Module 4
Organizing Files

Q) Explain the following file operations in Python with suitable example.
i) Copying files and folders
ii) Moving files and folders
iii) Permanently deleting files and folders

Q) List out the difference between shutil.copy() and shutil.copytree() method.

i) Copying files and folders

The shutil module provides functions for copying files, as well as entire folders.

Calling shutil.copy(source, destination) will copy the file at the path source to the folder at the path destination (Both source and destination are strings.). If destination is a filename, it will be used as the new name of the copied file. This function returns a string of the path of the copied file.

Example:

```
>>> import shutil, os
>>> os.chdir('C:\\')
>>> shutil.copy('C:\\spam.txt', 'C:\\delicious')
'C:\\delicious\\spam.txt'
>>> shutil.copy('eggs.txt', 'C:\\delicious\\eggs2.txt')
'C:\\delicious\\eggs2.txt'
```

shutil.copytree() will copy an entire folder and every folder and file contained in it.

Calling shutil.copytree(source,destination) will copy the folder at the path source, along with all of its files and subfolders, to the folder at the path destination. The source and destination parameters are both strings. The function returns a string of the path of the copied folder.

Example:

```
>>> import shutil, os
>>> os.chdir('C:\\')
>>> shutil.copytree('C:\\bacon', 'C:\\bacon_backup')
'C:\\bacon_backup’
```

ii) Moving files and folders

Calling shutil.move(source, destination) will move the file or folder at the path source to the path destination and will return a string of the absolute path of the new location. If destination points to a folder, the source file gets moved into destination and keeps its current filename.

Example:

```
>>> import shutil
>>> shutil.move('C:\\bacon.txt', 'C:\\eggs')
'C:\\eggs\\bacon.txt'
```

Assuming a folder named eggs already exists in the C:\ directory, shutil.move() calls says, “Move C:\bacon.txt into the folder C:\eggs.” If there had been a bacon.txt file already in C:\eggs, it would have been overwritten. Since it’s easy to accidentally overwrite files in this way, we should take some care when using move().

The destination path can also specify a filename. The following example, the source file is moved and renamed.

```
>>> shutil.move('C:\\bacon.txt', 'C:\\eggs\\new_bacon.txt')
'C:\\eggs\\new_bacon.txt'
```

This line says, “Move C:\bacon.txt into the folder C:\eggs, and rename that bacon.txt file to new_bacon.txt.”

Both of the previous examples worked under the assumption that there was a folder eggs in the C:\ directory. But if there is no eggs folder, then move() will rename bacon.txt to a file named eggs.

```
>>> shutil.move('C:\\bacon.txt', 'C:\\eggs')
'C:\\eggs'
```

iii) Permanently deleting files and folders

We can delete a single file or a single empty folder with functions in the os module, whereas to delete a folder and all of its contents, we use the shutil module.

Calling os.unlink(path) will delete the file at path.

Calling os.rmdir(path) will delete the folder at path. This folder must be empty of any files or folders.

Calling shutil.rmtree(path) will remove the folder at path, and all files and folders it contains will also be deleted.

Example:

```
import os
for filename in os.listdir():
        if filename.endswith('.rxt'):
                os.unlink(filename)
```

Q) List out the benefits of compressing file? Also explain reading of a Zip file with an example.

Data compression is a reduction in the number of bits needed to represent data. Compressing data can save storage capacity, speed up file transfer and decrease costs for storage hardware and network bandwidth.

There are many benefits to using file compression.

Increased computing efficiency: Compressed data permits users to back-up and store data faster, especially when dealing with larger files. *Note:* The ***advantage of compressing digital video*** is becoming more useful as video sales letters (VSLs) and personalized videos become more prevalent.

Quicker transfers: Not only does file compression enable you to move files around on a local device more efficiently, it also enables you to send large documents and data faster over the internet.
Improved file integrity: Uncompressed files can often become corrupt when sent over the web. Zipped files serve to preserve the integrity of your files and make sure your data goes uncorrupted.
Email/webpage accessibility: It's easier to compress larger files when uploading them to a webpage or sending them via email. Also, as previously mentioned, the most common email systems restrict the size of attachments. So, compression offers a way to send multiple files collectively, instead of one by one.

