This lesson is being piloted (Beta version)

Writing Functions

Overview

Teaching: 25 min
Exercises: 15 min
Questions

- How do I write include functions in my code?

Objectives

- Define functions to perform one computational task.
- Use functions within code and inside `for` loops.
- Write functions that accept user inputs.
- Assign default values for function inputs.

Why functions?

Most code is organized into blocks of code which perform a particular task. These code blocks are called *functions*. A commercial software package likely has hundreds of thousands or millions of functions. Functions break up our code into smaller, more easily understandable statements, and also allow our code to be more *modular*, meaning we can take pieces and reuse them. Functions also make your code easier to test, which we will see in a later lesson.

In general, each function should perform only one computational task.

Defining and running a function

In Python, the following syntax is used to declare a function:

Python

```
def function_name(parameters):
    ** function body code **
    return value_to_return
```

Functions are defined using the `def` keyword, followed by the name of the function. The function may have parameters which are passed to it, which are in parenthesis following the function name. A function can have no parameters as well. Most (though not all) functions return some type of information. It is important to note here that defining a function does not execute it.

Writing functions into our geometry analysis project

Let's go back and consider a possible solution for the geometry analysis project.

Python

```python
import numpy
import os

file_location = os.path.join('data', 'water.xyz')
xyz_file = numpy.genfromtxt(fname=file_location, skip_header=2, dtype='unicode')
symbols = xyz_file[:, 0]
coordinates = xyz_file[:, 1:]
coordinates = coordinates.astype(numpy.float)
num_atoms = len(symbols)
for num1 in range(0, num_atoms):
    for num2 in range(0, num_atoms):
        if num1 < num2:
            x_distance = coordinates[num1, 0] - coordinates[num2, 0]
            y_distance = coordinates[num1, 1] - coordinates[num2, 1]
            z_distance = coordinates[num1, 2] - coordinates[num2, 2]
            bond_length_12 = numpy.sqrt(x_distance ** 2 + y_distance ** 2 + z_distance ** 2)
            if bond_length_12 > 0 and bond_length_12 <= 1.5:
                print(F'{symbols[num1]} to {symbols[num2]} : {bond_length_12:.3f}')
```

To think about where we should write functions in this code, let's think about parts we may want to use again or in other places. One of the first places we might think of is in the bond distance calculation. Perhaps we'd want to calculate a bond distance in some other script. We can reduce the likelihood of errors in our code by defining this in a function (so that if we wanted to change our bond calculation, we would only have to do it in one place.)

Let's change this code so that we write a function to calculate the bond distance. As explained above, to define a function, you start with the word `def` and then give the name of the function. In parenthesis are in inputs of the function followed by a colon. The the statements the function is going to execute are indented on the next lines. For this function, we will `return` a value. The last line of a function shows the return value for the function, which we can use to store a variable with the output value. Let's write a function to calculate the distance between atoms.

Python

```python
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
```

Now we can change our `for` loop to just call the distance function we wrote above.

Python

```python
for num1 in range(0, num_atoms):
    for num2 in range(0, num_atoms):
        if num1 < num2:
            bond_length_12 = calculate_distance(coordinates[num1], coordinates[num2])
            if bond_length_12 > 0 and bond_length_12 <= 1.5:
                print(F'{symbols[num1]} to {symbols[num2]} : {bond_length_12:.3f}')
```

Next, let's write another function that checks to see if a particular bond distance represents a bond. This function will be called `bond_check`, and will return `True` if the bond distance is within certain bounds (At first we'll set this to be between 0 and 1.5 angstroms).

Python

```python
def bond_check(atom_distance):
    if atom_distance > 0 and atom_distance <= 1.5:
        return True
    else:
        return False
```

This is great! Our function will currently return `True` if our bond distance is less than 1.5 angstrom.

Exercise

Modify the `bond_check` function to take a minimum length and a maximum length as user input.

Answer

Function Documentation

Recall from our work with tabular data that we were able to use `help` to see a help message on a function. As a reminder, we used it like this.

Python

```
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, converters=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=Tr
ue, 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.
```

Let's try the same thing on our function.

Python

```
help(calculate_distance)
```

Output

```
Help on function calculate_distance in module __main__:

calculate_distance(atom1_coord, atom2_coord)
```

There is no help for our function! That is because you haven't written it yet. In Python, we can document our functions using something called `docstrings` . When you call help on something, it will display the docstring you have written. In fact, most Python libraries use docstrings and other automated tools to pull the docstrings out of the code to make online documentation. For example, see the documentation (https://docs.scipy.org/doc/numpy/reference/generated/numpy.genfromtxt.html) for the `genfromtxt` function online.

To add a docstring to our function, we simply add a block quote directly underneath the function definition. We do this in in the same way we type a string, except that we use three quotation marks to open and close the string instead of one.

Python

```
def calculate_distance(atom1_coord, atom2_coord):
    """Calculate the distance between two three-dimensional points."""

    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
```

We are using a very simple docstring in this example. However, there are many formats for docstrings. Now, you should see a message when you call help on this function.

Python

```
help(calculate_distance)
```

Output

```
Help on function calculate_distance in module __main__:

calculate_distance(atom1_coord, atom2_coord)
    Calculate the distance between two three-dimensional points
```

If you use a well-known format, you can use software to extract the docstring and make a webpage with your documentation. MolSSI recommends using numpy style docstrings. You can learn more about this in our Python Package Development Best Practices Workshop (https://molssi-education.github.io/python-package-best-practices/).

