Python #2
Analyzing Patient Data

Location: https://people.sc.fsu.edu/~jburkardt/classes/python_2022/python02/python02.pdf

Freely adapted from the Python lessons at https://software-carpentry.org/

Working with a data file

- *Data is organized into lists and tables, and stored in files;*
- *To work with such data, we will need functions from the* `numpy` *library;*
- *Our example will involve a file of medical data;*
- *We want to transfer the data from the file, and store it as a variable;*
- *This variable will use a new structure, the* `numpy` *array;*
- *Then we want to examine individual values, rows, and columns of the data;*
- *We can compute some simple data statistics;*
- *Arrays can be indexed, stacked, and differenced.*

1 Data stored in CSV format

A doctor has performed a medical experiment, to test a new treatment for arthritis. 60 patients were involved in the study, over a period of 40 days. On each day, each patient's level of inflammation was measured, on a scaled from 0 to 20. Thus, the results of the medical study can be thought of as a 60 row by 40 column table of numbers.

This data is stored in a file named `inflammation-01.csv`. The file can be downloaded from the class website. Another way to get this file is to go to

`https://people.sc.fsu.edu/~jburkardt/classes/python_2022/python_2022.html`

then choose `datasets`, which will give you a menu that includes the file we are interested in. The file extension `.csv` tells us that the data is stored as *comma separated values*, that is, the file consists of 60 lines of text, and each such line contains 40 numbers, separated by commas. Here is what the first three lines of the file look like:

```
0,0,1,3,1,2,4,7,8,3,3,3,10,5,7,4,7,7,12,18,6,13,11,11,7,7,4,6,8,8,4,4,5,7,3,4,2,3,0,0
0,1,2,1,2,1,3,2,2,6,10,11,5,9,4,4,7,16,8,6,18,4,12,5,12,7,11,5,11,3,3,5,4,4,5,5,1,1,0,1
0,1,1,3,3,2,6,2,5,9,5,7,4,5,4,15,5,11,9,10,19,14,12,17,7,12,11,7,4,2,10,5,4,2,2,3,2,2,1,1
```

In order to work with this data, we first need to transfer if from this file into the computer. To do so, we have to find the appropriate file reader function that can handle this type of data file.

2 The `numpy` library

Along with the basic Python programming language, there are many libraries of functions that you can access. The most useful is called `numpy`, which is designed to strengthen Python's ability to do numerical calculations. This library includes a file-reading function that we will want to use, and so it's time to learn a little about Python libraries.

In order to access functions within a library, we have to make a request for access, using the `import` statement. For us, this will be

```
>>> import numpy
```

If, for some reason, you have not installed the `numpy` library on your computer, you will unfortunately get the message

```
ModuleNotFoundError: No module named 'numpy'
```

and you will have to go back and check that you have properly installed Anaconda.

But assuming `import` statement was accepted, you now have access to all the functions in the library. These functions are named in the usual way, but because they come from a separate library, we need to specify that library name as well. Thus, for example, `numpy` has a square root function named `sqrt()`, but in order for us to use it, we type

```
>>> x = numpy.sqrt ( 20.0 )
```

It is also possible, and very common, to specify a shorter version of the library name, to cut down on typing. In that case, we might have started by saying

```
>>> import numpy as np
```

after which we can use the prefix `np` instead of `numpy`. So in many Python codes, you will see commands like

```
>>> x = np.sqrt ( 20.0 )
```

For now, we will stick with using the full library name.

The important thing to note is that `numpy()` includes a function `loadtxt()` that we can use in order to read our data file.

3 Reading a data file with `loadtxt()`

Our command to read the data file will be:

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

Of course, `data` is the variable name that will be storing our information. Inside the parentheses of the call to `loadtxt()` are two arguments, both of which use the format `keyword = value`:

- The argument `fname = 'inflammation-01.csv'` specifies the name of the file we want to read.
- The argument `delimiter= ','` indicates that the information in the file is separated by commas. Not all data files use commas as separators; sometimes a space or a TAB character is used instead, but loadtxt() can handle all the possibilities as long as we give it the correct information.

This command will fail if loadtxt() cannot find the file (you didn't copy it into your current directory), or if the data file is not formatted correctly (missing commas, or rows having different numbers of data items).

But assuming the command is accepted, we now have a variable storing the data from the file.

4 What's in the array?

The `print()` statement will give us an idea of what's stored in the variable we just created:

```
>>> print ( data )
```

Notice that, although the file contains 60 rows and 40 columns, the `print()` statement just gives us a selection of the data, enough to get an idea of what is in there. This is a graceful approach by Python. If we really want to see all the data, there are always ways to make that happen.

