1. To what does a relative path refer?

Answer: In Python, a relative path refers to the path to a file or directory relative to the current working directory of the Python script or the current directory from which the Python script is executed.

For example, if your Python script is located in a directory called "scripts" and you want to refer to a file "data.txt" located in a directory called "data" within the same parent directory, you can use a relative path like this:

relative_path = "data/data.txt"

This assumes that your current working directory or the directory from which your script is executed is the "scripts" directory.

Using relative paths is often preferred in Python because they make the code more portable. It allows you to move the entire project directory to another location without having to update absolute paths in your code.

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

Answer: In Python, an absolute path starts with the root directory of the file system. On most operating systems, including Windows, macOS, and Linux/Unix-based systems, the root directory is represented by either a drive letter (on Windows) or a forward slash (on Unix-like systems).

Here are examples of absolute paths in Python:

- On Windows:
  absolute_path = "C:\\Users\\Username\\Documents\\file.txt"

- On Unix-like systems (including macOS and Linux):
  absolute_path = "/home/username/documents/file.txt"

In Python, you can also use the **os.path.abspath()** function to obtain the absolute path of a file or directory. This function returns the absolute path of the specified file or directory by resolving any symbolic links and references to parent directories.

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

Answer: os.getcwd() returns the current working directory as a string. This is the directory that the Python script is currently executing in.

os.chdir(path) changes the current working directory to the specified path. This allows you to navigate and operate within different directories from your Python script.

4. What are the . and .. folders?

Answer: In most operating systems, including Unix-like systems such as Linux and macOS, as well as Windows, the **.** (dot) and **..** (dot-dot) folders have special meanings:

1. **.** (dot): This represents the current directory. When you refer to **.** in a file path, you are specifying the current directory.
2. **..** (dot-dot): This represents the parent directory. When you refer to **..** in a file path, you are specifying the directory that contains the current directory.

   These symbols are often used when navigating file systems, especially in command-line interfaces, to specify relative paths. For example, if you're in a directory called **folder1** and you want to refer to a file in its parent directory, you can use **../filename**. Similarly, if you're in **folder1** and want to refer to a file within **folder1**, you can use **./filename**, but **./** is usually optional and can be omitted.

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

Answer: In Python, you can use the **os.path** module to work with file paths. Specifically, you can use the **os.path.dirname()** function to get the directory name and the **os.path.basename()** function to get the base name.

Here's how you can use them:

```
import os

file_path = "C:/bacon/eggs/spam.txt"

dir_name = os.path.dirname(file_path)
base_name = os.path.basename(file_path)

print("Directory name:", dir_name)
print("Base name:", base_name)
```

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

Answer: The **open()** function in Python accepts three main mode arguments:

1. **Read mode ('r')**: This is the default mode. It opens the file for reading. If the file does not exist or cannot be opened for reading, it raises a **FileNotFoundError** exception.
2. **Write mode ('w')**: This mode opens the file for writing. If the file does not exist, it creates a new file. If the file exists, it truncates (i.e., clears) the existing content. If the file cannot be opened for writing, it raises an exception.
3. **Append mode ('a')**: This mode opens the file for appending. If the file does not exist, it creates a new file. If the file exists, it appends new data to the end of the file. It does not

truncate the existing content. If the file cannot be opened for appending, it raises an exception.

Additionally, you can specify a mode that combines reading and writing by adding a **'+'** to the mode string. For example, **'r+'** opens the file for both reading and writing, while **'w+'** and **'a+'** open the file for reading and writing, with **'w+'** truncating the file and **'a+'** appending to it.

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

Answer: If an existing file is opened in write mode (**'w'**) in Python, the following occurs:

1. If the file exists, its contents are truncated, meaning all existing data in the file is erased.
2. If the file does not exist, a new file is created.

Opening a file in write mode essentially gives you a fresh start, allowing you to write new content to the file. It's important to exercise caution when using write mode, as you can unintentionally delete existing data in the file.

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

Answer: In Python, **read()** and **readlines()** are both methods used to read data from a file, but they behave differently:

1. **read()**: This method reads the entire contents of the file as a single string, including newline characters (**'\n'**). It reads from the current position of the file pointer to the end of the file or up to the specified number of bytes if given.
   Example:
   ```
   with open("example.txt", "r") as file:
       content = file.read()
   ```

2. **readlines()**: This method reads all the lines of the file and returns them as a list of strings, with each string representing one line. The newline characters (**'\n'**) are included in each string. If you specify a size (in bytes) as an argument, it will read up to that many bytes.
   Example:
   ```
   with open("example.txt", "r") as file:
       lines = file.readlines()
   ```

In summary, **read()** returns the entire file content as a single string, while **readlines()** returns a list of strings, each representing one line from the file.

9. What data structure does a shelf value resemble?

Answer: A shelf value in Python resembles a dictionary data structure.

Python's **shelve** module provides a persistent, dictionary-like interface for storing and retrieving Python objects to and from a disk file. When you work with a shelf, you can store and retrieve data using keys, much like you would with a dictionary.

Here's a basic example of using a shelf:

```
import shelve

# Open a shelf file for read-write access
with shelve.open("mydata") as shelf:
    shelf["name"] = "John"
    shelf["age"] = 30
    shelf["city"] = "New York"

    print(shelf["name"])
    print(shelf["age"])
    print(shelf["city"])

    del shelf["age"]  # Deleting an entry
```

In this example, the shelf acts like a dictionary where you can store and retrieve data using keys like **"name"**, **"age"**, and **"city"**.