**Model Exam Questions with answers (UNIT – V)**

**Two marks**

1. **Define file.**

A file refers to a location with filename that stores information. The storage area is non- volatile memory like hard-disk. A file stores related data, information, settings, or commands in secondary storage device like magnetic disks, magnetic tapes and optical disks. A file can be a sequence of bits, bytes, lines or records depending of the application/software used to create it. For example a text file is organized as a sequence of lines.

1. **List the file opening modes**

|  |  |
| --- | --- |
| Modes | Description |
| r  r+  w  w+  a  a+ | Opens a file for reading only.  Opens a file for both reading and writing.  Opens a file for writing only. Overwrites the file if the file exists  Opens a file for both writing and reading.  Opens a file for appending. File pointer is at the end of the file  Opens a file for both appending and reading. |

1. **List the different ways to read a file**

The text files can be read in four different ways listed below

* Using Read method
* Using Readlines method
* Using for line In file method
* Using readline method

1. **What is the difference between append and write mode?**

The write pointer is set to end of file when a file is opened in “a” mode. In append mode the file can never be read. The contents can only be written at the end of the file. In the write mode the write pointer is set to the beginning of the file.

1. **What are the attributes in file objects**

|  |  |
| --- | --- |
| Atrribute | Description |
| File.closed  File.mode  File.name  File.softspace | If the file is closed returns true else false  Returns one of the mode  Returns name of the file  Returns false if space explicitly required with print, true otherwise |

**PART B(13 MARKS)**

1. **Discuss in detail about text 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+")**

**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.

**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.

**APPEND**

To add data to an existing file use the command

**SYNTAX:**

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

**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

**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**

**READING AND WRITING FILES**

**READING FILES**

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

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

**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()**

**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 ()**

**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 ()**

**WRITING FILES**

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

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

**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()**

**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 ()**

**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

**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,**

|  |  |
| --- | --- |
| 1. **Explain in detail about errors and exception.** |  |

**ERRORS AND EXCEPTIONS**

**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.

## 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.

## 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.

## 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**

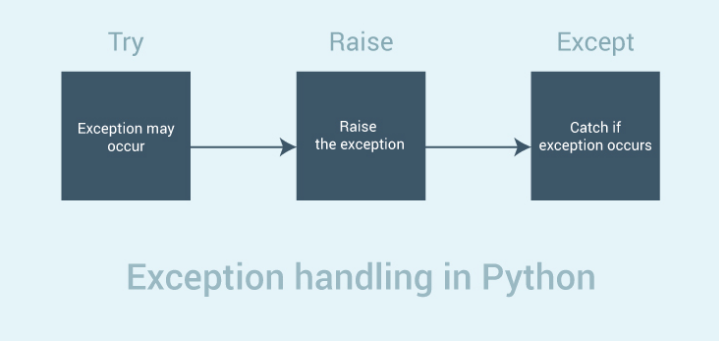
Raised when a built-in operation or function receives an argument that has the right type but an inappropriate value

**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



## 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.

**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**

**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. |

## What is Exception?

An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions. In general, when a Python script encounters a situation that it cannot cope with, it raises an exception. An exception is a Python object that represents an error.

When a Python script raises an exception, it must either handle the exception immediately otherwise it terminates and quits.

## Handling an exception

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

### Syntax

Here is simple syntax of *try....except...else* blocks −

**try:**

**You do your operations here;**

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

**except ExceptionI:**

**If there is ExceptionI, then execute this block.**

**except ExceptionII:**

**If there is ExceptionII, then execute this block.**

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

**else:**

**If there is no exception then execute this block.**

Here are few important points about the above-mentioned syntax −

* A single try statement can have multiple except statements. This is useful when the try block contains statements that may throw different types of exceptions.
* You can also provide a generic except clause, which handles any exception.
* After the except clause(s), you can include an else-clause. The code in the else-block executes if the code in the try: block does not raise an exception.
* The else-block is a good place for code that does not need the try: block's protection.

