UNIT – IV

1. Modules

The Python module—the highest-level program organization unit, which packages program code and data for reuse, and provides self contained namespaces that minimize variable name clashes across your programs. In concrete terms, modules typically correspond to Python program files. Each file is a module, and modules import other modules to use the names they define. Modules might also correspond to extensions coded in external languages such as C, Java, or C#, and even to directories in package imports. Modules are processed with two statements and one important function:

import-Lets a client (importer) fetch a module as a whole

from- Allows clients to fetch particular names from a module

imp.reload (reload in 2.X)- Provides a way to reload a module's code without stopping Python.

Why Use Modules?

In short, modules provide an easy way to organize components into a system by serving as self-contained packages of variables known as namespaces. All the names defined at the top level of a module file become attributes of the imported module object. Imports give access to names in a module's global scope. That is, the module file's global scope morphs into the module object's attribute namespace when it is imported. Ultimately, Python's modules allow us to link individual files into a larger program system.

More specifically, modules have at least three roles:

Code reuse

Modules let you save code in files permanently. Unlike code you type at the Python interactive prompt, which goes away when you exit Python, code in module files is persistent—it can be reloaded and rerun as many times as needed. Just as importantly, modules are a place to define names, known as attributes, which may be referenced by multiple external clients. When used well, this supports a modular program design that groups functionality into reusable units.

System namespace partitioning

Modules are also the highest-level program organization unit in Python. Although they are undamentally just packages of names, these packages are also self-contained— you can never see a name in another file, unless you explicitly import that file. Much like the local scopes of functions, this helps avoid name clashes across your programs. In fact, you can't avoid this feature—everything "lives" in a module, both the code you run and the objects you create are always implicitly enclosed in modules. Because of that, modules are natural tools for grouping system components.

Implementing shared services or data

From an operational perspective, modules are also useful for implementing components that are shared across a system and hence require only a single copy. For instance, if you need to provide a global object that's used by more than one function or file, you can code it in a module that can then be imported by many clients.

How to Structure a Program

At a base level, a Python program consists of text files containing Python statements, with one main top-level file, and zero or more supplemental files known as modules. Here's how this works. The top-level (a.k.a. script) file contains the main flow of control of your program—this is the file you run to launch your application. The module files are libraries of tools used to collect components used by the top-level file, and possibly elsewhere. Top-level files use tools defined in module files, and modules use tools defined in other modules.

Although they are files of code too, module files generally don't do anything when run directly; rather, they define tools intended for use in other files. A file imports a module to gain access to the tools it defines, which are known as its attributes—variable names attached to objects such as functions. Ultimately, we import modules and access their attributes to use their tools.

Imports and Attributes

Let's make this a bit more concrete. Figure sketches the structure of a Python program composed of three files: a.py, b.py, and c.py. The file a.py is chosen to be the top-level file; it will be a simple text file of statements, which is executed from top to bottom when launched. The files b.py and c.py are modules; they are simple text files of statements as well, but they are not usually launched directly. Instead, as explained previously, modules are normally imported by other files that wish to use the tools the modules define.

For instance, suppose the file b.py in Figure defines a function called spam, for external use. b.py will contain a Python def statement to generate the function, which you can later run by passing zero or more values in parentheses after the function's name:

```
# File b.py
def spam(text):
        print(text, 'spam')
```

Now, suppose a.py wants to use spam. To this end, it might contain Python statements such as the following:

```
import b                # File a.py
b.spam('gumby')         # Prints "gumby spam"
```

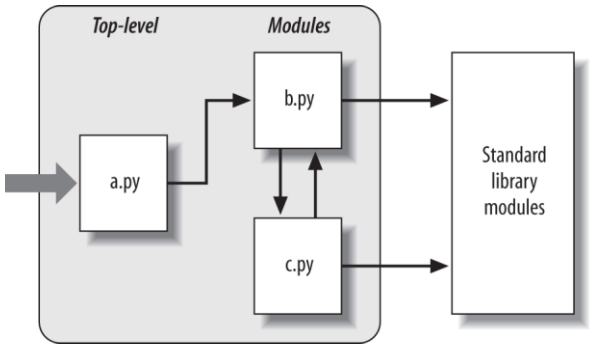

Fig: *Program architecture in Python*

The first of these, a Python import statement, gives the file a.py access to everything defined by top-level code in the file b.py. The code import b roughly means: Load the file b.py (unless it's already loaded), and give me access to all its attributes through the name b.

