UNIVERSITY of York

Using Python for shell scripts

Peter Hill

Outline

- Advantages/disadvantages of Python
- Running a parameter scan example
- Command line arguments
- Working with filesystem paths
- Working with string formatting

Why use Python?

- Nicer syntax:
  - Here's how to get the length of an array/list plus one in bash:

    ```
    $(( ${#array[@]} + 1 ))
    ```

  - and in Python:

    ```
    len(array) + 1
    ```

- Better data structures:
  - Associative arrays (`dicts` in Python) only in bash 4.0+
  - `classes` for encapsulating data and logic
- Error handling much easier in Python
- Easier to write portable Python vs portable shell scripts
  - e.g. `<( )` process substitution is bash-only
- Conditionals are more comprehensible in Python
  - e.g. `[[ -z "$foo" ]]` vs `if not foo`
- Testing is much easier!

Why not use Python?

- Not every machine has Python (and some only have Python 2)
  - Every *nix machine has some POSIX shell
  - Windows is a different matter...
- Might need to install external modules for Python
  - Only a problem on machines with e.g. IP whitelist
- Very simple things might be faster/easier using bash
  - e.g. `find . -name "*.inp" | xargs grep "nx = 4"`

Running a parameter scan for a simulation

Some different methods:

- Edit input file by hand, save a new copy
  - Very easy to make a mistake!
- Use `sed` and regular expressions to replace values in old file
  - Can take a long time to get that regular expression correct!
- Use variable substitution in bash to `echo` a string into a file
  - Careful about escaping variables!
- Use a template file/string and `format` it with Python

Typical things we might want to do in a shell script

Creating an input file for each set of parameters

- Parse arguments passed on command line
- Move about the file system
- Create/remove/copy files and directories
- Loop over multiple lists
- Read a file
- Replace text
- Write text to a file
- Run another program

Main

```
def create_directory():
    pass

def make_input_file():
    pass

def run_program():
    pass

for parameter in parameters:
    create_directory()
    make_input_file()
    run_program()
```

Maybe we can reuse things?

Traditional Python scripts

```
def create_directory():
    pass
...
if __name__ == "__main__":
    # Actually do work
```

But why?

- `__name__` for a file/module is only `__main__` when it is being run
- This allows us to not only run the program, but also import it to reuse the functions in other programs

A word about functions

- Wrapping logic up in functions is A Good Idea
- Enables reuse of bits of code
- Helps separate concerns
- Allows documentation and testing of individual functions

Best practices

```
def make_input_file(nx, species, dryrun=False, filename=None):
    """Some documentation
    Write down any assumptions about input parameters

    Returns: name of new input file
    """
    # Do stuff
```

Command line arguments

Not great in bash

```
while getopts ":n:" opt; do
  case ${opt} in
    n ) num_procs=$OPTARG ;;
    \? ) echo "Usage: scan [-n]" ;;
  esac
done
```

- Quickly becomes very complicated
- No support for long options
- Handling of options which require arguments is a pain

Command line arguments

- Can use built-in `argparse` module
  - Lots of other external modules to do this!
- Automatically handles `-h/--help` cases
- Allows us to specify expected type and number of arguments to an option
- Easy to specify both short and long forms
- Arguments are stored in the parameter name by default

Basic usage

```
import argparse
parser = argparse.ArgumentParser(description="Run a parameter scan")
parser.add_argument("-n", "--numprocs", type=int, default=1,
                    help="Number of processors")
args = parser.parse_args()
```

Command line arguments

Output

- Running "`scan --help`" then gives:

```
usage: scan [-h] [-n NUMPROCS]

Run a parameter scan

optional arguments:
  -h, --help            show this help message and exit
  -n NUMPROCS, --numprocs NUMPROCS
                        Number of processors
```

Command line arguments

Lots of options

```
parser.add_argument("inputfile", nargs=1,
                    help="""Positional argument
                    requiring exactly one argument""")

parser.add_argument("-n", "--dry-run", action="store_true",
                    default=False,
                    help="Set an optional flag to True")

parser.add_argument("--nx", nargs="+", dest="nx_list",
                    help="""Require at least one argument
                    if present, and store in a named variable")
```

Command line arguments

Accessing the arguments

```
results = parser.parse_args()

if results.flag:
    # Do something
if results.nx_list is not None:
    for nx in results.nx_list:
        # Iterate over parameters
```

Further reading

- https://pymotw.com/3/argparse/index.html
- https://docs.python.org/3/library/argparse.html

The pathlib module

- os and os.path modules more suited to lower-level operations
- pathlib makes manipulating paths much easier

Example

```
import pathlib
simpath = pathlib.Path().cwd()  # Current working directory
simpath.resolve()
# PosixPath('/data/user/simulation')
simpath.parent
# PosixPath('/data/user')
list(simpath.glob('*.inp'))
# [PosixPath('/data/user/simulation/template.inp'),
#  PosixPath('/data/user/simulation/C_nx4.inp')]
```

The pathlib module

Building paths

```
# Known in advance
run001 = simpath / 'run001'
print(run001)
# /data/user/simulation/run001

# Not known in advance
subdirs = ['nx', nx_value]
nx_path = simpath.joinpath(*subdirs)
print(nx_path)
# /data/user/simulation/nx/4
```

Making and removing directories

```
# Create a directory
nx_path.mkdir()
# Create a directory and its parents, don't throw if it already exists
nx_path.mkdir(parents=True, exist_ok=True)
# Delete a file (`rm`)
for temp_file in simpath.glob('*~'):
    temp_file.unlink()
# Delete an empty directory (`rm -r`)
simpath.rmdir()
```

