Defining a class

In object oriented programming classes and objects are the main features. A class creates a new data type and objects are instances of a class which follows the definition given inside the class. Here is a simple form of class definition.

```
class Student:
    Statement-1
    Statement-1
    ....
    ....
    ....
    Statement-n
```

A class definition started with the keyword 'class' followed by the name of the class and a colon.

The statements within a class definition may be function definitions, data members or other statements.

When a class definition is entered, a new namespace is created, and used as the local scope.

Creating a Class

Here we create a simple class using class keyword followed by the class name (Student) which follows an indented block of segments (student class, roll no., name).

view plaincopy to clipboardprint?

```
1. #studentdetails.py
2. class Student:
3.         stu_class = 'V'
4.         stu_roll_no = 12
5.         stu_name = "David"
```

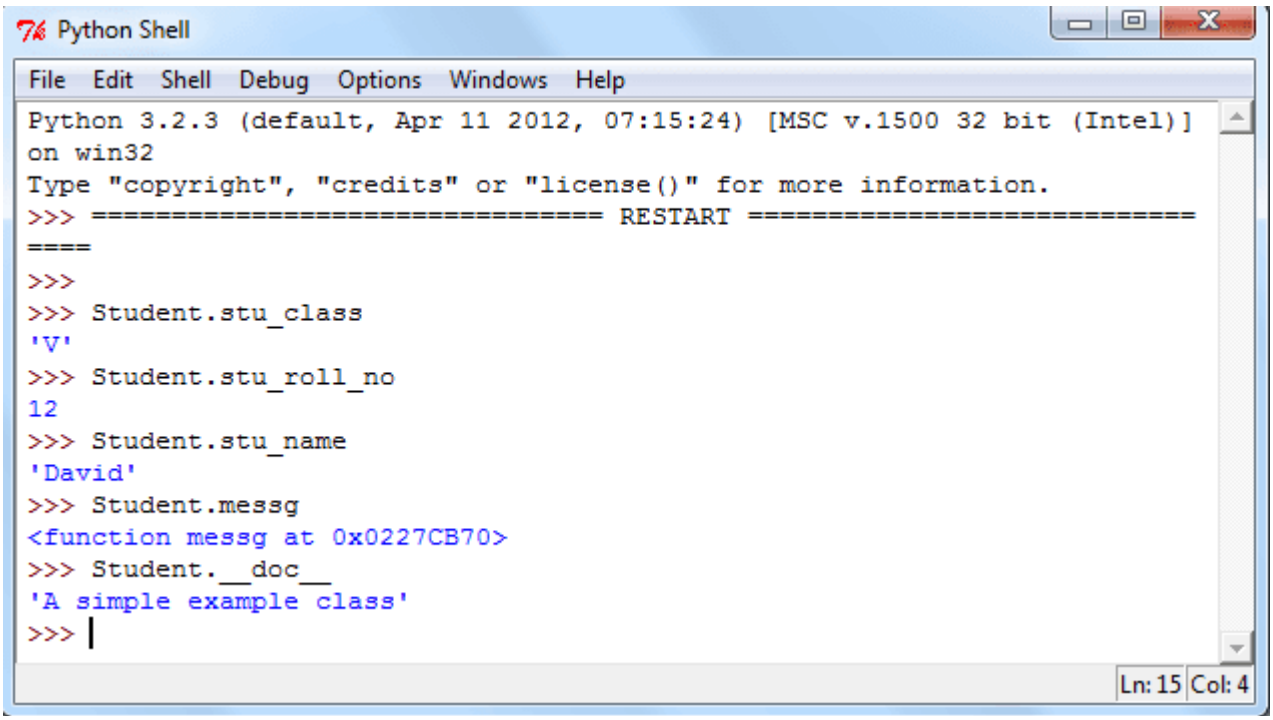
Class Objects

There are two kind of operations class objects supports : attribute references and instantiation. Attribute references use the standard syntax, obj.name for all attribute references in Python. Therefore if the class definition (add a method in previous example) look like this

view plaincopy to clipboardprint?

```
1. #studentdetails1.py
2. class Student:
3.     """A simple example class"""
4.     stu_class = 'V'
5.     stu_roll_no = 12
6.     stu_name = "David"
7.     def messg(self):
8.             return 'New Session will start soon.'
```

then Student.stu_class, Student.stu_roll_no, Student.stu_name are valid attribute reference and returns 'V', 12, 'David'. Student.messg returns a function object. In Python self is a name for the first argument of a method which is different from ordinary function. Rather than passing the object as a parameter in a method the word self refers to the object itself. For example if a method is defined as avg(self, x, y, z), it should be called as a.avg(x, y, z). See the output of the attributes in Python Shell.