### Example

This example opens a file, writes content in the, file and comes out gracefully because there is no problem at all −

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

**try:**

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

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

**except IOError:**

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

**else:**

**print "Written content in the file successfully"**

**fh.close()**

This produces the following result −

**Written content in the file successfully**

### Example

This example tries to open a file where you do not have write permission, so it raises an exception −

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

**try:**

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

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

**except IOError:**

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

**else:**

**print "Written content in the file successfully"**

This produces the following result −

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

## The *except* Clause with No Exceptions

You can also use the except statement with no exceptions defined as follows –

**try:**

**You do your operations here;**

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

**except:**

**If there is any exception, then execute this block.**

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

**else:**

**If there is no exception then execute this block.**

This kind of a **try-except** statement catches all the exceptions that occur. Using this kind of try-except statement is not considered a good programming practice though, because it catches all exceptions but does not make the programmer identify the root cause of the problem that may occur.

## The *except* Clause with Multiple Exceptions

You can also use the same *except* statement to handle multiple exceptions as follows −

**try:**

**You do your operations here;**

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

**except(Exception1[, Exception2[,...ExceptionN]]]):**

**If there is any exception from the given exception list,**

**then execute this block.**

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

**else:**

**If there is no exception then execute this block.**

## The try-finally Clause

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

**try:**

**You do your operations here;**

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

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

**finally:**

**This would always be executed.**

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

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

### Example

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

**try:**

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

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

**finally:**

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

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

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

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

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

**try:**

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

**try:**

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

**finally:**

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

**fh.close()**

**except IOError:**

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

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

## Argument of an Exception

An exception can have an *argument*, which is a value that gives additional information about the problem. The contents of the argument vary by exception. You capture an exception's argument by supplying a variable in the except clause as follows −

**try:**

**You do your operations here;**

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

**except ExceptionType, Argument:**

**You can print value of Argument here...**

If you write the code to handle a single exception, you can have a variable follow the name of the exception in the except statement. If you are trapping multiple exceptions, you can have a variable follow the tuple of the exception.

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

### Example

Following is an example for a single exception −

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

**# Define a function here.**

**def temp\_convert(var):**

**try:**

**return int(var)**

**except ValueError, Argument:**

**print "The argument does not contain numbers\n", Argument**

**# Call above function here.**

**temp\_convert("xyz");**

This produces the following result −

**The argument does not contain numbers**

**invalid literal for int() with base 10: 'xyz'**

## Raising an Exceptions

You can raise exceptions in several ways by using the raise statement. The general syntax for the **raise** statement is as follows.

### Syntax

**raise [Exception [, args [, traceback]]]**

Here, *Exception* is the type of exception (for example, NameError) and *argument* is a value for the exception argument. The argument is optional; if not supplied, the exception argument is None.

The final argument, traceback, is also optional (and rarely used in practice), and if present, is the traceback object used for the exception.

### Example

An exception can be a string, a class or an object. Most of the exceptions that the Python core raises are classes, with an argument that is an instance of the class. Defining new exceptions is quite easy and can be done as follows −

**def functionName( level ):**

**if level < 1:**

**raise "Invalid level!", level**

**# The code below to this would not be executed**

**# if we raise the exception**

**Note:** In order to catch an exception, an "except" clause must refer to the same exception thrown either class object or simple string. For example, to capture above exception, we must write the except clause as follows −

**try:**

**Business Logic here...**

**except "Invalid level!":**

**Exception handling here...**

**else:**

**Rest of the code here...**

## User-Defined Exceptions

Python also allows you to create your own exceptions by deriving classes from the standard built-in exceptions.

Here is an example related to *RuntimeError*. Here, a class is created that is subclassed from *RuntimeError*. This is useful when you need to display more specific information when an exception is caught.

In the try block, the user-defined exception is raised and caught in the except block. The variable e is used to create an instance of the class *Networkerror*.

**class Networkerror(RuntimeError):**

**def \_\_init\_\_(self, arg):**

**self.args = arg**

So once you defined above class, you can raise the exception as follows −

**try:**

**raise Networkerror("Bad hostname")**

**except Networkerror,e:**

**print e.args**

### Assertions in Python

An assertion is a sanity-check that you can turn on or turn off when you are done with your testing of the program.

The easiest way to think of an assertion is to liken it to a **raise-if** statement (or to be more accurate, a raise-if-not statement). An expression is tested, and if the result comes up false, an exception is raised.

Assertions are carried out by the assert statement, the newest keyword to Python, introduced in version 1.5.

Programmers often place assertions at the start of a function to check for valid input, and after a function call to check for valid output.

