This almost exactly follows SWC lesson 1, just replacing inflammation data with auto spectra.

I'm leaving out all the words/explanations in this notebook and just documenting the actual code typed into the notebook, so this notebook can only be understood in concert with SWC lesson 1.

In [ ]:
```
import numpy
```

In [ ]:
```
numpy.loadtxt(fname='spectra-01.csv', delimiter=',')
```

In [ ]:
```
weight_kg = 55
```

In [ ]:
```
print(weight_kg)
```

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

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

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

In [ ]:
```
weight_kg = 100.0
print('weight in kilograms is now:', weight_kg, 'and weight in pounds is sti
```

In [ ]:
```
%whos
```

In [ ]:
```
data = numpy.loadtxt(fname='spectra-01.csv', delimiter=',')
```

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

In [ ]:
```
print(type(data))
```

In [ ]:
```
print(data.dtype)
```

In [ ]:
```
print(data.shape)
```

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

In [ ]:
```
print('middle value in data:', data[63, 192])
```

In [ ]:
```
print(data[0:4, 0:3])
```

In [ ]:
```
print(data[4:8, 0:3])
```

In [ ]:
```
small = data[:4, 381:]
print('small is:')
print(small)
```

In [ ]:
```
doubledata = data * 2.0
```

In [ ]:
```
print('original:')
print(data[:4, 381:])
print('doubledata:')
print(doubledata[:4, 381:])
```

In [ ]:
```
tripledata = doubledata + data
```

In [ ]:
```
print('tripledata:')
print(tripledata[:4, 381:])
```

In [ ]:
```
print(numpy.mean(data))
```

In [ ]:
```
import time
print(time.ctime())
```

In [ ]:
```
maxval, minval, stdval = numpy.max(data), numpy.min(data), numpy.std(data)

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

In [ ]:
```
antenna_0 = data[0, :] # 0 on the first axis, everything on the second
print('maximum inflammation for antenna 0:', antenna_0.max())
```

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

In [ ]:
```
print(numpy.mean(data, axis=0))
```

In [ ]:
```
print(numpy.mean(data, axis=0).shape)
```

In [ ]:
```
print(numpy.mean(data, axis=1))
```

In [ ]:
```
print(numpy.mean(data, axis=1)).shape
```

In [ ]:
```
import matplotlib.pyplot
image = matplotlib.pyplot.imshow(data)
matplotlib.pyplot.show()
```

In [ ]:
```
%matplotlib inline
```

In [ ]:
```
image = matplotlib.pyplot.imshow(data)
matplotlib.pyplot.show()
```

In [ ]:
```
ave_spectrum = numpy.mean(data, axis=0)
ave_plot = matplotlib.pyplot.plot(ave_spectrum)
matplotlib.pyplot.show()
```

In [ ]:
```
max_plot = matplotlib.pyplot.plot(numpy.max(data, axis=0))
matplotlib.pyplot.show()
```

In [ ]:
```
min_plot = matplotlib.pyplot.plot(numpy.min(data, axis=0))
matplotlib.pyplot.show()
```

In [ ]:
```
import numpy
import matplotlib.pyplot

data = numpy.loadtxt(fname='spectra-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()
```

End of lesson 1

Lessons 2 & 3 exactly follows SWC, no inflammation data is invoked.

Begin Lesson 4. Unlikely to actually get to this point, but better to be prepared.

In [ ]:
```
import glob
```

In [ ]:
```
print(glob.glob('spectra*.csv'))
```

In [ ]:
```
import numpy
import matplotlib.pyplot

filenames = sorted(glob.glob('data/inflammation*.csv'))
filenames = filenames[0:3]
for f in filenames:
    print(f)

    data = numpy.loadtxt(fname=f, 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()
```

^ (../)

Programming with Python (../)

> (../02-loop/)

# Analyzing Patient Data

## Overview

**Teaching:** 30 min
**Exercises:** 0 min
**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 functions 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.
- Plot simple graphs from data.

In this lesson we will learn how to manipulate the inflammation dataset with Python. But before we discuss how to deal with many data points, we will show how to store a single value on the computer.

The line below assigns (reference.html#assignment) the value `55` to a variable (reference.html#variable) `weight_kg`:

```
weight_kg = 55
```

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 (reference.html#case-sensitive). We can create a new variable by assigning a value to it using `=`. When we are finished typing and press Shift+Enter, the notebook runs our command.

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

```
print(weight_kg)
```

```
55
```

and do arithmetic with it:

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

```
weight in pounds: 121.0
```

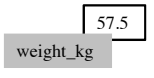
As the example above shows, we can print several things at once by separating them with commas.

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
```

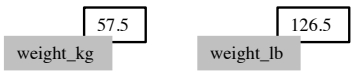
If we imagine the variable as a sticky note with a name written on it, assignment is like putting the sticky note on a particular value:




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)
```

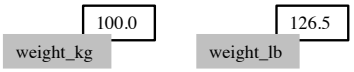
```
weight in kilograms: 57.5 and in pounds: 126.5
```



and then 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
```




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.

