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

A relative path in computing refers to a file or directory path that is relative to the current working directory of the program or user.

In other words, a relative path describes the location of a file or directory relative to the current directory, rather than describing the complete path from the root directory.

For example, if the current working directory is `/home/user/`, and we have a file named `file.txt` located in the directory `/home/user/docs/`, then the relative path to `file.txt` would be `docs/file.txt`.

Relative paths are often used in programming and file systems to reference files and directories within the current directory or its subdirectories. They are useful for moving or copying files or directories between directories without having to update the full path every time.

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

An absolute path in computing refers to the complete path to a file or directory, starting from the root directory of the file system.

In most operating systems, the root directory is represented by a special character or name.

For example, in Unix-based systems (including Linux and macOS), an absolute path starts with a forward slash (/). For instance, `/home/user/documents/myfile.txt` is an absolute path to a file named `myfile.txt` located in the `documents` directory under the `user` home directory.

In Windows operating systems, an absolute path typically starts with the drive letter followed by a colon and a backslash (e.g., `C:\Users\JohnDoe\Documents\myfile.txt`). The root directory for each drive is represented by the drive letter followed by a colon and a backslash (e.g., `C:\`).

Using absolute paths ensures that the file or directory can be located from any directory in the file system, regardless of the current working directory.

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

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

`os.getcwd()` returns the current working directory as a string. It returns a string that represents the current working directory of the Python script, which is the directory that the script is currently running in. This function can be used to determine the current working directory, which is useful for opening files or accessing other resources that are located in the same directory as the script.

`os.chdir(path)` changes the current working directory to the specified `path`. It takes in a string representing the path to the new working directory and changes the current working directory to that directory. This function is useful for navigating to different directories within a script.

For example, the following code demonstrates how to use `os.getcwd()` and `os.chdir()` to navigate to a directory called `myfolder`, which is located in the same directory as the Python script:

```

import os

# get the current working directory

cwd = os.getcwd()

print("Current working directory:", cwd)

# change the working directory to the 'myfolder' directory

os.chdir('myfolder')

# get the new working directory

new\_cwd = os.getcwd()

print("New working directory:", new\_cwd)

```

The output of the above code should be something like:

```

Current working directory: /home/user/projects/myproject

New working directory: /home/user/projects/myproject/myfolder

```

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

In most operating systems, `.` refers to the current directory and `..` refers to the parent directory.

For example, suppose you are currently in the directory `/home/user/docs`. The `.` folder refers to the same directory (`/home/user/docs`), while the `..` folder refers to the parent directory (`/home/user`).

These folders can be useful in navigating file systems or specifying file paths in code. For instance, to navigate to the parent directory of the current directory in the command line, you can use `cd ..`. In Python code, you can use `os.path.join('..', 'filename.txt')` to specify a file path in the parent directory.

**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 (dir name) is `C:\bacon\eggs` and the base name is `spam.txt`.

The directory name refers to the path of the directory containing the file, and the base name refers to the file name itself.

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

The `open()` function can take the following three mode arguments:

1. `'r'` (read mode): This is the default mode. It opens the file for reading, and raises an error if the file does not exist.

2. `'w'` (write mode): It opens the file for writing, truncating the file if it already exists, or creating the file if it doesn't exist.

3. `'a'` (append mode): It opens the file for writing, but does not truncate the file. Instead, it appends any new data to the end of the file. If the file does not exist, it creates a new file.

Each of these mode arguments can be combined with additional characters to indicate the file's type of access and encoding. For example, `'rb'` opens the file in binary read mode, and `'w+'` opens the file for both reading and writing.

**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, the file is truncated to zero length. This means that any existing data in the file is deleted and the file is essentially "reset" to an empty file. Any new data written to the file will overwrite any existing data in the file. It is important to exercise caution while using write mode with an existing file as it may result in the permanent loss of data.

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

The `read()` method and `readlines()` method are both used to read data from a file in Python, but they differ in how they handle the data.

The `read()` method reads the entire contents of the file as a single string and returns it. It takes an optional argument that specifies the number of bytes to read from the file. If no argument is provided, `read()` reads the entire contents of the file.

The `readlines()` method reads the contents of the file line by line and returns a list of strings, where each string represents a line from the file. The newline character at the end of each line is included in the string. If called without arguments, `readlines()` reads the entire contents of the file and returns a list where each element of the list represents a line of the file.

In summary, `read()` reads the entire contents of the file as a single string, while `readlines()` reads the file line by line and returns a list of strings, where each string represents a line from the file.

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

A shelf value in Python resembles a dictionary data structure. It has key-value pairs that can be manipulated like a dictionary but the data is stored on disk rather than memory. The shelve module is used to work with the shelf values.