Copying and renaming files

- `pathlib` doesn't provide a copy function
- Instead, we can use `shutil` module
- Also, we only need `str` here if we're not using Python 3.6

```
import shutil
restart_file = pathlib.Path("/data/user/old_simulation/restart")
destination = pathlib.Path("/data/user/simulation/")
shutil.copy(str(restart_file), str(destination))
```

- Just renaming or moving a file can be done with `pathlib`:

```
old_file = pathlib.Path("output.dat")
backup = old_file.with_suffix(".bak")
old_file.rename(backup)
```

Formatting text (“string interpolation”)

- Python now has three different ways of formatting strings:
  - C `printf` style: `print('%s' % "hello, world!")`
  - `format` string method: `print("{}".format("hello, world!"))`
  - “f-strings” (only in 3.6):

```
hello = "hello, world!"
print(f"{hello}")
```

- The `format` method is the most powerful and widely supported

Further reading

- https://pyformat.info/
- https://docs.python.org/3.5/library/string.html#formatstrings

Template files

```
# Dictionary with all our parameters in
parameters = {
    'nx': 4,
    'species': 'C',
}
# How we want new input files to be called
filename = "{species}_nx{nx}.inp"
# The "**" operator unpacks a dictionary into "key=value" pairs
new_inputfile = pathlib.Path(filename.format(**parameters))
# Read in template file and then write our formatted one
template_file = pathlib.Path('template.inp')
template = template_file.read_text()
new_inputfile.write_text(template.format(**parameters))
```

Template files

Output

Turns this. . . :

```
# template.inp
[grid]
nx = {nx}
[species]
name = {species}
```

. . . into this:

```
# C_nx4.inp
[grid]
nx = 4
[species]
name = 'C'
```

Other methods

- The `configparser` deals very well with “INI” style files like the above
- Allows treatment of such files very much like dictionaries

Running other programs

The subprocess module

```
import subprocess
output = subprocess.run(['mpirun', '-n', str(num_procs), 'runsim'])
```

- Arguments passed as a list of strings
- Avoids problems with shell quoting, etc.

Running other programs

Capturing output

- Sending the output into a pipe allows us to capture the output for later parsing

```
output = subprocess.run(['mpirun', '-n', str(num_procs), 'runsim'],
                        stdout=subprocess.PIPE)
# output.stdout is `bytes`, so we need to decode it into text
print(output.stdout.decode("utf-8"))
```

Further reading

- https://pymotw.com/3/subprocess/index.html
- https://docs.python.org/3.6/library/subprocess.html
- https://docs.python.org/3.6/library/subprocess.html#subprocess-replacements

Looping over multiple sets of parameters

- For the Cartesian product of lists, we can use `itertools.product`:

```
import itertools
nx_list = [4, 8]
species_list = ['C', 'N']
for nx, species in itertools.product(nx_list, species_list):
    print(nx, species)
# 4 C
# 4 N
# 8 C
# 8 N
```

- Lots of other methods for iterating over or combining sets of lists

Other useful things

Timestamps

- Very useful to keep track of when you ran something
  - Could be done directly in file name or directory structure
- Use the `datetime` module:

```
import datetime
print(datetime.datetime.today())
# 2018-01-25 09:48:58.141256
print("{:%a %b %d %H:%M}".format(today))
# Thu Jan 25 09:48
```

Keeping a log

- Also useful to keep track of *what* you ran as well as *when*
- Lots of options for this:
  - Plain text file
  - Excel spreadsheet
  - Pandas database

Comma-separated values (CSV)

- Simplest, actually useful file format:

```
Heading 1, Heading 2, Heading 3
value 1, value 2, value 3
value 1, value 2, value 3
value 1, value 2, value 3
```

Working with CSV files

Use the csv module

```
import csv

def write_heading():
    with open("simulation_log.csv", "w") as f:
        writer = csv.writer(f)
        writer.writerow(("Date", "nx", "species"))

def log_simulation(nx, species):
    with open("simulation_log.csv", "a") as f:
        writer = csv.writer(f)
        writer.writerow(("{}".format(today), nx, species))
```

Remote connections

paramiko + scp

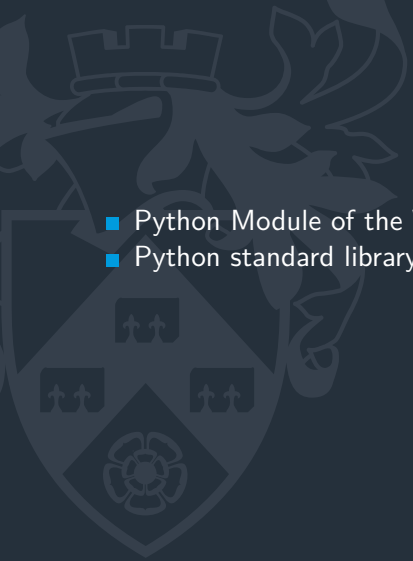
- Need two third-party modules, paramiko and scp, for transferring files:

```
from paramiko import SSHClient
from scp import SCPClient

ssh = SSHClient()
ssh.load_system_host_keys()
ssh.connect('example.com')

with SCPClient(ssh.get_transport()) as scp:
    scp.put('test.txt', 'test2.txt')
    scp.get('test2.txt')
```

Further reading



- Python Module of the Week: https://pymotw.com/3/index.html
- Python standard library: https://docs.python.org/3/library/index.html