This lesson is being piloted (Beta version)

‹ Python Scripting for Computational Molecular Science (../) (../03-multiple_files/index.html) › (../05-plottin

Working with Tabular Data

Overview

Teaching: 45 min
Exercises: 60 min
Questions

- How do I work with numerical data presented in tables?

Objectives

- Use functions in `numpy` to read in tabular data.
- Take 2D slices of data in numpy arrays.
- Use 2D slices to work with particular rows or columns of data.
- Use the `range()` function in `for` loops.
- Use `numpy` functions to analyze data.

Most scientists work with a lot of numerical data. In this module we will focus on reading in and analyzing numerical data, visualizing the data, and working with arrays.

Reading in Tabular Data

As we already discussed, there are many ways to read in data from files in python. In our last module, we used the `readlines()` function to read in a complex output file. In theory, you could always use the `readlines()` function, and then use the data parsing tools we learned in the previous module to format the data as you needed. But sometimes there are other ways that make more sense, particularly if the data is (1) all or mostly one type of data (for example, all numbers) and/or (2) formatted in a table. Frequently, a table will be mostly numbers, but have column or row labels.

A common table format is the CSV file or comma separated values. This is exactly what it sounds like. Data is presented in rows, with each value separated by a comma. If you have data in a spreadsheet program that you need to import into a python code, you can save the data as a csvfile to read it in.

In this example, we have a CSV file that contains data from a molecular dynamics trajectory. We have a 20 ns simulation that used a 2 fs timestep. The data was saved to the trajectory file every 1000 steps, so our file has 10,000 timesteps. At each timestep, we are interested in the distance between particular atoms. These trajectories were generated with the AMBER molecular dynamics program and the distances were measured with the python program MDAnalysis. The table of atomic distances was saved as a CVS file called “distance_data_headers.csv”. This file was downloaded as part of your lesson materials. Open the file in a text editor and study it to determine its structure.

In analyzing tabular data, we often need to perform the same types of calculations (averaging, calculating the minimum or maximum of the data set), so we are once again going to use a python **library**, this time a library that contains lots of functions to perform math operations. This library is called `numpy` . The `numpy` library has several functions available to read in tabular data. One of these functions is the `genfromtxt()` function. We will use the `help()` function to learn more about `genfromtxt()` and how it works.

Python

```
import numpy
help(numpy.genfromtxt)
```

Output

```
Help on function genfromtxt in module numpy.lib.npyio:

genfromtxt(fname, dtype=<class 'float'>, comments='#', delimiter=None, skip_header=0, skip_footer=0, convert
ers=None, missing_values=None, filling_values=None, usecols=None, names=None, excludelist=None, deletechars=
None, replace_space='_', autostrip=False, case_sensitive=True, defaultfmt='f%i', unpack=None, usemask=False,
loose=True, invalid_raise=True, max_rows=None, encoding='bytes')
    Load data from a text file, with missing values handled as specified.

    Each line past the first `skip_header` lines is split at the `delimiter`
    character, and characters following the `comments` character are discarded.
```

The help menu shows us all the options we can use with this function. The first input `fname` is the filename we are reading in. We must put a values for this option because it does not have a default value. All the other options have a default value that is shown after the = sign. We only need to specify these options if we don't want to use the default value. For example, in our file, all the values were not numbers so we don't want to use the datatype `float`, we want to use something else. If you have mixed datatypes, like we do here, we want to use `'unicode'`. In our file, our values are separated by commas; we indicate that with `delimiter=','`.

📌 Should you skip the headers?

The clever student may notice the `skip_header` option, where you can specify a number of lines to skip at the beginning of the file. If we did this, then our values would all be numbers and we could use dtype='float', which is the default. In this example, we are not going to do that because we might want to use the headers later to label things, but keep this option in mind because you might want to use it in a later project.

Now we have have our plan, we are ready to import our data with `genfromtxt()`.

First, we have to get the path to our file. Remember from previous lessons that we use the `os.path` module to do this.

