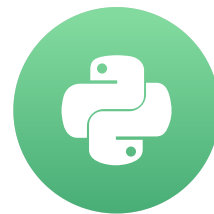


# Execute shell commands in subprocess

COMMAND LINE AUTOMATION IN PYTHON

**Noah Gift**

Lecturer, Northwestern & UC Davis & UC  
Berkeley | Founder, Pragmatic AI Labs



# Using subprocess.run

- Simplest way to run shell commands using Python 3.5+
- Takes a list of strings

```
subprocess.run(["ls", "-l"])
```

# Dealing with Byte Strings

- Byte Strings are default in subprocess

```
res = b'repl          24  0.0  0.0  36072  3144 pts/0    R+   03:15   0:00 ps aux\n'
print(type(res))
```

bytes

- Byte Strings decode

```
regular_string = res.decode("utf-8")
'repl          24  0.0  0.0  36072  3144 pts/0    R+   03:15   0:00 ps aux\n'
print(type(regular_string))
```

# Unix status codes

- Successful completion returns 0

```
ls -l  
echo $?  
0
```

- Unsuccessful commands return non-zero values

```
ls --bogus-flag  
echo $?  
1
```

# Checking status codes

- Run shell command and assign output

```
out = run(["ls", "-l"])
```

- CompletedProcess object

```
subprocess.CompletedProcess
```

- Check status code

```
print(out.returncode)
```

```
0
```

# Non-zero status codes in subprocess.run

- Successful status code

```
out = run(["ls", "-l"])  
print(out.returncode)
```

- Unsuccessful status code

```
bad_out = run(["ls", "--turbo"])  
print(bad_out.returncode)  
1
```

# Control flow for status codes

- Handling user input

```
good_user_input = "-l"  
out = run(["ls", good_user_input])
```

- Controlling flow based on response

```
if out.returncode == 0:  
    print("Your command was a success")  
else:  
    print("Your command was unsuccessful")
```

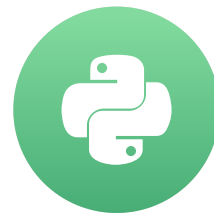
# Practicing executing shell commands

COMMAND LINE AUTOMATION IN PYTHON



# Capture output of shell commands

COMMAND LINE AUTOMATION IN PYTHON



**Noah Gift**

Lecturer, Northwestern & UC Davis & UC  
Berkeley | Founder, Pragmatic AI Labs

# Using the subprocess.Popen module

- Captures the output of shell commands
- In bash a directory listing using `ls`

```
bash-3.2$ ls
some_file.txt      some_other_file.txt
```

- In Python output can be captured with `Popen`

```
with Popen(["ls"], stdout=PIPE) as proc:
    out = proc.readlines()
print(out)

['some_file.txt', 'some_other_file.txt']
```

# "with" statement

- Context manager handles closing file

```
with open("somefile.txt", "r") as output:
```

```
# uses context manager  
with Popen(["ls", "/tmp"], stdout=PIPE) as proc:  
    # perform file operations
```

- Simplifies using Popen
- Also simplifies other Python statements like reading files.

# Breaking down a real example

```
# import Popen and PIPE to manage subprocesses
from subprocess import (Popen, PIPE)
```

```
with Popen(["ls", "/tmp"], stdout=PIPE) as proc:
    result = proc.stdout.readlines()
```

