Python Part II - Analyzing Patient Data

Jean-Yves Sgro

February 16, 2017

Contents

```
## Warning: package 'knitr' was built under R version 3.5.2
```

1 Software Carpentry: Analyzing Patient Data

1.1 Overview:

Questions

- How can I process tabular data files in Python?

Objectives

- Explain what a library is, and what libraries are used for.
- Import a Python library and use the things it contains.
- Read tabular data from a file into a program.
- Assign values to variables.
- Select individual values and subsections from data.
- Perform operations on arrays of data.
- Display simple graphs.

1.2 Key points summary

- Import a library into a program using import `libraryname`.
- Use the `numpy` library to work with arrays in Python.
- Use `variable = value` to assign a value to a variable in order to record it in memory.
- Variables are created on demand whenever a value is assigned to them.
- Use `print(something)` to display the value of `something`.
- The expression `array.shape` provides the shape of an array (*i.e.* its dimensions.)
- Use `array[x, y]` to select a single element from an array.
- Array indices start at **0**, not **1**.
- Use `low:high` to specify a slice that includes the indices from low to `high-1`.
- All the indexing and slicing that works on arrays also works on strings.
- Use `# some kind of explanation` to add comments to programs.
- Use `numpy.mean(array)`, `numpy.max(array)`, and `numpy.min(array)` to calculate simple statistics.
- Use `numpy.mean(array, axis=0)` or `numpy.mean(array, axis=1)` to calculate statistics across the specified axis.
- Use the `pyplot` library from `matplotlib` for creating simple visualizations.

2 Patient data

Earlier we downloaded and *unzipped* a file that we placed withing a **desktop** directory called **python-novice-inflammation** containing the unzipped files within a directory called `data`.

This should contain the downloaded files as well as the *ipython notebook* we started earlier that we saved as `notebook1.ipynb`.

A simple Unix command placed from the (`ls -R ~/Desktop//python-novice-inflammation`) within a Terminal would show the follwing result for 1 directory (`data`) and 15 comma-separated files.

```
data/

/Users/jsgro/Desktop/python-novice-inflammation/data:
inflammation-01.csv inflammation-07.csv notebook1.ipynb
inflammation-02.csv inflammation-08.csv small-01.csv
inflammation-03.csv inflammation-09.csv small-02.csv
inflammation-04.csv inflammation-10.csv small-03.csv
inflammation-05.csv inflammation-11.csv
inflammation-06.csv inflammation-12.csv
```

3 Libraries

We used libraries `sys` and `platform` above but there are many more libraries available. When working with set of numbers, tables, matrices etc. the library `numpy` is very useful and widely used.

However, it does not come standard with the python software and has to be installed first. How the installation is done varies with the operating system and the python software used. `numpy` has already been installed on the computer you are using in class.

However, if you are trying to do this on your own computer you will need to install `numpy`.

Since we are using Anaconda, we just need to add to the collection of installed libraries. For anaconda the command would be **issused from a Terminal** using the **Unix** line command (NOT on the python notebook or a python console!)

`Unix/bash` command:

```
conda install numpy
```

It is then possible to list all installed libraries with the command:

`Unix/bash` command:

```
conda list
```

If you are using a python software different than Anaconda you may need to refer to the help for that software or perhaps seach online with a search engine. Some python software use the `pip` command (also from a Unix Terminal.)

```
import numpy
numpy.loadtxt(fname='inflammation-01.csv', delimiter=',')
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.]])
```

```
        [[ 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.]]
```

3.1 Dotted notation and functions

> **What is a function?** Functions can be part of a *library* or created by the user as “user-defined functions.”" A Function is a block of code written to perform a specific task, and can be re-used to provide modularity. A simple example of a function is `print()` that is *built-in* the python langage.
>
> **What is dotted notation?** Functions that are built-in the langage, like `print()` are simply called by their name. Functions that are part of an imported library, as the above example of `numpy.loadtxt()` are written with the library name as a suffix, and separated by a dot for clarity. A general term could be that the function is a `component` of the `library`.

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 to refer to the parts of `things` as `thing.component`.

`numpy.loadtxt` has two *parameters*: the *name of the file* 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.

When we are finished typing and press Shift+Enter, the notebook runs our command. 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.

4 Variables

A variable is just a name for a value, such as `x`, `current_temperature`, or `subject_id`. Python's variables must begin with a letter and are case sensitive. We can create a new variable by assigning a value to it using `=`. As an illustration, let's step back and instead of considering a table of data, consider the simplest “collection” of data, a single value. The line below assigns the value `55` to a variable `weight_kg`:

```
# Define a variable and assign a numeric value:
weight_kg = 55
```

Once a variable has a value, we can print it to the screen:

```
print(weight_kg)
```

```
55
```

We can also perform arithmetic with the variable:

```
print('weight in pounds:', 2.2 * weight_kg)
```

```
weight in pounds: 121.0
```

As the example above shows, we can print several things at once by separating them with commas.

We can also change a variable's value by assigning it a new one:

```
weight_kg = 57.5
print('weight in kilograms is now:', weight_kg)
('weight in kilograms is now:', 57.5)
```

If we imagine the variable as a sticky note with a name written on it, assignment is like putting the sticky note on a particular value. Here we place a sticky note called `weight_kg` onto a value of `57.5`:

Figure 1.

This means that assigning a value to one variable does *not* change the values of other variables. For example, let's store the subject's weight in pounds in a variable:

```
weight_lb = 2.2 * weight_kg
print('weight in kilograms:', weight_kg, 'and in pounds:', weight_lb)
```

```
weight in kilograms: 57.5 and in pounds: 126.5
```

Figure 2.