### 📌 Who's Who in Memory

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
```

Words are useful, but what's more useful are the sentences and stories we build with them. Similarly, while a lot of powerful, general tools are built into languages like Python, specialized tools built up from these basic units live in libraries (reference.html#library) that can be called upon when needed.

In order to load our inflammation data, we need to access (import (reference.html#import) in Python terminology) a library called NumPy (http://docs.scipy.org/doc/numpy/). In general you should use this library if you want to do fancy things with numbers, especially if you have matrices or arrays. We can import NumPy using:

```
import numpy
```

Importing a library is like getting a piece of lab equipment out of a storage locker and setting it up on the bench. Libraries provide additional functionality to the basic Python package, much like a new piece of equipment adds functionality to a lab space. Just like in the lab, importing too many libraries can sometimes complicate and slow down your programs - so we only import what we need for each program. Once you've imported the library, we can ask the library to read our data file for us:

```
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.]])
```

The expression `numpy.loadtxt(...)` is a function call (reference.html#function-call) that asks Python to run the function (reference.html#function) `loadtxt` which belongs to the `numpy` library. This dotted notation (reference.html#dotted-notation) is used everywhere in Python to refer to the parts of things as `thing.component` .

`numpy.loadtxt` has two parameters (reference.html#parameter): the name of the file we want to read, and the delimiter (reference.html#delimiter) that separates values on a line. These both need to be character strings (or strings (reference.html#string) 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 its result:

```
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:

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

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

```
<class '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.

### 📌 Data Type

A Numpy array contains one or more elements of the same type. `type` will only tell you that a variable is a NumPy array. We can also find out the type of the data contained in the NumPy array.

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

```
dtype('float64')
```

This tells us that the NumPy array's elements are floating-point numbers (../reference/#floating-point number).

We can see what the array's shape (../reference/#shape) is like this:

```
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 (../reference/#member) 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.

If we want to get a single number from the array, we must provide an index (../reference/#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, MATLAB and R 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 it represents an offset from the first value in the array (the second value is offset by one index from the first value). This is closer to the way that computers represent arrays (if you are interested in the historical reasons behind counting indices from zero, you can read Mike Hoye's blog post (http://exple.tive.org/blarg/2013/10/22/citation-needed/)). 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.

### 📌 In the 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.

An index like `[30, 20]` selects a single element of an array, but we can 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:

```
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.]]
```

The slice (../reference/#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.

We don't have to start slices 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.]]
```

We also 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, and if we don't include either (i.e., if we just use ':' on its own), the slice includes everything:

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

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

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. Thus:

```
doubledata = data * 2.0
```

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('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.]]
```

If, instead of taking an array and doing arithmetic with a single value (as above) 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.]]
```

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
```

`mean` is a function (../reference/#function) that takes an array as an argument (../reference/#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())
```

```
'Sat Mar 26 13:07:33 2016'
```

For functions that don't take in any arguments, we still need parentheses ( `()` ) to tell Python to go and do something for us.

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.

```
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)` .

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. 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
```

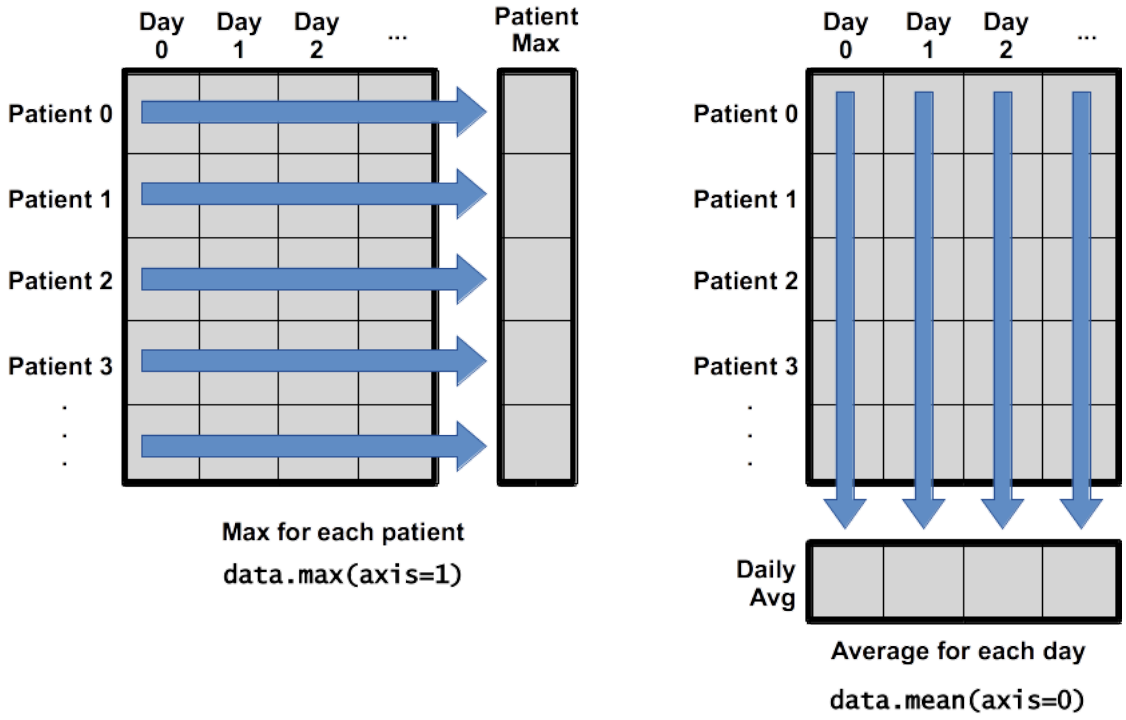
Everything in a line of code following the '#' symbol is a comment (../reference/#comment) that is ignored by the computer. Comments allow programmers to leave explanatory notes for other programmers or their future selves.

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 each patient over all days (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:




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
   0.56666667]
```

As a quick check, we can ask this array what its shape is:

```
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 get:

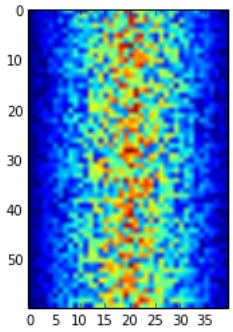
```
print(numpy.mean(data, axis=1))
```

```
[ 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.