### The *assert* Statement

When it encounters an assert statement, Python evaluates the accompanying expression, which is hopefully true. If the expression is false, Python raises an *AssertionError* exception.

The syntax for assert is −

**assert Expression[, Arguments]**

If the assertion fails, Python uses ArgumentExpression as the argument for the AssertionError. AssertionError exceptions can be caught and handled like any other exception using the try-except statement, but if not handled, they will terminate the program and produce a traceback.

### Example

Here is a function that converts a temperature from degrees Kelvin to degrees Fahrenheit. Since zero degrees Kelvin is as cold as it gets, the function bails out if it sees a negative temperature −

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

**def KelvinToFahrenheit(Temperature):**

**assert (Temperature >= 0),"Colder than absolute zero!"**

**return ((Temperature-273)\*1.8)+32**

**print KelvinToFahrenheit(273)**

**print int(KelvinToFahrenheit(505.78))**

**print KelvinToFahrenheit(-5)**

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

**32.0**

**451**

**Traceback (most recent call last):**

**File "test.py", line 9, in**

**print KelvinToFahrenheit(-5)**

**File "test.py", line 4, in KelvinToFahrenheit**

|  |  |
| --- | --- |
| **3.Write in detail about modules and packages with examples.** |  |

A module is a file containing Python definitions and statements. The file name is the module name with the suffix .py appended. Within a module, the module’s name (as a string) is available as the value of the global variable \_\_name\_\_. For instance, use your favorite text editor to create a file called fibo.py in the current directory with the following contents:

*# Fibonacci numbers module*

**def** fib(n): *# write Fibonacci series up to n*

a, b = 0, 1

**while** b < n:

print b,

a, b = b, a+b

**def** fib2(n): *# return Fibonacci series up to n*

result = []

a, b = 0, 1

**while** b < n:

result.append(b)

a, b = b, a+b

**return** result

Now enter the Python interpreter and import this module with the following command:

**>>> import** **fibo**

This does not enter the names of the functions defined in fibo directly in the current symbol table; it only enters the module name fibo there. Using the module name you can access the functions:

**>>>** fibo.fib(1000)

1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987

**>>>** fibo.fib2(100)

[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]

**>>>** fibo.\_\_name\_\_

'fibo'

If you intend to use a function often you can assign it to a local name:

**>>>** fib = fibo.fib

**>>>** fib(500)

1 1 2 3 5 8 13 21 34 55 89 144 233 377

More on Modules

A module can contain executable statements as well as function definitions. These statements are intended to initialize the module. They are executed only the *first* time the module name is encountered in an import statement. [[1]](https://docs.python.org/2/tutorial/modules.html#id2) (They are also run if the file is executed as a script.)

Each module has its own private symbol table, which is used as the global symbol table by all functions defined in the module. Thus, the author of a module can use global variables in the module without worrying about accidental clashes with a user’s global variables. On the other hand, if you know what you are doing you can touch a module’s global variables with the same notation used to refer to its functions, modname.itemname.

Modules can import other modules. It is customary but not required to place all [**import**](https://docs.python.org/2/reference/simple_stmts.html#import) statements at the beginning of a module (or script, for that matter). The imported module names are placed in the importing module’s global symbol table.

There is a variant of the [**import**](https://docs.python.org/2/reference/simple_stmts.html#import) statement that imports names from a module directly into the importing module’s symbol table. For example:

**>>> from** **fibo** **import** fib, fib2

**>>>** fib(500)

1 1 2 3 5 8 13 21 34 55 89 144 233 377

This does not introduce the module name from which the imports are taken in the local symbol table (so in the example, fibo is not defined).

There is even a variant to import all names that a module defines:

**>>> from** **fibo** **import** \*

**>>>** fib(500)

1 1 2 3 5 8 13 21 34 55 89 144 233 377

This imports all names except those beginning with an underscore (\_).

Note that in general the practice of importing \* from a module or package is frowned upon, since it often causes poorly readable code. However, it is okay to use it to save typing in interactive sessions.

**Note**

For efficiency reasons, each module is only imported once per interpreter session. Therefore, if you change your modules, you must restart the interpreter – or, if it’s just one module you want to test interactively, use [**reload()**](https://docs.python.org/2/library/functions.html#reload), e.g. reload(modulename).

