**Assignment 9**

1. To what does a relative path refer?

In Python, a relative path refers to the path of a file or directory relative to the current working directory of the Python script. The current working directory is the directory in which the Python script is being executed.

For example, if the Python script is located in the directory **/home/user/scripts/** and you want to access a file located in the directory **/home/user/data/**, the relative path of the file would be **../data/** because you need to move up one directory level (**..**) from the current working directory (**scripts/**) to reach the directory containing the file (**data/**).

Relative paths are usually used when accessing files or directories within the same project or folder structure. They provide a concise way of specifying the location of a file or directory relative to the location of the Python script, without the need to specify the full path from the root directory of the file system.

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

In most operating systems, including Windows, macOS, and Linux, an absolute path starts with the root directory of the file system.

On Windows, an absolute path typically starts with the drive letter (e.g., **C:**) followed by a backslash (e.g., **\**), and then the path to the file or directory. For example: **C:\Users\JohnDoe\Documents\example.txt**.

On macOS and Linux, an absolute path typically starts with a forward slash (e.g., **/**) and then the path to the file or directory. For example: **/Users/JohnDoe/Documents/example.txt**.

Absolute paths provide a complete and unambiguous way of specifying the location of a file or directory on the file system, starting from the root directory. They can be used to access files or directories from anywhere on the file system, regardless of the current working directory of the Python script or the user's current location in the file system hierarchy.

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

The **os.getcwd()** and **os.chdir()** functions are used to work with the current working directory in Python.

**os.getcwd()**: This function returns the current working directory as a string. It returns the path of the directory in which the Python script is being executed. For example:

pythonCopy code

import os current\_directory = os.getcwd() print(current\_directory)

This will print the current working directory to the console.

**os.chdir()**: This function is used to change the current working directory to the specified path. It takes a string argument that specifies the path to the new directory. For example:

pythonCopy code

import os os.chdir('/home/user/documents/')

This will change the current working directory to **/home/user/documents/**.

Together, these two functions can be used to navigate the file system and work with files and directories relative to the current working directory.

4. What are the . and .. folders?

In the context of file systems, the **.** and **..** folders are special directory names that are used to represent the current directory and the parent directory, respectively.

The **.** directory, also known as the current directory or current working directory, represents the directory in which the user or process is currently located. It is often used in relative file paths to indicate the path of a file or directory relative to the current working directory.

The **..** directory, also known as the parent directory, represents the directory that is one level above the current directory in the directory hierarchy. It is also often used in relative file paths to indicate the path of a file or directory relative to the parent directory.

For example, suppose the current working directory is **/home/user/documents/** and there is a file called **example.txt** located in the parent directory **/home/user/**. To access the file using a relative file path, we can use the **..** directory to indicate the parent directory:

pythonCopy code

import os parent\_directory = os.path.abspath('..') file\_path = os.path.join(parent\_directory, 'example.txt')

Here, the **os.path.abspath('..')** function returns the absolute path of the parent directory, and the **os.path.join()** function is used to join the path of the parent directory with the file name to create the full file path.

In summary, the **.** and **..** directories are special directory names used to represent the current directory and the parent directory, respectively, in file systems.

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, also known as the path or folder name, is the part of the file path that specifies the location of the file in the directory hierarchy. In this example, the directory name is **C:\bacon\eggs**.

The base name, also known as the file name or file extension, is the part of the file path that specifies the name of the file itself. In this example, the base name is **spam.txt**.

When working with file paths in Python, the **os.path** module provides functions for working with directory names and base names separately. For example, the **os.path.dirname()** function returns the directory name of a file path, and the **os.path.basename()** function returns the base name of a file path:

pythonCopy code

import os file\_path = 'C:\\bacon\\eggs\\spam.txt' directory\_name = os.path.dirname(file\_path) base\_name = os.path.basename(file\_path) print(directory\_name) # 'C:\\bacon\\eggs' print(base\_name) # '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 and return a file object, which provides a set of methods for working with the contents of the file. The **open()** function takes two arguments: the file path and the mode in which the file should be opened. The mode argument specifies how the file should be opened and what operations can be performed on it.