The mathematician Richard Hamming 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 (of 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. 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()
```




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.

### 📌 Some IPython Magic

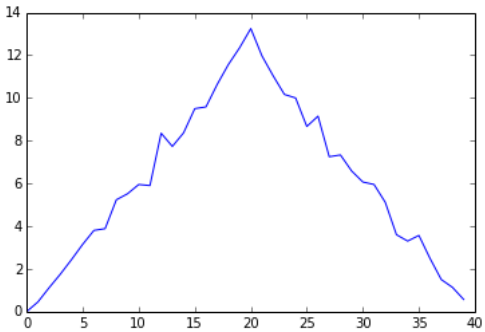
If you're using an IPython / Jupyter notebook, you'll need to execute the following command in order for your matplotlib images to appear in the notebook when `show()` is called:

```
%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.

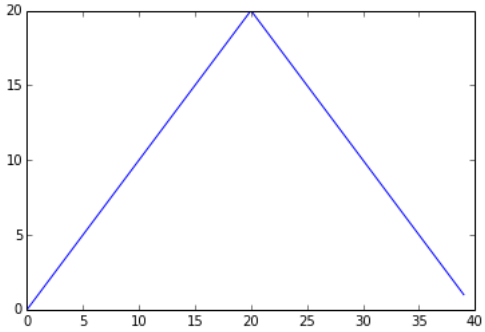
Let's take a look at the average inflammation over time:

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

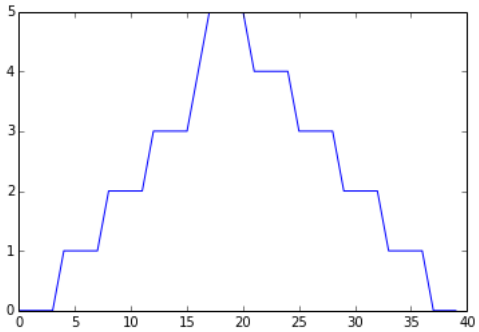



Here, we have put the average per day across all patients in the variable `ave_inflammation`, then asked `matplotlib.pyplot` to create and display a line graph of those values. The result is roughly a linear rise and fall, which is suspicious: based on other studies, we expect a sharper rise and slower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

```
max_plot = matplotlib.pyplot.plot(numpy.max(data, axis=0))
matplotlib.pyplot.show()
```



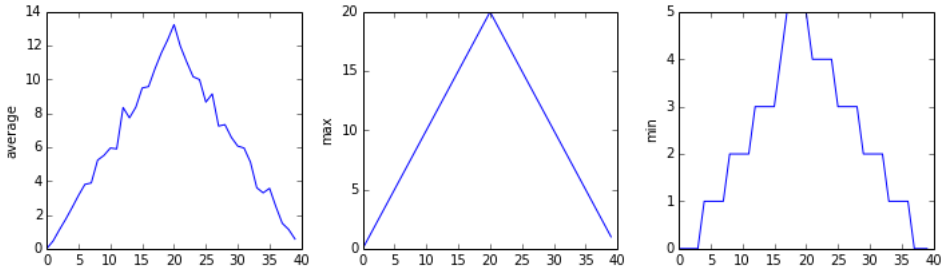

```
min_plot = matplotlib.pyplot.plot(numpy.min(data, axis=0))
matplotlib.pyplot.show()
```



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.

You can group similar plots in a single figure using subplots. This 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 (../reference/#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()
```




The call (../reference/#function-call) to `loadtxt` reads our data, and the rest of the program tells the plotting library how large we want the figure to be, that we're creating three subplots, what to draw for each one, and that we want a tight layout. (Perversely, if we leave out that call to `fig.tight_layout()` , the graphs will actually be squeezed together more closely.)

### 📌 Scientists Dislike Typing

We will always use the syntax `import numpy` to import NumPy. However, in order to save typing, it is often suggested (http://www.scipy.org/getting-started.html#an-example-script) 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. When working with other people, it is important to agree on a convention of how common libraries are imported.

## Check Your Understanding

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
```

## Sorting Out References

What does the following program print out?

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

**Solution**

## Slicing Strings

A section of an array is called a slice (../reference/#slice). We can take slices of character strings as well:

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

```
first three characters: oxy
last three characters: gen
```

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

**Solution**

What is `element[-1]` ? What is `element[-2]` ?

**Solution**

Given those answers, explain what `element[1:-1]` does.

**Solution**

## Thin Slices

The expression `element[3:3]` produces an empty string (../reference/#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, :]` ?

**Solution**

## Plot Scaling

Why do all of our plots stop just short of the upper end of our graph?

**Solution**

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.)

**Solution**

**Solution**

## Drawing Straight Lines

In the center and right subplots above, we expect all lines to look like step functions, because non-integer value are not realistic for the minimum and maximum values. However, you can see that the lines are not always vertical or horizontal, and in particular the step function in the subplot on the right looks slanted. Why is this?