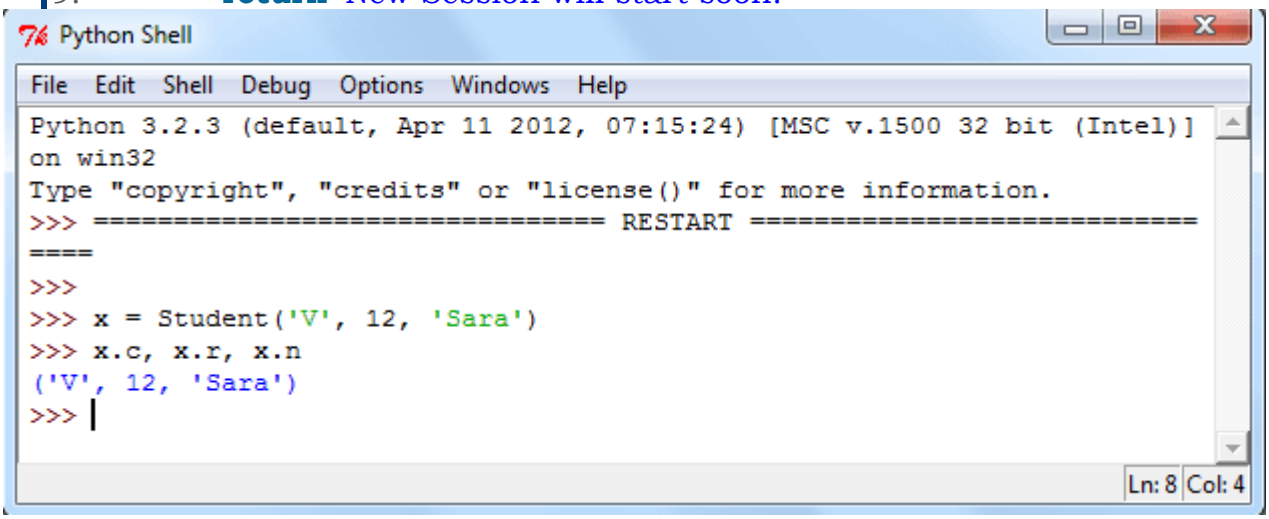
__doc__ is also a valid attribute which returns the docstring of the class.

__init__ method

There are many method names in Python which have special importance. A class may define a special method named __init__ which does some initialization work and serves as a constructor for the class. Like other functions or methods __init__ can take any number of arguments. The __init__ method is run as soon as an object of a class is instantiated and class instantiation automatically invokes __init__() for the newly-created class instance. See the following example a new, initialized instance can be obtained by:

view plaincopy to clipboardprint?

```
1. #studentdetailsinit.py
2. class Student:
3.     """A simple example class"""
4.     def __init__(self, sclass, sroll, sname):
5.         self.c = sclass
6.         self.r = sroll
7.         self.n = sname
8.     def messg(self):
9.             return 'New Session will start soon.'
```



Methods

We have already discussed that classes/objects can have methods just like functions except that we have an extra self variable. We will now see an example (save as oop_method.py).

```
class Person:
    def say_hi(self):
        print('Hello, how are you?')

p = Person()
p.say_hi()
# The previous 2 lines can also be written as
# Person().say_hi()
```

Output:

```
$ python oop_method.py
Hello, how are you?
```

How It Works

Here we see the self in action. Notice that the say_hi method takes no parameters but still has the self in the function definition.

