04_py_fundementals_reading_data

August 12, 2021

1 Working with data

The best way to learn how to program is to do something useful, so this introduction to Python is built around a common scientific task: data analysis.

So far we have typed in all our data directly. This is good for teaching, but not realistic. In the real world we are going to have data formatted in a file of some sort, and we will want to be able to read that data into python to work with it.

Let's dive right in and do this!

1.1 Credit:

things here are a mix of the really excellent Software Carpentry tutorial on Python: http://swcarpentry.github.io/python-novice-inflammation/

I've made some slight adaptations here and there, but the credit goes to those organizations. I hope I am using this correctly under the licences:

https://earth-env-data-science.github.io/LICENSE.html https://swcarpentry.github.io/python-novice-inflammation/LICENSE.html

2 Loading data into python

In order to do something more useful, we are going to need to be able to access data. How do we load data into python? To load the test data we will work with, we need to access (`import` in Python terminology) a library of code (think of it as a toolbox) called `NumPy`.

In general you should use this library if you want to do fancy things with numbers, especially if you have matrices or arrays.

`NumPy` is a toolbox that is not part of the most basic part of python. It's a code library that lots and lots of people work on imporving. It forms the basis of almost all other scientific python tool boxes, so it's super important.

2.0.1 installing numpy with (Ana)conda (*shouldn't be necessary because it comes installed with Anaconda by default, but just in case...*)

To get `NumPy` (you only need to do this once):

1. go to the little `+` on the left to open a `launcher` window.
2. click the 'terminal' tile to open a terminal
3. type `conda env list` and hit enter. Make sure that there is a `*` next to the line that says swbc
4. if that's right type `conda install numpy`
5. when it asks `proceed ([y]/n)?` type `y` and hit enter

Great! You just used the anaconda manager to download and install this important python library! You only need to do this once to get `NumPy` on your computer. We will follow the same proceedure to get a lot of other code, but lets start with the NumPy tools.

2.0.2 importing NumPy into this notebook

Ok, one more critical step! before we can use the tools from numpy in this code notebook, we need to tell python we are going to use it. In python-speak we need to `import` the `NumPy` library. We will always have a step like this in our code where we gather all the tools we need for our work. We can import `NumPy` using:

```
[1]: import numpy
```

`NumPy` has *many* different tools for working with data. There are so many I don't know all of them, and I wouldn't expect you to either. I will show you important ones, and as you code more you will start to remember what tools you need.

All python tool libraries (also called packages) have documentation online Almost always it is very good, including examples and tutorials that help you learn how to use the tools. The documentation for `NumPy` is here:

https://numpy.org/doc/stable/

Let's get started useing `NumPy`.

2.0.3 test data:

in a directory called `data` you have a bunch of files like `inflammation-01.csv`. These are simple data files with data in comma-seperated-value format. **Open one up useing the file browser in Jupyterlab (on left)**. You can also do this with notepad on a PC, with textEdit on a mac , or with Excel.

Each row of this data represents a patient, and the columns are the patients inflammation readings on subsequent days. Take a look and lets make sure we get the basic picture of the data.

Ok, now let's read that data into python so we can work with it.

I happen to know that `NumPy` has a function that can help us: its called `loadtxt()`. What do you guess it does?

We use `loadtxt()` like other methods we have seen before (remember `b.capitalize()`?):

`numpy.loadtxt(filename, delimeter=',')`

what is the `filename` argument? what do you think the `delimeter=','` means?

```
[2]: numpy.loadtxt('../data/inflammation-01.csv', delimiter=',')
```

```
[2]: array([[0., 0., 1., …, 3., 0., 0.],
            [0., 1., 2., …, 1., 0., 1.],
            [0., 1., 1., …, 2., 1., 1.],
            …,
            [0., 1., 1., …, 1., 1., 1.],
            [0., 0., 0., …, 0., 2., 0.],
            [0., 0., 1., …, 1., 1., 0.]])
```

The expression `numpy.loadtxt(...)` is a function call that asks Python to run the function `loadtxt` which belongs to the `numpy` library. **This dotted notation is used everywhere in Python: the thing that appears before the dot contains the thing that appears after.**

`numpy.loadtxt()` has two 'inputs': the name of the file (and path to it if it lives in another directory) we want to read and the delimiter that separates values on a line. These both need to be character strings (or strings for short), so we put them in quotes.

Since we haven't told it to do anything else with the function's output, the notebook displays it. In this case, that output is the data we just loaded. By default, only a few rows and columns are shown (with `...` to omit elements when displaying big arrays). To save space, Python displays numbers as 1. instead of 1.0 when there's nothing interesting after the decimal point.

Our call to `numpy.loadtxt` read our file but didn't save the data in memory. To do that, we need to assign the array to a variable. Just as we can assign a single value to a variable, we can also assign an array of values to a variable using the same syntax. Let's re-run `numpy.loadtxt` and save the returned data:

[3]:
```
data = numpy.loadtxt('../data/inflammation-01.csv', delimiter=',')
```

This statement doesn't produce any output because we've assigned the output to the variable `data`. If we want to check that the data have been loaded, we can print the variable's value:

[4]:
```
print(data)
```

```
[[0. 0. 1. … 3. 0. 0.]
 [0. 1. 2. … 1. 0. 1.]
 [0. 1. 1. … 2. 1. 1.]
 …
 [0. 1. 1. … 1. 1. 1.]
 [0. 0. 0. … 0. 2. 0.]
 [0. 0. 1. … 1. 1. 0.]]
```