Now let's change `weight_kg`:

```
weight_kg = 100.0
print('weight in kilograms is now:', weight_kg, 'and weight in pounds is still:', weight_lb)
```

```
weight in kilograms is now: 100.0 and weight in pounds is still: 126.5
```

Figure 3.

Originally the weight in *pounds* as `weight_lb` was calculated from the value of weight in *kilograms* as `weight_kg` with `print('weight in pounds:', 2.2 * weight_kg)`.

However, since `weight_lb` doesn't "remember" where its value came from, it isn't automatically updated when `weight_kg` changes. *This is different from the way spreadsheets work.*

4.0.1 Checking variables remembered by Python

You can use the `%whos` command at any time to see what variables you have created and what modules you have loaded into the computer's memory. **As this is an IPython command, it will only work if you are in an IPython terminal or the Jupyter Notebook**.

```
%whos
```

```
Variable    Type       Data/Info
--------------------------------
numpy       module     <module 'numpy' from '/Us<...>kages/numpy/__init__.py'>
weight_kg   float      100.0
weight_lb   float      126.5
```

4.1 Variables containing large data

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 its result *within* a variable called `data`:

```
data = numpy.loadtxt(fname='inflammation-01.csv', delimiter=',')
```

This statement doesn't produce any output because assignment doesn't display anything. If we want to check that our data has been loaded, we can print the variable's value:

```
[[ 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 our data is in memory, we can start doing things with it. First, let's ask what type of thing data refers to:

```
print(type(data))
```

```
<type 'numpy.ndarray'>
```

The output tells us that `data` currently refers to an N-dimensional array created by the **NumPy** library. These data correspond to arthritis patients' inflammation.

The rows are the individual patients and the columns are their daily inflammation measurements.

5 Attributes and dot operator

Above we created a variable named `data` into which we loaded data from the file `'inflammation-01.csv'` with the command: `data = numpy.loadtxt(fname='inflammation-01.csv', delimiter=',')`

The variable `data` has some specific attributes that can be inspected with the "dot operator" `.` followed by the attribute.

Attributes can be listed with the command:

```
dir(data)
```

Some attributes are not useful to the human eye, but a few maybe.

5.1 Data type

From the list obtained with `dir` we can ask what type of information contained within `data` by using the `dtype` attribute:

```
print(data.dtype)
```

```
float64
```

This tells us that the NumPy array's elements are floating-point numbers.

The "dot operator" to access object properties is a widely used method in Python.

5.2 Data shape

Another attribute built-in the `data` object at creation is `shape` which would describe the number of columns and rows of the table that was read. We can see what the array's shape is with the following command:

```
print(data.shape)
```

```
(60, 40)
```

This tells us that `data` has **60 rows** and **40 columns**. When we created the variable `data` to store our arthritis 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.

5.3 Accessing values

5.3.1 Selecting single values

If we want to get a single number from the array, we must provide an index in square brackets, just as we do in math:

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

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

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

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

The expression `data[30, 20]` may not surprise you, but `data[0, 0]` might. Programming languages like Fortran and MATLAB start counting at 1, because that's what human beings have done for thousands of years. Languages in the C family (including C++, Java, Perl, and Python) count from 0 because that's more convenient when indices are computed rather than constant.

As a result, if we have an M×N array in Python, its indices go from 0 to M-1 on the first axis and 0 to N-1 on the second. It takes a bit of getting used to, but one way to remember the rule is that the index is how many steps we have to take from the start to get the item we want.

> Upper left corner: What may also surprise you is that when Python displays an array, it shows the element with index [0, 0] in the upper left corner rather than the lower left. This is consistent with the way mathematicians draw matrices, but different from the Cartesian coordinates. The indices are (row, column) instead of (column, row) for the same reason, which can be confusing when plotting data.

5.3.2 Selecting subsets with `:`

With an index such as `[30, 20]` we previously selected a single element from the array of data contained within`data`. However, we can also select larger sub-sections rather than a single value.

For example, we can select the first ten days (columns) of values for the first four patients (rows) like this:

```
print(data[0:4, 0:10])
```

```
[[ 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.]]
```

Note: The slice `0:4` means, *"Start at index 0 and go up to, but not including, index 4."* Again, the *up-to-but-not-including* takes a bit of getting used to, but **the rule is that the difference between the upper and lower bounds is the number of values in the slice**.

A "slice" can also be taken from within `data` and not necessarily start at `0`:

```
print(data[5:10, 0:10])
```

```
[[ 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.]]
```

5.3.2.1 Upper and lower bound

We don't have to include the upper and lower bound on the slice.

If we don't include the lower bound, Python uses 0 by default.

If we don't include the upper, the slice runs to the end of the axis.

If we don't include either (*i.e.*, if we just use `:` on its own without any numbers), the slice includes everything.

We know that the "shape" of `data` is `(60, 40)` (see `print(data.shape)` above.)

In the code below, we create a small subset of `data` demonstrating inferred values for upper and lower bounds. `:3` means `0:3` and `36:` means to the end which we know is`40`.

```
small = data[:3, 36:]
print('small is:')
print(small)
```

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

6 Vectorization

Arrays also know how to perform common mathematical operations on their values. The simplest operations with data are arithmetic: add, subtract, multiply, and divide. When you do such operations on arrays, the operation is done on each individual element of the array.

This is a style of computer programming where operations are applied to whole arrays instead of individual elements; it is called **"vectorization."**

6.1 Multiplication

For example, we will now multiply each element of `data` by a factor `2` with a single multiplication operation.