Help vs Online Documentation

Many python libraries we have used such as numpy and matplotlib have extensive online documentation. It is a good idea to use online documentation if it is available. Typically, documentation for functions will be pulled from docstrings in the code, but additional information the code developers have provided will also be available through online documentation.

However, if you are offline or using a library without online documentation, you can check for documentation using the `help` function.

Remember, help for your code only exists if you write it! Every time you write a function, you should take some time to also write a docstring describing what the function does.

Function Default arguments

When there are parameters in a function definition, we can set these parameters to default values. This way, if the user does not input values, the default values can be used instead of the code just not working. For example, if we want the default values in bond check to be 0 and 1.5, we can change the function definition to the following:

Python

```
def bond_check(atom_distance, minimum_length=0, maximum_length=1.5):
    """
    Check if a distance is a bond based on a minimum and maximum bond length.
    """

    if atom_distance > minimum_length and atom_distance <= maximum_length:
        return True
    else:
        return False
```

Let's try out the function now.

Python

```
print(bond_check(1.5))
print(bond_check(1.6))
```

Output

```
True
False
```

However, we can overwrite `minimum_length` or `maximum_length` .

Python

```
print(bond_check(1.6, maximum_length=1.6))
```

Now that we have our `bond_check` function, we can use it in our `for` loop to only print the bond lengths that are really bonds.

Python

```
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(coordinates[num1], coordinates[num2])
            if bond_check(bond_length_12) is True:
                print(F'{symbols[num1]} to {symbols[num2]} : {bond_length_12:.3f}')
```

Output

```
O to H1 : 0.969
O to H2 : 0.969
```

Exercise

We might also want to write a function that opens and processes our `xyz` file for us. Write a function called open_xyz which takes an xyz file as a parameter and returns the symbols and coordinates.

Hint: You can return two values from a function by typing `return variable1, variable2` .

Answer

We've now written three functions. Using these functions, our script to print bonded atoms now looks like this:

Python

```python
import numpy
import os

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}')
```

Time Check

If you are running out of time, this is a good place to end the lesson. You can still complete the testing lesson even if you only got this far.

You can probably think of a further extension to use functions here. What if we wanted to print the bonds for another `xyz` file besides water? One option would be to copy and paste the two two `for` loops we've written. However, the smarter move is to put them in a function.

We could encapsulate the `for` loops into a function as well. Let's do this, then perform the same analysis using both the `water.xyz` file and the `benzene.xyz` file.

First, we will define a new function `print_bonds` , which takes bond symbols and coordinates from an `xyz` file as an input.

Python

```python
def print_bonds(atom_symbols, atom_coordinates):
    num_atoms = len(atom_symbols)
    for num1 in range(0, num_atoms):
        for num2 in range(0, num_atoms):
            if num1 < num2:
                bond_length_12 = calculate_distance(atom_coordinates[num1], atom_coordinates[num2])
                if bond_check(bond_length_12) is True:
                    print(F'{atom_symbols[num1]} to {atom_symbols[num2]} : {bond_length_12:.3f}')
```

If you were to put all the functions we wrote into a single cell, it looks like this:

Python

```python
import numpy
import os

def calculate_distance(atom1_coord, atom2_coord):
    """
    Calculate the distance between two three-dimensional points.
    """
    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):
    """Check if a distance is a bond based on a minimum and maximum bond length"""

    if atom_distance > minimum_length and atom_distance <= maximum_length:
        return True
    else:
        return False

def open_xyz(filename):
     """
     Open and read an xyz file. Returns tuple of symbols and coordinates.
     """
     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

def print_bonds(atom_symbols, atom_coordinates):
    """
    Prints atom symbols and bond length for a set of atoms.
    """
    num_atoms = len(atom_symbols)
    for num1 in range(0, num_atoms):
        for num2 in range(0, num_atoms):
            if num1 < num2:
                bond_length_12 = calculate_distance(atom_coordinates[num1], atom_coordinates[num2])
                if bond_check(bond_length_12) is True:
                    print(F'{atom_symbols[num1]} to {atom_symbols[num2]} : {bond_length_12:.3f}')
```

We can now open an arbitrary `xyz` file and print the bonded atoms. For example, to do this for water and benzene, we could execute a cell like this:

Python

```python
water_file_location = os.path.join('data', 'water.xyz')
water_symbols, water_coords = open_xyz(water_file_location)

benzene_file_location = os.path.join('data', 'benzene.xyz')
benzene_symbols, benzene_coords = open_xyz(benzene_file_location)

print(F'Printing bonds for water.')
print_bonds(water_symbols, water_coords)

print(F'Printing bonds for benzene.')
print_bonds(benzene_symbols, benzene_coords)
```

Output

```
Printing bonds for water.
O to H1 : 0.969
O to H2 : 0.969
Printing bonds for benzene.
C to H : 1.088
C to C : 1.403
C to C : 1.403
C to H : 1.088
C to C : 1.403
C to H : 1.088
C to C : 1.403
C to H : 1.088
C to C : 1.403
C to H : 1.088
C to C : 1.403
C to H : 1.088
```

Extension

In earlier lessons, we used `glob` to process multiple files. How could you use `glob` to print bonds for all the xyz files?

Solution

Key Points

- Functions make your code easier to read, more reuseable, more portable, and easier to test.
- If a function returns True or False, you can use it in an `if` statement as a condition.

‹ (../05-plotting/index.html)

› (../07-comma