software carpentry

Programming with Python

Command-Line Programs

Learning Objectives

- Use the values of command-line arguments in a program.
- Handle flags and files separately in a command-line program.
- Read data from standard input in a program so that it can be used in a pipeline.

The IPython Notebook and other interactive tools are great for prototyping code and exploring data, but sooner or later we will want to use our program in a pipeline or run it in a shell script to process thousands of data files. In order to do that, we need to make our programs work like other Unix command-line tools. For example, we may want a program that reads a dataset and prints the average inflammation per patient.

Switching to Shell Commands

In this lesson we are switching from typing commands in a Python interpreter to typing commands in a shell terminal window (such as bash). When you see a `$` in front of a command that tells you to run that command in the shell rather than the Python interpreter.

This program does exactly what we want - it prints the average inflammation per patient for a given file.

```
$ python readings.py --mean inflammation-01.csv
5.45
5.425
6.1
...
6.4
7.05
5.9
```

We might also want to look at the minimum of the first four lines

```
$ head -4 inflammation-01.csv | python readings.py --min
```

or the maximum inflammations in several files one after another:

```
$ python readings.py --max inflammation-*.csv
```

Our scripts should do the following:

1. If no filename is given on the command line, read data from standard input.
2. If one or more filenames are given, read data from them and report statistics for each file separately.
3. Use the `--min`, `--mean`, or `--max` flag to determine what statistic to print.

To make this work, we need to know how to handle command-line arguments in a program, and how to get at standard input. We'll tackle these questions in turn below.

Command-Line Arguments

Using the text editor of your choice, save the following in a text file called `sys-version.py`:

```
import sys
print('version is', sys.version)
```

The first line imports a library called `sys`, which is short for “system”. It defines values such as `sys.version`, which describes which version of Python we are running. We can run this script from the command line like this:

```
$ python sys-version.py
```

```
version is 3.4.3+ (default, Jul 28 2015, 13:17:50)
[GCC 4.9.3]
```

Create another file called `argv-list.py` and save the following text to it.

```
import sys
print('sys.argv is', sys.argv)
```

The strange name `argv` stands for “argument values”. Whenever Python runs a program, it takes all of the values given on the command line and puts them in the list `sys.argv` so that the program can determine what they were. If we run this program with no arguments:

```
$ python argv-list.py
```

```
sys.argv is ['argv-list.py']
```

the only thing in the list is the full path to our script, which is always `sys.argv[0]`. If we run it with a few arguments, however:

```
$ python argv-list.py first second third
```

```
sys.argv is ['argv-list.py', 'first', 'second', 'third']
```

then Python adds each of those arguments to that magic list.

With this in hand, let's build a version of `readings.py` that always prints the per-patient mean of a single data file. The first step is to write a function that outlines our implementation, and a placeholder for the function that does the actual work. By convention this function is usually called `main`, though we can call it whatever we want:

```
$ cat readings-01.py
```

```
import sys
import numpy

def main():
    script = sys.argv[0]
    filename = sys.argv[1]
    data = numpy.loadtxt(filename, delimiter=',')
    for m in data.mean(axis=1):
        print(m)
```

This function gets the name of the script from `sys.argv[0]`, because that's where it's always put, and the name of the file to process from `sys.argv[1]`. Here's a simple test:

```
$ python readings-01.py inflammation-01.csv
```

There is no output because we have defined a function, but haven't actually called it. Let's add a call to `main`:

```
$ cat readings-02.py
```

```
import sys
import numpy

def main():
    script = sys.argv[0]
    filename = sys.argv[1]
    data = numpy.loadtxt(filename, delimiter=',')
    for m in data.mean(axis=1):
        print(m)

main()
```

and run that:

```
$ python readings-02.py inflammation-01.csv
```

```
5.45
5.425
6.1
5.9
5.55
6.225
5.975
6.65
6.625
6.525
6.775
5.8
6.225
5.75
5.225
```

```
6.3
6.55
5.7
5.85
6.55
5.775
5.825
6.175
6.1
5.8
6.425
6.05
6.025
6.175
6.55
6.175
6.35
6.725
6.125
7.075
5.725
5.925
6.15
6.075
5.75
5.975
5.725
6.3
5.9
6.75
5.925
7.225
6.15
5.95
6.275
5.7
6.1
6.825
5.975
6.725
5.7
6.25
6.4
7.05
5.9
```

