Automation with Python

Sidheswar Routray
Department of Computer Science & Engineering
School of Technology

Automation with Python

- Automation is the process of using technology to perform repetitive or predefined tasks with minimal human intervention.

Why Python for Automation?

- Simple and human-readable syntax.
- Platform-independent (works on Windows, macOS, Linux).
- Large standard library (comes with many built-in modules).
- Extensive third-party ecosystem of libraries for specialized automation.
- Integration with databases, APIs, web services, and operating systems.

Categories of Python Automation and Tools

1. **File and Folder Automation**

- **os** → Create, delete, and manage directories and files.
- **shutil** → Copying, moving, archiving files.
- **pathlib** → Modern, object-oriented file path manipulation.

2. **Web Automation**

- **Selenium** → Automates browsers for form filling, testing, and data scraping.
- **Requests** → Makes HTTP requests (e.g., fetching API data).
- **BeautifulSoup / lxml** → Parse and extract information from HTML.
- **PyAutoGUI** → Automates mouse and keyboard actions on the desktop.

Categories of Python Automation and Tools

3. System & Process Automation

- **subprocess** → Run system commands.
- **psutil** → Monitor CPU, memory, processes.

4. Task Scheduling and Workflow

- **schedule** → Simple time-based task automation.
- **APScheduler** → Advanced scheduling with cron-like jobs.
- **Airflow** → Workflow management for data pipelines.

5. Email & Communication Automation

- **smtplib** → Send emails.
- **imaplib** → Read emails.
- **twilio** → Send SMS/WhatsApp messages.

Subprocess

Sidheswar Routray
Department of Computer Science & Engineering
School of Technology

Subprocess

- The subprocess is a standard Python module designed to start new processes from within a Python script.
- run multiple processes in parallel or call an external program or external command from inside your Python code
- the subprocess module is that it allows the user to manage the inputs, the outputs, and even the errors raised by the child process from the Python code.

Subprocess.run

- **subprocess.run()**: The most straightforward way to run a subprocess. It runs the command described by args, waits for it to complete, then returns a CompletedProcess instance.
- **Syntax:**

subprocess.**run**(*args, *, stdin=None, input=None, stdout=None, stderr=None, capture_output=False, shell=False, cwd=None, timeout=None, check=False, encoding=None, errors=None, text=None, env=None, universal_newlines=None, **other_popen_kwargs*)

- **args:** The command to run and its arguments, passed as a list of strings.
- **capture_output:** When set to True, will capture the standard output and standard error.
- **text:** When set to True, will return the stdout and stderr as string, otherwise as bytes.

Subprocess.run

- **check:** a boolean value that indicates whether to check the return code of the subprocess, if check is true and the return code is non-zero, then subprocess `CalledProcessError` is raised.
- **timeout:** A value in seconds that specifies how long to wait for the subprocess to complete before timing out.
- **shell:** A boolean value that indicates whether to run the command in a shell. This means that the command is passed as a string, and shell-specific features, such as wildcard expansion and variable substitution, can be used.

Using subprocess.run()

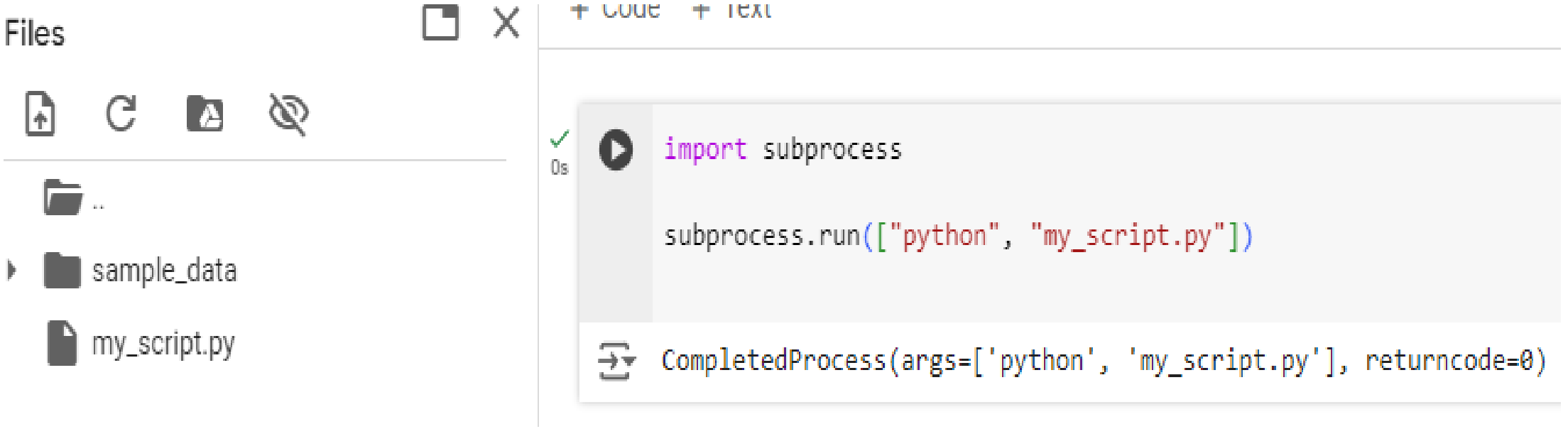
- The run function of the subprocess module in Python is a great way to run commands in the background without worrying about opening a new terminal or running the command manually.

```
import subprocess


subprocess.run(["python", "my_script.py"])
```