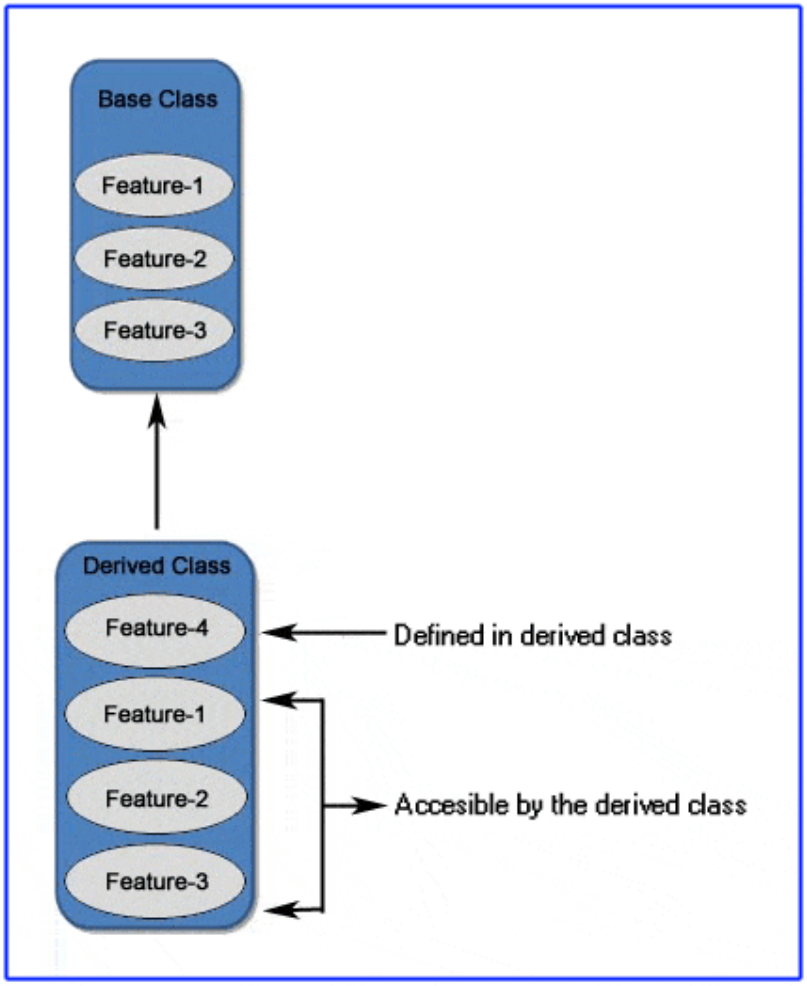
Constructor

Python doesn't have explicit constructors like C++ or Java, but the __init__() method in Python is something similar, though it is strictly speaking not a constructor. It behaves in many ways like a constructor, e.g. it is the first code which is executed, when a new instance of a class is created. The name sounds also like a constructor "__init__". But strictly speaking it would be wrong to call it a constructor, because a new instance is already "constructed" by the time the method __init__ is called. But anyway, the __init__ method is used - like constructors in other object oriented programming languages - to initialize the instance variables of an object. The definition of an init method looks like any other method definition:

```
def __init__(self, holder, number, balance,credit_line=1500):
        self.Holder = holder
        self.Number = number
        self.Balance = balance
        self.CreditLine = credit_line
```

Inheritance

The concept of inheritance provides an important feature to the the object-oriented programming is reuse of code. Inheritance is the process of creating a new class (derived class) to be based on an existing (base class) one where the new class inherits all the attributes and methods of the existing class. Following diagram shows the inheritance of a derived class from the parent (base) class.



Like other object-oriented language, Python allows inheritance from a parent (or base) class as well as multiple inheritances in which a class inherits attributes and methods from more than one parent. See the single and multiple inheritance syntaxes :class DerivedClassName(BaseClassName):

```
    Statement-1
    Statement-1
    ....
    ....
    ....
    Statement-n
```

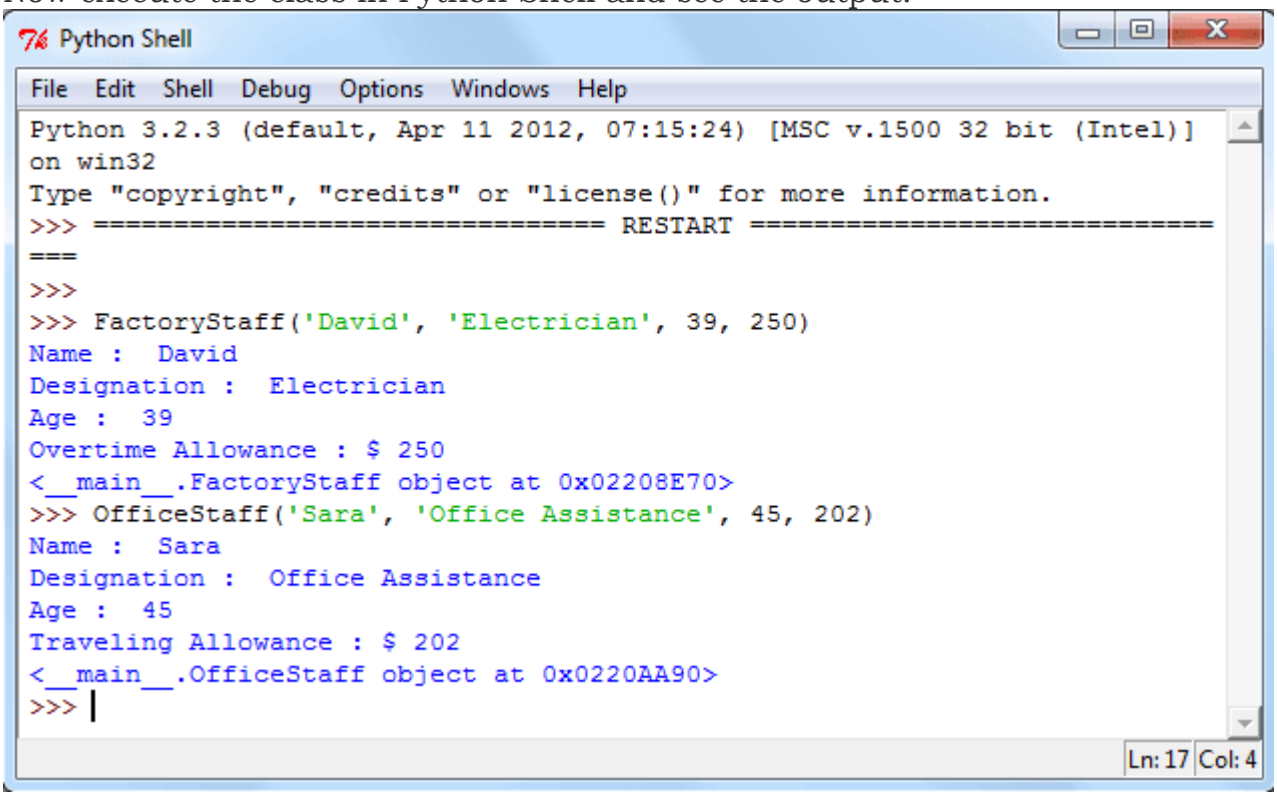
```
class DerivedClassName(BaseClassName1, BaseClassName2, BaseClassName3):
    Statement-1
    Statement-1
    ....
    ....
    ....
    Statement-n
```

