

< Python Scripting for Computational Molecular Science (../06-functions/index.html)

> (../08-testing/i

Running code from the Linux Command Line

Overview

Teaching: 15 min

Exercises: 10 min

Questions

- How do I move my code from the interactive jupyter notebook to run from the Linux command line?

Objectives

- Make code executable from the Linux command line.
- Use argparse to accept user inputs.

Creating and running a python input file

We are now going to move our geometry analysis code out of the Jupyter notebook and into a format that can be run from the Linux command line. Open your favorite text editor and create a new file called "geom_analysis.py" (or choose another filename, just make sure the extension is .py). Paste in your geometry analysis code (the version with your functions) from your jupyter notebook and save your file.

The best practice is to put all your functions at the top of the file, right after your import statements. Your file will look something like this.

Python

```
import numpy
import os

def calculate_distance(atom1_coord, atom2_coord):
    x_distance = atom1_coord[0] - atom2_coord[0]
    y_distance = atom1_coord[1] - atom2_coord[1]
    z_distance = atom1_coord[2] - atom2_coord[2]
    bond_length_12 = numpy.sqrt(x_distance**2+y_distance**2+z_distance**2)
    return bond_length_12

def bond_check(atom_distance, minimum_length=0, maximum_length=1.5):
    if atom_distance > minimum_length and atom_distance <= maximum_length:
        return True
    else:
        return False

def open_xyz(filename):
    xyz_file = numpy.genfromtxt(fname=filename, skip_header=2, dtype='unicode')
    symbols = xyz_file[:,0]
    coord = (xyz_file[:,1:])
    coord = coord.astype(numpy.float)
    return symbols, coord

file_location = os.path.join('data', 'water.xyz')
symbols, coord = open_xyz(file_location)
num_atoms = len(symbols)
for num1 in range(0,num_atoms):
    for num2 in range(0,num_atoms):
        if num1<num2:
            bond_length_12 = calculate_distance(coord[num1], coord[num2])
            if bond_check(bond_length_12) is True:
                print(F'{symbols[num1]} to {symbols[num2]} : {bond_length_12:.3f}')
```

This lesson is being piloted (Beta version)

Exit your text editor and go back to the command line. Now all you have to do to run your code is type

Bash

```
$ python geom_analysis.py
```

in your Terminal window. Your code should either print the output to the screen or write it to a file, depending on what you have it set up to do. (The code example given prints to the screen.)

Changing your code to accept user inputs

In your current code, the name of the xyzfile to analyze, "water.xyz", is hardcoded; in order to change it, you have to open your code and change the name of the file that is read in. If you were going to use this code to analyze geometries in your research, you would probably want to be able to specify the name of the input file when you run the code, so that you don't have to change it every single time. You might want to use the script like this:

Bash

```
$ python geom_analysis.py water.xyz
```

These types of user inputs are called *arguments* and to make our code accept arguments, we have to import a new python library in our code.

Open your geometry analysis code in your text editor and add this line at the top.

Python

```
import argparse
```

We are importing a library called <https://docs.python.org/3/library/argparse.html> (argparse) which can be used to easily make scripts with command line arguments. Argparse has the ability to allow us to easily write documentation for our scripts as well.

We tell argparse that we want to add a command line interface. The syntax for this is

Python

```
parser = argparse.ArgumentParser(description="This script analyzes a user given xyz file and outputs the length of the bonds.")
```

We've included a description of the script for our users using `description=`. This description does not need to explain what the arguments are, that will be done automatically for us in the next steps.

Next, we have to tell `argparse` what arguments it should expect. In general, the syntax for this is

Python

```
parser.add_argument("argument_name", help="Your help message for this argument.")
```

Let's add one for the xyz file the user should specify.

```
parser.add_argument("xyz_file", help="The filepath for the xyz file to analyze.")
```

Next, we have to get the arguments.

Python

```
args = parser.parse_args()
```

Our arguments are in the `args` variable. We can get the value of an argument by using `args.argument_name`, so to get the xyz file the user puts in, we use `args.xyz_file`. Notice that what follows after the dot is the same thing we but in quotation marks when using `add_argument`.

```
xyzfilename = args.xyz_file  
symbols, coord = open_xyz(xyzfilename)
```

Save your code and go back to the Terminal window. Make sure you are in the directory where your code is saved and type

Bash

```
$ python geom_analysis.py --help
```

This will print a help message. The `argparse` library has written this help message for us based on the descriptions and arguments we added.

Output

```
usage: analyze.py [-h] xyz_file  
  
This script analyzes a user given xyz file and outputs the length of the  
bonds.  
  
positional arguments:  
  xyz_file    The filepath for the xyz file to analyze.  
  
optional arguments:  
  -h, --help  show this help message and exit
```

Now try running your script with an xyz file.