Executing modules as scripts

When you run a Python module with

python fibo.py <arguments>

the code in the module will be executed, just as if you imported it, but with the \_\_name\_\_ set to "\_\_main\_\_". That means that by adding this code at the end of your module:

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

**import** **sys**

fib(int(sys.argv[1]))

you can make the file usable as a script as well as an importable module, because the code that parses the command line only runs if the module is executed as the “main” file:

**$** python fibo.py 50

1 1 2 3 5 8 13 21 34

If the module is imported, the code is not run:

**>>> import** **fibo**

>>>

This is often used either to provide a convenient user interface to a module, or for testing purposes (running the module as a script executes a test suite).

The Module Search Path

When a module named **spam** is imported, the interpreter first searches for a built-in module with that name. If not found, it then searches for a file named spam.py in a list of directories given by the variable **[sys.path](https://docs.python.org/2/library/sys.html" \l "sys.path" \o "sys.path)**. **[sys.path](https://docs.python.org/2/library/sys.html" \l "sys.path" \o "sys.path)** is initialized from these locations:

* the directory containing the input script (or the current directory).
* [**PYTHONPATH**](https://docs.python.org/2/using/cmdline.html#envvar-PYTHONPATH) (a list of directory names, with the same syntax as the shell variable **PATH**).
* the installation-dependent default.

After initialization, Python programs can modify **[sys.path](https://docs.python.org/2/library/sys.html" \l "sys.path" \o "sys.path)**. The directory containing the script being run is placed at the beginning of the search path, ahead of the standard library path. This means that scripts in that directory will be loaded instead of modules of the same name in the library directory. This is an error unless the replacement is intended. See section [Standard Modules](https://docs.python.org/2/tutorial/modules.html#tut-standardmodules) for more information.

“Compiled” Python files

As an important speed-up of the start-up time for short programs that use a lot of standard modules, if a file called spam.pyc exists in the directory where spam.py is found, this is assumed to contain an already-“byte-compiled” version of the module **spam**. The modification time of the version of spam.py used to create spam.pyc is recorded in spam.pyc, and the .pyc file is ignored if these don’t match.

Normally, you don’t need to do anything to create the spam.pyc file. Whenever spam.py is successfully compiled, an attempt is made to write the compiled version to spam.pyc. It is not an error if this attempt fails; if for any reason the file is not written completely, the resulting spam.pyc file will be recognized as invalid and thus ignored later. The contents of the spam.pyc file are platform independent, so a Python module directory can be shared by machines of different architectures.

Some tips for experts:

* When the Python interpreter is invoked with the [**-O**](https://docs.python.org/2/using/cmdline.html#cmdoption-o) flag, optimized code is generated and stored in .pyo files. The optimizer currently doesn’t help much; it only removes [**assert**](https://docs.python.org/2/reference/simple_stmts.html#assert) statements. When [**-O**](https://docs.python.org/2/using/cmdline.html#cmdoption-o) is used, *all* [bytecode](https://docs.python.org/2/glossary.html" \l "term-bytecode) is optimized; .pyc files are ignored and .py files are compiled to optimized bytecode.
* Passing two [**-O**](https://docs.python.org/2/using/cmdline.html#cmdoption-o) flags to the Python interpreter ([**-OO**](https://docs.python.org/2/using/cmdline.html#cmdoption-oo)) will cause the bytecode compiler to perform optimizations that could in some rare cases result in malfunctioning programs. Currently only \_\_doc\_\_ strings are removed from the bytecode, resulting in more compact .pyo files. Since some programs may rely on having these available, you should only use this option if you know what you’re doing.
* A program doesn’t run any faster when it is read from a .pyc or .pyo file than when it is read from a .py file; the only thing that’s faster about .pyc or .pyo files is the speed with which they are loaded.
* When a script is run by giving its name on the command line, the bytecode for the script is never written to a .pyc or .pyo file. Thus, the startup time of a script may be reduced by moving most of its code to a module and having a small bootstrap script that imports that module. It is also possible to name a .pyc or .pyo file directly on the command line.
* It is possible to have a file called spam.pyc (or spam.pyo when [**-O**](https://docs.python.org/2/using/cmdline.html#cmdoption-o) is used) without a file spam.py for the same module. This can be used to distribute a library of Python code in a form that is moderately hard to reverse engineer.
* The module **[compileall](https://docs.python.org/2/library/compileall.html" \l "module-compileall" \o "compileall: Tools for byte-compiling all Python source files in a directory tree.)** can create .pyc files (or .pyo files when [**-O**](https://docs.python.org/2/using/cmdline.html#cmdoption-o) is used) for all modules in a directory.

