**UNIT V FILES, MODULES, PACKAGES**

**5.1 FILES AND EXCEPTION: TEXT FILES**

**5.1.1 FILES**

A **file** is an object on a computer that stores data, information, settings, or commands used with a computer program and in order for easy reference.

**EXAMPLE:** a folder or box for holding loose papers together, student having all their certificates together in a file

**SYNTAX:**

**open("filename","w+")**

**5.1.2 OPEN( ) FUNCTION**

In order to open a file for writing or use in Python, you must rely on the built-in open () function.

As explained above, open ( ) will return a file object, so it is most commonly used with two arguments.

An argument is nothing more than a value that has been provided to a function, which is relayed when you call it. So, for instance, if we declare the name of a file as “Test File,” that name would be considered an argument.

**SYNTAX:**

**file\_object  = open(“filename”, “mode”)** where file\_object is the variable to add the file object.

The second argument you see – mode – tells the interpreter and developer which way the file will be used.

**5.1.3 MODE**

Including a mode argument is optional because a default value of ‘r’ will be assumed if it is omitted. The ‘r’ value stands for read mode, which is just one of many.

The modes are:

**‘r’** – Read mode which is used when the file is only being read

**‘w’** – Write mode which is used to edit and write new information to the file (any existing files with the same name will be erased when this mode is activated)

‘r+’ – Special read and write mode, which is used to handle both actions when working with a file

**EXAMPLE:**

F = open(“workfile”,”w”)

Print f

This snippet opens the file named “workfile” in writing mode so that we can make changes to it. The current information stored within the file is also displayed – or printed – for us to view.

Once this has been done, you can move on to call the objects functions. The two most common functions are read and write.

**5.1.4 APPEND**

To add data to an existing file use the command

**SYNTAX:**

**open("Filename", "a")**

**5.1.6 TEXT FILE**

A **text file** is a sequence of characters stored on a permanent medium like a hard drive, flash memory, or CD-ROM

**EXAMPLE:** Resume or assignments stored in a pen drive or CD

**5.1.6.1 OPEN TEXT FILE**

The built-in function open takes the name of the file as a parameter and returns a **file object** you can use to read the file.

**SYNTAX:**

**open("filename.txt","r")**

**EXAMPLE:**

**fh=open(“hello.txt”,”r”)**

Mode 'r' indicates that this file is open for reading (as opposed to 'w' for

writing).

We declared the variable fh to open a file named hello.txt. Open takes 2 arguments, the file that we want to open and a string that represents the kinds of permission or operation we want to do on the file.

 If you open the text file – or look at it – using Python you will see only the text we told the interpreter to add.

**$ cat hello.txt**

**Hello World**

**put the text you want to add here**

**5.2 READING AND WRITING FILES**

**5.2.1 READING FILES**

The **read** function is to read the contents of a file. It is of three categories. They are

* **read ()**
* **readline ()**
* **readlines ()**

**5.2.1.1 READ ()**

The read function is to read the ENTIRE contents of a file.

**SYNTAX:**

**fh = open(“file name.txt”, ”r”)**

**print fh.read()**

**EXAMPLE:**

**fh = open(“hello.txt”, ”r”)**

**print fh.read()**

**5.2.1.2 READLINE ()**

The readline function is to read the contents of a file line by line. i.e one line at a time

**SYNTAX:**

**fh = open(“file name.txt”, ”r”)**

**print fh.readline()**

**EXAMPLE:**

**fh = open(“hello.txt”, ”r”)**

**print fh.readline ()**

**5.2.1.3 READLINES ()**

The readlines function is to read list of lines in a text file.

**SYNTAX:**

**fh = open(“file name.txt”, ”r”)**

**print fh.readlines()**

**EXAMPLE:**

**fh = open(“hello.txt”, ”r”)**

**print fh.readlines ()**

**5.2.2 WRITING FILES**

The **write** function is to write the contents of a file. It is of three categories. They are

* **write ()**
* **writeline ()**
* **writelines ()**

**5.2.1.1 WRITE ()**

The write function is to write the ENTIRE contents of a file.

**SYNTAX:**

**fh = open(“file name.txt”, ”w”)**

**print fh.write()**

**EXAMPLE:**

**fh = open(“hello.txt”, ”w”)**

**fh.write(“put the text you want to add here”)**

**fh.close()**

**5.2.1.2 WRITELINE ()**

The writeline function is to write the contents of a file line by line. i.e one line at a time

**SYNTAX:**

**fh = open(“file name.txt”, ”w”)**

**print fh. writeline()**

**EXAMPLE:**

**fh = open(“hello.txt”, ”w”)**

**print fh. writeline ()**

**5.2.1.3 WRITELINES ()**

The writelines function is to write list of lines in a text file.