Oddly, if we ask for the type of our variable `data`, we get something we haven't seen before:

```
>>> print ( type ( data ) )
<class 'numpy.ndarray'>
```

Instead of an integer, string, or decimal number, we have an `ndarray`. This is simply a shorthand for "n-dimensional array", a list, table, or even more complicated structure.

If we wish to know more about the quantities inside the array, we have to use a new command that is valid for any `numpy` array. This command uses a different format than we have seen before. We state the name of the array, followed by a period, followed by the string `dtype`, which stands for *data type*.

```
>>> print ( data.dtype )
float64
```

This tells us that the entries in the array are floating point numbers, (what we sometimes think of as decimal numbers). The `64` tells us that 64 binary digits are used to describe the numbers, which means we have about 16 decimal digits of accuracy.

Since `data` is a table now, we may very well want to know its size, that is, the number of rows and columns. This can be found by another `numpy` command:

```
>>> print ( data.shape )
(60, 40)
```

In other words, `data` stores 60 rows and 40 columns of data.

5 Accessing an array element

If we want to look at a single value from the array, we specify it by listing the array name, followed by indexing information in square brackets. Since `data` is a table, we need to specify two index values, specifying the row and column, often symbolized by `i` and `j` respectively. Thus, the generic name for any entry in the array might be suggested by `data[i,j]`.

Here is how we could print the very first, middle, and very last entries of our array:

```
>>> print ( 'First entry is', data[0,0] )
>>> print ( 'Middle entry is', data[30,20] )
>>> print ( 'Last entry is', data[59,39] )
```

The expressions for the first and last entries may surprise you. Python uses 0-based indexing. This means that if `x` is a list of 10 numbers, then the "names" of the individual entries are `x[0], x[1], ..., x[9]`. And if `data` is an array of 60 rows and 40 columns, then the rows run from 0 to 59, and the columns from 0 to 39. If you have been used to a different convention, as in MATLAB, this difference can take some practice to get used to.

Once we know how to name an entry of our array, we have seen how to print it, but we can also ask for its type, or evaluate formulas involving it, or even reset its value:

```
>>> type ( data[0,0] )
>>> x = data[0,0] - 2.0 * data[0,1] + data[0,2]
>>> data[0,0] = 1.0
```

6 Accessing an array row, column or slice

Our medical study involves 60 patients. We can think of them as indexed by a patient index of `i`, where $0 \leq i < 60$. The measurements for patient i are stored in row i of the array, and consist of 40 values, one for each day. Suppose we want to see all this data for the patient with $i = 17$, and nothing else? We could issue 40 separate print statements.

```
>>> print ( data[17,0] )  # Trying to print row 17
>>> print ( data[17,1] )
...
>>> print ( data[17,39] )
```

Notice that the text following the # sign is a comment. We can freely add such comments to remind us of what we are doing. Python ignores the # sign and everything that follows it.

We could issue one print statement listing each of the 40 values.

```
>>> print ( data[17,0], data[17,1], ..., data[17,39] )
```

But the efficient way is to indicate in a single expression `0:40` the entire range of indices we want:

```
>>> print ( data[17,0:40] )
```

Again, something looks strange here. If we want indices 0 through 39, why do we seem to say we want 0 through 40? Actually, this is another special Python convention. When given a range of the form `a:b`, we start at `a`, but we stop before we get to `b`, that is, at `b-1`.

There are three more useful facts about this index range.

- If our lower bound is the first entry, we can write `:b`;
- If our upper bound is the last entry, we can write `a:`;
- If we want the entire row (or column), we can simply write `:`.

This means that to print row $i = 17$, any of these commands would work:

```
>>> print ( data[17,0:40] )
>>> print ( data[17,:40] )
>>> print ( data[17,0:] )
>>> print ( data[17,:] )
```

If, on the other hand, we think that the measurements made on the day with index $j = 25$ were made incorrectly, we can ask to see them by the command:

```
>>> print ( data[:,25] )
```

While requests for a particular row and column are common, we might also be interested in some small subtable of our data, for instance, the values with $0 \leq i < 3$ and $36 \leq j < 40$. In that case, we could print these values by

```
>>> print ( data[:3,36:] )
```

7 Statistical measurements of data

When facing a set of data, there are some common measurements that can capture some of its properties in a single value. In particular, the maximum, minimum average or mean value, and the standard deviation give a rough idea of the range and typical behavior. Functions for computing these quantities are available in `numpy()`.

