherwise.*

Pure functions

In the next few sections, we'll write two versions of a function called addTime, which calculates the sum of two Times. They will demonstrate two kinds of functions: pure functions and modifiers. The following is a rough version of addTime:
def addTime(t1, t2):

```
  sum = Time()
  sum.hours = t1.hours + t2.hours
  sum.minutes = t1.minutes + t2.minutes
  sum.seconds = t1.seconds + t2.seconds
  return sum
```

The function creates a new Time object, initializes its attributes, and returns a reference to the new object. This is called a pure function because it does not modify any of the objects passed to it as arguments and it has no side effects, such as displaying a value or getting user input. Here is an example of how to use this function. We'll create two Time objects: currentTime, which contains the current time; and breadTime, which contains the amount of time it takes for a breadmaker to make bread. Then we'll use addTime to figure out when the bread will be done. If you haven't finished writing printTime yet, take a look ahead to Section 14.2 before you try this:

```
>>> currentTime = Time()
>>> currentTime.hours = 9
>>> currentTime.minutes = 14
>>> currentTime.seconds = 30
>>> breadTime = Time()
>>> breadTime.hours = 3
>>> breadTime.minutes = 35
>>> breadTime.seconds = 0
>>> doneTime = addTime(currentTime, breadTime)
>>> printTime(doneTime)
```

The output of this program is 12:49:30, which is correct. On the other hand, there are cases where the result is not correct. Can you think of one?The problem is that this function does not deal with cases where the number of
seconds or minutes adds up to more than sixty. When that happens, we have to "carry" the extra seconds into the minutes column or the extra minutes into the hours column. Here's a second corrected version of the function:

```
def addTime(t1, t2):
sum = Time()
sum.hours = t1.hours + t2.hours
sum.minutes = t1.minutes + t2.minutes
sum.seconds = t1.seconds + t2.seconds
if sum.seconds >= 60:
sum.seconds = sum.seconds - 60
sum.minutes = sum.minutes + 1
if sum.minutes >= 60:
sum.minutes = sum.minutes - 60
sum.hours = sum.hours + 1
return sum
```

Although this function is correct, it is starting to get big. Later we will suggest
an alternative approach that yields shorter code.

Modifiers

There are times when it is useful for a function to modify one or more of the objects it gets as arguments. Usually, the caller keeps a reference to the objects it passes, so any changes the function makes are visible to the caller. Functions that work this way are called modifiers. increment, which adds a given number of seconds to a Time object, would be written most naturally as a modifier. A rough draft of the function looks like this:

```
def increment(time, seconds):
time.seconds = time.seconds + seconds
if time.seconds >= 60:
time.seconds = time.seconds - 60
time.minutes = time.minutes + 1
if time.minutes >= 60:
time.minutes = time.minutes - 60
time.hours = time.hours + 1
```

The first line performs the basic operation; the remainder deals with the special cases we saw before.Is this function correct? What happens if the parameter seconds is much greater than sixty? In that case, it is not enough to carry once; we have to keep doing it until seconds is less than sixty. One solution is to replace the if statements with
while statements:

```
def increment(time, seconds):
time.seconds = time.seconds + seconds
while time.seconds >= 60:
time.seconds = time.seconds - 60
time.minutes = time.minutes + 1
while time.minutes >= 60:
time.minutes = time.minutes - 60
time.hours = time.hours + 1
```

This function is now correct, but it is not the most efficient solution. *As an exercise, rewrite this function so that it doesn't contain any loops. As a second exercise, rewrite* increment *as a pure function, and write function calls to both versions.*

Python Exceptions Handling

Python provides two very important features to handle any unexpected error in your Python programs and to add debugging capabilities in them −

- **Exception Handling:** This would be covered in this tutorial. Here is a list standard Exceptions available in Python: Standard Exceptions.
- **Assertions:** This would be covered in Assertions in Python tutorial.

List of Standard Exceptions −

EXCEPTION NAME	DESCRIPTION
Exception	Base class for all exceptions
StopIteration	Raised when the next() method of an iterator does not point to any object.
SystemExit	Raised by the sys.exit() function.
StandardError	Base class for all built-in exceptions except StopIteration and SystemExit.
ArithmeticError	Base class for all errors that occur for numeric calculation.
OverflowError	Raised when a calculation exceeds maximum limit for a numeric type.
FloatingPointError	Raised when a floating point calculation fails.
ZeroDivisonError	Raised when division or modulo by zero takes place for all numeric types.
AssertionError	Raised in case of failure of the Assert statement.
AttributeError	Raised in case of failure of attribute reference or assignment.
EOFError	Raised when there is no input from either the raw_input() or input() function and the end of file is reached.
ImportError	Raised when an import statement fails.
KeyboardInterrupt	Raised when the user interrupts program execution, usually by pressing Ctrl+c.
LookupError	Base class for all lookup errors.
IndexError	Raised when an index is not found in a sequence.
KeyError	Raised when the specified key is not found in the dictionary.
NameError	Raised when an identifier is not found in the local or global namespace.
UnboundLocalError	Raised when trying to access a local variable in a function or method but no value has been assigned to it.
EnvironmentError	Base class for all exceptions that occur outside the Python environment.
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
IOError	

	exist.
	Raised for operating system-related errors.
SyntaxError	Raised when there is an error in Python syntax.
IndentationError	Raised when indentation is not specified properly.
SystemError	Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit.
SystemExit	Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit.
Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit.	Raised when an operation or function is attempted that is invalid for the specified data type.
ValueError	Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.
RuntimeError	Raised when a generated error does not fall into any category.
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 ArgumentExpression 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
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.
   ......................
```

Note that you can provide except clause(s), or a finally clause, but not both. 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 Exceptions

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
```