**SYNTAX:**

**fh = open(“file name.txt”, ”w”)**

**print fh.writelines()**

**EXAMPLE:**

**fh = open(“hello.txt”, ”w”)**

**lines\_of\_text = [“One line of text here”, “and another line here”, “and yet another here”, “and so on and so forth”]**

**fh.writelines(lines\_of\_text)**

**fh.close()**

In addition to write, writeline and writelines functions the following is used to create a file after checking whether the file is not there in library.

**EXAMPLE:**

**fh= open(“hello.txt”, “w+”)**

"w" letter in our argument indicates write and the plus sign that means it

will create a file if it does not exist in library

**5.2.1.4 WITH STATEMENT**

We can also work with file objects using the with statement. It is designed to provide much cleaner syntax and exceptions handling when you are working with code. That explains why it’s good practice to use the with statement where applicable.

One bonus of using this method is that any files opened will be closed automatically after you are done. This leaves less to worry about during cleanup.

To use the with statement to open a file:

**SYNTAX:**

**with open(“file name”) as file:**

**EXAMPLE:**

**With open (“hello.txt”) as file:**

**Data = file.read( )**

**Do something with data**

**5.2.1.5 LOOP OVER A FILE OBJECT**

When you want to write – or return – all the lines from a file in a more memory efficient, and fast manner, you can use the loop over method.

**EXAMPLE:**

**With open (“hello.txt”) as file:**

**for line in file:**

**print line,**

**5.3 FORMAT OPERATOR**

An operator,%, that takes a format string and a tuple and generates astring that includes the elements of the tuple formatted as specified by the format string.

When applied to integers, % is the modulus operator. But when the first operand is a string, % is the format operator.

The first operand is the **format string**, which contains one or more **format sequences**, which specify how the second operand is formatted. The result is a string.

**EXAMPLE 1:**

For example, the format sequence '%d' means that the second operand should be formatted as an integer (d stands for “decimal”):

1. **camels = 42**
2. **'%d' % camels**

**'42'**

The result is the string '42', which is not to be confused with the integer value 42.

**EXAMPLE 2:**

A format sequence can appear anywhere in the string, so you can embed a value in a sentence:

* + - 1. **camels = 42**

1. **'I have spotted %d camels. ' % camels 'I have spotted 42 camels.** '

If there is more than one format sequence in the string, the second argument has to be a tuple. Each format sequence is matched with an element of the tuple, in order.

**EXAMPLE 3:**

The following example uses '%d' to format an integer, '%g' to format a floating-point num-ber (don’t ask why), and '%s' to format a string:

**>>> 'In %d years I have spotted %g %s. ' % (3, 0.1, 'camels') 'In 3**

**years I have spotted 0.1 camels. '**

**EXAMPLE 4:**

The number of elements in the tuple has to match the number of format sequences in the string. Also, the types of the elements have to match the format sequences:

* + - 1. **'%d %d %d' % (1, 2)**

**TypeError: not enough arguments for format string**

* + 1. **'%d' % 'dollars'**

**TypeError: illegal argument type for built-in operation**

**5.4 COMMAND LINE ARGUMENTS**

The Python **sys** module provides access to any command-line arguments via the **sys.argv**. This serves two purposes:

* sys.argv is the list of command-line arguments.
* len(sys.argv) is the number of command-line arguments.

Here sys.argv[0] is the program ie. script name.

**EXAMPLE:**

Consider the following script test.py –

**#!/usr/bin/python**

**import sys**

**print 'Number of arguments:', len(sys.argv), 'arguments.'**

**print 'Argument List:', str(sys.argv)**

Now run above script as follows −

**$ python test.py arg1 arg2 arg3**

This produce following result −

**Number of arguments: 4 arguments.**

**Argument List: ['test.py', 'arg1', 'arg2', 'arg3']**

**NOTE:** As mentioned above, first argument is always script name and it is also being counted in number of arguments.

**5.4.1 PARSING COMMAND-LINE ARGUMENTS**

Python provided a **getopt** module that helps you parse command-line options and arguments. This module provides two functions and an exception to enable command line argument parsing.