If we want to find these characteristics for the entire set of data, we simply compute

```
>>> # Get statistics for our data array
>>> maxval = numpy.max ( data )
>>> minval = numpy.min ( data )
>>> average = numpy.mean ( data )
>>> stdval = numpy.std ( data )
```

Note that we have begun this set of statements with a comment, to remind us of what we are doing here.

Suppose, however, that we again want to focus on patient $i = 17$. We could compute the statistical quantities for row 17 as follows:

```
>>> maxi17 = numpy.max ( data[17,:] )
>>> mini17 = numpy.min ( data[17,:] )
>>> avei17 = numpy.mean ( data[17,:] )
>>> stdi17 = numpy.std ( data[17,:] )
```

and if we wanted to focus on the data collected on day $j = 25$, we write

```
>>> maxj25 = numpy.max ( data[:,25] )
>>> minj25 = numpy.min ( data[:,25] )
>>> avej25 = numpy.mean ( data[:,25] )
>>> stdj25 = numpy.std ( data[:,25] )
```

8 Row and column measurements

We have already computed the mean value of the data, and the mean for patient $i = 17$. Suppose we want to compute, separately, the mean for every patient, that is, a list of 60 values, with the i-th value recording the mean data for patient i. This means that, for each row index i, we want to compute:

```
  meanrows[i] = np.mean ( data[i,:] )
```

But we don't want to do this one row at a time! Since, for each fixed row, we are taking the mean over all the columns, Python describes this as working along axis 1 (the columns). And so the corresponding command to compute a mean value for every row is:

```
  meanrows = np.mean ( data, axis = 1 )
```

giving us, in particular, a 1-dimensional array of length 60. Yes, `axis = 1` means "columns", but what we are doing is, essentially, "eliminating" the columns from the data.
Similarly, if we want to compute the maximum inflammation value, day by day, then for day j, we start with one data item in each row, and we want to "eliminate" the row data, replacing it with as single number, the daily maximum. Thus, for each j, we are computing

```
maxcols[j] = np.max ( data[:,j] )
```

but we do this over the entire set of days by the command

```
maxcols = np.max ( data, axis = 0 )
```

giving us, in particular, a 1-dimensional array of length 40.
Let's verify this:

```
>>> meanrows = numpy.mean ( data, axis = 1 )  #
>>> print ( meanrows.shape )     # The result is not a number, but a list
>>> print ( meanrows[17] )       # We already computed this value before!
```

And of course, instead of measuring the maximum value on day $j = 25$, we could ask for a list of the maximum recorded each day, by a similar command, carrying out the operation along axis = 0:

```
>>> maxcols = numpy.max ( data, axis = 0 )
>>> print ( maxcols.shape )     # The result is not a number, but a list
>>> print ( maxcols[25] )       # We already computed this value before!
```

9 String indexing

We understand that a string is made up of a sequence of characters. If we have a string of n characters, we can think of it as a list, whose entries are indexed from 0 to $n-1$. Just as we have seen for numeric arrays, we can use indexing to extract from a string any single character, or a substring of consecutive characters:

```
>>> s1 = 'breakfast'
>>> print ( s1 )
>>> print ( s1[0] )
>>> print ( s1[0:0] )  # Surprise!
>>> print ( s1[0:2] )
```

As we already saw with our numerical table, if we can omit the lower index if we want to start from the beginning, and we can omit the upper index if we want to go all the way to the end. Omitting both lower and upper indices is the same as using the whole string:

```
>>> s1 = 'breakfast'
>>> s2 = 'lunch'
>>> print ( s2[1:] )
>>> s3 = s1[:2] + s2[1:]
print ( s3[:] )
```

We know how to reference the last element of a string s: count n, the number of characters, and write $s[n-1]$. Sometimes, counting the length of the string can be inconvenient. It turns out that, to index the last element, you can write -1 instead of $n-1$. Although it's of less use, you can write -2 for the next to the last entry and so on. Given this information, what is s1[5:-1]?

```
>>> s1 = 'breakfast'
>>> print ( s1[5:-1] )
```

Suppose we want to write a single expression that gets the last three characters of any string. Choose values for the indices a and b so that this happens:

```
>>> a = ?
>>> b = ?
>>> s = 'buffalo'
>>> print ( s[a:b] )
>>> s = 'bugblatter beast of traal'
>>> print ( s[a:b] )
>>> s = 'me'
>>> print ( s[a:b] )
```

If s is the string 'buffalo', then `s[3:3]` is a legal expression; it's a string of length zero. You could make such a string yourself directly with the command

```
>>> s2 = ''
```

Sometimes, an empty string can be useful.

