Python #6
Processing data from multiple files

Location: https://people.sc.fsu.edu/~jburkardt/classes/python_2022/python06/python06.pdf

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

Multiple File Processing

- *Our medical study data is stored in several separate files;*
- *We want to compute statistics over all this data;*
- *We need to initialize some variables and then update them one file at a time;*
- *We will need to use a* `for()` *loop to get all the files;*
- *The* `for()` *loop will need a list of the file names;*

1 How the medical data is stored

In a previous Python exercise, we wrote code to plot values of interest from a file containing medical data, `inflammation-01.csv`. In fact, the researcher has supplied us with 12 such sets of data, and there may be more on the way. To save work, we'd like to be able to process all the data with a single statement. In order to do that, we need to learn about Python loops, which allow us to repeat a given action on multiple sets of data.

2 Practice on one small file

Before we tackle the medical study, let's consider a smaller problem. Although we want to work with multiple files, we will start with a single data file involves `m` = 18 rows and `n` = 10 columns, stored in `maxi.csv`.

If you do not have a copy of this file, it can be downloaded from the class website. Another way to get it 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.

Let's recall, from our earlier work, how to read this file into a variable, and compute its statistics.

```
data = np.loadtxt ( fname = 'maxi.csv', delimiter = ',' )
maxval = np.max ( data )
minval = np.min ( data )
average = np.mean ( data )
stdev = np.std ( data )
```

Be sure to make a record of these statistical values for later comparison.

3 Merge little data arrays, compute statistics

Now in our medical study, we will soon be dealing with data spread across 12 files. To practice for this case, let's suppose that our data from the previous problem has been split among the 3 files `mini1.csv, mini2.csv, mini3.csv`, each have `m` = 6 rows and `n` = 10 columns.

If you do not have a copy of these files, they can be downloaded from the class website. Another way to get it 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.

Now we want to read the data from all three files and compute a set of statistics, as though the numbers had been in a single file.

One fairly crude way is to create three arrays, `data1, data2, data3` by reading each file, and then use the appropriate `numpy` stack function to recreate the `data` array, after which we can easily compute our statistics.

```
data1 = np.loadtxt ( fname = 'mini1.csv', delimiter = ',' )
data2 = np.loadtxt ( fname = 'mini2.csv', delimiter = ',' )
data3 = np.loadtxt ( fname = 'mini3.csv', delimiter = ',' )
data = np.vstack ( [ data1, data2, data3 ] )
```

Compute the statistics you get from this approach, and make sure they match the orginal results.

What's wrong with this approach? It's not very flexible, you have to have access to all the data files at the same time. And, if the data arrays are large, then we are using up a lot of memory storing the data chunks and the merged data as well. This is not an issue for our small examples, but in a real-life situation, it may be unwise or even impossible to load all the data at once. We will try to work towards an approach that avoids this issue.

4 Compute statistics on little arrays, then merge

So suppose that we are only allowed to have one mini-dataset at a time in the program, and we still want to compute the maximum, mean, and minimum values of the full set of data. How would this be possible? Let's start with the maximum. We know how to compute the maximum of each dataset, and the maximum of the whole dataset is simply the maximum of these three numbers. So isn't that easy? Well yes, but there are a couple of things we have to think about first:

- We are **not** allowed to create three variables, `max1, max2, max3`;
- We have to have a single variable, called `maxval`, which we update as we process the data.
- We have to initialize `maxval` before the loop begins, with a value that is so small that it is immediately replaced by `np.max(data1)`
- While `np.max()` computes the maximum of a numpy array, we need to use plain old `max()` to compute the maximum of two numbers.

Similar remarks apply for computing `minval`.

It might seem that `average` will be simpler to handle. However if `average` is the average of `n` things, and `mini_average` is the average of `mini_n` things, then we need a way to update to the average of `n + mini_n` things.

If you think about the definition of the average of n things, you can see that it is the sum of n values, divided by their number. If we multiply that average by n, we get back the original sum. Thus, if $average_1$ is the average of n_1 things and $average_2$ is the average of n_2 things, then the average of $n_1 + n_2$ things is found by recovering the two sums, adding them, and then dividing by the total number of things in the combination:

$$\text{average}_{12} = \frac{n_1 * \text{average}_1 + n_2 * \text{average}_2}{n_1 + n_2}$$

So look closely at how this computation works. First we initialize the statistical quantities. Then we read each mini dataset, compute the mini statistics, and update the full set of statistics. We never have the full array `data` in the computer at one time. If we had to process a thousand, or a million files containing partial data, we could do it this way too:

```
  n = 0                 # Start out with no data
  maxval = - np.inf     # Initial value is smallest possible
  minval =   np.inf     # Initial value is largest possible
  average = 0.0

  files = [ 'mini1.csv', 'mini2.csv', 'mini3.csv' ]

  for filename in files:
    mini_data = np.loadtxt ( fname = filename, delimiter = ',' )
    mini_n = mini_data.shape[0]
    mini_maxval = np.max ( mini_data )
    mini_minval = np.min ( mini_data )
    mini_average = np.mean ( mini_data )
#
#  Have to use "max", not "np.max", when computing maximum of two numbers...
#
    maxval = max ( maxval, mini_maxval )
    minval = min ( minval, mini_minval )
#
#  Here is how we merge two average values:
#
    average = ( n * average + mini_n * mini_average ) / ( n + mini_n )
    n = n + mini_n
```

Verify that the statistical quantities come out to be the same as when we worked directly on the full dataset.