Reading ZIP Files

To read the contents of a ZIP file, first we must create a ZipFile object (note the capital letters Z and F). ZipFile objects are conceptually similar to the File objects returned by the open() function.
To create a ZipFile object, call the zipfile.ZipFile() function, passing it a string of the .zip file's filename.

Note: zipfile is the name of the Python module, and ZipFile() is the name of the function.
Example:

```
>>> import zipfile, os
>>> os.chdir('C:\\') # move to the folder with example.zip
>>> exampleZip = zipfile.ZipFile('example.zip')
>>> exampleZip.namelist()
['spam.txt', 'cats/', 'cats/catnames.txt', 'cats/zophie.jpg']
>>> spamInfo = exampleZip.getinfo('spam.txt')
>>> spamInfo.file_size
13908
>>> spamInfo.compress_size
3828
>>> 'Compressed file is %sx smaller!' % (round (spamInfo.file_size / spamInfo
.compress_size, 2))
'Compressed file is 3.63x smaller!'
>>> exampleZip.close()
```

Q) Differentiate between extract() and extractall() method with an example.

The extract() method for ZipFile objects will extract a single file from the ZIP file.

```
>>> exampleZip.extract('spam.txt')
'C:\\spam.txt'
>>> exampleZip.extract('spam.txt', 'C:\\some\\new\\folders')
'C:\\some\\new\\folders\\spam.txt'
>>> exampleZip.close()
```

The extractall() method for ZipFile objects extracts all the files and folders from a ZIP file into the current working directory.

```
>>> import zipfile, os
>>> os.chdir('C:\\') # move to the folder with example.zip
>>> exampleZip = zipfile.ZipFile('example.zip')
>>> exampleZip.extractall()
>>> exampleZip.close()
```

Q) Briefly explain assertions and raising exception.

An assertion is a sanity check to make sure our code isn’t doing something obviously wrong. These sanity checks are performed by assert statements. If the sanity check fails, then an AssertionError exception is raised.

In code, an assert statement consists of the following:

- The assert keyword
- A condition (that is, an expression that evaluates to True or False)
- A comma

A string to display when the condition is False

Example:

```
>>> podBayDoorStatus = 'open'
>>> assert podBayDoorStatus == 'open', 'The pod bay doors need to be "open".'
>>> podBayDoorStatus = 'I\'m sorry, Dave. I\'m afraid I can't do that.''
>>> assert podBayDoorStatus == 'open', 'The pod bay doors need to be "open".'
Traceback (most recent call last):
        File "<pyshell#10>", line 1, in <module>
                assert podBayDoorStatus == 'open', 'The pod bay doors need to be "open".'
AssertionError: The pod bay doors need to be "open".
```

Python raises an exception whenever it tries to execute invalid code. We can also raise our own exceptions in our code. Raising an exception is a way of saying, “Stop running the code in this function and move the program execution to the except statement.” Exceptions are raised with a raise statement.

In code, a raise statement consists of the following:

- The raise keyword
- A call to the Exception() function

A string with a helpful error message passed to the Exception() function

Example:

```
def boxPrint(symbol, width, height):
        if len(symbol) != 1:
                raise Exception('Symbol must be a single character string.')
        if width <= 2:
                raise Exception('Width must be greater than 2.')
        if height <= 2:
```

```
        raise Exception('Height must be greater than 2.')
    print(symbol * width)
    for i in range(height - 2):
        print(symbol + (' ' * (width - 2)) + symbol)
    print(symbol * width)
    for sym, w, h in (('*', 4, 4), ('O', 20, 5), ('x', 1, 3), ('ZZ', 3, 3)):
        try:
            boxPrint(sym, w, h)
        except Exception as err:
            print('An exception happened: ' + str(err))
```

Q) List out the benefits of using logging module with an example.