**Solution**

## Make Your Own Plot

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

**Solution**

## Moving Plots Around

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

**Solution**

## 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

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

B = numpy.hstack([A, A])
print('B = ')
print(B)

C = numpy.vstack([A, A])
print('C = ')
print(C)
```

```
A =
[[1 2 3]
 [4 5 6]
 [7 8 9]]
B =
[[1 2 3 1 2 3]
 [4 5 6 4 5 6]
 [7 8 9 7 8 9]]
C =
[[1 2 3]
 [4 5 6]
 [7 8 9]
 [1 2 3]
 [4 5 6]
 [7 8 9]]
```

Write some additional code that slices the first and last columns of `A`, and stacks them into a 3x2 array. Make sure to `print` the results to verify your solution.

### Solution

### Solution

## Change In Inflammation

This patient data is *longitudinal* in the sense that each row represents a series of observations relating to one individual. This means that change inflammation is a meaningful concept.

The `numpy.diff()` function takes a NumPy array and returns the difference along a specified axis.

Which axis would it make sense to use this function along?

### Solution

If the shape of an individual data file is `(60, 40)` (60 rows and 40 columns), what would the shape of the array be after you run the `diff()` function and why?

### Solution

How would you find the largest change in inflammation for each patient? Does it matter if the change in inflammation is an increase or a decrease?

### Solution

< (../01-numpy/)

Programming with Python (../)

> (../03-lists/)

# Repeating Actions with Loops

## Overview

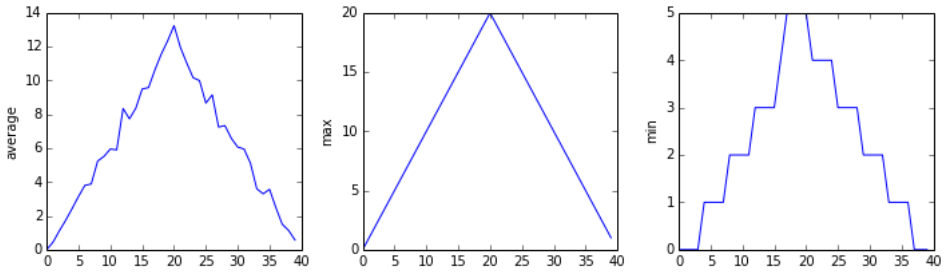
**Teaching:** 30 min
**Exercises:** 0 min
**Questions**

- How can I do the same operations on many different values?

**Objectives**

- Explain what a for loop does.
- Correctly write for loops to repeat simple calculations.
- Trace changes to a loop variable as the loop runs.
- Trace changes to other variables as they are updated by a for loop.

In the last lesson, we wrote some code that plots some values of interest from our first inflammation dataset, and reveals some suspicious features in it, such as from `inflammation-01.csv`




We have a dozen data sets right now, though, and more on the way. We want to create plots for all of our data sets with a single statement. To do that, we'll have to teach the computer how to repeat things.

An example task that we might want to repeat is printing each character in a word on a line of its own.

```
word = 'lead'
```

We can access a character in a string using its index. For example, we can get the first character of the word 'lead', by using word[0]. One way to print each character is to use four `print` statements:

```
print(word[0])
print(word[1])
print(word[2])
print(word[3])
```

```
l
e
a
d
```

This is a bad approach for two reasons:

1. It doesn't scale: if we want to print the characters in a string that's hundreds of letters long, we'd be better off just typing them in.
2. It's fragile: if we give it a longer string, it only prints part of the data, and if we give it a shorter one, it produces an error because we're asking for characters that don't exist.

```
word = 'tin'
print(word[0])
print(word[1])
print(word[2])
print(word[3])
```

```
t
i
n
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-3-7974b6cdaf14> in <module>()
      3 print(word[1])
      4 print(word[2])
----> 5 print(word[3])

IndexError: string index out of range
```

Here's a better approach:

```
word = 'lead'
for char in word:
    print(char)
```

```
l
e
a
d
```

This is shorter—certainly shorter than something that prints every character in a hundred-letter string—and more robust as well:

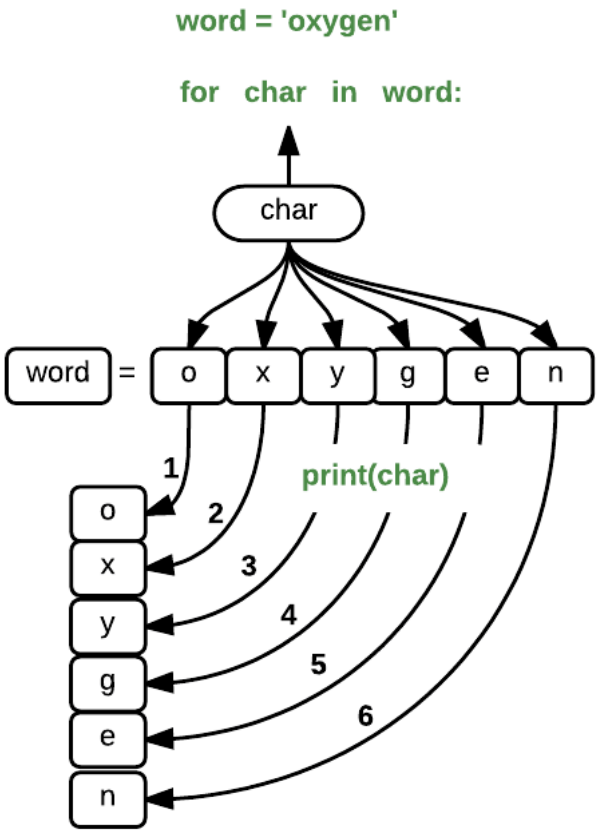
```
word = 'oxygen'
for char in word:
    print(char)
```

```
o
x
y
g
e
n
```

The improved version uses a for loop (../reference/#for-loop) to repeat an operation—in this case, printing—once for each thing in a sequence. The general form of a loop is:

```
for element in variable:
    do things with element
```

Using the oxygen example above, the loop might look like this:



where each character ( `char` ) in the variable `word` is looped through and printed one character after another. The numbers in the diagram denote which loop cycle the character was printed in (1 being the first loop, and 6 being the final loop).

We can call the loop variable (../reference/#loop-variable) anything we like, but there must be a colon at the end of the line starting the loop, and we must indent anything we want to run inside the loop. Unlike many other languages, there is no command to signify the end of the loop body (e.g. end for); what is indented after the for statement belongs to the loop.

### 📌 What's in a name?

In the example above, the loop variable was given the name `char` as a mnemonic; it is short for 'character'. 'Char' is not a keyword in Python that pulls the characters from words or strings. In fact when a similar loop is run over a list rather than a word, the output would be each member of that list printed in order, rather than the characters.