Chicken statement: For this example, we have stopped working with the standard deviation. That's because, while there is a formula to compute the standard deviation in pieces, it is much more complicated than we would like to see. If you know the definition of the standard deviation, it is a worthwhile exercise to figure out the standard deviation of a vector $v = v_1 + v_2$, given the size, average, and standard deviations of the separate vectors v_1 and v_2. As yet another complication to be aware of, in the usual definition of the standard deviation of n quantities, we divide by $n - 1$, not n, for technical reasons. This makes the update formula ever so slightly more complicated! Rather than spend too much time on this side issue, I will chicken out!

5 Create filenames one at a time

Although we say that our previous code could handle data that is stored in any number of files, it does require a way to access the name of each one of those files. When there are only three files, this is a simple

matter of typing in the names. But who wants to type in hundreds of filenames? We need to understand that our solution really can extend to bigger problems, by figuring out a way past this bottleneck.

The key is that in a typical case like this, the many filenames are likely to have a simple pattern, in which the name involves an index that increases by 1 each time. That means that we should think of our list of three files in the previous example in the symbolic form `filename = 'mini' + index number + '.csv'`. In fact, if we could just figure out how to fill in the index number in a way that lets us create the filename, we will have a solution.

There is a handy Python function `str()` that turns a number into a string. If the number is an integer, then this is almost equivalent to wrapping the value with quote marks, that is, for instance, $123 \rightarrow' 123'$. So if the variable `i` is set to 1, 2 or 3, then the corresponding formula `filename = 'mini' + str(i) + '.csv'` will exactly create the desired file name, starting with `'mini1.csv'`.

To make this happen, we need to modify our `for` statement so that it involves an index `i` which runs from 1 to 3 (careful! How exactly do we go from 1 to 3 with a `for` statement?), and then inside the loop immediately uses `i` to create the filename.

Here's what that part of the revised code would look like:

```
...
average = 0.0

for i in range ( 1, 4 ):
  filename = 'mini' + str ( i ) + '.csv'
  mini_data = np.loadtxt ( fname = filename, delimiter = ',' )
  ...
```

Can you revise the previous code to use this approach? Can you see why this new version could easily handle hundreds of files if necessary?

6 Create filenames using `glob`

Assuming our set of files has a simple pattern, there is another way to get that list. Python has a library called `glob` which can return all the filenames that match a given pattern. The pattern allows a description in which special characters **?** and ***** can be inserted to represent the pattern.

The question mark symbol must be replaced by exactly one (arbitrary) character in order for a match to be declared. Thus *'file?.txt'* will match *'file0.txt'*, *'filer.txt'*, *'filem.txt'*, but NOT *'file.txt'*, *'file12.txt'*.

The star symbol must be replaced by any number, including zero, of characters in order for a match to be declared. Thus*file*.txt* will match all the patterns above, but NOT *'tile0.txt'*, *'file1.jpg'* or *'fille2.txt'*.

In our example, the filenames follow either pattern *'mini?.csv* or *'mini*.csv*. The second pattern might be safer, since it would allow matching names like *'mini12.csv'* if our set of data was later to be increased with more files.

So, as an alternative to constructing each filename ourselves, we can generate the necessary list as follows:

```
import glob    # need to import the library

filenames = glob.glob ( 'mini*.csv' )
filenames = sorted ( filenames )    # Optional. Sorts the names in the list.
print ( '  glob found the following files:' )
print ( filenames )

...  # Usual initialization of the statistical variables.
```

```
for filename in filenames:
  ...    # The processing is the same as before.
```

7 Compute statistics of the full medical study data

Now we have all the tools we need in order to compute statistics for the medical study, which was split into 12 data files named *'inflammation-01.csv'* through *'inflammation-12.csv'*.

If you do not have a copy of these files, they can be downloaded from the class website. Another way to get it 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.

A new problem arises because the files 1 though 9 use indices 01 through 09. The way we wrote the code to convert the loop index into a character string was

```
filename = 'inflammation-' + str ( i ) + '.csv'
```

but this will incorrectly generate the file name *'inflammation-1.csv'* when we need to generate *'inflammation-01.csv'*. As long as we are aware that all the numeric indices use two digits, so that small indices must be padded with zero, we can easily fix our problem by

```
filename = 'inflammation-' + str ( i ).zfill(2) + '.csv'
```

If we had hundreds of filenames to generate, starting with *'inflammation-001.csv'*, what would we do?

Instead of generating the filenames ourselves, we could ask `glob` to generate them from the pattern of either `'inflammation-??.csv'` or *'inflammation-*.csv'*.

```
filenames = glob.glob ( 'inflammation-*.csv' )
```

Once we have managed to generate the list of filenames, our program should be easy to construct. The actual reading of each file, computation of statistics for each partial dataset, and combination of the statistics for the full data, will be the same as for our small example.

8 Plot max, mean, min for each medical dataset

Now, instead of computing statistics for the data, we'd like to generate a plot, for each dataset, of the maximum, mean, and minimum values over all the patients in that dataset. After we plot the data for file *'inflammation-01.csv'* we want to save the plot in the file *'inflammation-01.png'*, and so on for each of our 12 files. Here is an outline of what you must do:

```
For each of the 12 values of i
  Generate the data filenames using str(i).zfill(2)
  read the data
  compute the max, mean, and min vectors over all rows (axis = 0)
  plot the max, mean, and min vectors, combined in a single plot
  generate the plot filenames using str(i).zfill(2)
  save the plotfile using plt.savefig(filename)
```

If you compare the plots, you may see some data is very different from others, and some data is ... extremely the same. This suggests that the data for this experiment is fake.