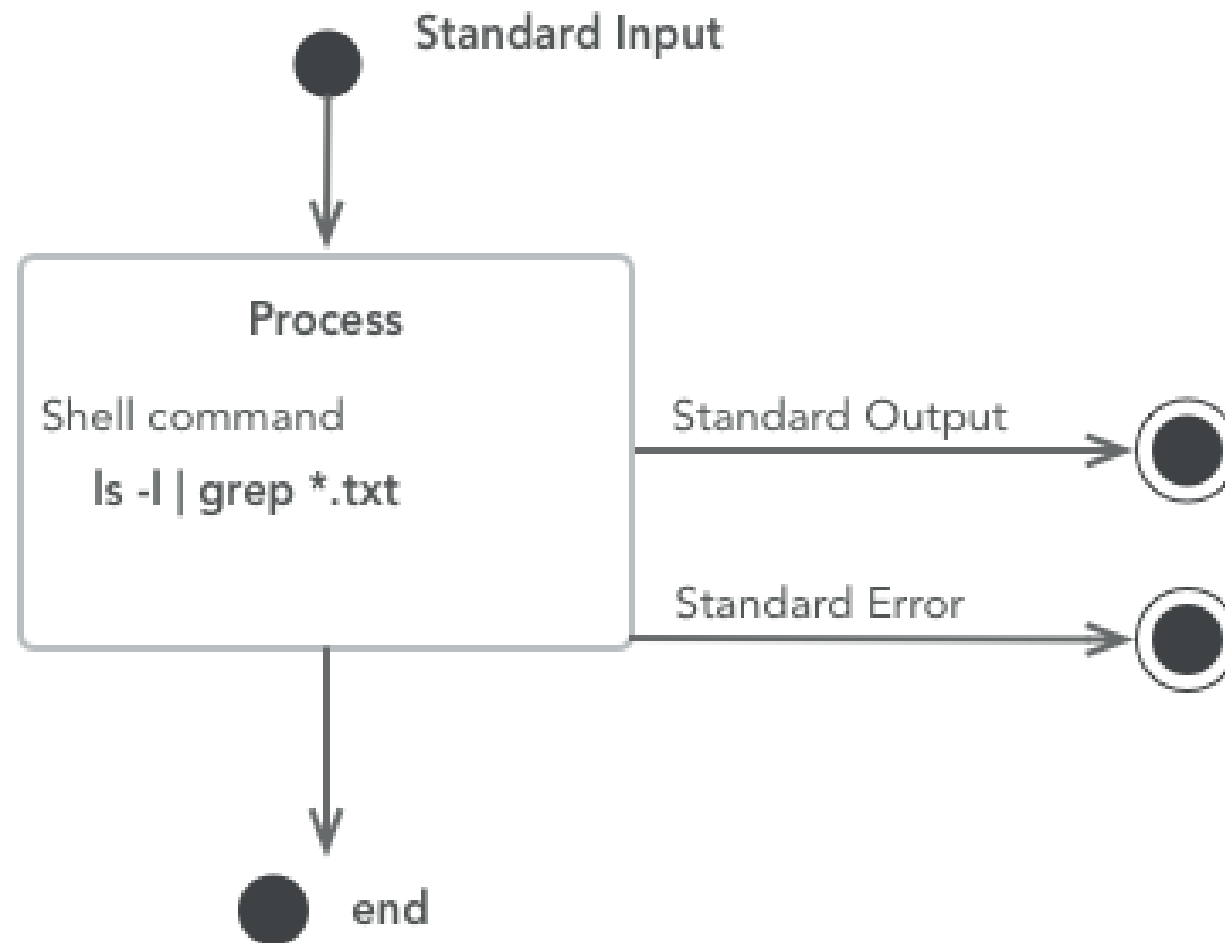
# Using communicate

- `communicate` : A way of communicating with streams of a process, including waiting.

```
proc = subprocess.Popen(...)
# Attempt to communicate for up to 30 seconds
try:
    out, err = proc.communicate(timeout=30)
except TimeoutExpired:
    # kill the process since a timeout was triggered
    proc.kill()
    # capture both standard output and standard error
    out, error = proc.communicate()
```

# Using PIPE

- **PIPE** : Connects a standard stream (stdin, stderr, stdout)
- One intuition about **PIPE** is to think of it as tube that connect to other tubes



# Required components of subprocess.Popen

- `stdout` : Captures output of command
- `stdout.read()` : returns output as a string
- `stdout.readlines()` : returns outputs as an interator
- `shell=False`
  - is default and recommended

```
# Unsafe!
```

```
with Popen("ls -l /tmp", shell=True, stdout=PIPE) as proc:
```