Now that the data are in memory, we can manipulate them. First, let's ask what type of thing data refers to:

[5]:
```
# print the data type
type(data)
```

[5]:
```
numpy.ndarray
```

This is a new data structure. It's like a `list` or a `dictonary` that we saw last time. Except now the data structure is an `ndarray`, which stands for: N-dimensional array (matrix). N refers to the number of dimentions. In this case, there are two dimentions: one is the rows, the other is the columns.

in plan english, what are other names for the 2 demensions of our particular test data? I mean, what is a descriptive name of those rows and columns? If you made a plot where they were the x an y axis, what would you label them? This is one way to think about the dimentions in a matrix we can check out how big the data is by using the `attribute` (or feature, characteristic) or the `ndarray` data structure called `shape`

```
[6]: # print the shape of the data
     data.shape
```

```
[6]: (60, 40)
```

The output tells us that the `data` array variable contains 60 rows and 40 columns. When we created the variable `data` to store our test data, we didn't just create the array; we also created information about the array, called members or attributes. This extra information describes `data` in the same way an adjective describes a noun. `data.shape` is an attribute of `data` which describes the dimensions of `data`. We use the same dotted notation for the attributes of variables that we use for the functions in libraries because they have the same part-and-whole relationship.

What does the shape mean? think about the patients and observations that the data represent

3 Accessing part of the data

If we want to get a single number from the array, we must provide an index in square brackets after the variable name, just as we do in math when referring to an element of a matrix. Our inflammation data has two dimensions, so we will need to use two indices to refer to one specific value.

Accessing the first value:

```
[7]: print('first value in data:', data[0, 0])
```

```
first value in data: 0.0
```

Accessing a value in the middle.

```
[8]: print('middle value in data:', data[30, 20])
```

```
middle value in data: 13.0
```

4 Slicing data

An expression like `data[30,20]` grabs a single data point. But we can also easily select whole sections as well. For example, we can select the first ten days (columns) of values for the first four patients (rows) like this:

```
[9]: # get the first 4 rows and the first 10 columns
     data[0:4, 0:10]
```

```
[9]: array([[0., 0., 1., 3., 1., 2., 4., 7., 8., 3.],
            [0., 1., 2., 1., 2., 1., 3., 2., 2., 6.],
            [0., 1., 1., 3., 3., 2., 6., 2., 5., 9.],
            [0., 0., 2., 0., 4., 2., 2., 1., 6., 7.]])
```

Here, `0:4` is saying, "start at index 0 and go up to, but not including, index 4." Likewise, `0:10` is saying "start at index 0 and go up to, but don't include index 10". It's important to note, while the starting index of a slice is *inclusive* the ending index is *exclusive*.

One way to remember how slices work is to think of the indices as pointing *between* characters, with the left edge of the first character numbered 0. Then the right edge of the last character of a string of *n* characters has index *n*, for example:

```
 +---+---+---+---+---+---+
 | P | y | t | h | o | n |
 +---+---+---+---+---+---+
 0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

The first row of numbers gives the position of the indices 0...6 in the string; the second row gives the corresponding negative indices. The slice from *i* to *j* consists of all characters between the edges labeled *i* and *j*, respectively.

We don't have to start the slice at the 0 index, either. For example, we can see the next five rows of data by putting the starting index at `5`.

[10]:
```
data[5:10,0:10]
```

[10]:
```
array([[0., 0., 1., 2., 2., 4., 2., 1., 6., 4.],
       [0., 0., 2., 2., 4., 2., 2., 5., 5., 8.],
       [0., 0., 1., 2., 3., 1., 2., 3., 5., 3.],
       [0., 0., 0., 3., 1., 5., 6., 5., 5., 8.],
       [0., 1., 1., 2., 1., 3., 5., 3., 5., 8.]])
```

In fact, we're not even required to declare the starting AND ending points. If we omit the staring index, Python starts at he beginning, or the 0 index. If we omit the ending index, the slice continues to the the end of the array. We can even store the results of a slice into a variable.

[11]:
```
# get a subset of the data and give it a new name
small = data[:3, 36:]
print('small is:')
print(small)
```

```
small is:
[[2. 3. 0. 0.]
 [1. 1. 0. 1.]
 [2. 2. 1. 1.]]
```

5 Math

Just as we performed mathematical operations on variables containing single real numbers (converting kilograms to pounds), we can performed operations on whole arrarys. The simplest operations with data are arithmetic: add, subtract, multiply, and divide. When you do such operations on arrays, the operation is done element-by-element. Thus:

[12]:
```
# multiply data by 2 and give that a name
doubledata = data * 2.0
```

[13]:
```
# add our two arrays

tripledata = doubledata + data

print("small")
print(small)
print("triple")
print(tripledata[:3, 36:])
```

```
small
[[2. 3. 0. 0.]
 [1. 1. 0. 1.]
 [2. 2. 1. 1.]]
triple
[[6. 9. 0. 0.]
 [3. 3. 0. 3.]
 [6. 6. 3. 3.]]
```

Often, we want to do more than add, subtract, multiply, and divide array elements. NumPy knows how to do more complex operations, too. If we want to find the average inflammation for all patients on all days, for example, we can ask NumPy to compute `data`'s mean value:

[14]:
```
# compute the mean value of the data

numpy.mean(data)

# alternative:
# data.mean()
```

[14]:
```
6.14875
```

Here, we are calling the function `mean` from the library `numpy`. Some functions, such as `numpy.mean()` take arguments. Here our variable `data` is passed to the function `mean()`.

The end…

6 Exercise 04 / Breakout