To satisfy such goals, import (and, as you'll see later, from) statements execute and load other files on request. More formally, in Python, cross-file module linking is not resolved until such import statements are executed at runtime; their net effect is to assign module names—simple variables like b—to loaded module objects. In fact, the module name used in an import statement serves two purposes: it identifies the external file to be loaded, but it also becomes a variable assigned to the loaded module. Similarly, objects defined by a module are also created at runtime, as the import is executing: import literally runs statements in the target file one at a time to create its contents. Along the way, every name assigned at the top-level of the file becomes an attribute of the module, accessible to importers.

For example, the second of the statements in a.py calls the function spam defined in the module b—created by running its def statement during the import—using object attribute notation. The code b.spam means:

Fetch the value of the name spam that lives within the object b.

This happens to be a callable function in our example, so we pass a string in parentheses ('gumby'). If you actually type these files, save them, and run a.py, the words "gumby spam" will be printed.

The notion of importing is also completely general throughout Python. Any file can import tools from any other file. For instance, the file a.py may import b.py to call its function, but b.py might also import c.py to leverage different tools defined there. Import chains can go as deep as you like: in this example, the module a can import b, which can import c, which can import b again, and so on.

How Imports Work

They are really runtime operations that perform three distinct steps the first time a program imports a given file:

1. Find the module's file.

2. Compile it to byte code (if needed).

3. Run the module's code to build the objects it defines.

To better understand module imports, we'll explore these steps in turn. Bear in mind that all three of these steps are carried out only the first time a module is imported during a program's execution; later imports of the same module in a program run bypass all of these steps and simply fetch the already loaded module object in memory.

Technically, Python does this by storing loaded modules in a table named sys.modules and checking there at the start of an import operation. If the module is not present, a three-step process begins.

1. Find It

First, Python must locate the module file referenced by an import statement. Notice that the import statement in the prior section's example names the file without a .py extension and without its directory path: it just says import b, instead of something like import c:\dir1\b.py. Path and extension details are omitted on purpose; instead, Python uses a standard module search path and known file types to locate the module file corresponding to an import statement. Because this is the main part of the import operation that programmers must know about, we'll return to this topic in a moment

2. Compile It (Maybe)

After finding a source code file that matches an import statement by traversing the module search path, Python next compiles it to byte code, if necessary. During an import operation Python checks both file modification times and the byte code's Python version number to decide how to proceed. The former uses file "timestamps," and the latter uses either a "magic" number embedded in the byte code or a filename, depending on the Python release being used. This step chooses an action as follows:

Compile

If the byte code file is older than the source file (i.e., if you've changed the source) or was created by a different Python version, Python automatically regenerates the byte code when the program is run.

Don't compile

If, on the other hand, Python finds a .pyc byte code file that is not older than the corresponding .py source file and was created by the same Python version, it skips the source-to-byte-code compile step. In addition, if Python finds only a byte code file on the search path and no source, it simply loads the byte code directly; this means you can ship a program as just byte code files and avoid sending source. In other words, the compile step is bypassed if possible to speed program startup. Notice that compilation happens when a file is being imported. Because of this, you will not usually see a .pyc byte code file for the top-level file of your program, unless it is also imported elsewhere—only imported files leave behind .pyc files on your machine.

3. Run It

The final step of an import operation executes the byte code of the module. All statements in the file are run in turn, from top to bottom, and any assignments made to names during this step generate attributes of the resulting module object. This is how the tools defined by the module's code are created. For instance, def statements in a file are run at import time to create functions and assign attributes within the module to those functions. The functions can then be called later in the program by the file's importers.