Using subprocess.run()

- subprocess.run: This function is used to run a command in the form of a list, where the first item is the command (in this case, "python") and the following items are the arguments (here, "my_script.py").
- "python": This specifies the command to run Python.
- "my_script.py": This is the argument passed to the Python interpreter, telling it to execute the my_script.py script.



Using subprocess.run()

- You can capture the output or handle errors by adding more options, as in the examples I shared earlier. For instance, to capture the output:

```
result = subprocess.run(["python", "my_script.py"],
capture_output=True, text=True)
print(result.stdout)
```

Using subprocess.run()

+ Code + Text

[1]
```
import subprocess

subprocess.run(["python", "my_script.py"])
```

```
CompletedProcess(args=['python', 'my_script.py'], returncode=0)
```

```
result = subprocess.run(["python", "my_script.py"], capture_output=True, text=True)
print(result.stdout)
```

```
Hello from my_script.py
```

Run code directly

- It's also possible to write Python code directly to the function instead of passing a .py file. Here's an example of running such a subprocess:

```
result = subprocess.run(["/usr/local/bin/python", "-c", "print('This is a subprocess')"])
```

- subprocess.run([...]): Executes the command specified in the list.
- "/usr/local/bin/python": Specifies the Python interpreter to use.
- "-c": Tells Python to execute the following string as a command.
- "print('This is a subprocess')": The Python command to be executed.

Subprocess execution

```
import sys
import subprocess
result = subprocess.run([sys.executable, "-c", "print('This is a subprocess')"])
```

- sys.executable: This gives the path of the Python interpreter that is currently running the script.
- subprocess.run: This function runs the command described by the arguments
- -c: This flag tells the Python interpreter to run the string following it as a command.
- "print('This is a subprocess')": This is the command that will be run by the subprocess, which simply prints "This is a subprocess".

Send output and error to file from shell command

- SomeCommand >> SomeFile.txt

- SomeCommand &> SomeFile.txt

Controlling the Outputs

```
import subprocess

# Running a subprocess to execute a Python command

result = subprocess.run(["python", "-c", "print('This is a subprocess')"], capture_output=True)

# Printing the result object

print(result)
```

Output:

```
CompletedProcess(args=['python', '-c', "print('This is a subprocess')"], returncode=0, stdout=b'This is a subprocess\n', stderr=b'')
```

- returncode is 0, indicating that the subprocess executed successfully.
- stdout contains the captured output "This is a subprocess\n", in bytes format (indicated by b'').
- stderr is empty, as there were no errors.

- If you want to see just the output of the subprocess

```
print(result.stdout.decode().strip())
```

Output:

```
This is a subprocess
```

Stdout and stderr

- print(result.stdout)
- print(result.stderr)

Output:

```
b'This is a subprocess\n’
b''
```

Result in text

- text=True simplifies handling the input/output as strings, making the code easier to read and write when working with text data.
- Without text=True, you would need to handle the binary data and manually convert it to strings where needed.

```
import subprocess
# Running a subprocess to execute a Python command
result = subprocess.run(["python", "-c", "print('This
is a subprocess')"], capture_output=True, text=True)
# Printing the result object
print('output: ', result.stdout)
print('error: ', result.stderr)
```

- check=True parameter in subprocess.run() is used to control whether the function should raise an exception if the subprocess exits with a non-zero exit code, which typically indicates an error.
- When you set check=True, subprocess.run() will raise a subprocess.CalledProcessError if the subprocess returns a non-zero exit code. This is useful for detecting and handling errors in subprocesses without manually checking the exit code.

```
import subprocess
result = subprocess.run(["python", "-c", "print('Hello from subprocess')"], capture_output=True, text=True, check=True)
print('output: ', result.stdout)
print('error: ', result.stderr)
```

timeout

- timeout parameter in the subprocess.run() function sets a time limit for how long the process is allowed to run.
- If the process exceeds this time limit, a subprocess.TimeoutExpired exception is raised.

```
subprocess.run(["python", "-c", "import time; time.sleep(5)"],
capture_output=True, text=True)
```