- If we have ever put a print() statement in our code to output some variable's value while our program is running, we have used a form of *logging* to debug our code.
- Logging is a great way to understand what's happening in our program and in what order its happening.
- Python's logging module makes it easy to create a record of custom messages that we write.
- These log messages will describe when the program execution has reached the logging function call and list any variables we have specified at that point in time.
- On the other hand, a missing log message indicates a part of the code was skipped and never executed.
- To enable the logging module to display log messages on our screen as our program runs, copy the following to the top of our program (but under the #! python shebang line):

```
import logging
logging.basicConfig(level=logging.DEBUG, format=' %(asctime)s - %(levelname)s - %(message)s')
```

Example:

```
import logging
logging.basicConfig(level=logging.DEBUG, format=' %(asctime)s - %(levelname)s - %(message)s')
logging.debug('Start of program')
def factorial(n):
    logging.debug('Start of factorial( %)' % (n))
    total = 1
    for i in range(n + 1):
        total *= i
        logging.debug('i is ' + str(i) + ', total is ' + str(total))
    logging.debug('End of factorial( %)' % (n))
    return total
print(factorial(5))
logging.debug('End of program')
```

Output:

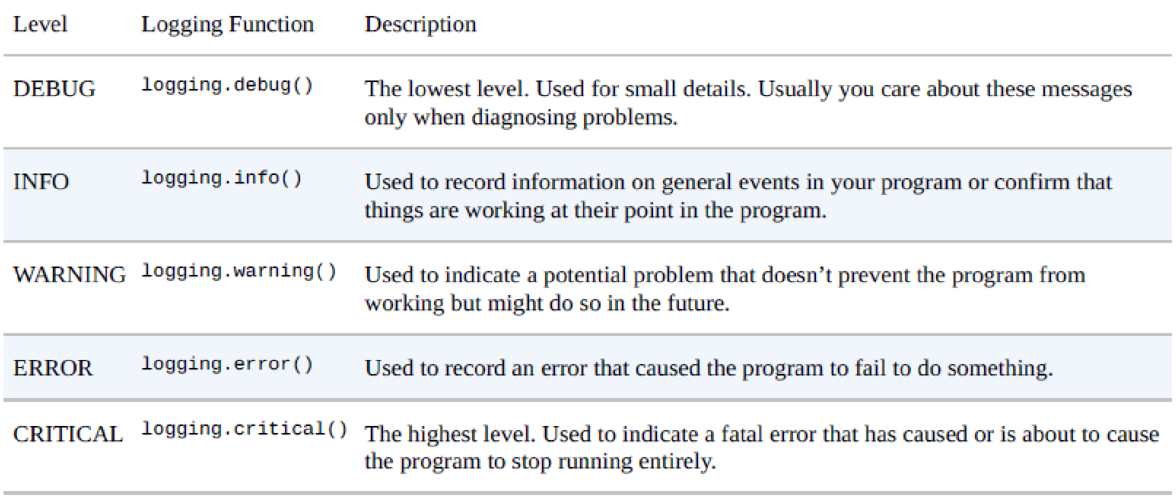
```
2015-05-23 16:20:12,664 - DEBUG - Start of program
2015-05-23 16:20:12,664 - DEBUG - Start of factorial(5)
2015-05-23 16:20:12,665 - DEBUG - i is 0, total is 0
2015-05-23 16:20:12,668 - DEBUG - i is 1, total is 0
2015-05-23 16:20:12,670 - DEBUG - i is 2, total is 0
2015-05-23 16:20:12,673 - DEBUG - i is 3, total is 0
2015-05-23 16:20:12,675 - DEBUG - i is 4, total is 0
2015-05-23 16:20:12,678 - DEBUG - i is 5, total is 0
2015-05-23 16:20:12,680 - DEBUG - End of factorial(5)
0
2015-05-23 16:20:12,684 - DEBUG - End of program
```