Because this last import step actually runs the file's code, if any top-level code in a module file does real work, you'll see its results at import time. For example, top-level print statements in a module show output when the file is imported. Function def statements simply define objects for later use.

Any given module is imported only once per process by default. Future imports skip all three import steps and reuse the already loaded module in memory.

Python modules are easy to *create*; they're just files of Python program code created with a text editor. You don't need to write special syntax to tell Python you're making a module; almost any text file will do. Because Python handles all the details of finding and loading modules, modules are also easy to *use*; clients simply import a module, or specific names a module defines, and use the objects they reference.

1.1 Module Creation

To define a module, simply use your text editor to type some Python code into a text file, and save it with a ".py" extension; any such file is automatically considered a Python module. All the names assigned at the top level of the module become its attributes (names associated with the module object) and are exported for clients to use —they morph from variable to module object attribute automatically.

For instance, if you type the following def into a file called module1.py and import it, you create a module object with one attribute—the name printer, which happens to be a reference to a function object:

```
def printer(x): # Module attribute

        print(x)
```

Module Filenames

You can call modules just about anything you like, but module filenames should end in a .py suffix if you plan to import them. The .py is technically optional for top-level files that will be run but not imported, but adding it in all cases makes your files' types more obvious and allows you to import any of your files in the future.

Because module names become variable names inside a Python program (without the .py), they should also follow the normal variable name rules. For instance, you can create a module file named if.py, but you cannot import it because if is a reserved word—when you try to run import if, you'll get a syntax error. In fact, both the names of module files and the names of directories used in package imports must conform to the rules for variable names. They may, for instance, contain only letters, digits, and underscores. Package directories also cannot contain platform-specific syntax such as spaces in their names.

When a module is imported, Python maps the internal module name to an external filename by adding a directory path from the module search path to the front, and a .py or other extension at the end. For instance, a module named M ultimately maps to some external file <directory>\M.<extension> that contains the module's code.

Other Kinds of Modules

It is also possible to create a Python module by writing code in an external language such as C, C++, and others (e.g., Java, in the Jython implementation of the language). Such modules are called extension modules, and they are generally used to wrap up external libraries for use in Python scripts. When imported by Python code, extension modules look and feel the same as modules coded as Python source code files—they are accessed with import statements, and they provide functions and objects as module attributes.

Module Usage

Clients can use the simple module file we just wrote by running an import or from statement. Both statements find, compile, and run a module file's code, if it hasn't yet been loaded. The chief difference is that import fetches the module as a whole, so you must qualify to fetch its names; in contrast, from fetches (or copies) specific names out of the module.

Let's see what this means in terms of code. All of the following examples wind up calling the printer function defined in the prior section's module1.py module file, but in different ways.

1.3 The import Statement

In the first example, the name module1 serves two different purposes—it identifies an external file to be loaded, and it becomes a variable in the script, which references the module object after the file is loaded:

```
>>> import module1              # Get module as a whole (one or more)

>>> module1.printer('Hello world!')  # Qualify to get names

Hello world!
```

The import statement simply lists one or more names of modules to load, separated by commas. Because it gives a name that refers to the whole module object, we must go through the module name to fetch its attributes (e.g., module1.printer).

The from Statement

By contrast, because from copies specific names from one file over to another scope, it allows us to use the copied names directly in the script without going through the module (e.g., printer):

```
>>> from module1 import printer       # Copy out a variable (one or more)

>>> printer('Hello world!')            # No need to qualify name

Hello world!
```

This form of from allows us to list one or more names to be copied out, separated by commas. Here, it has the same effect as the prior example, but because the imported name is copied into the scope where the from statement appears, using that name in the script requires less typing—we can use it directly instead of naming the enclosing module. In fact, we must; from doesn't assign the name of the module itself. The from statement is really just a minor extension to the import statement—it imports the module file as usual , but adds an extra step that copies one or more names (not objects) out of the file. The entire file is loaded, but you're given names for more direct access to its parts.

