Python Lab

BD2K-LINCS: Computer Programming for Big Data Biomedicine

Programming with Python

Since the best way to learn how to program is to write code, this lab will focus on a common task in big data biomedical research: parsing and cleaning high-throughput gene expression profiling data. The goal of this lab is not to provide you with an in-depth understanding of Python but rather to give you a feel for Python and what you can do with it.

In this lab, we will parse a text file containing raw data, clean it, and output it into a new file. We will using a dataset from the Gene Expression Omnibus (GEO), an NCBI-run repository of gene expression profiling data. After the lab, you will have a working parser that can be used on many GEO DataSets.

Why Python? As a researcher, you may want several tools in your tool belt: a language that is good for statistical or mathematical computing, such as R, MatLab, or Mathematica; some basic knowledge of web development (HTML, CSS, and JavaScript) for sharing your work online; and a general-purpose programming language for writing scripts and programs. Python is a good choice for this last tool because it...

- Is free and runs on nearly every machine.
- Is well-documented: https://docs.python.org/3/.
- Has a large community of users, particularly in academia, the sciences, research, and finance.
- Has many libraries for high-performance numerical computing, such as NumPy, SciPy, and Pandas.
- Is the language most frequently used in introductory computer science courses.

Installation

Please install Python 3:

1. Go to https://www.python.org/downloads/.
2. Select the latest release (Version 3.4.3 as of March 2015).
3. Follow the installation instructions.

Once you have Python 3 installed, open a console (also called a "terminal" in OS X or Linux or a "command prompt" in Windows) and then type `python3`. You will then see something like this:

```
$ python3
Python 3.4.2 (v3.4.2:ab2c023a9432, Oct  5 2014, 20:42:22)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

This means Python 3 is properly installed. To exit, type `exit()` and press enter.

Download the dataset

Go to the GEO DataSet we will be using: http://www.ncbi.nlm.nih.gov/sites/GDSbrowser?acc=GDS1001. This GEO webpage contains basic information about the experiment, plus links to download the raw data in plain text format or Simple Omnibus Format in Text (SOFT). On the right-hand side, you'll see a download link called "DataSet SOFT File". Download that file and save it in a directory called `python-lab`.

Parsing gene expression data

The REPL

Now that we have our dataset, we can begin programming. In your console, navigate to the `python-lab` directory. In OS X, Linux, and Windows, the command to change directories is `cd`; for example:

```
$ cd python-lab
```

You may also give the full path. In OS X and Linux, that would look something like this:

```
$ cd /Users/<user>/python-lab
```

Now, start Python's interactive interpreter. Type:

```
$ python3
```

You should see the same interactive interpreter as you did when you installed Python. This program is called the REPL. "REPL" stands for "Read-eval-print loop"; anything you type into the REPL will be evaluated and executed by the Python interpreter. The ">>>" indicates that Python is waiting for user input.

Before we begin writing our data parsing script, we should appreciate how powerful a tool the REPL is. To evaluate an expression in the REPL, simply type it and then press enter. For example:

```
>>> 4+4*2
12
>>>
```

Notice that Python prints the evaluated expression on a new line, without angle brackets, and then prompts you for input again. In Python, the binary operations `+`, `-`, `*`, and `/` work as you might expect. Next, enter a *string*, a sequence of characters, into the REPL:

```
>>> 'This is a string.'
'This is a string.'
```

Python is particularly good at string manipulation. For just a taste, see how easy it is to manipulate them in a few intuitive ways:

```
>>> 'hello ' + 'world!'
'hello world!'
```

```
>>> 'I ' + 'really ' * 6 + 'like you'
'I really really really really really really like you'
```

Opening files

First, we want to open the dataset. Python makes this easy. Type

```
>>> dataset = open('GDS1001.soft', 'r')
```

and press enter. You may wonder if anything happened. The REPL is silent but it created an object representing the text file and stored in it a variable named `dataset`. You can open many files in Python using the built-in function `open()`.

Variables

Variables are names associated with values in a computer. We just assigned the data in the file to a variable `dataset` using the assignment operator `=`. See for yourself with another example:

```
>>> num = 9
>>> num
9
```

The above assignment binds the value 9 to the variable `num`. Next, when we typed `num` into the REPL, Python gave us back the value. Let's see what `dataset` is:

```
>>> dataset
<class '_io.TextIOWrapper'>
```

For loops

If you are familiar with another programming language, you may expect looping over a collection of data to be verbose. For example, to iterate over an array in JavaScript, one does the following:

```
for (var i = 0; i < my_array.length; i++) console.log(my_array[i]);
```

Python's syntax for looping is clean and the language defaults to iterating over the values in a collection, not the indices. Here is how to loop over every line in `dataset`.