Standard Modules

Python comes with a library of standard modules, described in a separate document, the Python Library Reference (“Library Reference” hereafter). Some modules are built into the interpreter; these provide access to operations that are not part of the core of the language but are nevertheless built in, either for efficiency or to provide access to operating system primitives such as system calls. The set of such modules is a configuration option which also depends on the underlying platform. For example, the **winreg** module is only provided on Windows systems. One particular module deserves some attention: [**sys**](https://docs.python.org/2/library/sys.html#module-sys), which is built into every Python interpreter. The variables sys.ps1 and sys.ps2 define the strings used as primary and secondary prompts:

**>>> import** **sys**

**>>>** sys.ps1

'>>> '

**>>>** sys.ps2

'... '

**>>>** sys.ps1 = 'C> '

C> print 'Yuck!'

Yuck!

C>

These two variables are only defined if the interpreter is in interactive mode.

The variable sys.path is a list of strings that determines the interpreter’s search path for modules. It is initialized to a default path taken from the environment variable [**PYTHONPATH**](https://docs.python.org/2/using/cmdline.html#envvar-PYTHONPATH), or from a built-in default if [**PYTHONPATH**](https://docs.python.org/2/using/cmdline.html#envvar-PYTHONPATH) is not set. You can modify it using standard list operations:

**>>> import** **sys**

**>>>** sys.path.append('/ufs/guido/lib/python')

The [**dir()**](https://docs.python.org/2/library/functions.html#dir) Function

The built-in function [**dir()**](https://docs.python.org/2/library/functions.html#dir) is used to find out which names a module defines. It returns a sorted list of strings:

**>>> import** **fibo**, **sys**

**>>>** dir(fibo)

['\_\_name\_\_', 'fib', 'fib2']

**>>>** dir(sys)

['\_\_displayhook\_\_', '\_\_doc\_\_', '\_\_excepthook\_\_', '\_\_name\_\_', '\_\_package\_\_',

'\_\_stderr\_\_', '\_\_stdin\_\_', '\_\_stdout\_\_', '\_clear\_type\_cache',

'\_current\_frames', '\_getframe', '\_mercurial', 'api\_version', 'argv',

'builtin\_module\_names', 'byteorder', 'call\_tracing', 'callstats',

'copyright', 'displayhook', 'dont\_write\_bytecode', 'exc\_clear', 'exc\_info',

'exc\_traceback', 'exc\_type', 'exc\_value', 'excepthook', 'exec\_prefix',

'executable', 'exit', 'flags', 'float\_info', 'float\_repr\_style',

'getcheckinterval', 'getdefaultencoding', 'getdlopenflags',

'getfilesystemencoding', 'getobjects', 'getprofile', 'getrecursionlimit',

'getrefcount', 'getsizeof', 'gettotalrefcount', 'gettrace', 'hexversion',

'long\_info', 'maxint', 'maxsize', 'maxunicode', 'meta\_path', 'modules',

'path', 'path\_hooks', 'path\_importer\_cache', 'platform', 'prefix', 'ps1',

'py3kwarning', 'setcheckinterval', 'setdlopenflags', 'setprofile',

'setrecursionlimit', 'settrace', 'stderr', 'stdin', 'stdout', 'subversion',

'version', 'version\_info', 'warnoptions']

Without arguments, [**dir()**](https://docs.python.org/2/library/functions.html#dir) lists the names you have defined currently:

**>>>** a = [1, 2, 3, 4, 5]

**>>> import** **fibo**

**>>>** fib = fibo.fib

**>>>** dir()

['\_\_builtins\_\_', '\_\_name\_\_', '\_\_package\_\_', 'a', 'fib', 'fibo', 'sys']

Note that it lists all types of names: variables, modules, functions, etc.