Example :

In a company Factory, staff and Office staff have certain common properties - all have a name, designation, age etc. Thus they can be grouped under a class called CompanyMember. Apart from sharing those common features, each subclass has its own characteristic - FactoryStaff gets overtime allowance while OfficeStaff gets traveling allowance for an office job. The derived classes (FactoryStaff & OfficeStaff) has its own characteristic and, in addition, they inherit the properties of the base class (CompanyMember). See the example code.

```
1.  # python-inheritance.py
2.  class CompanyMember:
3.     '''Represents Company  Member.'''
4.     def __init__(self, name, designation, age):
5.        self.name = name
6.        self.designation = designation
7.        self.age = age
8.     def tell(self):
9.        '''Details of an employee.'''
10.              print('Name: ', self.name,'\nDesignation : ',self.designation, '\nAge
    : ',self.age)
11.
12.       class FactoryStaff(CompanyMember):
13.          '''Represents a Factory Staff.'''
14.          def __init__(self, name, designation, age, overtime_allow):
15.             CompanyMember.__init__(self, name, designation, age)
16.             self.overtime_allow = overtime_allow
17.             CompanyMember.tell(self)
18.             print('Overtime Allowance : ',self.overtime_allow)
19.
20.       class OfficeStaff(CompanyMember):
21.          '''Represents a Office Staff.'''
22.          def __init__(self, name, designation, age, travelling_allow):
23.             CompanyMember.__init__(self, name, designation, age)
24.             self.marks = travelling_allow
25.             CompanyMember.tell(self)
26.             print('Traveling Allowance : ',self.travelling_allow)
```

Now execute the class in Python Shell and see the output.



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
```

Data Hiding:

An object's attributes may or may not be visible outside the class definition. For these cases, you can name attributes with a double underscore prefix, and those attributes will 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 would work for you:

.....
print counter._JustCounter__secretCount

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

1
2
2

Difference between an error and Exception

Exceptions occur when *exceptional* situations occur in your program. For example, what if you are going to read a file and the file does not exist? Or what if you accidentally deleted it when the program was running? Such situations are handled using **exceptions**.

Similarly, what if your program had some invalid statements? This is handled by Python which **raises** its hands and tells you there is an **error**.

Errors

Consider a simple print function call. What if we misspelt print as Print? Note the capitalization. In this case, Python *raises* a syntax error.

```
>>> Print("Hello World")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'Print' is not defined
>>> print("Hello World")
Hello World
```

Observe that a NameError is raised and also the location where the error was detected is printed. This is what an **error handler** for this error does.

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 can't cope with, it raises an exception. An exception is a Python object that represents an error.

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

Handling an exception:

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

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:

Here is simple example, which opens a file and writes the 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 will produce the following result:

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

Example:

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

```
#!/usr/bin/python

try:
   fh = open("testfile", "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"
```

This will produce 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 can not 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 are writing the code to handle a single exception, you can have a variable follow the name of the exception in the except statement. If you are trapping multiple exceptions, you can have a variable follow the tuple of the exception.

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

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 would produce 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.

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 our 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 your exception as follows:

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