```
>>> for line in dataset: print(line)
...
```

You'll need to press enter twice for the data to be printed. Also notice that if you try to run this command a second time, it doesn't work. If we want to iterate over the file again, we need to open it again.

Collections

Printing every line was easy, but now we want to do something with the data. Python provides some simple data structures for grouping together multiple values. These data structures are called *collections*. For the assignment, we will use a collection called a `list`. A list is a sequence of values. Order is significant and uniqueness is not enforced. Lists are comma-separated and grouped by brackets:

```
>>> x = [1,2,5]
```

Python does not care about the kind of data you store in a list:

```
>>> y = ['a', 29, [1]]
```

And we can add lists...

```
>>> x + y
[1, 2, 5, 'a', 29, [1]]
```

...calculate their length...

```
>>> len(x)
3
```

...and sum them...

```
>>> sum(x)
8
```

...and add to them:

```
>>> x.append(6)
>>> x
[1, 2, 5, 6]
```

Now, let's create a `list` and store every line of our file in it. First we need to reopen the dataset file, then we need to create an empty list, then we need to iterate over every line in the file, and finally we need to store every line in our `list`. This is the code to do all that:

```
>>> dataset = open('GDS1001.soft', 'r')
>>> lines = []
>>> for line in dataset: lines.append(line)
...
```

Now we have successfully stored every line in the file in a data structure. To verify, let's spot check the list by indexing it. An *index* is a number that maps to a value in a collection of values. In the REPL, execute these commands:

```
>>> line[0]
'^DATABASE = Geo\n'

>>> line[100]
'100069_at\tCyp2f2\t3159.3\t2305.7\t5035.4\t4045.1\n'
```

In Python, a `list` is 0-indexed, meaning that the two lines correspond to the 1 and 101 lines of the file. Many Python data types can be indexed; for example:

```
>>> 'Queens, NYC'[6]
','
```

Built-in functions

The functions `open()`, `print()`, `len()`, and `sum()` are called built-in functions. They are globally available to you in your programs and operate on a wide range of data types. A full list of built-ins can be found here: https://docs.python.org/3/library/functions.html.

Advanced indexing and slicing

Python's indexing is very expressive. For example, we can *slice* the first 5 lines from the `list` like this...

```
>>> lines[1:6]
['!Database_name = Gene Expression Omnibus (GEO)\n', '!Database_institute = NCBI
NLM NIH\n', '!Database_web_link = http://www.ncbi.nlm.nih.gov/geo\n',
'!Database_email = geo@ncbi.nlm.nih.gov\n', '!Database_ref = Nucleic Acids Res.
2005 Jan 1;33 Database Issue:D562-6\n']
```

...and the last line like this:

```
>>> lines[-1]
'!dataset_table_end\n'
```

Now notice that the first few dozen lines of the file contain metadata. For this assignment, we aren't interested in that metadata. Notice that the raw expression data starts after line 40, "`!dataset_table_begin`". How can we programmatically remove it? In Python, this is also fairly straightforward:

```
>>> lines.index('!dataset_table_begin\n')
39
```

The `index()` function returns a number indicating where in the `list` the value is. Notice we include the newline character "`\n`" because Python added that character to the end of every line in `lines`. If we store 39 in a variable, we can more easily express what we want to do next:

```
>>> idx = lines.index('!dataset_table_begin\n')
>>> lines = lines[idx+1:]
```

Notice we can omit the second index in our slice notation, and Python does what you might expect. Also, make sure you add +1 since we want to omit the last metadata line! We can verify this by checking the length of the new `list` using `len()`.

Comprehensions

One of Python's most expression notations is called a *comprehension*. Comprehensions are probably new to most students and even some experienced Python programmers. But they are a compact, concise, and powerful notation that closely resembles their mathematical counterpart.

Let's say we have a `list` of numbers and we want to multiply each number in the `list` by 2. With a comprehension, this code is concise:

```
>>> num = [1,2,3,4,5]
>>> [2*x for x in nums]
[2, 4, 6, 8, 10]
```

Compare this to the mathematical notation of defining a set:

```
{ 2x : x ∈ {1,2,3,4,5} }
```

Let's try a few more examples:

```
>>> list_of_lists = [[2,4,6], [1,3,5], [9,9,9]]
>>> [sum(x) for x in list_of_lists]
[12, 9, 27]

>>> alpha = ['a','b','c','d']
>>> [c+c for c in alpha]
['aa', 'bb', 'cc', 'dd']
```

Now that we have comprehensions, let's clean up every line in the file:

```
>>> lines = [line[:-1].split('\t') for line in lines]
```

This line is dense, so let's take examine each part of it. First, notice that we are iterating over every line in `lines`. Next, notice that every line is sliced to remove the newline character. Try this notation yourself:

```
>>> ['a', 'b', 'c', 'd'][:-1]
['a', 'b', 'c']
```

Finally, we split the line on the tab character. `split()` operates on a string, takes a separator as an argument, and returns a list:

```
>>> 'alea.jacta.est'.split('.')
['alea', 'jacta', 'est']
```

In summary, `lines` is now a `list` of lists, in which every value in the inner `list` is a string representing the probe ID, gene symbol, and raw expression data.