[**dir()**](https://docs.python.org/2/library/functions.html#dir) does not list the names of built-in functions and variables. If you want a list of those, they are defined in the standard module [**\_\_builtin\_\_**](https://docs.python.org/2/library/__builtin__.html#module-__builtin__):

**>>> import** **\_\_builtin\_\_**

**>>>** dir(\_\_builtin\_\_)

['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',

'BufferError', 'BytesWarning', 'DeprecationWarning', 'EOFError',

'Ellipsis', 'EnvironmentError', 'Exception', 'False', 'FloatingPointError',

'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning',

'IndentationError', 'IndexError', 'KeyError', 'KeyboardInterrupt',

'LookupError', 'MemoryError', 'NameError', 'None', 'NotImplemented',

'NotImplementedError', 'OSError', 'OverflowError',

'PendingDeprecationWarning', 'ReferenceError', 'RuntimeError',

'RuntimeWarning', 'StandardError', 'StopIteration', 'SyntaxError',

'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'True',

'TypeError', 'UnboundLocalError', 'UnicodeDecodeError',

'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError',

'UnicodeWarning', 'UserWarning', 'ValueError', 'Warning',

'ZeroDivisionError', '\_', '\_\_debug\_\_', '\_\_doc\_\_', '\_\_import\_\_',

'\_\_name\_\_', '\_\_package\_\_', 'abs', 'all', 'any', 'apply', 'basestring',

'bin', 'bool', 'buffer', 'bytearray', 'bytes', 'callable', 'chr',

'classmethod', 'cmp', 'coerce', 'compile', 'complex', 'copyright',

'credits', 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval',

'execfile', 'exit', 'file', 'filter', 'float', 'format', 'frozenset',

'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input',

'int', 'intern', 'isinstance', 'issubclass', 'iter', 'len', 'license',

'list', 'locals', 'long', 'map', 'max', 'memoryview', 'min', 'next',

'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'quit',

'range', 'raw\_input', 'reduce', 'reload', 'repr', 'reversed', 'round',

'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super',

'tuple', 'type', 'unichr', 'unicode', 'vars', 'xrange', 'zip']

**Packages**

Packages are a way of structuring Python’s module namespace by using “dotted module names”. For example, the module name **A.B** designates a submodule named B in a package named A. Just like the use of modules saves the authors of different modules from having to worry about each other’s global variable names, the use of dotted module names saves the authors of multi-module packages like NumPy or the Python Imaging Library from having to worry about each other’s module names.

Suppose you want to design a collection of modules (a “package”) for the uniform handling of sound files and sound data. There are many different sound file formats (usually recognized by their extension, for example: .wav, .aiff, .au), so you may need to create and maintain a growing collection of modules for the conversion between the various file formats. There are also many different operations you might want to perform on sound data (such as mixing, adding echo, applying an equalizer function, creating an artificial stereo effect), so in addition you will be writing a never-ending stream of modules to perform these operations. Here’s a possible structure for your package (expressed in terms of a hierarchical filesystem):

sound/ Top-level package

\_\_init\_\_.py Initialize the sound package

formats/ Subpackage for file format conversions

\_\_init\_\_.py

wavread.py

wavwrite.py

aiffread.py

aiffwrite.py

auread.py

auwrite.py

...

effects/ Subpackage for sound effects

\_\_init\_\_.py

echo.py

surround.py

reverse.py

...

filters/ Subpackage for filters

\_\_init\_\_.py

equalizer.py

vocoder.py

karaoke.py

...

When importing the package, Python searches through the directories on sys.path looking for the package subdirectory.

The \_\_init\_\_.py files are required to make Python treat the directories as containing packages; this is done to prevent directories with a common name, such as string, from unintentionally hiding valid modules that occur later on the module search path. In the simplest case, \_\_init\_\_.py can just be an empty file, but it can also execute initialization code for the package or set the \_\_all\_\_ variable, described later.

Users of the package can import individual modules from the package, for example:

**import** **sound.effects.echo**

This loads the submodule **sound.effects.echo**. It must be referenced with its full name.

sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)

An alternative way of importing the submodule is:

**from** **sound.effects** **import** echo

This also loads the submodule **echo**, and makes it available without its package prefix, so it can be used as follows:

echo.echofilter(input, output, delay=0.7, atten=4)

Yet another variation is to import the desired function or variable directly:

**from** **sound.effects.echo** **import** echofilter

Again, this loads the submodule **echo**, but this makes its function **echofilter()** directly available:

echofilter(input, output, delay=0.7, atten=4)

Note that when using from package import item, the item can be either a submodule (or subpackage) of the package, or some other name defined in the package, like a function, class or variable. The import statement first tests whether the item is defined in the package; if not, it assumes it is a module and attempts to load it. If it fails to find it, an **[ImportError](https://docs.python.org/2/library/exceptions.html" \l "exceptions.ImportError" \o "exceptions.ImportError)** exception is raised.