```
doubledata = data * 2.0
```

This will create a new array `doubledata` whose elements have the value of two times the value of the corresponding elements in `data`:

```
print('original:')
print(data[:3, 36:])
print()
print('doubledata:')
print(doubledata[:3, 36:])
```

```
original:
[[ 2.  3.  0.  0.]
 [ 1.  1.  0.  1.]
 [ 2.  2.  1.  1.]]

doubledata:
[[ 4.  6.  0.  0.]
 [ 2.  2.  0.  2.]
 [ 4.  4.  2.  2.]]
```

6.2 Addition

If, instead of taking an array and doing arithmetic with a single value (as above when we used `2` to multiply) you did the arithmetic operation with *another array* of the *same shape*, the operation will be done **on corresponding elements** of the two arrays. Thus:

```
tripledata = doubledata + data
```

will give you an array where `tripledata[0,0]` will equal `doubledata[0,0`] plus `data[0,0]`, and so on for ***all*** other elements of the arrays.

```
print('tripledata:')
print(tripledata[:3, 36:])
```

```
tripledata:
[[ 6.  9.  0.  0.]
 [ 3.  3.  0.  3.]
 [ 6.  6.  3.  3.]]
```

6.3 Complex arithmetic with Numpy

Often, we want to do more than add, subtract, multiply, and divide values of data. **NumPy** knows how to do more complex operations on arrays. 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:

```
print(numpy.mean(data))
```

```
6.14875
```

7 Functions

`mean` (above) is a function that takes an array as an argument. If variables are nouns, functions are verbs: they do things with variables.

> Not all functions have input. Generally, a function uses inputs to produce outputs. However, some functions produce outputs without needing any input. For example, checking the current time doesn't require any input.

```
import time
print(time.ctime())
Thu Apr 18 13:23:03 2019
```

> For functions that don't take in any arguments, we still need parentheses `(())` to tell Python to go and do something for us.
> We already saw functions example with *e.g.* `print()` and `numpy.loadtxt()`.

7.1 Numpy functions

NumPy has lots of useful functions that take an array as input.

Let's use three of those functions to get some descriptive values about the dataset.

We'll also use multiple assignment, a convenient Python feature that will enable us to do this all in one line. The separation between the statement of declaration of the variables is a comma `,`.

The first line defines variables`maxval`, `minval`, and `stdval` which are respectively given values after the equal sign `=` with atributes functions recognized by the `dot-notation`. `numpy.max(data)` assigns the highest value found in `data`, `numpy.min(data)` assigns the lowest value and `numpy.std(data)` calculates the standard deviation. These values are assigned, in respective order of declaration before the equal sign to the declared variables.

The next lines simply print the value of the variables.

```
maxval, minval, stdval = numpy.max(data), numpy.min(data), numpy.std(data)

print('maximum inflammation:', maxval)
print('minimum inflammation:', minval)
print('standard deviation:', stdval)
```

```
maximum inflammation: 20.0
minimum inflammation: 0.0
standard deviation: 4.61383319712
```

> Mystery functions in IPython
> How did we know what functions **NumPy** has and how to use them? If you are working in the *IPython/Jupyter Notebook* there is an easy way to find out. If you type the name of something **with a full-stop** then you can use tab completion (*e.g.* type `numpy.` and then **press tab**) to see a list of all functions and attributes that you can use. After selecting one you can also add a question mark (*e.g.* `numpy.cumprod?`) and IPython will return an explanation of the method! This is the same as doing `help(numpy.cumprod)`.

Note: If you are not within IPython you can use the command `dir(numpy)` to get a list of all embedded functions and attributes and then manually type the help command `help(numpy.cumprod)`.

8 Partial statistics

When analyzing data, though, we often want to look at partial statistics, such as the **maximum value per patient** or the **average value per day**.

8.1 Temporary array:

One way to do this is to create a new temporary array of the data we want, then ask it to do the calculation:

```
patient_0 = data[0, :] # 0 on the first axis, everything on the second
print('maximum inflammation for patient 0:', patient_0.max())
```

```
maximum inflammation for patient 0: 18.0
```

Everything in a line of code following the `#` symbol is a comment that is ignored by the computer. Comments allow programmers to leave explanatory notes for other programmers or their future selves.

We don't actually need to store the row in a variable of its own. Instead, we can combine the selection and the function call:

```
print('maximum inflammation for patient 2:', numpy.max(data[2, :]))
```

```
maximum inflammation for patient 2: 19.0
```

What if we need the maximum inflammation for all patients (as in the next diagram on the left), or the average for each day (as in the diagram on the right)? As the diagram below shows, we want to perform the operation across an axis:

We know from above that `data` has 60 rows and 40 columns.

`axis=0` represents each of the 40 (vertical) columns.
`axis=1` represents each of the 60 (horiontal) rows.

To support this, most array functions allow us to specify the axis we want to work on. If we ask for the average across axis 0 (rows in our 2D example), we get:

```
print(numpy.mean(data, axis=0))
```

```
[  0.           0.45         1.11666667   1.75         2.43333333   3.15
   3.8          3.88333333   5.23333333   5.51666667   5.95         5.9
   8.35         7.73333333   8.36666667   9.5          9.58333333
  10.63333333  11.56666667  12.35        13.25        11.96666667
  11.03333333  10.16666667  10.           8.66666667   9.15         7.25
   7.33333333   6.58333333   6.06666667   5.95         5.11666667   3.6
   3.3          3.56666667   2.48333333   1.5          1.13333333
```

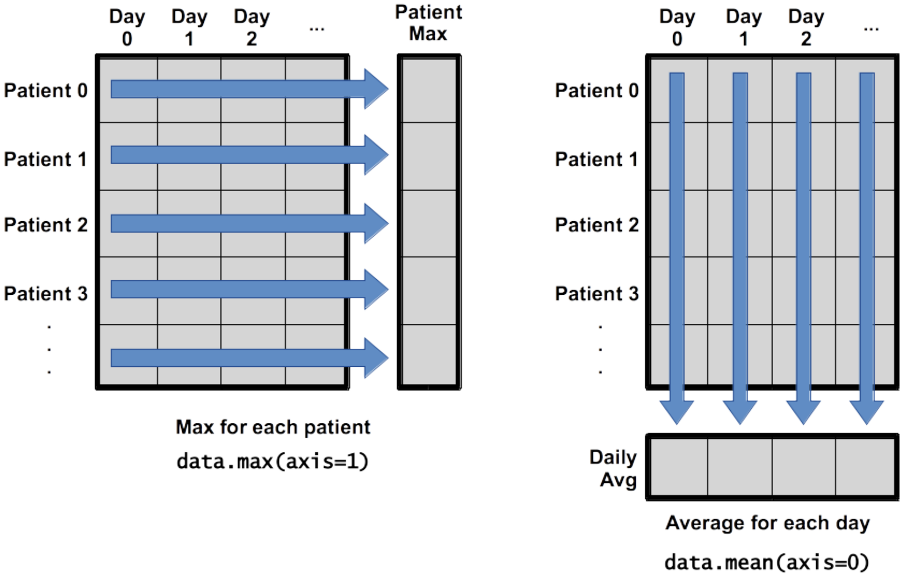

Figure 1:

```
   0.56666667]
```

As a quick check, we can ask this array what its shape is, and verify that we obtain the same number of columns (40) as in `data`:

```
print(numpy.mean(data, axis=0).shape)
```

```
(40,)
```

The expression `(40,)` tells us we have an N×1 vector, so this is the average inflammation per day for all patients.

If we average across axis 1 (columns in our 2D example), we obtain:

```
print(numpy.mean(data, axis=1))
```

```
[ 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  ]
```

which is the average inflammation per patient across all days.

9 Visualization as insight: `matplotlib`

The mathematician Richard Hamming (Hamming 1962) once said, *"The purpose of computing is insight, not numbers,"* and the best way to develop insight is often to visualize data.

Visualization deserves an entire lecture (or course) of its own, but we can explore a few features of Python's `matplotlib` library here.

While there is no "official" plotting library, this package is the *de facto* standard.

9.1 Heat map

First, we will import the `pyplot` module from `matplotlib` and use two of its functions to create and display a heat map of our data:

```
import matplotlib.pyplot
image  = matplotlib.pyplot.imshow(data)
matplotlib.pyplot.show()
```

The last command will open a new window, perhaps *behind* your Notebook window, titled `Figure 1` and showing the heatmap. On the bottom part of the window you can click in the floppy disk icon to save the image to your local computer.

The heat-map image within the browser window will look like this:

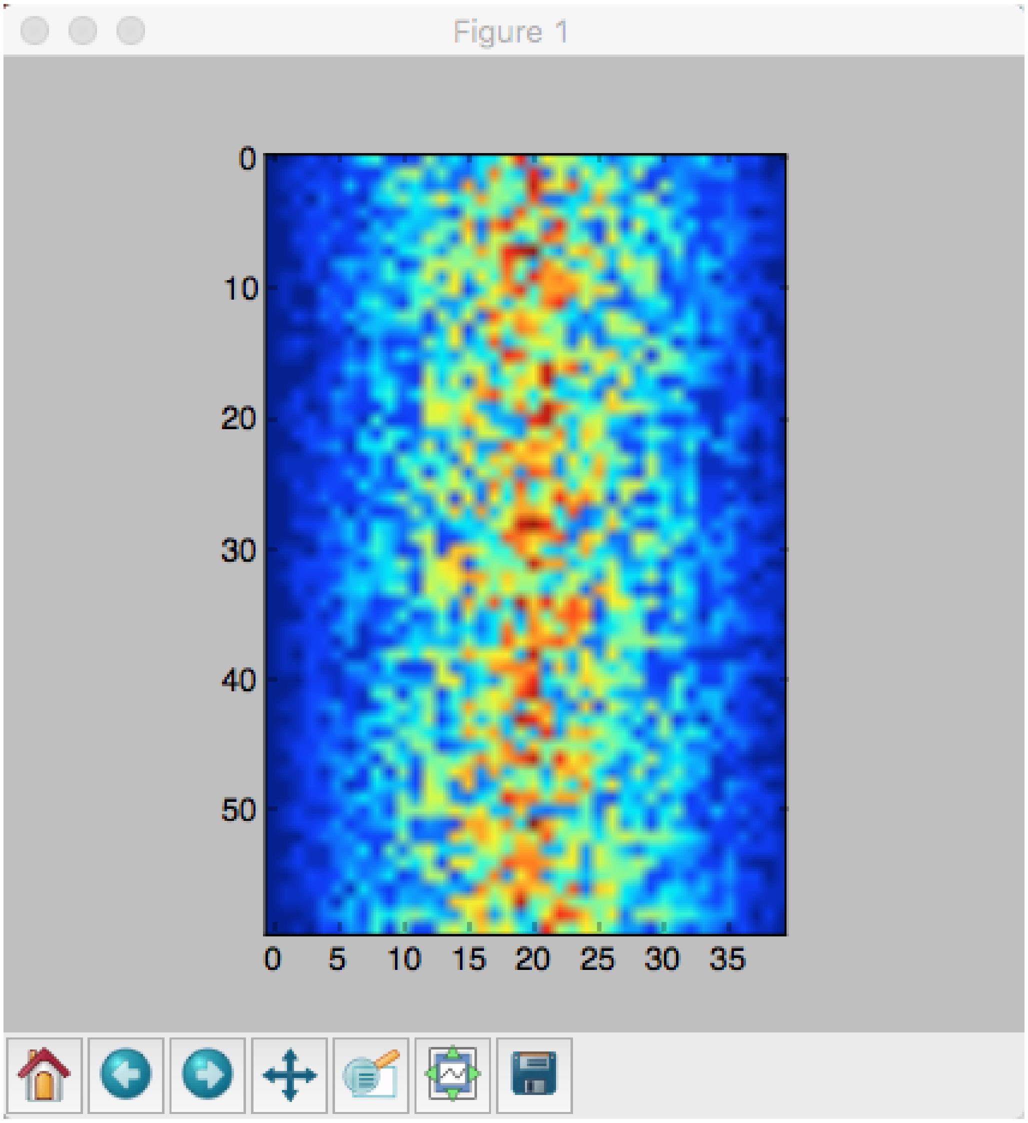

Figure 4.

Heatmap of the Data.

Blue regions in this heat map are low values, while red shows high values. As we can see, inflammation rises and falls over a 40-day period.

> IPython -
> If you're using an IPython / Jupyter notebook, you'll need to execute the following command in order for your matplotlib images to appear in the notebook when `show()` is called:
>
> ```
> % matplotlib inline
> ```
>
> The `%` indicates an IPython magic function - a function that is only valid within the notebook environment. Note that you only have to execute this function once per notebook. This command has to be used ***after*** `show()` is called. Therefore the complete command sequence would be:

```
import matplotlib.pyplot
image  = matplotlib.pyplot.imshow(data)
matplotlib.pyplot.show()
% matplotlib inline
```

9.2 Line plots

9.2.1 Average inflammation over time

Let's take a look at the average **inflammation over time**:

```
ave_inflammation = numpy.mean(data, axis=0)
ave_plot = matplotlib.pyplot.plot(ave_inflammation)
matplotlib.pyplot.show()
```

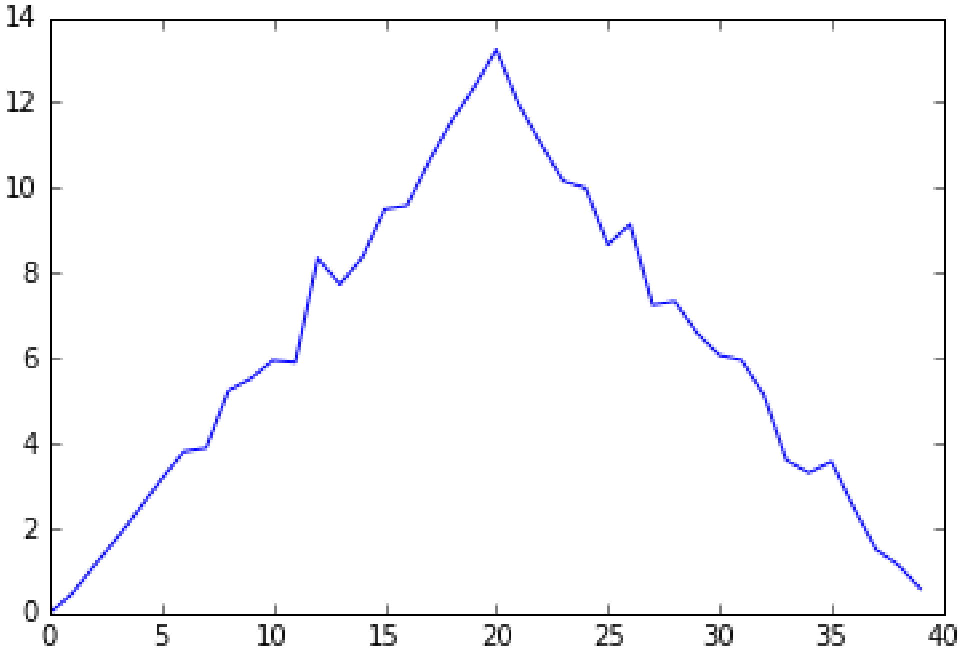

Figure 5.

Average Inflammation Over Time.

Here, we have put the average per day across all patients in the variable `ave_inflammation`, then asked `matplotlib.pyplot` to create and display a **line graph** of those values. The result is roughly a linear rise and fall, which is suspicious: based on other studies, we expect a sharper rise and slower fall. We'll check two other statistics below.

9.2.2 Maximum and minimum values along first axis