The from * Statement

Finally, the next example uses a special form of from: when we use a * instead of specific names, we get copies of all names assigned at the top level of the referenced module.Here again, we can then use the copied name printer in our script without going through the module name:

```
>>> from module1 import *              # Copy out _all_ variables

>>> printer('Hello world!')
```

Hello world!

Technically, both import and from statements invoke the same import operation; the from * form simply adds an extra step that copies all the names in the module into the importing scope. It essentially collapses one module's namespace into another; again, the net effect is less typing for us. Note that only * works in this context; you can't use pattern matching to select a subset of names.

The from ...* statement form described here can be used only at the top level of a module file, not within a function. Python 2.X allows it to be used within a function, but issues a warning anyhow.

When import is required The only time you really must use import instead of from is when you must use the same name defined in two different modules. For example, if two files define the same name differently:

```
# M.py

def func():

        ...do something...

# N.py

def func():

        ...do something else...
```

and you must use both versions of the name in your program, the from statement will fail—you can have only one assignment to the name in your scope:

```
# O.py

from M import func

from N import func              # This overwrites the one we fetched from M

func()                          # Calls N.func only!
```

An import will work here, though, because including the name of the enclosing module makes the two names unique:

```
# O.py

import M, N    # Get the whole modules, not their names

M.func()       # We can call both names now

N.func()        # The module names make them unique
```

This case is unusual enough that you're unlikely to encounter it very often in practice.

Example-2

A module allows you to logically organize your Python code. Grouping related code into a module makes the code easier to understand and use. A module is a Python object with arbitrarily named attributes that you can bind and reference. Simply, a module is a file consisting of Python code. A module can define functions, classes and variables. A module can also include runnable code. The Python code for a module named aname normally resides in a file named aname.py. Here is an example of a simple module, support.py

```
def print_func( par ):

        print "Hello : ", par

        return
```

The import Statement

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

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

When the interpreter encounters an import statement, it imports the module if the module is present in the search path. A search path is a list of directories that the interpreter searches before importing a module. For example, to import the module hello.py, you need to put the following command at the top of the script:

```
import support
# Now you can call defined function that module as follows

support.print_func("Zara")
```

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

```
Hello : Zara
```

A module is loaded only once, regardless of the number of times it is imported. This prevents the module execution from happening over and over again if multiple imports occur.

The from...import Statement

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

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

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

```
from fib import fibonacci
```

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

The from...import * Statement:

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

```
from modname import *
```

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

Example: A module or file with three functions.

```
Exampl1.py
def add():
   print("sum=",2+3)

def sub():
   print("sum=",2-3)

def hello():
   print("hello to all")
```

```
Example2.py
import python48
python48.add()
```

output: sum= 5

The above program imports python48 modules. It adds all the functions and attributes defined in python48.py

To import the specific functions then we need to use the following statement:

```
Example2.py
from python48 import add
add()
```

output:

```
sum=5
```

Here we need not to use the module name.

Locating Modules:

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

- The current directory.

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

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

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

The PYTHONPATH Variable

The PYTHONPATH is an environment variable, consisting of a list of directories. The syntax of PYTHONPATH is the same as that of the shell variable PATH.

Here is a typical PYTHONPATH from a Windows system:

```
set PYTHONPATH=c:\python20\lib;
```

And here is a typical PYTHONPATH from a UNIX system:

```
set PYTHONPATH=/usr/local/lib/python
```

1.4 Namespaces and Scoping

Variables are names (identifiers) that map to objects. A namespace is a dictionary of variable names (keys) and their corresponding objects (values). A Python statement can access variables in a local namespace and in the global namespace. If a local and a global variable have the same name, the local variable shadows the global variable. Each function has its own local namespace. Class methods follow the same scoping rule as ordinary functions. Python makes educated guesses on whether variables are local or global. It assumes that any variable assigned a value in a function is local.