Python

```
import os

distance_file = os.path.join('data', 'distance_data_headers.csv')

distances = numpy.genfromtxt(fname=distance_file, delimiter=',', dtype='unicode')
print(distances)
```

Output

```
[['Frame' 'THR4_ATP' 'THR4_ASP' 'TYR6_ATP' 'TYR6_ASP']
 ['1' '8.9542' '5.8024' '11.5478' '9.9557']
 ['2' '8.6181' '6.0942' '13.9594' '11.6945']
 ...
 ['9998' '8.6625' '7.7306' '9.5469' '10.3063']
 ['9999' '9.2456' '7.8886' '9.8151' '10.7564']
 ['10000' '8.8135' '7.917' '9.9517' '10.7848']]
```

The output of this function is a list of lists; that is, each row is a entry in our list, but each row is itself a list of values. We can see that the first row is our column headings and all the other rows contain numerical data.

If we were to read this in with the readlines() function, we would have to split each line of the file, use the `append` function to make a new list for each row, and THEN put all those lists together into a list of lists. Using the appropriate `numpy` function makes our life much easier.

Manipulating Tabular Data

Even now, we can see that our first line of data is headings for our columns, and will need to be stored as strings, whereas all the rest of the data is numerical and will need to be stored as floats. Let's take a slice of the data that is just the headers.

Python

```
headers = distances[0]
print(headers)
```

Output

```
['Frame' 'THR4_ATP' 'THR4_ASP' 'TYR6_ATP' 'TYR6_ASP']
```

Check Your Understanding

Take a slice of the data that is just the numerical values. To be uniform for later activities, call this slice data.

Answer

Even though we now have a list of lists that is just the numbers, the numbers are all still strings. We know this because (1) we read them all in as unicode and (2) if we look at the output of the print statement, we can see that each number is enclosed in single quotes, indicating that it is a string. We need to recast these values as floats. The `numpy` library has a built-in function to accomplish this. *In this case, keeping a variable with all the same information as strings is not useful to us, so this is a case where we are going to overwrite our variable data.*

Python

```
data = data.astype(numpy.float)
print(data)
```

Output

```
[[1.00000e+00 8.95420e+00 5.80240e+00 1.15478e+01 9.95570e+00]
 [2.00000e+00 8.61810e+00 6.09420e+00 1.39594e+01 1.16945e+01]
 [3.00000e+00 9.00660e+00 6.06370e+00 1.30924e+01 1.13043e+01]
 ...
 [9.99800e+03 8.66250e+00 7.73060e+00 9.54690e+00 1.03063e+01]
 [9.99900e+03 9.24560e+00 7.88860e+00 9.81510e+00 1.07564e+01]
 [1.00000e+04 8.81350e+00 7.91700e+00 9.95170e+00 1.07848e+01]]
```

We already learned how to address a particular element of a list and how to take a slice of a list to create a new list. Now that we have an array, we now need two indices to address a particular element of the array. The notation to address an element of the array is always

Python

```
array_name[row,column]
```

Check Your Understanding

What will be the output of these lines of code?

Python

```
print(data[0,1])
print(data[1,0])
```

Answer

There you can also take two-dimensional slices of an array where you specify a range of rows and a range of columns for the slice. For example, sometimes it is easier to work with a small subset of our data for testing rather than the full data set. This command takes a slice that includes only the first ten rows and the first three columns of our data.

Python

```
small_data = data[0:10,0:3]
print(small_data)
```

Output

```
[[ 1.      8.9542  5.8024]
 [ 2.      8.6181  6.0942]
 [ 3.      9.0066  6.0637]
 [ 4.      9.2002  6.0227]
 [ 5.      9.1294  5.9365]
 [ 6.      9.0462  6.2553]
 [ 7.      8.8657  5.9186]
 [ 8.      9.3256  6.2351]
 [ 9.      9.4184  6.1993]
 [10.      9.06    6.0478]]
```

Remember that counting starts at zero, so 0:10 means start at row zero and include all rows, up to but *not* including 10. Just as with the one-dimensional list slices, if you don't include a number before the `:` the slice automatically starts with `list_name[0]`. If you don't include a number after the `:` the slice goes to the end of the list. Therefore, if you don't include either, a `:` means *every row* or *every column*.

Check Your Understanding

What will be the output of these lines of code?