```
# Maximum value along first axis
max_plot = matplotlib.pyplot.plot(numpy.max(data, axis=0))
matplotlib.pyplot.show()
```

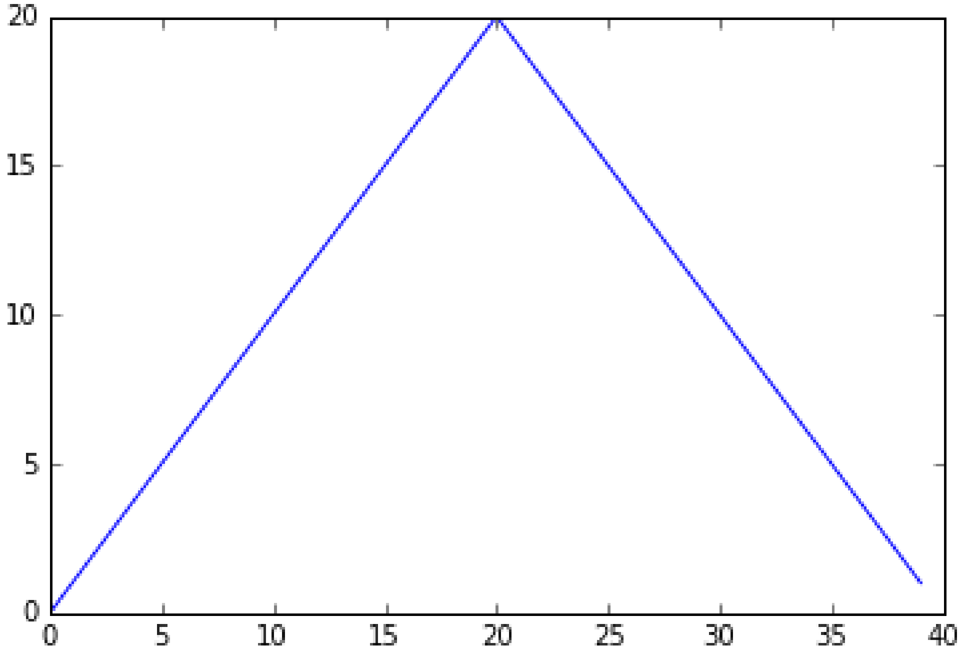

Figure 6.

Maximum value along the first axis.

```
# Maximum value along first axis
min_plot = matplotlib.pyplot.plot(numpy.min(data, axis=0))
matplotlib.pyplot.show()
```

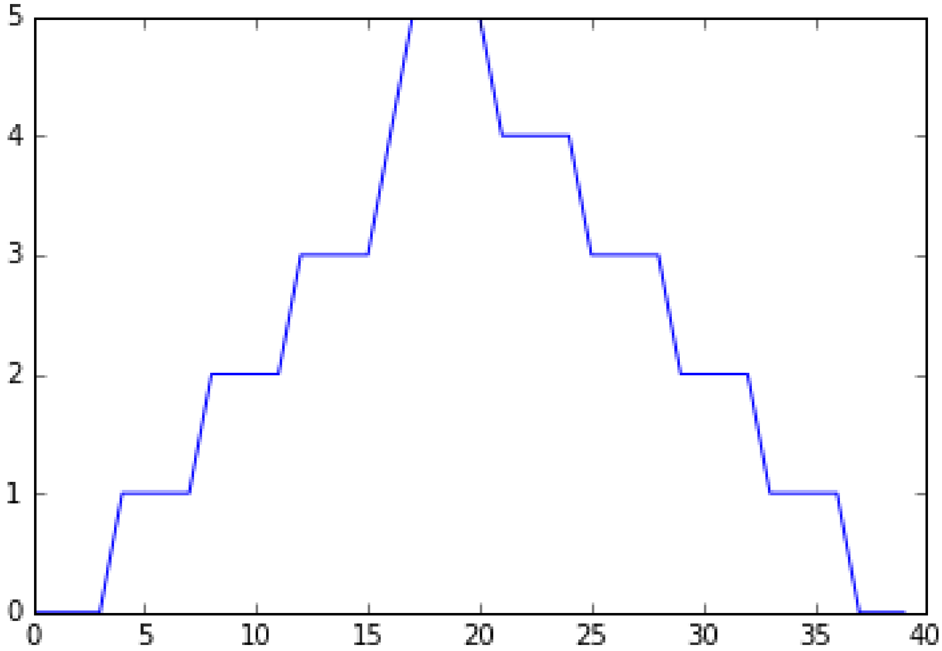

Figure 7.

Minimum value along the first axis.

The maximum value rises and falls perfectly smoothly, while the minimum seems to be a step function. Neither result seems particularly likely, so either there's a mistake in our calculations or something is wrong with our data. This insight would have been difficult to reach by examining the data without visualization tools.

Note: It is not clear from the above remark if the data is OK or not (is this made-up data?) It seems that they are stressing the fact that without making these images we would not be able to be aware of these variations in the data.

9.3 Combining plots

You can group similar plots in a single figure using subplots. The script below uses a number of new commands.

The function `matplotlib.pyplot.figure()` creates a space into which we will place all of our plots. The parameter `figsize` tells Python how big to make this space. Each subplot is placed into the figure using its `add_subplot method`. The `add_subplot` method takes 3 parameters. The first denotes how many total rows of subplots there are, the second parameter refers to the total number of subplot columns, and the final parameter denotes which subplot your variable is referencing (left-to-right, top-to-bottom). Each subplot is stored in a different variable (`axes1`, `axes2`, `axes3`). Once a subplot is created, the axes can be titled using the `set_xlabel()` command (or `set_ylabel()`).

Here are our three plots side by side:

```
import numpy
import matplotlib.pyplot
```

```
data = numpy.loadtxt(fname='inflammation-01.csv', delimiter=',')

fig = matplotlib.pyplot.figure(figsize=(10.0, 3.0))

axes1 = fig.add_subplot(1, 3, 1)
axes2 = fig.add_subplot(1, 3, 2)
axes3 = fig.add_subplot(1, 3, 3)

axes1.set_ylabel('average')
axes1.plot(numpy.mean(data, axis=0))

axes2.set_ylabel('max')
axes2.plot(numpy.max(data, axis=0))

axes3.set_ylabel('min')
axes3.plot(numpy.min(data, axis=0))

fig.tight_layout()