Bash

```
$ python analyze.py data/water.xyz
```

Check that the output of your code is what you expected.

What would happen if the user forgot to specify the name of the xyz file?

Error

```
usage: analyze.py [-h] xyz_file  
analyze.py: error: the following arguments are required: xyz_file
```

Argparse handles this for us and prints an error message. It tells us that we must specify an xyz file.

Try out your program with other XYZ files in your `data` folder.

The “main” part of our script

We need to add one more thing to our code. When you write a code that includes function definitions and a main script, you need to tell python which part is the main script. (This becomes very important later when we are talking about testing.) *After* your import statements and function definitions and *before* use `argparse`

Python

```
if __name__ == "__main__":
```

Since this is an `if` statement, you now need to indent each line of your main script below this `if` statement. Be very careful with your indentation! Don't use a mixture of tabs and spaces! A good way to indent multiple lines in many text editors is to highlight the lines you would like to indent, then press `tab`.

Save your code and run it again. It should work exactly as before. If you now get an error message, it is probably due to inconsistent indentation.

Python

```
import os
import numpy
import argparse

def calculate_distance(atom1_coord, atom2_coord):
    x_distance = atom1_coord[0] - atom2_coord[0]
    y_distance = atom1_coord[1] - atom2_coord[1]
    z_distance = atom1_coord[2] - atom2_coord[2]
    bond_length_12 = numpy.sqrt(x_distance**2+y_distance**2+z_distance**2)
    return bond_length_12

def bond_check(atom_distance, minimum_length=0, maximum_length=1.5):
    if atom_distance > minimum_length and atom_distance <= maximum_length:
        return True
    else:
        return False

def open_xyz(filename):
    xyz_file = numpy.genfromtxt(fname=filename, skip_header=2, dtype='unicode')
    symbols = xyz_file[:,0]
    coord = (xyz_file[:,1:])
    coord = coord.astype(numpy.float)
    return symbols, coord

if __name__ == "__main__":

    ## Get the arguments.
    parser = argparse.ArgumentParser(description="This script analyzes a user given xyz file and outputs the length of the bonds.")
    parser.add_argument("xyz_file", help="The filepath for the xyz file to analyze.")

    args = parser.parse_args()

    symbols, coord = open_xyz(args.xyz_file)
    num_atoms = len(symbols)

    for num1 in range(0,num_atoms):
        for num2 in range(0,num_atoms):
            if num1<num2:
                bond_length_12 = calculate_distance(coord[num1], coord[num2])
                if bond_check(bond_length_12, minimum_length=args.minimum_length, maximum_length=args.maximum_length) is True:
                    print(f'{symbols[num1]} to {symbols[num2]} : {bond_length_12:.3f}')
```

Extension - Optional Arguments

What's another argument we might want to include? We also might want to let the user specify a minimum and maximum bond length on the command line. We would want these to be optional, just like they are in our function.

We can add optional arguments by putting a dash (`-`) or two dashes (`--`) in front of the argument name when we add an argument. Add this line below where you added the first argument. Note that all `add_argument` lines should be above the line with `parse_args`.

Python

```
parser.add_argument('-minimum_length', help='The minimum distance to consider atoms bonded.', type=float, default=0)
```

We've added two new things to our argument as well. We have told `argparse` that the argument people will pass for this will be a decimal number (float) and that the default value will be zero. If a user does not use `-minimum_length`, the value will be zero.

Now we will change our code to use this value. Find the line where you call `bond_check` and change it to use the value from `argparse`.

Python

```
if bond_check(bond_length_12, minimum_length=args.minimum_length) is True:
```

Bash

```
$ python analyze.py data/water.xyz -minimum_length 1
```

Now we can override the minimum length when running our script. If we don't specify it, it will default to using zero.

We can do the same thing for our maximum bond length.

Python

```
parser.add_argument('-maximum_length', help='The maximum distance to consider atoms bonded.', type=float, default=1.5)
```

And, don't forget to update your `bond_check` function.

Python

```
if bond_check(bond_length_12, minimum_length=args.minimum_length, maximum_length=args.maximum_length) is True:
```

Our final program looks like this:

Python

```
import os
import numpy
import argparse

def calculate_distance(atom1_coord, atom2_coord):
    x_distance = atom1_coord[0] - atom2_coord[0]
    y_distance = atom1_coord[1] - atom2_coord[1]
    z_distance = atom1_coord[2] - atom2_coord[2]
    bond_length_12 = numpy.sqrt(x_distance**2+y_distance**2+z_distance**2)
    return bond_length_12

def bond_check(atom_distance, minimum_length=0, maximum_length=1.5):
    if atom_distance > minimum_length and atom_distance <= maximum_length:
        return True
    else:
        return False

def open_xyz(filename):
    xyz_file = numpy.genfromtxt(fname=filename, skip_header=2, dtype='unicode')
    symbols = xyz_file[:,0]
    coord = (xyz_file[:,1:])
    coord = coord.astype(numpy.float)
    return symbols, coord

if __name__ == "__main__":

    ## Get the arguments.
    parser = argparse.ArgumentParser(description="This script analyzes a user given xyz file and outputs the length of the bonds.")
    parser.add_argument("xyz_file", help="The filepath for the xyz file to analyze.")

    parser.add_argument('-minimum_length', help='The minimum distance to consider atoms bonded.', type=float, default=0)
    parser.add_argument('-maximum_length', help='The maximum distance to consider atoms bonded.', type=float, default=1.5)

    args = parser.parse_args()

    symbols, coord = open_xyz(args.xyz_file)
    num_atoms = len(symbols)

    for num1 in range(0,num_atoms):
        for num2 in range(0,num_atoms):
            if num1<num2:
                bond_length_12 = calculate_distance(coord[num1], coord[num2])
                if bond_check(bond_length_12, minimum_length=args.minimum_length, maximum_length=args.maximum_length) is True:
                    print(F'{symbols[num1]} to {symbols[num2]} : {bond_length_12:.3f}')
```

Project Assignment

For this homework assignment, you will return to your first project where you processed the file `03_Prod.mdout`.

Create a command line script using `argparse` which can take in an `mdout` file from Amber, pull out total energy for each time step, and write a new file containing these values. The script should take a file name (`03_Prod.mdout`) and output a file with the names `filename_Etot.txt`. Modify your week 1 homework to do this.

In the week 1 homework, the file we wrote had two values at the end which we did not want for the total energy. The last two values were some statistics associated with the md simulation and were not total energies. Excluded these two values from the file you write this time.

You can download a directory containing more mdout files here (`./data/mdout.zip`)

Call your script `analyze_mdout.py`. You should be able to call the script in the following way:

Bash

```
$ python analyze_mdout.py 03_Prod.mdout
```

When you call help, you should get the following output:

Bash

```
$ python analyze_mdout.py --help
```

Output

```
usage: This script parses amber mdout files to extract the total energy. You can also use it to create plots.
       [-h] [--make_plots] path
```

positional arguments:

path The filepath to the file to be analyzed.

optional arguments:

-h, --help show this help message and exit

Solution

Python

```
import os
import glob
import argparse

if __name__ == "__main__":

    # Create the argument parser
    parser = argparse.ArgumentParser("This script parses amber mdout files to extract the total energy.")
    parser.add_argument("path", help="The filepath to the file(s) to be analyzed.", type=str)

    args = parser.parse_args()
    filename = args.path

    # Read the data from the specified file.
    f = open(filename)
    data = f.readlines()
    f.close()

    # Figure out the file name for writing the output.
    fname = os.path.basename(args.path).split('.')[0]

    etot = []
    # Loop through the lines
    for line in data:
        split_line = line.split()
        if 'Etot' in line:
            etot.append(float(split_line[2]))

    # Get rid of values we don't need.
    values = etot[:-2]

    # Open a file for writing
    outfile_location = F'{fname}_Etot.txt'
    outfile = open(outfile_location, 'w+')

    for value in values:
        outfile.write(f'{value}\n')

    outfile.close()
```

Extension 1

Modify your script so that people can specify a wildcard character to read and write multiple files at once.

You should be able to call the script using:

Bash

```
$ python analyze_mdout.py data/*.mdout
```

You will need find a way for an argument in argparse to be a list for this. See the documentation (<https://docs.python.org/3/library/argparse.html#the-add-argument-method>) for the `add_argument` function - you will find the answer there!

Solution

Extension 2


Add an additional optional argument to your script which allows the user to output a plot of the total energy. The names of the plot files should be `file_name.png` where the name of the file was `file_name.mdout`.

Solution

Key Points

- You must `import argparse` in your code to accept user arguments.
- You add must first create an argument parser using `parser = argparse.ArgumentParser`
- You add arguments using `parser.add_argument`


(../06-
functions/index.html)


(../08-
testing/i

Copyright © 2018–2020 The Molecular Sciences Software Institute (<http://molssi.org>)

Edit on GitHub (https://github.com/MolSSI-Education/python_scripting_cms/edit/gh-pages/_episodes/07-command_line.md) / Contributing (https://github.com/MolSSI-Education/python_scripting_cms/blob/gh-pages/CONTRIBUTING.md) / Source (https://github.com/MolSSI-Education/python_scripting_cms/) / Cite (https://github.com/MolSSI-Education/python_scripting_cms/blob/gh-pages/CITATION) / Contact (<mailto:education@molssi.org>)

Using The Carpentries style (<https://github.com/carpentries/styles/>) version 9.5.0 (<https://github.com/carpentries/styles/releases/tag/v9.5.0>).