Therefore, in order to assign a value to a global variable within a function, you must first use the global statement.

The statement global VarName tells Python that VarName is a global variable. Python stops searching the local namespace for the variable.

For example, we define a variable *Money* in the global namespace. Within the function *Money*, we assign *Money* a value, therefore Python assumes *Money* as a local

variable. However, we accessed the value of the local variable *Money* before setting it, so an UnboundLocalError is the result. Uncommenting the global statement fixes the problem.

```
Money = 2000
def AddMoney():
        # Uncomment the following line to fix the code:
        # global Money
        Money = Money + 1
        print Money
AddMoney()
print Money
```

Namespace Dictionaries: __dict__

In fact, internally, module namespaces are stored as dictionary objects. These are just normal dictionaries with all the usual methods. When needed—for instance, to write tools that list module content generically we can access a module's namespace dictionary through the module's __dict__ attribute.

```
>>> list(module2.__dict__.keys())

['__loader__', 'func', 'klass', '__builtins__', '__doc__', '__file__', '__name__',

'name', '__package__', 'sys', '__initializing__', '__cached__']
```

The names we assigned in the module file become dictionary keys internally, so some of the names here reflect top-level assignments in our file. However, Python also adds some names in the module's namespace for us; for instance, __file__ gives the name of the file the module was loaded from, and __name__ gives its name as known to importers (without the .py extension and directory path).

```
>>> module2.name, module2.__dict__['name']

(42, 42)
```

2. package

A package is a hierarchical file directory structure that defines a single Python application environment that consists of modules and subpackages and sub-subpackages, and so on. Consider a file Pots.py available in Phone directory. This file has following line of source code:

```
Pots.py
def Pots():
        print ("I'm Pots Phone" )

Isdn.py
def Isdn():
        print ("I'm Isdn Phone" )

G3.py
def G3():
        print ("I'm G3 Phone" )


__init__.py
from Pots import Pots
from Isdn import Isdn
from G3 import G3
```

Similar way, we have another two files having different functions with the same name as above:

- Phone/Isdn.py file having function Isdn()
- Phone/G3.py file having function G3()

Now, create one more file __init__.py in Phone directory:

- Phone/__init__.py

To make all of your functions available when you've imported Phone, you need to put explicit import statements in __init__.py as shown.

```
import Phone
Phone.Pots()
Phone.Isdn()
Phone.G3()
```

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

```
I'm Pots Phone
I'm 3G Phone
I'm ISDN Phone
```

In the above example, we have taken example of a single functions in each file, but you can keep multiple functions in your files. You can also define different Python classes in those files and then you can create your packages out of those classes.

3. Exception Handling

3.1 Difference between an error and Exception

Until now error messages haven't been more than mentioned, but if you have tried out the examples you have probably seen some. There are (at least) two distinguishable kinds of errors: *syntax errors* and *exceptions*.

1. Syntax Errors
Syntax errors, also known as parsing errors, are perhaps the most common kind of complaint you get while you are still learning Python:

```
>>> while True print('Hello world')
  File "<stdin>", line 1
    while True print('Hello world')
                   ^
SyntaxError: invalid syntax
```

The parser repeats the offending line and displays a little 'arrow' pointing at the earliest point in the line where the error was detected. The error is caused by (or at least detected at) the token *preceding* the arrow: in the example, the error is detected at the function print(), since a colon (':') is missing before it. File name and line number are printed so you know where to look in case the input came from a script.

2. Exceptions
Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called *exceptions* and are not unconditionally fatal: you will soon learn how to handle them in Python programs. Most exceptions are not handled by programs, however, and result in error messages as shown here:

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

The last line of the error message indicates what happened. Exceptions come in different types, and the type is printed as part of the message: the types in the example are ZeroDivisionError, NameError and TypeError. The string printed as the exception type is the name of the built-in exception that occurred. This is true for all built-in exceptions, but need not be true for user-defined exceptions (although it is a useful convention). Standard exception names are built-in identifiers (not reserved keywords). The rest of the line provides detail based on the type of exception and what caused it. The preceding part of the error message shows the context where the exception happened, in the form of a stack traceback. In general it contains a stack traceback listing source lines; however, it will not display lines read from standard input.