# Using stderr

- `stderr`: Captures shell `stderr` (error output)

```
with Popen(["ls", "/a/bad/path"], stdout=PIPE, stderr=PIPE) as proc:  
    print(proc.stderr.read())
```

stderr output

```
b'ls: /a/bad/path: No such file or directory\n'
```



# Analyzing Results

```
# Printing raw result  
print(result)
```

```
[b'bar.txt\n', b'foo.txt\n']
```

```
#print each file  
for file in result:  
    print(file.strip())
```

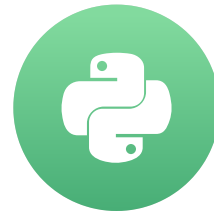
```
b'bar.txt'  
b'foo.txt'
```

# Practicing with the subprocess.Popen Class

COMMAND LINE AUTOMATION IN PYTHON

# Sending input to processes

COMMAND LINE AUTOMATION IN PYTHON



**Noah Gift**

Lecturer, Northwestern & UC Davis & UC Berkeley | Founder, Pragmatic AI Labs

# Using Unix Pipes as input

- Two ways of connecting input
  - `Popen` method

```
proc1 = Popen(["process_one.sh"], stdout=subprocess.PIPE)
Popen(["process_two.sh"], stdin=proc1.stdout)
```

- `run` method (Higher Level Abstraction)

```
proc1 = run(["process_one.sh"], stdout=subprocess.PIPE)
run(["process_two.sh"], input=proc1.stdout)
```

# Input Pipe from Unix

- Contents of the directory

```
ls -l
```

```
total 160
-rw-r--r--  1 staff  staff  13 Apr 15 06:56
-rw-r--r--  1 staff  staff  12 Apr 15 06:56 file_9.txt
```

- Sends output of one command to another

```
ls | wc
```

```
20      20     220
```

# The string language of Unix Pipes

- Strings are the language of shell pipes
- Pass strings via STDOUT

```
echo "never odd or even" | rev
```

```
neve ro ddo reven
```

# Translating between objects and strings

- Python objects contain
  - data
  - methods
- Unix strings are
  - data only
  - often columnar

# User input

- Bash uses `read` .
- Python uses `input` .
- Python can also accept input from command-line libraries.
- Subprocess can pipe input to scripts that wait for user input.



# Practicing Input

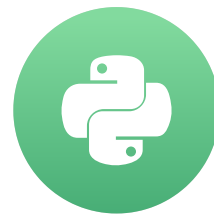
COMMAND LINE AUTOMATION IN PYTHON

# Passing arguments safely to shell commands

COMMAND LINE AUTOMATION IN PYTHON

**Noah Gift**

Lecturer, Northwestern & UC Davis & UC  
Berkeley | Founder, Pragmatic AI Labs



# User input is unpredictable

- Expected input to a script

```
"/some/dir"
```

- Actual input to a script

```
"/some/dir && rm -rf /all/your/dirs"
```

# Understanding shell=True in subprocess

- By default `shell=False`
- `shell=True` allows arbitrary code
- Best practice is to avoid `shell=True`

```
#shell=False is default  
run(["ls", "-l"], shell=False)
```

# Using the shlex module

- `shlex` can sanitize strings

```
shlex.split("/tmp && rm -rf /all/my/dirs")
```

```
['/tmp', '&&', 'rm', '-rf', '/all/my/dirs']
```

```
directory = shlex.split("/tmp")  
cmd = ["ls"]  
cmd.extend(directory)  
run(cmd, shell=True)
```

```
CompletedProcess(args=['ls', '/tmp'], returncode=0)
```

# Defaulting to items in a list

- Best practice is using a list
- Limits mistakes

```
with subprocess.Popen(["find", user_input, "-type", "f"],  
                      stdout=subprocess.PIPE) as find:  
  
    #do something else in Python....
```

# The problem with security by obscurity

- House key under the doormat
- Key cards for every door
- Integrated security is best

# Security best practices for subprocess

- Always use shell=False
- Assume all users are malicious
- Never use security by obscurity
- Always use the principle of least privilege
- Reduce complexity



# Security focused practice!

COMMAND LINE AUTOMATION IN PYTHON