subprocess vs os.system

When running shell commands in Bash from a Python script, the **subprocess** module is generally considered a better approach compared to **os.system**. Here is why:

1. Flexibility and Functionality: The **subprocess** module provides a more powerful and flexible way to interact with shell commands. It allows you to have more control over the execution and capture of command output, error handling, process management, and other features. With **subprocess**, you can capture the output of the command, handle input/output streams, set environment variables, and control the execution flow more effectively.

2. Security: The **subprocess** module offers better security features, especially when dealing with user-provided input or dynamically constructing commands. It allows you to pass arguments and command options explicitly, reducing the risk of command injection vulnerabilities that could arise from using string interpolation with **os.system**.

3. Compatibility: The **subprocess** module is more versatile and compatible across different platforms, including Windows, macOS, and Linux. It provides a consistent API for executing shell commands, regardless of the underlying operating system.

4. Code Readability: Using **subprocess** typically results in cleaner and more readable code. You can specify the command and its arguments as a list of strings, making it easier to understand and maintain compared to constructing a command string with **os.system**.

5. Advanced Features: **subprocess** offers additional functionalities such as subprocess piping, input/output redirection, and the ability to communicate interactively with the subprocess, which may be necessary for complex command execution scenarios.

In summary, **subprocess** is the recommended approach for running shell commands in Python due to its flexibility, security, compatibility, and advanced features. It provides a more robust and reliable way to interact with the command-line environment from your Python scripts.

Return Vs. Print Built In functions.

print is used to **display output to the console**, while **return** is **used to provide a value or object as the result of a function**. print is mainly used for informational purposes, while return is essential for returning values from functions, allowing for further computation or assignment in the program.

print: The **print** statement is used to display output to the console or standard output. It takes one or more values as arguments and displays them as text. The primary purpose of **print** is to provide information or debugging messages during program execution. The output is visible to the user or can be redirected to a file or other output streams.

For example:

```
def greet(name):
    print("Hello,", name)

greet("John")  # Output: Hello, John
```

The function greet has a parameter called name. When calling the function greet(), you must add 1 parameter.

return: The **return** statement is used to exit a function and return a value or object to the caller. It allows a function to provide a result or output that can be used in further computations or assignments. The **return** statement terminates the execution of the function and transfers control back to the point where the function was called.

For example:

```
def add_numbers(a, b):
    return a + b

result = add_numbers(3, 5)
print(result)  # Output: 8
```

In this case, the **add_numbers** function returns the sum of **a** and **b**, which is then assigned to the **result** variable. The value **8** is printed to the console using **print**.

Result function is great in helping us get the result of one command and use it for the next command.

Why use import shutil when checking whether a binary of an application such as docker, git exists.

Using the **shutil.which()** function can be beneficial when checking for the existence of a binary application like Docker or Git. Here are a few reasons why **shutil.which()** is commonly used:

1. Cross-platform compatibility: The **shutil.which()** function is designed to work across different operating systems, including Windows, macOS, and Linux. It searches the system's executable search path to locate the specified binary, making it convenient for writing platform-independent code.
2. Reliable binary detection: **shutil.which()** performs a thorough search for the binary, considering the directories listed in the **PATH** environment variable. It checks each directory to find the presence of the binary and returns the full path if found. This

approach ensures a more reliable detection compared to simply relying on a command like **os.system()** or **subprocess.run()**.

3. No shell interpretation: Unlike executing shell commands through **os.system()** or **subprocess.run()**, **shutil.which()** does not involve interpreting shell commands. It directly searches for the binary in the system's path, avoiding any potential issues or vulnerabilities associated with shell command interpretation.

4. Flexibility for further actions: With **shutil.which()**, you can obtain the full path to the binary, allowing you to perform additional actions, such as executing the binary with **subprocess.run()** or using the path in your code logic.