Why Use Exceptions?

In Python programs, exceptions are typically used for a variety of purposes. Here are some of their most common roles:

Error handling

Python raises exceptions whenever it detects errors in programs at runtime. You can catch and respond to the errors in your code, or ignore the exceptions that are raised. If an error is ignored, Python's default exception-handling behavior kicks in: it stops the program and prints an error message. If you don't want this default behavior, code a try statement to catch and recover from the exception—Python will jump to your try handler when the error is detected, and your program will resume execution after the try.

Event notification

Exceptions can also be used to signal valid conditions without you having to pass result flags around a program or test them explicitly. For instance, a search routine might raise an exception on failure, rather than returning an integer result code— and hoping that the code will never be a valid result!

Special-case handling

Sometimes a condition may occur so rarely that it's hard to justify convoluting your code to handle it in multiple places. You can often eliminate special-case code by handling unusual cases in exception handlers in higher levels of your program.

An Termination actions

The try/finally statement allows you to guarantee that required closing-time operations will be performed, regardless of the presence or absence of exceptions in your programs. The newer with statement offers an alternative in this department for objects that support it.

Unusual control flows

Finally, because exceptions are a sort of high-level and structured "go to," you can use them as the basis for implementing exotic control flows. For instance, although the language does not explicitly support backtracking, you can implement it in Python by using exceptions and a bit of support logic to unwind assignments. There is no "go to" statement in Python (thankfully!), but exceptions can sometimes serve similar roles; a raise, for instance, can be used to jump out of multiple loops.

Default Exception Handler

Suppose we write the following function:

```
>>> def fetcher(obj, index):

        return obj[index]
```

There's not much to this function—it simply indexes an object on a passed-in index. In normal operation, it returns the result of a legal index:

```
>>> x = 'spam'

>>> fetcher(x, 3) # Like x[3]

'm'
```

However, if we ask this function to index off the end of the string, an exception will be triggered when the function tries to run obj[index]. Python detects out-of-bounds indexing for sequences and reports it by raising (triggering) the built-in IndexError exception:

```
>>> fetcher(x, 4)        # Default handler - shell interface

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

File "<stdin>", line 2, in fetcher

IndexError: string index out of range
```

Because our code does not explicitly catch this exception, it filters back up to the top level of the program and invokes the default exception handler, which simply prints the standard error message

In a more realistic program launched outside the interactive prompt, after printing an error message the default handler at the top also terminates the program immediately. That course of action makes sense for simple scripts; errors often should be fatal, and the best you can do when they occur is inspect the standard error message.

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:

```
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:

```
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.

3.2 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

```
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:

```
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.

3.3 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 can 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. 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:

```
def temp_convert(var):
	try:
		return int(var)
	except ValueError, Argument:
		print "The argument does not contain numbers\n", Argument
temp_convert("xyz");
```

This produces the following result:

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

3.4 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...
```

3.5 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):
    print( "hello")
```

Output:

```
hello
```

4. Object Oriented Programming OOP in Python

Python has been an object-oriented language since it existed. Because of this, creating and using classes and objects are downright easy.

Overview of OOP Terminology

1. **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.
2. **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.
3. **Data member**: A class variable or instance variable that holds data associated with a class and its objects.
4. **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.
5. **Instance variable**: A variable that is defined inside a method and belongs only to the current instance of a class.
6. **Inheritance**: The transfer of the characteristics of a class to other classes that are derived from it.
7. **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.
8. **Instantiation**: The creation of an instance of a class.
9. **Method**: A special kind of function that is defined in a class definition.
10. **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.
11. **Operator overloading**: The assignment of more than one function to a particular operator.