The Right Way to Do It

If our programs can take complex parameters or multiple filenames, we shouldn't handle `sys.argv` directly. Instead, we should use Python's `argparse` library, which handles common cases in a systematic way, and also makes it easy for us to provide sensible error messages for our users. We will not cover this module in this lesson but you can go to Tshepang Lekhonkhobe's Argparse tutorial that is part of Python's Official Documentation.

Handling Multiple Files

The next step is to teach our program how to handle multiple files. Since 60 lines of output per file is a lot to page through, we'll start by using three smaller files, each of which has three days of data for two patients:

```
$ ls small-*.csv
```

```
small-01.csv small-02.csv small-03.csv
```

```
$ cat small-01.csv
```

```
0,0,1
0,1,2
```

```
$ python readings-02.py small-01.csv
```

```
0.333333333333
1.0
```

Using small data files as input also allows us to check our results more easily: here, for example, we can see that our program is calculating the mean correctly for each line, whereas we were really taking it on faith before. This is yet another rule of programming: *test the simple things first*.

We want our program to process each file separately, so we need a loop that executes once for each filename. If we specify the files on the command line, the filenames will be in `sys.argv`, but we need to be careful: `sys.argv[0]` will always be the name of our script, rather than the name of a file. We also need to handle an unknown number of filenames, since our program could be run for any number of files.

The solution to both problems is to loop over the contents of `sys.argv[1:]`. The '1' tells Python to start the slice at location 1, so the program's name isn't included; since we've left off the upper bound, the slice runs to the end of the list, and includes all the filenames. Here's our changed program `readings-03.py`:

```
$ cat readings-03.py
```

```
import sys
import numpy

def main():
    script = sys.argv[0]
    for filename in sys.argv[1:]:
        data = numpy.loadtxt(filename, delimiter=',')
        for m in data.mean(axis=1):
            print(m)

main()
```

and here it is in action:

```
$ python readings-03.py small-01.csv small-02.csv
```

```
0.333333333333
1.0
13.6666666667
11.0
```

The Right Way to Do It

At this point, we have created three versions of our script called `readings-01.py`, `readings-02.py`, and `readings-03.py`. We wouldn't do this in real life: instead, we would have one file called `readings.py` that we committed to version control every time we got an enhancement working. For teaching, though, we need all the successive versions side by side.

Handling Command-Line Flags

The next step is to teach our program to pay attention to the `--min`, `--mean`, and `--max` flags. These always appear before the names of the files, so we could just do this:

```
$ cat readings-04.py
```

```
import sys
import numpy

def main():
    script = sys.argv[0]
    action = sys.argv[1]
    filenames = sys.argv[2:]

    for f in filenames:
        data = numpy.loadtxt(f, delimiter=',')

        if action == '--min':
            values = data.min(axis=1)
        elif action == '--mean':
            values = data.mean(axis=1)
        elif action == '--max':
            values = data.max(axis=1)

        for m in values:
            print(m)

main()
```

This works:

```
$ python readings-04.py --max small-01.csv
```

```
1.0
2.0
```

but there are several things wrong with it:

1. `main` is too large to read comfortably.
2. If we do not specify at least two additional arguments on the command-line, one for the **flag** and one for the **filename**, but only one, the program will not throw an exception but will run. It assumes that the file list is empty, as `sys.argv[1]` will be considered the `action`, even if it is a filename. Silent failures like this are always hard to debug.
3. The program should check if the submitted `action` is one of the three recognized flags.