Contrarily, when using syntax like import item.subitem.subsubitem, each item except for the last must be a package; the last item can be a module or a package but can’t be a class or function or variable defined in the previous item.

Importing \* From a Package

Now what happens when the user writes from sound.effects import \*? Ideally, one would hope that this somehow goes out to the filesystem, finds which submodules are present in the package, and imports them all. This could take a long time and importing sub-modules might have unwanted side-effects that should only happen when the sub-module is explicitly imported.

The only solution is for the package author to provide an explicit index of the package. The [**import**](https://docs.python.org/2/reference/simple_stmts.html#import) statement uses the following convention: if a package’s \_\_init\_\_.py code defines a list named \_\_all\_\_, it is taken to be the list of module names that should be imported when from package import \* is encountered. It is up to the package author to keep this list up-to-date when a new version of the package is released. Package authors may also decide not to support it, if they don’t see a use for importing \* from their package. For example, the file sound/effects/\_\_init\_\_.py could contain the following code:

\_\_all\_\_ = ["echo", "surround", "reverse"]

This would mean that from sound.effects import \* would import the three named submodules of the **sound** package.

If \_\_all\_\_ is not defined, the statement from sound.effects import \* does *not* import all submodules from the package **sound.effects** into the current namespace; it only ensures that the package **sound.effects** has been imported (possibly running any initialization code in \_\_init\_\_.py) and then imports whatever names are defined in the package. This includes any names defined (and submodules explicitly loaded) by \_\_init\_\_.py. It also includes any submodules of the package that were explicitly loaded by previous [**import**](https://docs.python.org/2/reference/simple_stmts.html#import) statements. Consider this code:

**import** **sound.effects.echo**

**import** **sound.effects.surround**

**from** **sound.effects** **import** \*

In this example, the **echo** and **surround** modules are imported in the current namespace because they are defined in the **sound.effects** package when the from...import statement is executed. (This also works when \_\_all\_\_ is defined.)

Although certain modules are designed to export only names that follow certain patterns when you use import \*, it is still considered bad practice in production code.

Remember, there is nothing wrong with using from Package import specific\_submodule! In fact, this is the recommended notation unless the importing module needs to use submodules with the same name from different packages.

Intra-package References

The submodules often need to refer to each other. For example, the **surround** module might use the **echo** module. In fact, such references are so common that the [**import**](https://docs.python.org/2/reference/simple_stmts.html#import) statement first looks in the containing package before looking in the standard module search path. Thus, the **surround** module can simply use import echo or from echo import echofilter. If the imported module is not found in the current package (the package of which the current module is a submodule), the [**import**](https://docs.python.org/2/reference/simple_stmts.html#import) statement looks for a top-level module with the given name.

When packages are structured into subpackages (as with the **sound** package in the example), you can use absolute imports to refer to submodules of siblings packages. For example, if the module **sound.filters.vocoder** needs to use the **echo** module in the **sound.effects** package, it can use fromsound.effects import echo.

Starting with Python 2.5, in addition to the implicit relative imports described above, you can write explicit relative imports with the from module import nameform of import statement. These explicit relative imports use leading dots to indicate the current and parent packages involved in the relative import. From the **surround** module for example, you might use:

**from** **.** **import** echo

**from** **..** **import** formats

**from** **..filters** **import** equalizer

Note that both explicit and implicit relative imports are based on the name of the current module. Since the name of the main module is always "\_\_main\_\_", modules intended for use as the main module of a Python application should always use absolute imports.

Packages in Multiple Directories

Packages support one more special attribute, **\_\_path\_\_**. This is initialized to be a list containing the name of the directory holding the package’s \_\_init\_\_.py before the code in that file is executed. This variable can be modified; doing so affects future searches for modules and subpackages contained in the package.

While this feature is not often needed, it can be used to extend the set of modules found in a package.

|  |  |
| --- | --- |
| **4.Write in detail about modules and packages with examples.** |  |

**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**

**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 "**