4.1 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:

```
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:

1. The getattr(obj, name[, default]) : to access the attribute of object.
2. The hasattr(obj,name) : to check if an attribute exists or not.
3. The setattr(obj,name,value) : to set an attribute. If attribute does not exist, then it would be created.
4. 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
```

self variable

Methods (or, more technically, bound methods) have their first parameter bound to the instance they belong to, so you don’t have to supply it. While you can certainly bind an attribute to a plain function, it won’t have that special self parameter:

```
>>> class Class:
        def method(self):
            print 'I have a self!'
>>> def function():
        print "I don't..."
>>> instance = Class()
>>> instance.method()
I have a self!
>>> instance.method = function
>>> instance.method()
I don't...
```

Note that the self parameter is not dependent on calling the method the way I’ve done until now, as instance.method. You’re free to use another variable that refers to the same method:

```
>>> class Bird:
        song = 'Squaawk!'
        def sing(self):
            print self.song
>>> bird = Bird()
>>> bird.sing()
Squaawk!
>>> birdsong = bird.sing
>>> birdsong()
Squaawk!
```

Even though the last method call looks exactly like a function call, the variable birdsong refers to the bound method bird.sing, which means that it still has access to the self parameter (that is, it is still bound to the same instance of the class).

4.2 METHODS

Instance methods and class methods are the two ways to call Python methods. As a matter of fact, instance methods are automatically converted into class methods by Python.

```
class PrintClass :
        def printMethod ( self , input ) :
                print input
```

Now we'll call the class' method using the normal instance method and the "new" class method:

```
>>>x = PrintClass ( )
>>>x . printMethod ( "Try spam ! " )          #instance method
Try spam!

>>> PrintClass. printMethod ( x , "Buy more spam ! " )       #class method
Buy more spam!
```

So, what is the benefit of using class methods? Well, when using inheritance you can extend, rather than replace, inherited behavior by calling a method via the class rather than the instance. Here's a generic example:

Class methods and inheritance

```
>>>class Super :
        def method ( self) :
                print "now in Super .method"
>>>class Subclass ( Super ) :
        def method ( self ) :              #override method
                print " starting Subclass .method"      #new actions
                Super .method( self )
                print "ending Subclass .method"

>>>x = Super ( )        #make a Super i n s t a n c e
>>>x . method ( )        #run Super . method
now in Super .method
>>>x = Subclass( )     #make a S u b c l a s s i n s t a n c e
>>>x . method ( )                    #run S u b c l a s s . method which c a l l s
Super .method
Starting  Subclass .method
now in Super .method
ending Subclas s .method
```

Using class methods this way, you can have a subclass extend thedefault method actions by having specialized subclass actions yet still call the original default behavior via the superclass.

4.3 Constructor

The first magic method we'll take a look at is the constructor. In case you have never heard the word constructor before, it's basically a fancy name for the kind of initializing method I have already used in some of the examples, under the name init. What separates constructors from ordinary methods, however, is that the constructors are called automatically right after an object has been created. Thus, instead of doing what I've been doing up until now:

```
>>> f = FooBar()
>>> f.init()
```

constructors make it possible to simply do this:

```
>>> f = FooBar()
```

Creating constructors in Python is really easy; simply change the init method’s name from the plain old init to the magic version, __init__:

```
class FooBar:
def __init__(self):
        self.somevar = 42
>>> f = FooBar()
>>> f.somevar
42
```

Now, that’s pretty nice. But you may wonder what happens if you give the constructor some parameters to work with. Consider the following:

```
class FooBar:
        def __init__(self, value=42):
                self.somevar = value
```

How do you think you could use this?

```
>>> f = FooBar('This is a constructor argument')
>>> f.somevar
'This is a constructor argument'
```

Of all the magic methods in Python, __init__ is quite certainly the one you’ll be using the most.

4.4 Inheritance Class

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 calss 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

Methods Overriding

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

```
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, and 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) :Evaluatable 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)
```

4.4 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 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
```