**Python Assignment –9**

**By Abhishek Sachan**

**1. To what does a relative path refer?**

A relative path refers to a file or directory path that is relative to the current working directory of the program or the user's current location within the file system. It specifies the path to a file or directory relative to the current directory, rather than the absolute path which starts from the root directory.

For example, if the current working directory is **/home/user/** and there is a file named **file.txt** in the directory **/home/user/documents/**, then the relative path of the file from the current working directory would be **./documents/file.txt**. The. represents the current directory and the. /**documents** represents the path to the **document's** directory relative to the current directory.

**2. What does an absolute path start with your operating system?**

The absolute path in an operating system refers to the complete path of a file or directory starting from the root directory of the file system. In most operating systems, an absolute path starts with the root directory symbol, which varies depending on the operating system:

* On Windows, an absolute path starts with the drive letter (e.g., **C:**) followed by a colon and a backslash (e.g., **C:\**), since Windows uses backslashes to separate directories in a path.
* On Unix-based systems (e.g., Linux, macOS), an absolute path starts with a forward slash (**/**), which is used as the root directory separator.

**3. What do the functions os.getcwd() and os.chdir() do?**

The **os.getcwd()** and **os.chdir()** functions are part of the **os** module in Python and are used to manipulate the current working directory of a Python script.

* **os.getcwd()** returns the current working directory as a string. The current working directory is the directory in which the Python script is currently executing. The returned string represents the absolute path to the current working directory.
* **os.chdir(path)** changes the current working directory to the directory specified by the **path** argument. The **path** argument can be either an absolute path or a relative path. If the **path** argument is a relative path, it is relative to the current working directory

**4. What are** **the . and** **.. folders?**

In Python, the **.** and **..** folders are represented by the strings **'.'** and **'..'**, respectively. They can be used in file paths to refer to the current directory or the parent directory, just like in most operating systems.

In Python, the **.** and **..** folders can also be used in module imports to refer to the current package or the parent package, respectively. For example, **from .module1 import function1** imports **function1** from the current package, while **from** **..module2 import function2** imports **function2** from the parent package.

**5. In C:\bacon\eggs\spam.txt, which part is the dir name, and which part is the base name?**

In the file path **C:\bacon\eggs\spam.txt**, the directory name is **C:\bacon\eggs**, and the base name is **spam.txt**.

* The directory name is the path to the folder or directory containing the file. In this case, the directory name is **C:\bacon\eggs**.
* The base name is the name of the file itself, without the path to the directory. In this case, the base name is **spam.txt**

**6.What are the three “mode” arguments that can be passed to the** **open() function?**

The **open()** function in Python is used to open files in various modes, which determine how the file can be read from or written to. The three main mode arguments that can be passed to the **open()** function are:

1. **'r'** (read mode): This is the default mode for the **open()** function. It is used to open a file for reading, and the file pointer is positioned at the beginning of the file. If the file does not exist, a **FileNotFoundError** is raised. Example: **file =** **open('example.txt', 'r')**
2. **'w'** (write mode): This mode is used to open a file for writing. If the file already exists, its contents are truncated (i.e., deleted), and if it doesn't exist, a new file is created. The file pointer is positioned at the beginning of the file. Example: **file =** **open('example.txt', 'w')**
3. **'a'** (append mode): This mode is used to open a file for appending. If the file already exists, new data is appended to the end of the file, and if it doesn't exist, a new file is created. The file pointer is positioned at the end of the file. Example: **file =** **open('example.txt', 'a')**

**7. What happens if an existing file is opened in write mode?**

If an existing file is opened in write mode using the **open()** function in Python, its contents will be truncated (i.e., deleted), and any data that was previously stored in the file will be lost. This means that if you try to open an existing file in write mode and write data to it, the new data will overwrite the existing data, starting from the beginning of the file.

**8.How do you tell the difference between** **read() and readlines()?**

In Python, the **read()** and **readlines()** methods are used to read data from a file object. The main difference between the two methods is that **read()** returns the entire contents of the file as a single string, while **readlines()** returns the contents of the file as a list of strings, where each string represents a line of text.

Let file taken eg.text

1.Bacon love racon

2.Bacon love data analyst

3.Bacon like singing

file=open(‘eg.text’,r)

content=file.read()

conetnt2=file.readlines()

print(contnet)

print(contnet2)

Output for content

1.Bacon love racon

2.Bacon love data analyst

3.Bacon like singing

Output for content 2

[‘1. Bacon love racon’,’ 2. Bacon love data analyst’,’ 3. Bacon like singing’]

**9.What data structure does a shelf value resemble?**

In Python, the **shelve** module provides a simple interface to store and retrieve persistent objects using a key-value mapping. A **shelve** value resembles a dictionary data structure, in which the keys are strings, and the values can be any Python object. However, unlike a regular dictionary, the values stored in a **shelve** object are persistent, meaning that they are stored on disk and can be retrieved even after the program that created them has existed.

The **shelve** module is built on top of the **dbm** module, which provides a simple key-value database interface using files. When a **shelve** object is created, a new file is opened (or an existing one is used if one exists) and the key-value mappings are stored in the file. The **shelve** object provides a dictionary-like interface for accessing and modifying the values stored in the file.

import shelve

# create a new shelf object

my\_shelf = shelve.open('my\_data')

# store some data in the shelf

my\_shelf['foo'] = 'bar'

my\_shelf['spam'] = [1, 2, 3, 4] # retrieve some data from the shelf print(my\_shelf['foo']) # prints 'bar'

print(my\_shelf['spam']) # prints [1, 2, 3, 4]

# modify some data in the shelf

my\_shelf['foo'] = 'baz' # close the shelf object my\_shelf.close()