Python

```
print(small_data[5,:])
print(small_data[:,1:])
```

Answer

Analyzing Tabular Data

The `numpy` library has numerous built-in functions. For example, to calculate the average (mean) of a data set, the syntax is

```
data_average = numpy.mean(data_set)
```

Let's say we want to calculate the average distance for a particular measurement over the whole simulation. We want to calculate the average of one of the columns. We can take a slice of our data array that is just one column. Then we can find the average of that column. It doesn't make sense to average the frame numbers, so let's do the THR4_ATP column first.

Python

```
thr4_atp = data[:,1]  # Every row, just the THR4_ATP column
avg_thr4_atp = numpy.mean(thr4_atp)
print(avg_thr4_atp)
```

Output

```
10.876950930000001
```

This is correct, but now we would like to calculate the average of *every* column. This seems like a job for a `for` loop, but unlike last time, we don't want to count over a particular list and do something for every item, we want to do something a particular number of times. Basically, we want to take that `1` and let it be every number, up to the number of columns. This is a task for the `range()` function. The general syntax is

```
range(start,end)
```

and we can use `range()` in a `for` loop.

In our example, the "end" value needs to be the number of columns of data.

Check Your Understanding

Determine the number of columns in our data set. Save this value as a variable called num_columns.

Answer

Now that we know the number of columns, we can use the `range()` function to set up our `for` loop.

Python

```
for i in range(1,num_columns):
    column = data[:,i]
    avg_col = numpy.mean(column)
    print(F'{headers[i]} : {avg_col}')
```

Output

```
THR4_ATP : 10.876950930000001
THR4_ASP : 7.342344959999999
TYR6_ATP : 11.209791329999998
TYR6_ASP : 10.9934435
```

Geometry Analysis Project

In the lesson materials, there is a file in the `data` folder called "water.xyz". This is a very simple, standard file format that is often used to distribute molecular coordinates. The first line of the file is the number of atoms in the molecule, the second line is a title line (or may be blank), and the coordinates begin on the third line. The format of the coordinates is

```
Atom_Label  XCoor   YCoor   ZCoor
```

and the default units (which are used in this example) are angstroms.

Write a code to read in the information from the xyz file and determine the bond lengths between all the atoms. There is a numpy function to take the square root, `numpy.sqrt()`. To raise a number to a power, use `**`, as in `3**2 = 9`. Your code output should look something like this.

Output

```
O to O : 0.0
O to H1 : 0.969
O to H2 : 0.969
H1 to O : 0.969
H1 to H1 : 0.0
H1 to H2 : 1.527
H2 to O : 0.969
H2 to H1 : 1.527
H2 to H2 : 0.0
```

Hint: You will need a double `for` loop to measure the distance between all the atoms. If you aren't sure how to get started, print the variables inside your for loop.

Solution

Variable Names

In our solution above, we called our bond length variable `bond_length_AB`. We could have called this variable anything we wanted. Consider the following two potential variable names for bond length - `BL_AB` and `bond_length_AB`. Which is more clear to you? While you might know what `BL` means, and it is possible for others to figure it out through context, it's easier on others if you give your variables clear names.

Project Extension 1

Your initial project calculated the distance between every set of atoms. However, some of these atoms aren't really bonded to each other. H1 and H2 are not bonded for example, and all of the distances between an atom and itself are zero. Use a distance cutoff of 1.5 angstroms to define a bond (that is, if the bond length is greater than 1.5 angstroms, consider the atoms not bonded). Modify your code to only print the atoms that are actually bonded to each other.

Solution

Project Extension 2

Some of these are actually the same bond length; for example, O to H1 and H1 to O refer to the same bond length. Remove the duplicates from your list.

Solution

Project Extension 3

Write a your output to a text file called bond_lengths.txt instead of just printing it to the screen.

Solution

Key Points

- If you are reading in a file that is mostly numerical data, there are better ways to read in the data than using `readlines()`.
- The notation to refer to a particular element of an array is `array_name[row,column]`.
- Typically, you should not write a function to perform a standard math operation. The function probably already exists in `numpy`.

‹ (../03-multiple_files/index.html)

› (../05-plottin