```
>>> lines[99]
['100130_at', 'Jun', '466.2', '579.3', '1644.8', '1431.3']
```

Now that we are becoming comfortable with comprehensions, we can quickly see how to parse this data efficiently. For example, how would we trim the first symbol—the probe ID, which we do not care about—from every line?

```
>>> lines = [line[1:] for line in lines]
>>> lines[99]
['Jun', '466.2', '579.3', '1644.8', '1431.3']
```

Writing scripts

So far, we've been working in the REPL. The REPL is great for quickly exploring computations in Python. But what if we want to save our work in a file? Any filename ending in `.py` is executable by the Python interpreter. For example, if we have a file named `example.py`, we would execute it from the console by typing:

```
$ python3 example.py
```

Create a file called `parser.py`. On OS X and Linux, you type:

```
$ touch parser.py
```

On Windows, you type:

```
$ type NUL > parser.py
```

Open the file in your favorite text editor. Now we can copy the snippets of code we have been using into this file. Your `parser.py` should look something like this:

```
"""This module parses a GEO DataSet SOFT file.
"""

# Open SOFT file and initialize an empty list.
data = open('GDS1001.soft', 'r')
lines = []

# Iterate over every line in the file, storing it in the list.
for line in data:
    lines.append(line)

# Remove lines containing just metadata.
idx = lines.index('!dataset_table_begin\n')
lines = lines[idx+1:]

# Remove the newline character and split on the tab, creating a list of lists.
lines = [line[:-1].split('\t') for line in lines]

# Remove the first element from every list.
lines = [line[1:] for line in lines]
```

Now try executing the file. You'll notice there is no output. If you want to "see" what the program is doing, use the built-in function `print()` to print data to the console.

Validating and cleaning our data

We have a working script that opens a dataset file, parses every line, and discards metadata. Hopefully, you have seen how expressive Python can be. But there are a few clean up steps we need to perform, and it is a bit easier to express them in a `for loop`. Notice near the end of the file, some lines have the string "`null`" where an expression value should be. Also notice some lines have the string "`--Control`" where a gene symbol should be. Let's remove these lines. In order to check for these values, we need two new powerful keywords, `if` and `in`.

`if`

`if` must be followed by a boolean or `bool` in Python. A `bool` has two possible values, `True` or `False`. We can create a `bool` by comparing values of the same type using comparison operations. You can compare strings and numbers using the operators `==`, `!=`, `<`, `>`, `<=`, `>=`. The `!=` operator indicates inequality.

```
>>> 4 == 4
True
>>> 4 > 2
True
```

The boolean operators `or` and `and` can be used for more complex expressions:

```
>>> True and (7 == 9)
False
```

Now let's use a conditional expression with an `if` statement:

```
>>> if (7 > 3): print('Math is real')
...
Math is real
```

`in`

The second keyword is `in`. `in` returns `True` or `False` depending on whether or not the left-hand side of the expression is in the right-hand side:

```
>>> 'foo' in 'foobar'
True
>>> 2 in [1,2,3]
True
```

Validating

We have already seen a one-line `for loop` but Python let's us specify a block of code that will be executed on every iteration. For example:

```
for line in lines:
    if 'null' in line or '--Control' in line:
        continue
    if len(line) == 0:
        continue
```

This iterates over every line in our cleaned lines `list` and skips over any line that has a value we would like to ignore or any line that is empty.

Outputting

The last step is to output our cleaned data into a new file. We can use the `open()` built-in and the same `for loop`. We will also use a function `upper()` to convert the gene symbols to uppercase and the method `join()` to convert the `list` back to a `str`. This is the new `for loop`:

```
out = open('out.txt', 'w+')
for line in lines:
    if 'null' in line or '--Control' in line:
        continue
    if len(line) == 0:
        continue
    line[0] = line[0].upper()
    line = '\t'.join(line) + '\n'
    out.write(line)
```

Now open `out.txt`. You should have cleaned data.

Resources

Hopefully this lab gave you some insight into programming in Python. Python is a free, well-documented, well-supported, easy to learn, and expressive language with a growing community in academia, the sciences, and industry. While we did not use them today, Python has a number of high-performance libraries for numerical computing, so performance is typically not an issue even for very large datasets.

If you enjoyed Python and would like to learn more, you can do so with a number of great online resources. Here are a few:

- Official tutorial: https://docs.python.org/3/tutorial/
- Software Carpentry: http://software-carpentry.org/v5/novice/python/index.html
- Learn Python the Hard Way: http://learnpythonthehardway.org/