Introduction to Scripting

Scripts, modules, interfaces and deployment

You might have hear the term "Python script" or "scripting". What this means is that Python programming code is typically organised in a file that can be run like a program. You are probably familiar with command line programs like `ls`, `cd` and `cat`. A Python script is similar to them, in that it can be run from the command line and takes input and output.

Conceptually, Python scripts are an important way of deploying code. Deployment means putting in place everything necessary for making code work in an environment: the interpreter program must be there, the operating system needs to know how to run code with the interpreter, all modules needed for the program need to be present and findable by the interpreter and the code must be able to accept input parameters.

The Python interpreter

Let's examine these elements in turn. Firstly, the interpreter. Python is run using a program that is called python or python3. The 3 in python3 refers to the fact that this is an interpreter for version 3 of the Python programming language. [1]

Then, how does does the Python interpreter end up on a computer? It is either installed as an operating system package (this would, on Linux or MacOS, typically installed `/usr/bin/python3`) or you can install it yourself with an installer like Miniconda. While "system python" (the one installed along with the operating system) has the advantage of not requiring any user intervention, the Miniconda installer will install a recent version of Python.

[1] In 2020 the Python community completed a more than 10 year process of transitioning from version 2 to version 3 of the programming language. As part of that transition, many operating systems have adopted the convention that the python command runs a version of the Python interpreter that understands Python version 3. To simplify the user experience, however, the Python community strongly recommends that the python3 command should be available wherever a Python version 3 interpreter is installed. In short, python and python3 might both interpret Python version 3 code, but python3 is *guaranteed* to understand Python version 3.

Modules and dependencies

Python comes with a standard library containing many useful modules to add to the basic features of the language. The broader community has created many more modules that do not come with Python and have to be installed. These modules (imported into your code with `import`) and the software that they depend are called a the "dependencies" of a program (or script). They need to be available wherever the program needs to work.

If you have installed Python using Miniconda or Anaconda, your installation comes with the `conda` tool for installing packages and managing environments[2]. You might also have installed a compatible, but faster, tool called mamba. Besides suiting your own needs for package installation, these tools take on the job of "dependency management", that is ensuring that you can make a list of required packages and have them installed wherever your script runs.

Connecting the interpreter with the code

When it comes to Python, "running the code" means sending Python code to the Python interpreter program. In the Jupyter notebook environment the notebook server takes the code you submit (when you run one or more code cells) and sends it to a copy of python3. On the command line, you might run python3 directly, for example:

```
$ python3 myscript.py
```

Or you might make a script executable. In Linux and Linux-like environments, you can start your script with a "shebang line" to indicate how it should be run, for example, the contents of myscript.py could be:

```
#!/usr/bin/env python3

print("Hello World")
```

The `#!/usr/bin/env python3` indicates that `/usr/bin/env` should be used to find `python3` and then run the script. This only works if the script is both executable and findable. To make a script executable, one needs to give it executable permission with `chmod` e.g.

```
$ chmod a+x myscript.py
```

[2]Conda "environments" are workspaces with collections of packages installed. For example, you might make an environment with Jupyter lab and associated data science tools in it, and another with tools related to your domain of bioinformatics.

The "+x" means make `myscript.py` executable and the "a" means "let anyone run the script". Finally findable: one way to run the script is to give a path, either relative or absolute, to the script, e.g.

```
$ ./myscript.py
$ /tools/software/other_script.py
```

Another way to make the script findable is to put it in one of the directories mentioned in the PATH environment variable. PATH is a colon (":") separated list of directories where the shell looks for scripts or programs to run. It contains directories like `/usr/bin/` and `/usr/sbin` that are not writeable by ordinary users but it can be extended to include user directories. For example if a script `myscript.py` is executable and in `$HOME/bin` (i.e. the `~/bin` directory) then one can add that to PATH:

```
$ PATH=$PATH:$HOME/bin
$ export PATH
$ myscript.py
```

Notice that to add to PATH you use `PATH=$PATH:$HOME/bin`. If you didn't include the existing PATH variable in this line you would *replace* PATH so that it *only* contained `$HOME/bin`. This is almost certainly not what you want because it would mean that programs in `/usr/bin` are no longer findable and tools like `ls` would stop working.

Passing parameters to the code

The final piece of the puzzle being described here is passing parameters to the code. Just like Python functions, blocks of Python code (whether in Jupyter notebooks or as scripts) need input and output. In the Jupyter lab or notebook environment, input is provided by directly setting variables in a code cell. This is because one typically uses Jupyter for "exploratory data analysis". Scripts and programs are more often used when one is building workflows. A workflow is a collection of tools that implement an analysis or a repetitive task. For example, you might write a script to run some steps of sequence analysis and then use the same script to analyse many different sequences.