**5.4.1.1 getopt.getopt method**

This method parses command line options and parameter list. Following is simple syntax for this method −

**getopt.getopt(args, options, [long\_options])**

Here is the detail of the parameters −

* + - * **args**: This is the argument list to be parsed.
      * **options**: This is the string of option letters that the script wants to recognize, with options that require an argument should be followed by a colon (:).
      * **long\_options**: This is optional parameter and if specified, must be a list of strings with the names of the long options, which should be supported. Long options, which require an argument should be followed by an equal sign ('='). To accept only long options, options should be an empty string.
      * This method returns value consisting of two elements: the first is a list of **(option, value)** pairs. The second is the list of program arguments left after the option list was stripped.
      * Each option-and-value pair returned has the option as its first element, prefixed with a hyphen for short options (e.g., '-x') or two hyphens for long options (e.g., '--long-option').

**5.4.2 EXCEPTION GETOPT.GETOPT ERROR**

This is raised when an unrecognized option is found in the argument list or when an option requiring an argument is given none.

The argument to the exception is a string indicating the cause of the error. The attributes **msg** and **opt** give the error message and related option

**EXAMPLE**

Consider we want to pass two file names through command line and we also want to give an option to check the usage of the script. Usage of the script is as follows −

**usage: test.py -i <inputfile> -o <outputfile>**

Here is the following script to test.py −

**#!/usr/bin/python**

**import sys, getopt**

**def main(argv):**

**inputfile = ''**

**outputfile = ''**

**try:**

**opts, args = getopt.getopt(argv,"hi:o:",["ifile=","ofile="])**

**except getopt.GetoptError:**

**print 'test.py -i <inputfile> -o <outputfile>'**

**sys.exit(2)**

**for opt, arg in opts:**

**if opt == '-h':**

**print 'test.py -i <inputfile> -o <outputfile>'**

**sys.exit()**

**elif opt in ("-i", "--ifile"):**

**inputfile = arg**

**elif opt in ("-o", "--ofile"):**

**outputfile = arg**

**print 'Input file is "', inputfile**

**print 'Output file is "', outputfile**

**if \_\_name\_\_ == "\_\_main\_\_":**

**main(sys.argv[1:])**

**Now, run above script as follows −**

**$ test.py -h**

**usage: test.py -i <inputfile> -o <outputfile>**

**$ test.py -i BMP -o**

**usage: test.py -i <inputfile> -o <outputfile>**

**$ test.py -i inputfile**

**Input file is " inputfile**

**Output file is "**

**5.5. ERRORS AND EXCEPTIONS**

**5.5.1. What is an Exception?**

An exception is an error that happens during execution of a program. When that error occurs, Python generate an exception that can be handled, which avoids your program to crash.

## 5.5.2. Why use Exceptions?

Exceptions are convenient in many ways for handling errors and special conditions in a program. When you think that you have a code which can produce an error there you can use exception handling.

## 5.5.3. Raising an Exception

You can raise an exception in your own program by using the raise exception statement. Raising an exception breaks current code execution and returns the exception back until it is handled.

## 5.5.4. Exception Errors

Below are some common exceptions errors in Python:

**IOError**

If the file cannot be opened.

**ImportError**

If python cannot find the module

**ValueError**

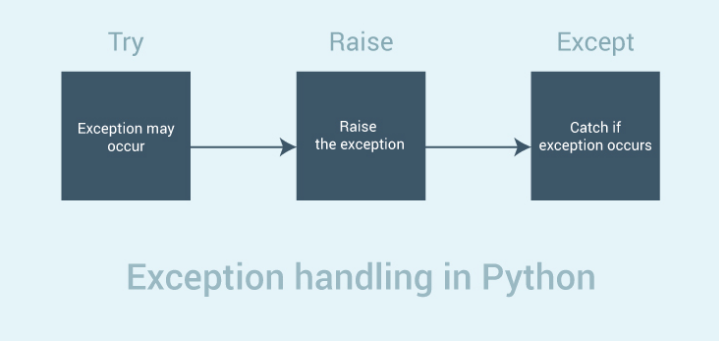
Raised when a built-in operation or function receives an argument that has the right type but an inappropriate value

**KeyboardInterrupt**

Raised when the user hits the interrupt key (normally Control-C or Delete)

**EOFError**

Raised when one of the built-in functions (input() or raw\_input()) hits an end-of-file condition (EOF) without reading any data



## 5.5.5. Exception Errors Examples

Now, when we know what some of the exception errors means, let's see some examples:

**except IOError:**

print('An error occurred trying to read the file.')

**except ValueError:**

print('Non-numeric data found in the file.')

**except ImportError:**

print "NO module found"

**except EOFError:**

print('Why did you do an EOF on me?')

**except KeyboardInterrupt:**

print('You cancelled the operation.')

**except:**

print('An error occurred.')

Try to use as few try blocks as possible and try to distinguish the failure conditions by the kinds of exceptions they throw.

