# Python: how to execute shell commands

Writing a Python script that executes shell commands provides a powerful tool for automating system operations, managing processes, and interacting with the operating system. Python provides several libraries for running shell commands, each with its own benefits and use cases. In this article, we'll explore some of the main ways to execute shell commands in Python, covering the `subprocess`, `os`, and `sh` libraries.. p>

## 1. Using `subprocess`

The `subprocess` library is the most flexible and recommended tool for executing shell commands in Python. It offers great flexibility by allowing the execution of new commands, connecting to their input/output/error pipes and obtaining their return codes.

```
import subprocess

# Run a simple command
result = subprocess.run(["ls", "-l"],
capture_output=True, text=True)
print(result.stdout)

# Running a command and capturing the output
try:
     output = subprocess.check_output(["ls", "-l"],
text=True)
     print(output)
```

```
except subprocess.CalledProcessError as e:
     print(e)
```

## 2. Using `os.system` and `os.popen`

Before the introduction of `subprocess`, the `os` module was commonly used to execute shell commands. However, compared to `subprocess`, it offers less control and flexibility.

`os.system()`: Executes the indicated command in a subshell.

```
import os

# Example of os.system usage
os.system('ls -l')
```

`os.popen()`: Executes the indicated command in a subshell and returns a file object linked to the standard input or output of the command (depending on the parameters).

```
# Example of using os.popen
stream = os.popen('ls -l')
output = stream.read()
print(output)
```

## 3. Using sh (only on Unix-like)

The `sh` module is a wrapper for `subprocess` designed to make executing shell commands easier and more intuitive. It is not included in the Python standard library and only works on Unix-like systems.

```
from sh import ls

# Example of using sh
print(ls("-l"))
```

## Security Considerations

When running shell commands from a Python script, it is important to consider the security implications, especially if you are working with user-supplied input. Be sure to sanitize all inputs to avoid vulnerabilities such as command injection.

## Conclusion

Python offers several options for executing shell commands, each with its own advantages. `subprocess` is the most powerful and flexible choice for most applications, while `os` and `sh` may be simpler options for use cases specific. Selecting the right library depends on your project's specific needs and security requirements.