This version pulls the processing of each file out of the loop into a function of its own. It also checks that `action` is one of the allowed flags before doing any processing, so that the program fails fast:

```
$ cat readings-05.py
```

```
import sys
import numpy

def main():
    script = sys.argv[0]
    action = sys.argv[1]
    filenames = sys.argv[2:]
    assert action in ['--min', '--mean', '--max'], \
           'Action is not one of --min, --mean, or --max: ' + action
    for f in filenames:
        process(f, action)

def process(filename, action):
    data = numpy.loadtxt(filename, delimiter=',')

    if action == '--min':
        values = data.min(axis=1)
    elif action == '--mean':
        values = data.mean(axis=1)
    elif action == '--max':
        values = data.max(axis=1)

    for m in values:
        print(m)

main()
```

This is four lines longer than its predecessor, but broken into more digestible chunks of 8 and 12 lines.

Handling Standard Input

The next thing our program has to do is read data from standard input if no filenames are given so that we can put it in a pipeline, redirect input to it, and so on. Let's experiment in another script called `count-stdin.py`:

```
$ cat count-stdin.py
```

```
import sys

count = 0
for line in sys.stdin:
    count += 1

print(count, 'lines in standard input')
```

This little program reads lines from a special “file” called `sys.stdin`, which is automatically connected to the program’s standard input. We don’t have to open it — Python and the operating system take care of that when the program starts up — but we can do almost anything with it that we could do to a regular file. Let’s try running it as if it were a regular command-line program:

```
$ python count-stdin.py < small-01.csv
```

```
2 lines in standard input
```

A common mistake is to try to run something that reads from standard input like this:

```
$ python count_stdin.py small-01.csv
```

i.e., to forget the `<` character that redirect the file to standard input. In this case, there’s nothing in standard input, so the program waits at the start of the loop for someone to type something on the keyboard. Since there’s no way for us to do this, our program is stuck, and we have to halt it using the `Interrupt` option from the `Kernel` menu in the Notebook.

We now need to rewrite the program so that it loads data from `sys.stdin` if no filenames are provided. Luckily, `numpy.loadtxt` can handle either a filename or an open file as its first parameter, so we don’t actually need to change `process`. Only `main` changes:

```
$ cat readings-06.py
```

```
import sys
import numpy

def main():
    script = sys.argv[0]
    action = sys.argv[1]
    filenames = sys.argv[2:]
    assert action in ['--min', '--mean', '--max'], \
           'Action is not one of --min, --mean, or --max: ' + action
    if len(filenames) == 0:
        process(sys.stdin, action)
    else:
        for f in filenames:
            process(f, action)

def process(filename, action):
    data = numpy.loadtxt(filename, delimiter=',')

    if action == '--min':
        values = data.min(axis=1)
    elif action == '--mean':
        values = data.mean(axis=1)
    elif action == '--max':
        values = data.max(axis=1)

    for m in values:
        print(m)

main()
```

Let’s try it out:

```
$ python readings-06.py --mean < small-01.csv
```

```
0.333333333333
1.0
```

That’s better. In fact, that’s done: the program now does everything we set out to do.

Arithmetic on the command line

Write a command-line program that does addition and subtraction:

```
$ python arith.py add 1 2
```

```
3
```

```
$ python arith.py subtract 3 4
```

```
-1
```

Finding particular files

Using the `glob` module introduced earlier, write a simple version of `ls` that shows files in the current directory with a particular suffix. A call to this script should look like this:

```
$ python my_ls.py py
```

```
left.py
right.py
zero.py
```

Changing flags

Rewrite `readings.py` so that it uses `-n`, `-m`, and `-x` instead of `--min`, `--mean`, and `--max` respectively. Is the code easier to read? Is the program easier to understand?

Adding a help message

Separately, modify `readings.py` so that if no parameters are given (i.e., no action is specified and no filenames are given), it prints a message explaining how it should be used.

Adding a default action

Separately, modify `readings.py` so that if no action is given it displays the means of the data.

A file-checker

Write a program called `check.py` that takes the names of one or more inflammation data files as arguments and checks that all the files have the same number of rows and columns. What is the best way to test your program?

Counting lines

Write a program called `line-count.py` that works like the Unix `wc` command:

- If no filenames are given, it reports the number of lines in standard input.
- If one or more filenames are given, it reports the number of lines in each, followed by the total number of lines.