```
elements = ['oxygen', 'nitrogen', 'argon']
for char in elements:
   print(char)
```

```
oxygen
nitrogen
argon
```

We can choose any name we want for variables. We might just as easily have chosen the name `banana` for the loop variable, as long as we use the same name when we invoke the variable inside the loop:

```
word = 'oxygen'
for banana in word:
    print(banana)
```

```
o
x
y
g
e
n
```

It is a good idea to choose variable names that are meaningful so that it is easier to understand what the loop is doing.

Here's another loop that repeatedly updates a variable:

```
length = 0
for vowel in 'aeiou':
    length = length + 1
print('There are', length, 'vowels')
```

```
There are 5 vowels
```

It's worth tracing the execution of this little program step by step. Since there are five characters in `'aeiou'` , the statement on line 3 will be executed five times. The first time around, `length` is zero (the value assigned to it on line 1) and `vowel` is `'a'` . The statement adds 1 to the old value of `length` , producing 1, and updates `length` to refer to that new value. The next time around, `vowel` is `'e'` and `length` is 1, so `length` is updated to be 2. After three more updates, `length` is 5; since there is nothing left in `'aeiou'` for Python to process, the loop finishes and the `print` statement on line 4 tells us our final answer.

Note that a loop variable is just a variable that's being used to record progress in a loop. It still exists after the loop is over, and we can re-use variables previously defined as loop variables as well:

```
letter = 'z'
for letter in 'abc':
    print(letter)
print('after the loop, letter is', letter)
```

```
a
b
c
after the loop, letter is c
```

Note also that finding the length of a string is such a common operation that Python actually has a built-in function to do it called `len` :

```
print(len('aeiou'))
```

```
5
```

`len` is much faster than any function we could write ourselves, and much easier to read than a two-line loop; it will also give us the length of many other things that we haven't met yet, so we should always use it when we can.

### From 1 to N

Python has a built-in function called `range` that creates a sequence of numbers. Range can accept 1-3 parameters. If one parameter is input, range creates an array of that length, starting at zero and incrementing by 1. If 2 parameters are input, range starts at the first and ends just before the second, incrementing by one. If range is passed 3 parameters, it starts at the first one, ends just before the second one, and increments by the third one. For example, `range(3)` produces the numbers 0, 1, 2, while `range(2, 5)` produces 2, 3, 4, and `range(3, 10, 3)` produces 3, 6, 9. Using `range`, write a loop that uses `range` to print the first 3 natural numbers:

```
1
2
3
```

**Solution**

### Computing Powers With Loops

Exponentiation is built into Python:

```
print(5 ** 3)
```

```
125
```

Write a loop that calculates the same result as `5 ** 3` using multiplication (and without exponentiation).

**Solution**

### Reverse a String

Knowing that two strings can be concatenated using the `+` operator, write a loop that takes a string and produces a new string with the characters in reverse order, so `'Newton'` becomes `'notweN'`.

**Solution**

### Computing the Value of a Polynomial

The built-in function `enumerate` takes a sequence (e.g. a list) and generates a new sequence of the same length. Each element of the new sequence is a pair composed of the index (0, 1, 2,...) and the value from the original sequence:

```
for i, x in enumerate(xs):
    # Do something with i and x
```

The loop above assigns the index to `i` and the value to `x`.

Suppose you have encoded a polynomial as a list of coefficients in the following way: the first element is the constant term, the second element is the coefficient of the linear term, the third is the coefficient of the quadratic term, etc.

```
x = 5
cc = [2, 4, 3]
```

```
y = cc[0] * x**0 + cc[1] * x**1 + cc[2] * x**2
y = 97
```

Write a loop using `enumerate(cc)` which computes the value `y` of any polynomial, given `x` and `cc`.

**Solution**

### Key Points

- Use `for variable in sequence` to process the elements of a sequence one at a time.
- The body of a for loop must be indented.
- Use `len(thing)` to determine the length of something that contains other values.

< (../01-numpy/)

› (../03-lists/)



Edit on GitHub (https://github.com/swcarpentry/python-novice-inflammation/edit/gh-pages/_episodes/02-loop.md) / Contributing (https://github.com/swcarpentry/python-novice-inflammation/blob/gh-pages/CONTRIBUTING.md) / Source (https://github.com/swcarpentry/python-novice-inflammation/) / Cite (https://github.com/swcarpentry/python-novice-inflammation/blob/gh-pages/CITATION) / Contact ()

‹ (../02-loop/)

Programming with Python (../)

› (../04-files/)

# Storing Multiple Values in Lists

## Overview

**Teaching:** 30 min
**Exercises:** 0 min
**Questions**

- How can I store many values together?

**Objectives**

- Explain what a list is.
- Create and index lists of simple values.

Just as a `for` loop is a way to do operations many times, a list is a way to store many values. Unlike NumPy arrays, lists are built into the language (so we don't have to load a library to use them). We create a list by putting values inside square brackets and separating the values with commas:

```
odds = [1, 3, 5, 7]
print('odds are:', odds)
```

```
odds are: [1, 3, 5, 7]
```

We select individual elements from lists by indexing them:

```
print('first and last:', odds[0], odds[-1])
```

```
first and last: 1 7
```

and if we loop over a list, the loop variable is assigned elements one at a time:

```
for number in odds:
    print(number)
```

```
1
3
5
7
```

There is one important difference between lists and strings: we can change the values in a list, but we cannot change individual characters in a string. For example:

```
names = ['Newton', 'Darwing', 'Turing'] # typo in Darwin's name
print('names is originally:', names)
names[1] = 'Darwin' # correct the name
print('final value of names:', names)
```

```
names is originally: ['Newton', 'Darwing', 'Turing']
final value of names: ['Newton', 'Darwin', 'Turing']
```

works, but:

```
name = 'Darwin'
name[0] = 'd'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-8-220df48aeb2e> in <module>()
      1 name = 'Darwin'
----> 2 name[0] = 'd'

TypeError: 'str' object does not support item assignment
```

does not.

## 📌 Ch-Ch-Ch-Changes

Data which can be modified in place is called mutable (../reference/#mutable), while data which cannot be modified is called immutable (../reference/#immutable). Strings and numbers are immutable. This does not mean that variables with string or number values are constants, but when we want to change the value of a string or number variable, we can only replace the old value with a completely new value.

Lists and arrays, on the other hand, are mutable: we can modify them after they have been created. We can change individual elements, append new elements, or reorder the whole list. For some operations, like sorting, we can choose whether to use a function that modifies the data in place or a function that returns a modified copy and leaves the original unchanged.

Be careful when modifying data in place. If two variables refer to the same list, and you modify the list value, it will change for both variables!

```
salsa = ['peppers', 'onions', 'cilantro', 'tomatoes']
mySalsa = salsa        # <-- mySalsa and salsa point to the *same* list data in memory
salsa[0] = 'hot peppers'
print('Ingredients in my salsa:', mySalsa)
```

```
Ingredients in my salsa: ['hot peppers', 'onions', 'cilantro', 'tomatoes']
```

If you want variables with mutable values to be independent, you must make a copy of the value when you assign it.

```
salsa = ['peppers', 'onions', 'cilantro', 'tomatoes']
mySalsa = list(salsa)        # <-- makes a *copy* of the list
salsa[0] = 'hot peppers'
print('Ingredients in my salsa:', mySalsa)
```

```
Ingredients in my salsa: ['peppers', 'onions', 'cilantro', 'tomatoes']
```

Because of pitfalls like this, code which modifies data in place can be more difficult to understand. However, it is often far more efficient to modify a large data structure in place than to create a modified copy for every small change. You should consider both of these aspects when writing your code.

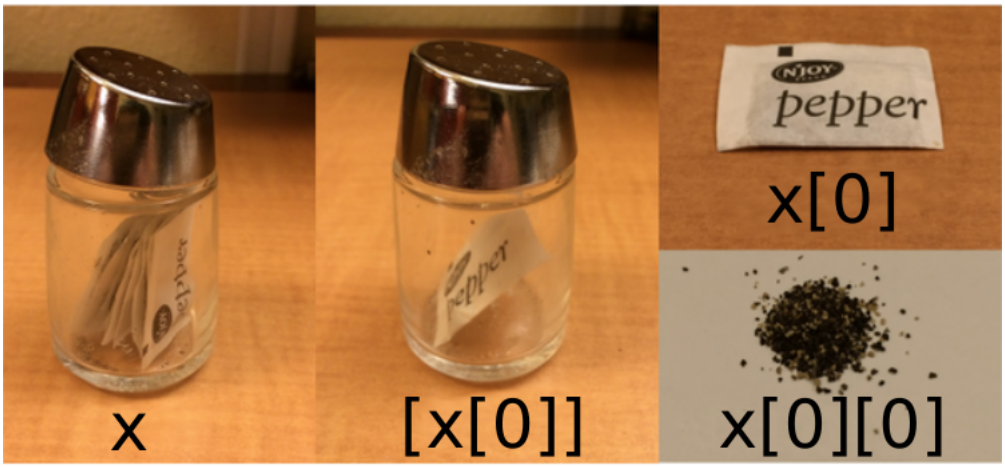
## 📌 Nested Lists

Since lists can contain any Python variable, it can even contain other lists.

For example, we could represent the products in the shelves of a small grocery shop:

```
x = [['pepper', 'zucchini', 'onion'],
     ['cabbage', 'lettuce', 'garlic'],
     ['apple', 'pear', 'banana']]
```

Here is a visual example of how indexing a list of lists `x` works:




(https://twitter.com/hadleywickham/status/643381054758363136)

Using the previously declared list `x`, these would be the results of the index operations shown in the image:

```
print([x[0]])
```

```
[['pepper', 'zucchini', 'onion']]
```

```
print(x[0])
```

```
['pepper', 'zucchini', 'onion']
```

```
print(x[0][0])
```

```
'pepper'
```

Thanks to Hadley Wickham (https://twitter.com/hadleywickham/status/643381054758363136) for the image above.

There are many ways to change the contents of lists besides assigning new values to individual elements:

```
odds.append(11)
print('odds after adding a value:', odds)
```

```
odds after adding a value: [1, 3, 5, 7, 11]
```

```
del odds[0]
print('odds after removing the first element:', odds)
```

```
odds after removing the first element: [3, 5, 7, 11]
```

```
odds.reverse()
print('odds after reversing:', odds)
```

```
odds after reversing: [11, 7, 5, 3]
```

While modifying in place, it is useful to remember that Python treats lists in a slightly counter-intuitive way.

If we make a list and (attempt to) copy it then modify in place, we can cause all sorts of trouble:

```
odds = [1, 3, 5, 7]
primes = odds
primes.append(2)
print('primes:', primes)
print('odds:', odds)
```

```
primes: [1, 3, 5, 7, 2]
odds: [1, 3, 5, 7, 2]
```

This is because Python stores a list in memory, and then can use multiple names to refer to the same list. If all we want to do is copy a (simple) list, we can use the `list` function, so we do not modify a list we did not mean to:

```
odds = [1, 3, 5, 7]
primes = list(odds)
primes.append(2)
print('primes:', primes)
print('odds:', odds)
```

```
primes: [1, 3, 5, 7, 2]
odds: [1, 3, 5, 7]
```

This is different from how variables worked in lesson 1, and more similar to how a spreadsheet works.

### Turn a String Into a List

Use a for-loop to convert the string “hello” into a list of letters:

```
["h", "e", "l", "l", "o"]
```

Hint: You can create an empty list like this:

```
my_list = []
```

**Solution**

Subsets of lists and strings can be accessed by specifying ranges of values in brackets, similar to how we accessed ranges of positions in a Numpy array. This is commonly referred to as “slicing” the list/string.

```
binomial_name = "Drosophila melanogaster"
group = binomial_name[0:10]
print("group:", group)

species = binomial_name[11:24]
print("species:", species)

chromosomes = ["X", "Y", "2", "3", "4"]
autosomes = chromosomes[2:5]
print("autosomes:", autosomes)

last = chromosomes[-1]
print("last:", last)
```

```
group: Drosophila
species: melanogaster
autosomes: ["2", "3", "4"]
last: 4
```

## Slicing From the End

Use slicing to access only the last four characters of a string or entries of a list.

```
string_for_slicing = "Observation date: 02-Feb-2013"
list_for_slicing = [["fluorine", "F"], ["chlorine", "Cl"], ["bromine", "Br"], ["iodine", "I"], ["astatine", "At"]]
```

```
"2013"
[["chlorine", "Cl"], ["bromine", "Br"], ["iodine", "I"], ["astatine", "At"]]
```

Would your solution work regardless of whether you knew beforehand the length of the string or list (e.g. if you wanted to apply the solution to a set of lists of different lengths)? If not, try to change your approach to make it more robust.

### Solution

## Non-Continuous Slices

So far we've seen how to use slicing to take single blocks of successive entries from a sequence. But what if we want to take a subset of entries that aren't next to each other in the sequence?

You can achieve this by providing a third argument to the range within the brackets, called the *step size*. The example below shows how you can take every third entry in a list:

```
primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37]
subset = primes[0:12:3]
print("subset", subset)
```

```
subset [2, 7, 17, 29]
```

Notice that the slice taken begins with the first entry in the range, followed by entries taken at equally-spaced intervals (the steps) thereafter. If you wanted to begin the subset with the third entry, you would need to specify that as the starting point of the sliced range:

```
primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37]
subset = primes[2:12:3]
print("subset", subset)
```

```
subset [5, 13, 23, 37]
```

Use the step size argument to create a new string that contains only every other character in the string "In an octopus's garden in the shade"

```
beatles = "In an octopus's garden in the shade"
```

```
I notpssgre ntesae
```

### Solution

If you want to take a slice from the beginning of a sequence, you can omit the first index in the range:

```
date = "Monday 4 January 2016"
day = date[0:6]
print("Using 0 to begin range:", day)
day = date[:6]
print("Omitting beginning index:", day)
```

```
Using 0 to begin range: Monday
Omitting beginning index: Monday
```

And similarly, you can omit the ending index in the range to take a slice to the very end of the sequence:

```
months = ["jan", "feb", "mar", "apr", "may", "jun", "jul", "aug", "sep", "oct", "nov", "dec"]
sond = months[8:12]
print("With known last position:", sond)
sond = months[8:len(months)]
print("Using len() to get last entry:", sond)
sond = months[8:]
("Omitting ending index:", sond)
```

```
With known last position: ["sep", "oct", "nov", "dec"]
Using len() to get last entry: ["sep", "oct", "nov", "dec"]
Omitting ending index: ["sep", "oct", "nov", "dec"]
```

### Swapping the contents of variables

Explain what the overall effect of this code is:

```
left = 'L'
right = 'R'

temp = left
left = right
right = temp
```

Compare it to:

```
left, right = [right, left]
```

Do they always do the same thing? Which do you find easier to read?

#### Solution

### Overloading

`+` usually means addition, but when used on strings or lists, it means “concatenate”. Given that, what do you think the multiplication operator `*` does on lists? In particular, what will be the output of the following code?

```
counts = [2, 4, 6, 8, 10]
repeats = counts * 2
print(repeats)
```

1. `[2, 4, 6, 8, 10, 2, 4, 6, 8, 10]`
2. `[4, 8, 12, 16, 20]`
3. `[[2, 4, 6, 8, 10],[2, 4, 6, 8, 10]]`
4. `[2, 4, 6, 8, 10, 4, 8, 12, 16, 20]`

The technical term for this is *operator overloading*: a single operator, like `+` or `*`, can do different things depending on what it’s applied to.

#### Solution

### Key Points

- `[value1, value2, value3, ...]` creates a list.
- Lists are indexed and sliced in the same way as strings and arrays.
- Lists are mutable (i.e., their values can be changed in place).
- Strings are immutable (i.e., the characters in them cannot be changed).

## < (../02-loop/)

## > (../04-files/)



Edit on GitHub (https://github.com/swcarpentry/python-novice-inflammation/edit/gh-pages/_episodes/03-lists.md) / Contributing (https://github.com/swcarpentry/python-novice-inflammation/blob/gh-pages/CONTRIBUTING.md) / Source

< (../03-lists/)

Programming with Python (../)

> (../05-cond/)

# Analyzing Data from Multiple Files

## Overview

**Teaching:** 20 min
**Exercises:** 0 min
**Questions**

- How can I do the same operations on many different files?

**Objectives**

- Use a library function to get a list of filenames that match a simple wildcard pattern.
- Write a for loop to process multiple files.

We now have almost everything we need to process all our data files. The only thing that's missing is a library with a rather unpleasant name:

```
import glob
```

The `glob` library contains a function, also called `glob`, that finds files and directories whose names match a pattern. We provide those patterns as strings: the character `*` matches zero or more characters, while `?` matches any one character. We can use this to get the names of all the CSV files in the current directory:

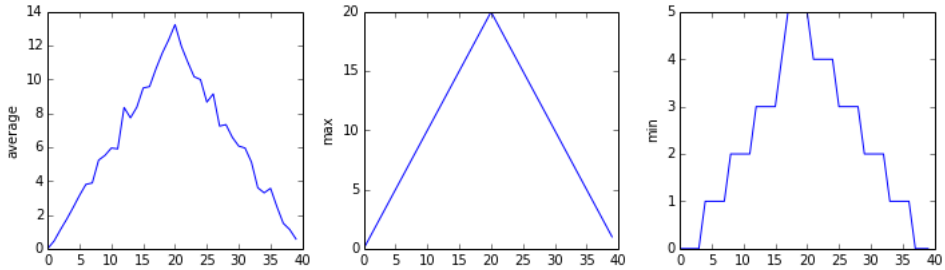
```
print(glob.glob('data/inflammation*.csv'))
```

```
['data/inflammation-05.csv', 'data/inflammation-11.csv', 'data/inflammation-12.csv', 'data/inflammation-08.csv', 'data/inflammation-03.csv', 'data/inflammation-06.csv', 'data/inflammation-09.csv', 'data/inflammation-07.csv', 'data/inflammation-10.csv', 'data/inflammation-02.csv', 'data/inflammation-04.csv', 'data/inflammation-01.csv']
```

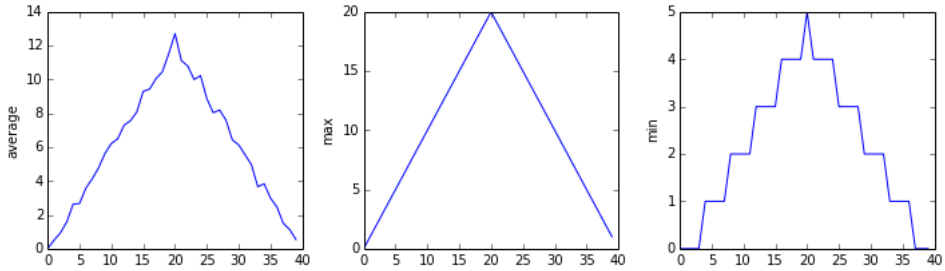
As these examples show, `glob.glob`'s result is a list of file and directory paths in arbitrary order. This means we can loop over it to do something with each filename in turn. In our case, the "something" we want to do is generate a set of plots for each file in our inflammation dataset. If we want to start by analyzing just the first three files in alphabetical order, we can use the `sorted` built-in function to generate a new sorted list from the `glob.glob` output:

```
import numpy
import matplotlib.pyplot

filenames = sorted(glob.glob('data/inflammation*.csv'))
filenames = filenames[0:3]
for f in filenames:
    print(f)

    data = numpy.loadtxt(fname=f, 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()
```

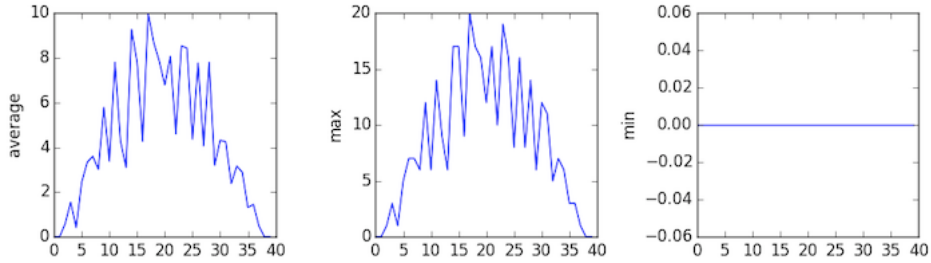
```
inflammation-01.csv
```




```
inflammation-02.csv
```




```
inflammation-03.csv
```




Sure enough, the maxima of the first two data sets show exactly the same ramp as the first, and their minima show the same staircase structure; a different situation has been revealed in the third dataset, where the maxima are a bit less regular, but the minima are consistently zero.

## Plotting Differences

Plot the difference between the average of the first dataset and the average of the second dataset, i.e., the difference between the leftmost plot of the first two figures.

### Solution

## Generate Composite Statistics

Use each of the files once to generate a dataset containing values averaged over all patients:

```
filenames = glob.glob('data/inflammation*.csv')
composite_data = numpy.zeros((60,40))
for f in filenames:
    # sum each new file's data into as it's read
# and then divide the composite_data by number of samples
composite_data /= len(filenames)
```

Then use pyplot to generate average, max, and min for all patients.

### Solution

## Key Points

- Use `glob.glob(pattern)` to create a list of files whose names match a pattern.
- Use `*` in a pattern to match zero or more characters, and `?` to match any single character.

(../03-lists/)