It is also possible to index numeric arrays and tables in this way. Usually, we do this because the resulting object will be the starting point for constructing some new item. For the data array that we set up earlier, consider the following expressions and state what they produce:

```
>>> data[3:3,4:4]
>>> data[3:3,:]
```

10 Creating an array

Our only array example so far, called `data`, was created by using the `numpy.loadtxt()` function, so the values were simply transferred from a file into an array. But often, we want to set up an array ourselves, with values that we simply type in. It's time to discuss how these arrays are represented.

The simplest array is a list stored as a single row, sometimes called a *row vector*. Mathematically, this might be represented by $[0, 1, 2]$. To create such an array in Python, we call the function `numpy.array()` as follows:

```
>>> rowvec = numpy.array ( [ 0, 1, 2 ] )
```

The `print()`, `type()`, `.dtype`, `.shape`, and indexing functions all work for this object, just as we have seen for other cases.

A table, matrix, or two-dimensional array can be thought of as a list of row vectors. Since this will actually be a list of lists, we are going to see two levels of square brackets. You can think of such an array as looking like this:

```
table = [ [  0,  1,  2 ]
          [ 10, 11, 12 ]
          [ 20, 21, 22 ] ]
```

If the array is small, you can create it by typing all the entries on one line:

```
>>> table = numpy.array ( [ [  0,  1,  2 ], [ 10, 11, 12 ], [ 20, 21, 22 ] ] )
```

But, especially if the array is large, you might prefer putting each row on a separate line:

```
>>> table = numpy.array ( [
...   [  0,  1,  2 ],
...   [ 10, 11, 12 ],
...   [ 20, 21, 22 ] ] )
```

The ... is Python's way of saying "Keep typing, I know there's more!".

A *column vector* is a little more complicated than a row vector. We have to think of it as a table in which each row has a single entry in it. To create such an array in Python, we call the function `numpy.array()` as follows:

```
>>> colvec = numpy.array ( [ [ 0 ], [ 10 ], [ 20 ] ] )
```

You can see that entering a column vector requires many more square brackets than when we are entering a row vector!

11 Stacking arrays

Sometimes we want to create a big array out of smaller pieces. This is called *stacking*. If `A`, `B`, and `C` are arrays, we can try horizontal stacking, making a new array `[ A, B, C ]`, or vertical stacking, in which case we expect to get something like

```
[ A ]
[ B ]
[ C ]
```

Arrays can be stacked if their shapes match. Horizontal stacking, for instance, requires that every component has the same number of rows. If the arrays do not have the appropriate shapes, the command will fail.

Look at what happens in the following examples.

```
>>> A = numpy.array ( [ [ 1, 2, 3 ], [ 4, 5, 6 ], [ 7, 8, 9 ] ] )
>>> B = numpy.hstack ( [ A, A ] )
>>> C = numpy.vstack ( [ A, A ] )
```

Exercise: Use indexing to "slice" the first and last columns of the array A, and them horizontall stack them into a new array of 3 rows and 2 columns.

12 Data differences

Row i of our `data` array contains a daily record of the inflammation level measured for patient i. An interesting secondary measurement is the change between one day's measurement and the next. The `numpy.diff()` function can compute this quantity. Let's start by looking at a short stretch of this data, and see what we get:

```
>>> sample = data[17,10:15]
>>> change = numpy.diff ( sample )
```

Could you predict the number of values in `sample`. Why does `change` have fewer values?

What would be the command to compute the changes for patient 17 over the entire medical study?

Of course, we probably want to compute the corresponding record of changes for every patient. We can do this by applying `diff()` to the entire `data` array, as long as we specify that we want to compute the changes by moving along rows, which Python thinks of as `axis = 0`. The following command will compute an array of length 60×39, containing in entry $[i, j]$ the change, for patient i, in inflammation levels between days j and $j + 1$.

```
>>> changes = numpy.diff ( data )
```

Suppose you wanted to know, for each patient, the value of the greatest change in inflammation from one day to the next. What command could you use?

Changes can be positive or negative. Your answer to the previous question probably used the `max)` function, so it only looked for the greatest increase in inflammation. The function `numpy.abs()` computes the absolute value of a number. So now suppose you wanted to know, for each patient, the value of the greatest change (positive or negative) in inflammation from one day to the next. What command could you use?