  - But if we set the timeout parameter to less than five, we'll have an exception:

```
subprocess.run(["python", "-c", "import time; time.sleep(5)"],
capture_output=True, text=True, timeout=2)
```

timeout

```
import subprocess
try:
    # This will run successfully as it sleeps for 5 seconds
    subprocess.run(["python", "-c", "import time; time.sleep(5)"],
capture_output=True, text=True)
    # This will raise a TimeoutExpired exception because it will try to run for 5
seconds
    # but the timeout is set to 2 seconds
    subprocess.run( ["python", "-c", "import time; time.sleep(5)"],
capture_output=True,text=True, timeout=2)
except subprocess.TimeoutExpired as e:
    print(f"Process exceeded the timeout and was terminated: {e}")
```

```
Process exceeded the timeout and was terminated: Command '['python', '-c', 'import
time; time.sleep(5)']' timed out after 2 seconds
```

Inputting in subprocess

```
import subprocess
result = subprocess.run(
    ["python", "-c", "import sys; my_input=sys.stdin.read();
print(my_input)"],
    capture_output=True,
    text=True,
    input='my_text')
print(result.stdout)
```

- input='my_text':Provides the string 'my_text' as input to the command's stdin.
- Execution:The Python command will receive 'my_text' via stdin due to the input parameter.sys.stdin.read() will read this input and print(my_input) will output it.
- Expected Output:The result.stdout will contain the output of the command, which should be 'my_text':

Command line arguments

```python
import sys

my_input = sys.argv

def sum_two_values(a=int(my_input[1]), b=int(my_input[2])):
    return a + b

if __name__=="__main__":
    print(sum_two_values())
```

result = subprocess.run(["python", "my_script.py", "2", "4"], capture_output=True, text=True) print(result.stdout)

Problem Statement

- Parallel Backup system
- Suppose we have very important data that we want to backup in 3 different place.

1. Create a program to copy files of a folder to the specific folder.
2. Run this process all three places parallelly using subprocess.

```
import subprocess
import os
from pathlib import Path

def backup_files(source_folder, dest_folders):
    # Ensure the source folder exists
    if not os.path.exists(source_folder):
        raise FileNotFoundError(f"Source folder '{source_folder}' does not exist.")

    # Ensure all destination folders exist, create them if they don't
    for folder in dest_folders:
        Path(folder).mkdir(parents=True, exist_ok=True)

    # Create a list to hold subprocesses
    processes = []
```

```
    # Loop through each destination and start the copy process in parallel
    for dest_folder in dest_folders:
        command = ["cp", "-r", source_folder, dest_folder]
        print(f"Starting backup to {dest_folder}")
        process = subprocess.Popen(command)
        processes.append(process)

    # Wait for all processes to finish
    for process in processes:
        process.wait()
        print(f"Backup to {process.args[-1]} completed")

if __name__ == "__main__":
    source_folder = "/path/to/source/folder"
    dest_folders = [
        "/path/to/destination1",
        "/path/to/destination2",
        "/path/to/destination3"
    ]

    backup_files(source_folder, dest_folders)
```

```
 # Loop through each destination and start the copy process in parallel
    for dest_folder in dest_folders:
        command = ["cp", "-r", source_folder, dest_folder]
        print(f"Starting backup to {dest_folder}")
        process = subprocess.Popen(command)
        processes.append(process)

    # Wait for all processes to finish
    for process in processes:
        process.wait()
        print(f"Backup to {process.args[-1]} completed")

if __name__ == "__main__":
    source_folder = "/path/to/source/folder"
    dest_folders = [
        "/path/to/destination1",
        "/path/to/destination2",
        "/path/to/destination3"
    ]

    backup_files(source_folder, dest_folders)
```

Explanations of the problem statement:

- Imports:subprocess: To run external commands (cp in this case).
- os: To check if the source folder exists.
- Path from pathlib: To create destination directories if they don't exist.
- backup_files function:Takes a source_folder and a list of dest_folders as inputs.
- Checks if the source_folder exists. If not, it raises an error.
- Creates the destination folders if they don't already exist.
- Uses subprocess.Popen to start the copy operation (cp -r) for each destination folder in parallel.
- Collects the processes in a list and waits for each one to finish using process.wait().
- Main Block:Specifies the source and destination folders.
- Calls the backup_files function with these folders.