matplotlib.pyplot.show()
```

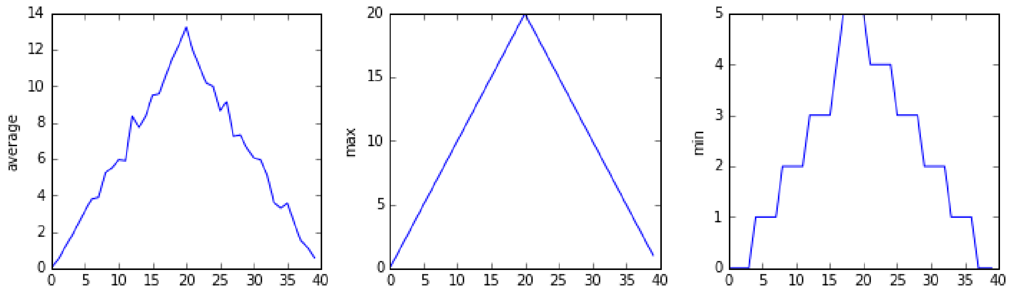

Figure 8.

Three previous plots as subplots.

The call to `loadtxt` as the NumPy command `numpy.loadtxt()` reads our data as we did before.

The rest of the program defines the following actions:

- `fig = matplotlib.pyplot.figure(figsize=(10.0, 3.0))` tells the plotting library how large we want the figure to be: 10.0 by 3.0 inches and we store this information in variable `fig`.
- `axes1 = fig.add_subplot(1, 3, 1)` and similar commands for `axes2` and `axes3` tell the plotting library that we're creating three subplots.
- `axes1.set_ylabel('average')` and similar commands for `axes2` and `axes3` tell the plotting library how to label the `Y axis` on each plot.
- `axes1.plot(numpy.mean(data, axis=0))` and similar commands for `axes2` and `axes3` tell the plotting library to plot the data contained within `data`.
- `fig.tight_layout()` is a misnomer as if we leave out that call the graphs will actually be squeezed together more closely!
- `matplotlib.pyplot.show()` tells the plotting library that all plots are ready and the final plot should put output.

9.3.1 Image size.

The image size requested in the program was 10.0 by 3.0 inches. If we paste the resulting image in PhotoShop and check the size it will be slightly smaller:

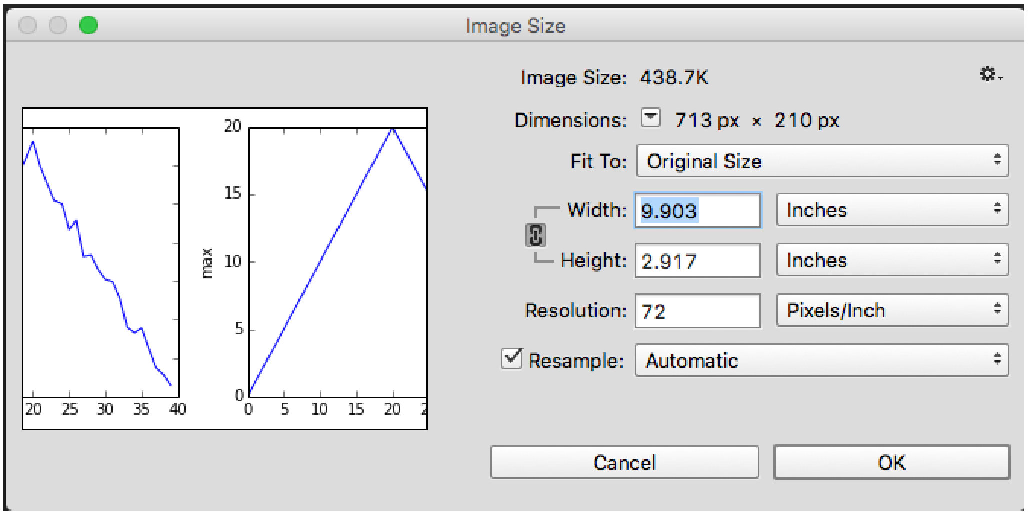

Image Size: 438.7K

Dimensions: 713 px × 210 px

Fit To: Original Size

Width: 9.903 Inches

Height: 2.917 Inches

Resolution: 72 Pixels/Inch

☑ Resample: Automatic

Figure 9.

Image size of the three plots as subplots.

The size will be in fact 9.903 x 2.917 inches. The number of pixels is 713 x 210 and the image has a default resolution of 72 dpi (dots or pixels per inch.)

10 Importing libraries “as”

Scientists dislike typing.
We will always use the syntax `import numpy` to import **NumPy**. However, in order to save typing, it is often suggested to make a shortcut like so: `import numpy as np`. If you ever see Python code online using a **NumPy** function with `np` (for example, `np.loadtxt(...)`), it’s because they’ve used this shortcut.
Note: It is therefore up to the programmer to remember that fact, and also not use the same shortcut for another imported module!

11 Check your understanding

11.1 Variable assigments

Draw diagrams showing what variables refer to what values after each statement in the following program:

```
mass = 47.5
age = 122
```

```
mass = mass * 2.0
age = age - 20
```

11.2 Sorting out references

```
first, second = 'Grace', 'Hopper'
third, fourth = second, first
print(third, fourth)
```

11.3 Slicing strings

A section of an array is called a slice. We can take slices of character strings as well:

```
element = 'oxygen'
print('first three characters:', element[0:3])
('first three characters:', 'oxy')
print('last three characters:', element[3:6])
('last three characters:', 'gen')
```

What is the value of `element[:4]`? What about `element[4:]`? Or `element[:]`?

What is `element[-1]`? What is `element[-2]`? Given those answers, explain what `element[1:-1]` does.

Once you have reflected on these questions you can use the following code to help you see if you got it right!

```
print(element[:4])
print(element[4:])
print(element[:])
print(element[-1])
print(element[-2])
print(element[1:-1])
```

11.4 Thin slices

The expression `element[3:3]` produces an ***empty string***, *i.e.*, a string that contains no characters.

If `data` holds our array of patient data, what does `data[3:3, 4:4]` produce? What about `data[3:3, :]`?

Once you have reflected on these questions you can use the following code to help you see if you got it right!