When you are running a script on the command line, the parameters that you type on the command line provide input to Python. For example:

```
$ python3 say_hello.py 'Roger Rabbit'
Hello Roger Rabbit
```

The command line here starts with `python3` which runs the Python interpreter, reading in `say_hello.py`. The rest of the command line is passed as input to `say_hello.py` and the

script prints "Hello Roger Rabbit". Notice the quotes around 'Roger Rabbit'. They are there because, before Python or the `say_hello.py` has a chance to see the command line contents, it is first interpreted the shell. The shell would consider "Roger Rabbit" to be two parameters if it is not surrounded by quotes.

Inside Python the command line parameters are made visible to as elements of a special list, `argv` in the `sys` module. The name `argv` standards for "argument vector" (a vector is just another word for a list). The `sys` module is a standard Python module that can be imported with `import`. In `argv` the command line parameters start from the 2nd element of the list. That is because the first element (`argv[0]`) contains the name of the Python script that is running. Here is an example of how the `say_hello.py` script could be implemented:

```
#!/usr/bin/env python3

import sys

name = sys.argv[1]
print("Hello", name)
```

The `argv` list is a list of strings. This is an important difference when compare to parameters to Python functions: the script must interpret each parameter (aka argument). So a number would need to be converted into a numeric type with `int()` or `float()` and a file would be represented by a filename rather than a file object. For example, this script takes three parameters: two numbers to add and the name of a file to write the result to. It is called `add_nums.py`:

```
#!/usr/bin/env python3

import sys

num1 = float(sys.argv[1])
num2 = float(sys.argv[2])
output_filename = sys.argv[3]

total = num1 + num2
output_file = open(output_filename, 'w')
print(total, file=output_file)
```

This is an example of running the script:

```
$ chmod a+x add_nums.py
$ ./add_nums.py 10 10 output.txt
```

The file `output.txt` will now contain the string `"20\n"`. The `add_nums.py` script illustrate that when dealing with a script, information about where to put output is typically specified as part of the input parameters. This is because there is no `return` from scripts. The only value that a script can return is a single number called the "exit code". The exit code is, by convention, zero if everything proceeded correctly and some other number otherwise. If you do not specifically set the exit code in your script, Python will return an exit code of zero for a script that completes with no errors and one for a script that ends with an error. In the bash shell the exit code of the most recently executed command is stored in the `$?` environment variable. This can be illustrated using these examples, firstly `goodscript.py`:

```
#!/usr/bin/env python3

print("I am a good script")
```

then `badscript.py`:

```
#!/usr/bin/env python3

print("this will fail", 1 + "one")
```

and finally running these scripts:

```
$ python3 goodscript.py
I am a good script
$ echo $?
0
$ python badscript.py
Traceback (most recent call last):
  File "/home/pvh/Documents/intro_to_python/badscript.py", line 3, in <module>
    print("this will fail", 1 + "one")
TypeError: unsupported operand type(s) for +: 'int' and 'str'
$ echo $?
1
```

The exit code can be set using the `exit` function in the `sys` module. For example one, this script (`add_nums2.py`) checks if the user provided the correct number of parameters before adding two numbers, and if the correct number of parameters is not provided it prints a message to the `stderr`[3]:

[3] In Linux terminal output is divided into two "streams", `stdout` and `stderr`. Normal output from `print()` goes to `stdout` and error messages are written to `stderr` by convention. The two output streams in show up on the same terminal but can be dealt with independently, for example, by shell redirects where `>` redirects the `stdout` stream and `2>` redirects the `stderr` stream. In Python these two streams are represented by

```
#!/usr/bin/env python3

import sys

if len(sys.argv) != 4:
    # 3 elements expected because parameters start at argv[1]
    print("Error: expected 3 parameters", file=sys.stderr)
    sys.exit(1)

num1 = float(sys.argv[1])
num2 = float(sys.argv[2])
output_filename = sys.argv[3]

total = num1 + num2
output_file = open(output_filename, 'w')
print(total, file=output_file)
```

The `exit` function can also take a string as a parameter. If it is given a string parameter the program will print the string to `stderr` and exit with exit code 1. For example (`add_nums3.py`):

```
#!/usr/bin/env python3

import sys

if len(sys.argv) != 4:
    # 3 elements expected because parameters start at argv[1]
    sys.exit("Error: expected 3 parameters")

num1 = float(sys.argv[1])
num2 = float(sys.argv[2])
output_filename = sys.argv[3]

total = num1 + num2
output_file = open(output_filename, 'w')
print(total, file=output_file)
```

In practice the output produced by these two methods is identical:

`stdout` and `stderr` file objects, both of which are found in the `sys` module.

```
$ python add_nums2.py
Error: expected 3 parameters
$ python add_nums3.py
Error: expected 3 parameters
```

Practice

Write a script that reads in a name and a greeting from the command line parameters and prints the greeting and name to the user, with an exclamation mark ("!") after the greeting.

Call the script `greet.py`. Here is an example of running the script:

```
$ python3 greet.py Hello John
Hello John!
$ python3 greet.py Ahoy Captain
Ahoy Captain!
```