1. To what does a relative path refer?

Ans:

A relative path refers to the location of a file or directory relative to the current working directory (CWD). Instead of specifying the full path from the root directory of the filesystem, a relative path indicates the location of the file or directory relative to where the script or program is currently running.

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

Ans:

In most operating systems, an absolute path starts with the root directory of the filesystem.

Windows: An absolute path typically starts with a drive letter followed by a colon (C:\, D:\, etc.) for each drive, and then the rest of the path specifying the location of the file or directory within that drive.

Unix-like systems (including Linux and macOS): An absolute path typically starts with a forward slash (/) representing the root directory, and then the rest of the path specifying the location of the file or directory within the filesystem.

For example:

Windows: C:\Users\username\Documents\file.txt

Linux/macOS: /home/username/Documents/file.txt

An absolute path uniquely identifies the location of a file or directory from the root of the filesystem, regardless of the current working directory.

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

Ans:

The functions os.getcwd() and os.chdir() are both part of the os module in Python, which provides a way to interact with the operating system.

os.getcwd():

This function returns the current working directory (CWD) as a string.

It does not take any arguments.

The current working directory is the directory in which the Python script is currently executing.

Example:

import os

cwd = os.getcwd()

print("Current working directory:", cwd)

os.chdir(path):

This function changes the current working directory to the specified path.

It takes one argument, which is the path of the directory to change to.

If the path is relative, it is interpreted relative to the current working directory.

If the path is absolute, it is interpreted as an absolute path.

Example:

import os

# Change the current working directory to 'new\_directory'

os.chdir('new\_directory')

After executing this code, the current working directory will be changed to 'new\_directory'.

Together, these functions provide a way to inspect and modify the current working directory within a Python script, allowing you to navigate and manipulate files and directories on the filesystem.

4. What are the . and .. folders?

Ans:

The . and .. folders are special directory entries found in most filesystems:

. (dot):

The . (dot) represents the current directory.

It is a reference to the directory in which it is located.

When used in a file path, it refers to the current directory itself.

For example, if you are in the directory /home/user, then . refers to /home/user.

.. (dot-dot):

The .. (dot-dot) represents the parent directory.

It is a reference to the directory immediately above the current directory in the filesystem hierarchy.

When used in a file path, it refers to the parent directory of the current directory.

For example, if you are in the directory /home/user, then .. refers to /home.

These special directory entries provide a way to navigate the filesystem hierarchy relative to the current directory. They are commonly used in command-line interfaces and file paths to refer to nearby directories without specifying their absolute paths.

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

Ans:

In the file path C:\bacon\eggs\spam.txt, the directory name and the base name can be identified as follows:

Directory Name: C:\bacon\eggs

This part of the path specifies the directory in which the file spam.txt is located.

It consists of the drive letter (C:) followed by a series of directory names separated by backslashes (\).

This represents the path to the parent directory of the file.

Base Name: spam.txt

This part of the path specifies the name of the file itself.

It is the last component of the path, following the last backslash (\).

It represents the name of the file without any directory information.

In summary:

Directory Name: C:\bacon\eggs

Base Name: spam.txt

Understanding the distinction between the directory name and the base name is useful when working with file paths, as it allows you to manipulate and extract information about the file and its location within the filesystem.

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

Ans:

The open() function in Python accepts three main "mode" arguments, which specify how the file should be opened. These mode arguments are:

Read mode ('r'):

This is the default mode for opening files.

It allows reading from the file, but does not allow writing to it.

If the file does not exist, it raises a FileNotFoundError.

Write mode ('w'):

This mode opens the file for writing.

If the file exists, its contents are overwritten.

If the file does not exist, a new file is created.

It does not allow reading from the file.

Append mode ('a'):

This mode opens the file for writing.

If the file exists, new data is appended to the end of the file.

If the file does not exist, a new file is created.

It does not allow reading from the file.

These mode arguments can be used alone or in combination with other characters to specify additional options. For example, 'r+' opens the file for both reading and writing, 'rb' opens the file in binary read mode, and so on.

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

Ans:

If an existing file is opened in write mode ('w') using the open() function in Python, the following will happen:

File Truncation:

If the file already exists, its contents will be truncated (i.e., deleted), and the file will be treated as empty.

All existing data in the file will be removed.

New File Creation:

If the file does not exist, a new file with the specified name will be created.

The file will be created in the specified location with the specified name.

Write Operation:

The file is opened for writing, allowing data to be written to it.

Subsequent write operations will start from the beginning of the file, overwriting any existing data.

It's important to exercise caution when opening existing files in write mode ('w'), as any existing data in the file will be lost. If you want to append data to an existing file without truncating its contents, you should use append mode ('a') instead.

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

Ans:

The read() and readlines() methods are both used to read data from a file object in Python, but they have different behaviors:

read():

The read() method reads the entire contents of the file as a single string.

It reads the file from the current position of the file pointer until the end of the file.

If no size argument is provided, it reads the entire file. Alternatively, you can specify the number of bytes to read as an argument.

It returns a single string containing all the data read from the file.

Example:

with open('file.txt', 'r') as file:

data = file.read()

print(data)

readlines():

The readlines() method reads all lines of the file into a list.

It reads the file from the current position of the file pointer until the end of the file, splitting it into lines based on newline characters (\n).

Each line is stored as a separate string element in the list.

If the size parameter is provided, it reads at most size bytes/characters from the file and returns them as a list of lines.

If size is negative or omitted, it reads until EOF and returns all lines.

Example:

with open('file.txt', 'r') as file:

lines = file.readlines()

print(lines)

In summary, read() returns the entire file as a single string, while readlines() returns a list of strings, with each string representing a line from the file. Use read() when you want to process the entire file as a single entity, and use readlines() when you want to process the file line by line.

9. What data structure does a shelf value resemble?

Ans:

A shelf value in Python, as provided by the shelve module, resembles a dictionary data structure.

Like a dictionary, a shelf value stores key-value pairs, where each key is unique within the shelf.

You can access and modify values in a shelf using keys, just like with a dictionary.

Shelf values support methods similar to dictionary methods, such as keys(), values(), items(), get(), pop(), update(), and others.

However, there are some differences:

Persistence: Unlike a regular dictionary, a shelf value is persistent. It is stored on disk and retains its data even after the program exits. This makes it useful for storing data across multiple program runs.

Serialization: Shelf values serialize the data when storing it on disk and deserialize it when retrieving it. This allows them to handle a wider range of data types compared to regular dictionaries.

Overall, while shelf values resemble dictionaries in their interface and usage, they provide additional features related to persistence and serialization, making them useful for certain types of applications where data storage between program runs is required.