**5.6. HANDLING EXCEPTIONS**

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

* **Exception Handling**
* **Assertions**

**5.6.1. Exception Handling**

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. |
| ZeroDivisionError | 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  KeyError | Raised when an index is not found in a sequence.  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  EnvironmentError | Raised when trying to access a local variable in a function or method but no value has been assigned to it.  Base class for all exceptions that occur outside the Python environment. |
| IOError  IOError | Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist.  Raised for operating system-related errors. |
| SyntaxError  IndentationError | Raised when there is an error in Python syntax.  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. |
| TypeError | 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. |

## 5.6.1.1. 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.

## 5.6.1.2. 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**

## 5.6.1.3. 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.

## 5.6.1.4. 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.**

## 5.6.1.5. The try-finally Clause

You can use a **finally:** block along with a **try:** block. The finally block is a place to put any code that must execute, whether the try-block raised an exception or not. The syntax of the try-finally statement is this –

**try:**

**You do your operations here;**

**......................**

**Due to any exception, this may be skipped.**

**finally:**

**This would always be executed.**

**......................**

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

### Example

**#!/usr/bin/python**

**try:**

**fh = open("testfile", "w")**

**fh.write("This is my test file for exception handling!!")**

**finally:**

**print "Error: can\'t find file or read data"**

**If you do not have permission to open the file in writing mode, then this will produce the following result:**

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

**Same example can be written more cleanly as follows −**

**#!/usr/bin/python**

**try:**

**fh = open("testfile", "w")**

**try:**

**fh.write("This is my test file for exception handling!!")**

**finally:**

**print "Going to close the file"**

**fh.close()**

**except IOError:**

**print "Error: can\'t find file or read data"**

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

## 5.6.1.6. 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'**

## 5.6.1.7. 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...**

## 5.6.1.8. 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**

### 5.6.2. 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.

### 5.6.2.1. 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!**

* 1. **MODULES**

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.

**Example**

The Python code for a module named *aname* normally resides in a file named *aname.py*. Here's an example of a simple module, support.py

**def print\_func( par ):**

**print "Hello : ", par**

**return**

**5.7.1. 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 support.py, you need to put the following command at the top of the script −

**#!/usr/bin/python**

**# Import module support**

**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.

* + 1. **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.

* + 1. **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.

* + 1. **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**

## 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.

**#!/usr/bin/python**

**Money = 2000**

**def AddMoney():**

**# Uncomment the following line to fix the code:**

**# global Money**

**Money = Money + 1**

**print Money**

**AddMoney()**

**print Money**

* + 1. **The dir( ) Function**

The dir() built-in function returns a sorted list of strings containing the names defined by a module.

The list contains the names of all the modules, variables and functions that are defined in a module. Following is a simple example −

**#!/usr/bin/python**

**# Import built-in module math**

**import math**

**content = dir(math)**

**print content**

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

**['\_\_doc\_\_', '\_\_file\_\_', '\_\_name\_\_', 'acos', 'asin', 'atan',**

**'atan2', 'ceil', 'cos', 'cosh', 'degrees', 'e', 'exp',**

**'fabs', 'floor', 'fmod', 'frexp', 'hypot', 'ldexp', 'log',**

**'log10', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh',**

**'sqrt', 'tan', 'tanh']**

Here, the special string variable *\_\_name\_\_* is the module's name, and *\_\_file\_\_* is the filename from which the module was loaded.

* + 1. **The *globals()* and *locals()* Functions −**
* The *globals()* and *locals()* functions can be used to return the names in the global and local namespaces depending on the location from where they are called.
* If locals() is called from within a function, it will return all the names that can be accessed locally from that function.
* If globals() is called from within a function, it will return all the names that can be accessed globally from that function.
* The return type of both these functions is dictionary. Therefore, names can be extracted using the keys() function.
  + 1. **The *reload()* Function**

When the module is imported into a script, the code in the top-level portion of a module is executed only once.

Therefore, if you want to reexecute the top-level code in a module, you can use the *reload()* function. The reload() function imports a previously imported module again. The syntax of the reload() function is this −

**reload(module\_name)**

Here, *module\_name* is the name of the module you want to reload and not the string containing the module name. For example, to reload *hello* module, do the following −

**reload(hello)**

* 1. **PACKAGES IN PYTHON**

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 −

**#!/usr/bin/python**

**def Pots():**

**print "I'm Pots Phone"**

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 follows −

**from Pots import Pots**

**from Isdn import Isdn**

**from G3 import G3**

After you add these lines to \_\_init\_\_.py, you have all of these classes available when you import the Phone package.

**#!/usr/bin/python**

**# Now import your Phone Package.**

**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.