Change the for i in range(n + 1): line to for i in range(**1,** n + 1):, and run the program again.
Output:

```
2015-05-23 17:13:40,650 - DEBUG - Start of program
2015-05-23 17:13:40,651 - DEBUG - Start of factorial(5)
2015-05-23 17:13:40,651 - DEBUG - i is 1, total is 1
2015-05-23 17:13:40,654 - DEBUG - i is 2, total is 2
2015-05-23 17:13:40,656 - DEBUG - i is 3, total is 6
2015-05-23 17:13:40,659 - DEBUG - i is 4, total is 24
2015-05-23 17:13:40,661 - DEBUG - i is 5, total is 120
2015-05-23 17:13:40,661 - DEBUG - End of factorial(5)
120
2015-05-23 17:13:40,666 - DEBUG - End of program
```

Q) Briefly explain logging levels and give example.
Logging levels provide a way to categorize our log messages by importance.
There are five logging levels, described in Table below from least to most important.
Messages can be logged at each level using a different logging function.

Table . Logging Levels in Python



Level	Logging Function	Description
DEBUG	`logging.debug()`	The lowest level. Used for small details. Usually you care about these messages only when diagnosing problems.
INFO	`logging.info()`	Used to record information on general events in your program or confirm that things are working at their point in the program.
WARNING	`logging.warning()`	Used to indicate a potential problem that doesn't prevent the program from working but might do so in the future.
ERROR	`logging.error()`	Used to record an error that caused the program to fail to do something.
CRITICAL	`logging.critical()`	The highest level. Used to indicate a fatal error that has caused or is about to cause the program to stop running entirely.

Our logging message is passed as a string to these functions. The logging levels are suggestions. Ultimately, it is up to us to decide which category your log message falls into.

Example:

```
>>> import logging
>>> logging.basicConfig(level=logging.DEBUG, format=' %(asctime)s -
%(levelname)s - %(message)s')
>>> logging.debug('Some debugging details.')
2015-05-18 19:04:26,901 - DEBUG - Some debugging details.
>>> logging.info('The logging module is working.')
2015-05-18 19:04:35,569 - INFO - The logging module is working.
>>> logging.warning('An error message is about to be logged.')
2015-05-18 19:04:56,843 - WARNING - An error message is about to be logged.
>>> logging.error('An error has occurred.')
2015-05-18 19:05:07,737 - ERROR - An error has occurred.
>>> logging.critical('The program is unable to recover!')
2015-05-18 19:05:45,794 - CRITICAL - The program is unable to recover!
```

Q) Briefly explain how we can disable the logging. Give example.

After we have debugged our program, we probably don't want all these log messages cluttering the screen.

The logging.disable() function disables these so that we don't have to go into our program and remove all the logging calls by hand.

We simply pass logging.disable() a logging level, and it will suppress all log messages at that level or lower.

So if we want to disable logging entirely, just add logging.disable(logging.CRITICAL) to our program.

```
>>>> import logging
>> logging.basicConfig(level=logging.INFO, format=' %(asctime)s -
%(levelname)s - %(message)s')
>>> logging.critical('Critical error! Critical error!')
2015-05-22 11:10:48,054 - CRITICAL - Critical error! Critical error!
>>> logging.disable(logging.CRITICAL)
>>> logging.critical('Critical error! Critical error!')
>>> logging.error('Error! Error!')
```

Since logging.disable() will disable all messages after it, we will probably want to add it near the import logging line of code in our program. This way, we can easily find it to comment out or uncomment that call to enable or disable logging messages as needed.

Q) Briefly explain about IDE's Debugger.

The debugger is a feature of IDLE that allows us to execute our program one line at a time.

The debugger will run a single line of code and then wait for us to tell it to continue. By running our program "under the debugger" like this, we can take as much time as we want to examine the values in the variables at any given point during the program’s lifetime. This is a valuable tool for tracking down bugs. To enable IDLE’s debugger, click **Debug▸Debugger** in the interactive shell window. This will bring up the Debug Control window.

When the Debug Control window appears, select all four of the **Stack**, **Locals**, **Source**, and **Globals** checkboxes so that the window shows the full set of debug information.
While the Debug Control window is displayed, any time you run a program from the file editor, the debugger will pause execution before the first instruction and display the following:

The line of code that is about to be executed

- A list of all local variables and their values
- A list of all global variables and their values