There are three mode arguments that can be passed to the **open()** function:

1. **"r"**: Read mode. This is the default mode and is used to open a file for reading. The file pointer is placed at the beginning of the file, and the file is read from start to end.
2. **"w"**: Write mode. This mode is used to open a file for writing. If the file already exists, its contents are truncated. If the file does not exist, a new file is created.
3. **"a"**: Append mode. This mode is used to open a file for appending. If the file already exists, new data is written to the end of the file. If the file does not exist, a new file is created.

For example, to open a file in read mode, we can use the following code:

pythonCopy code

file = open('example.txt', 'r')

To open a file in write mode, we can use the following code:

pythonCopy code

file = open('example.txt', 'w')

To open a file in append mode, we can use the following code:

pythonCopy code

file = open('example.txt', 'a')

It's worth noting that there are additional mode arguments that can be passed to the **open()** function, such as **"x"** for exclusive creation mode or **"b"** for binary mode. The mode arguments can also be combined, such as **"wb"** for writing a binary file.

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 existing contents will be truncated or overwritten, and the file will be treated as if it were a new file. This means that any data that was previously stored in the file will be lost.

For example, consider the following code:

pythonCopy code

file = open('example.txt', 'w') file.write('Hello, world!') file.close()

This code will create a new file called **example.txt** and write the string **'Hello, world!'** to it. If the file already exists, its contents will be overwritten with the new data.

It is important to exercise caution when opening files in write mode, as this can lead to unintended data loss or corruption. It is generally a good practice to make a backup copy of any files that will be modified or overwritten, or to use a version control system to track changes to the files over time.

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 key difference between the two methods is in the way they return the data.

The **read()** method reads the entire contents of the file object as a single string and returns it. It reads the file from the current file pointer position until the end of the file, or until the specified number of characters, if given. Here's an example:

pythonCopy code

file = open('example.txt', 'r') content = file.read() file.close() print(content)

This code will open the file **example.txt** in read mode and read the entire contents of the file into the **content** variable as a string.

On the other hand, the **readlines()** method reads the entire contents of the file object line by line and returns them as a list of strings. Each element of the list corresponds to a single line of the file. Here's an example:

pythonCopy code

file = open('example.txt', 'r') lines = file.readlines() file.close() for line in lines: print(line)

This code will open the file **example.txt** in read mode and read the contents of the file into the **lines** variable as a list of strings. Each element of the list corresponds to a single line of the file, and the **for** loop is used to print each line.

In summary, the **read()** method returns the entire contents of the file as a single string, while the **readlines()** method returns the entire contents of the file as a list of strings, with each element of the list corresponding to a single line of the file.

9. What data structure does a shelf value resemble?

In Python, the **shelve** module provides a way to store and retrieve persistent objects using a dictionary-like interface. A shelf value in Python resembles a dictionary data structure. It can be thought of as a persistent dictionary, where the keys are strings and the values can be any picklable Python object.

Like a dictionary, a shelf value stores data in a key-value format. However, unlike a regular dictionary, a shelf value persists its contents to disk, so the data is not lost when the program exits. A shelf value is implemented using a combination of a database file and a dictionary object, which provides a mapping between the keys and the values.

Here's an example of how to use the **shelve** module to create a shelf value:

pythonCopy code

import shelve # Open a shelf file shelf = shelve.open('myshelf') # Add some data to the shelf shelf['key1'] = 'value1' shelf['key2'] = 'value2' # Close the shelf file shelf.close()

In this example, we create a shelf value by opening a shelf file called **myshelf**. We then add some data to the shelf using dictionary-like syntax, and close the shelf file when we're done. The resulting shelf file can be reopened later to retrieve the stored data.

Overall, a shelf value in Python resembles a dictionary data structure, but with the added benefit of persisting its contents to disk for later use.