```
print(element[3:3])
print(data[3:3, 4:4])
print(data[3:3, :])
```

```
# added item:
print(data[3:4, :])
```

11.5 Plot scaling

Why do all of our plots stop just short of the upper end of our graph? If we want to change this, we can use the `set_ylim(min, max)` method of each 'axes', for example:

```
axes3.set_ylim(0,6)
```

Update your plotting code to automatically set a more appropriate scale.

Hint: you can make use of the `max` and `min` methods to help.

11.6 Make your own plot

Create a plot showing the standard deviation (`numpy.std`) of the inflammation data for each day across all patients.

Hint: you can use one of the single plots created above and alter the code to use `numpy.std`.

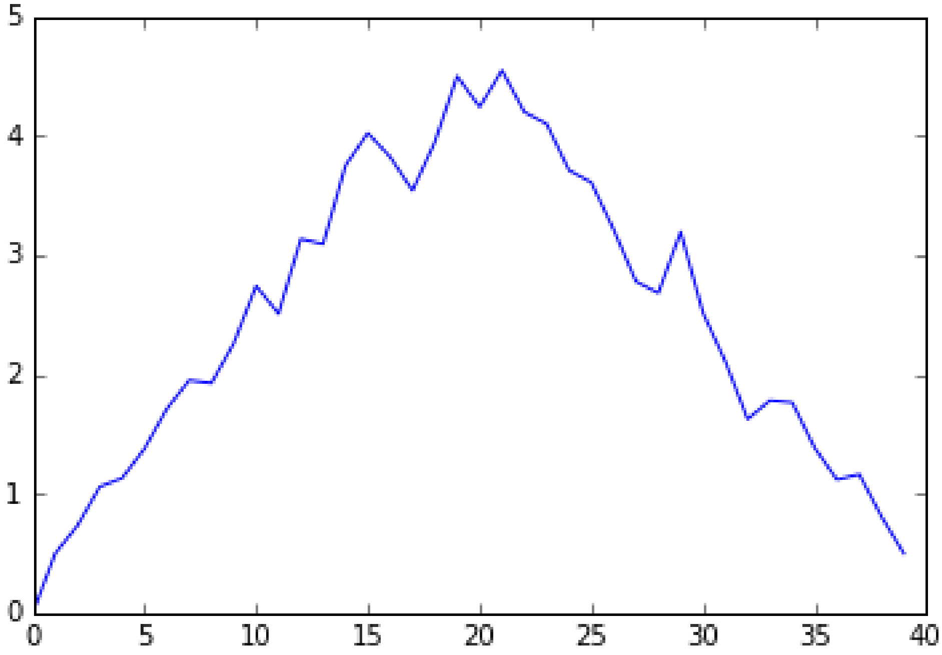

Figure 10.

Standard deviation of the inflammation data for each day across all patients using `numpy.std`.

11.7 Moving plots around

Modify the program to display the three plots on top of one another instead of side by side.

Hint: Probably need to change numbers within `fig.add_subplot()`

11.8 Stacking arrays

Arrays can be concatenated and stacked on top of one another, using NumPy's `vstack` and `hstack` functions for vertical and horizontal stacking, respectively.

```
import numpy

# Define array A
A = numpy.array([[1,2,3], [4,5,6], [7, 8, 9]])
print('A = ')
A =
print(A)

# "glue" array A side by side to make B with "horizontal stack 'numpy.hstack'"
[[1 2 3]
 [4 5 6]
 [7 8 9]]
B = numpy.hstack([A, A])
print('B = ')
B =
print(B)

# "glue" array A vertically on top of each other to make array C
[[1 2 3 1 2 3]
 [4 5 6 4 5 6]
 [7 8 9 7 8 9]]
C = numpy.vstack([A, A])
print('C = ')
C =
print(C)
[[1 2 3]
 [4 5 6]
 [7 8 9]
 [1 2 3]
 [4 5 6]
 [7 8 9]]
```

The solution to this problem is in a sense simple but at the same time tougher than one migh expect because there are no information in the above exercises that can exactly address this question.

If we decompose the question it is asked to take column one of A and "glue" it to column 2. So in theory we could extract columns 1 and 2 and then use one of the `stack` commands above.

Let's try that:

```
# get the first column
print(A[:,0])
```

```
[1 4 7]
```

```
# get the 3rd column
print(A[:,2])
```

```
[3 6 9]
```

So now we know how to extract each column, then perhaps we can “glue” them together.

```
print("trying hstack")
numpy.hstack([A[:,0], A[:,2]])
```

```
trying hstack

array([1, 4, 7, 3, 6, 9])
```

```
print("Trying vstack")
numpy.vstack([A[:,0], A[:,2]])
```

```
Trying vstack

array([[1, 4, 7],
       [3, 6, 9]])
```

The problem is that when we extracted a ***column*** the result was a transposed to fit on a single line. And no matter how we are trying to glue them it does not work!

Part of the answer can be found on stackoverflow.com[1]

```
A[:,[0,2]]
```

```
array([[1, 3],
       [4, 6],
       [7, 9]])
```

The result is indeed an array as expected. This would be hard to guess as it is mostly format-related.

References and/or Footnotes

Hamming, Richard. 1962. *Numerical Methods for Scientists and Engineers.* New York: McGraw-Hill. https://en.wikiquote.org/wiki/Richard_Hamming.

[1] http://stackoverflow.com/questions/4455076/numpy-access-an-array-by-column