Lecture 3: Functions and modular code

Dr Benjamin J. Morgan[1] and Dr Andrew R. McCluskey[1,2]

[1] *Department of Chemistry, University of Bath, email: b.j.morgan@bath.ac.uk*
[2] *Diamond Light Source, email: andrew.mccluskey@diamond.ac.uk*

September 24, 2019

Aim

This lecture will introduce how to simplify your code by writing functions and how to reuse code easily in many different places.

1 Functions

Occasionally, there is a particular section of your code that you would like to reuse over-and-over, without having to write the code over and over (as previously mentioned, programmers are lazy). For this, we make use of *functions*, the use of which will be familiar. For example, we have used the `print()` function already in previous weeks, and the NumPy library contains a wide variety of functions, some of which were introduced last week. However, this week we shall see how it is possible to write our own functions in Python.

The general syntax for *defining* a function in Python is as follows,

```
# Defining a function

def my_function(argument_1, argument_2):
    """
    Adds together two arguments.
    """
    result = argument_1 + argument_2
    return result
```

Above, we defined a function named `my_function` which took two *arguments*, added them together to produce a result, which was *returned*. Once defined it is possible to use this function in our code as follows,

```
# Using our function

a = 1
b = 2
c = my_function(a, b)

print(c)
```

It can be seen when looking at the above example, that the object that is returned from the function is then assigned to the variable `c`. The text immediately following the function definition line is the *docstring*, this is important to help others understand the purpose of the function without having to read the code (basically it is a description of the function's action).

The above example of a function is relatively simple, just adding together two numbers. However, a function can contain a large amount of code abstracted to a single line. Furthermore, a well named function can increase the readibility of code significantly (consider the atomic distance code from previous weeks, the abstraction of the distance calculation to an appropriately named function would make this more understandable).

The `my_function` example above contained two *required* arguments. These are the objects that are passed of the function when it is called (and typically these are operated on in some fashion) and are necessary for the function to run. In addition to these required arguments, other arugments may be included in a given function; such as *default* arguments and *variable-length* arguments. These are showing in the functions below,

```
# A function with defaults

def my_function(arg1, arg2, arg3=0):
    """
    Adds together two arguments,
    and an optional third.
    """
    result = arg1 + arg2 + arg3
    return result

print(my_function(1, 2))
print(my_function(1, 2, arg3=3))

# A variable length argument

def my_function(arg1, arg2, *arg3):
    """
    Definitely adds together two arguments,
    but can optionally add together any
    number!
    """
    result = arg1 + arg2
    for i in arg3:
        result += i
    return result

print(my_function(1, 2))
print(my_function(1, 2, 3, 4, 5))
```

Note that in default argument example, if no value was given it would default to `0`.

You should be aware that a Python function does not **need** to have any

arguments. For example, you could write a function that returns the value of the Planck constant (ideally using the SciPy constants library),

```
# A function with no arguments

from scipy import constants

def h():
    """
    Planck constant.
    """
    return constants.h

print('Planck constant = {:.3e}'.format(h()))
```

Exercise

Investigate the effect of returning two values from a function, using the `return a, b` syntax.

2 Modules

In addition to writing your own functions, it is possible to *clean up* your code further by creating your own *modules*. These modules are similar to the libraries (like NumPy and SciPy) that we discussed last week, however, they are created by you. This is achieved by creating a file in the folder where you would like to use the module, and then *importing* the module in the code where your would like to use it. For example, say you have a file called `atom_helper.py` in which you will put many functions related to atomistic calculations. Lets start it off with our distance calculation,

```
# atom_helper.py

import numpy as np

def distance(a, b):
    """
    Determines the distance between two points
    in any number of dimensions.
    """
    return np.sqrt(np.sum(np.square(a - b)))
```

Having created this module, we can then import it into a Jupyter Notebook that is in the same folder, and access the distances function as shown below,

```
# Using the atom_helper module

import numpy as np
```

```
import atom_helper

atom1 = np.array([1, 4, 2])
atom2 = np.array([5, 2, 6])

dist = atom_helper.distance(atom1, atom2)
print(dist)
```

It is clear from the above example, that the *user-defined* module can be imported in the same way as a library. Then any function defined in the module may be called from the Jupyter Notebook. This means that you can create modules containing a wide variety of functions and use them in many different Notebooks.

3 Problem

It is possible to estimate the interaction energy, E_{LJ} between two atoms using the Lennard-Jones expression,

$$E_{LJ} = \epsilon\left[\left(\frac{r_m}{r}\right)^{12} - 2\left(\frac{r_m}{r}\right)^{6}\right], \tag{1}$$

where, r is the distances between the two atoms, ϵ and r_m are constants describing the shape of the interaction potential surface. Add a function that can calculate the interaction energy, using the above relationship with constants of $\epsilon = -2.2\,\mathrm{kJ\,mol^{-1}}$ and $r_m = 3.7\,\text{Å}$, to the `atom_helper.py` module.

Use the new function to calculate the interactions energy for each of the pairs of atoms used in Week 2.

3.1 Energy minimisation

One of the most common operations in computational chemistry is energy minimisation, which is used to determine the most stable geometry for a given chemical molecule or structure. The energy minimum is where there are no net forces acting on any of the atoms; the structure is relaxed. This process will be covered in future lectures, however, in this exercise you will code the steps that can be found at the core of many energy-minimisation programs.

If r_0 is the initial distance between two atoms, and E' and E'', respectively, are the first and second derivatives of the interaction energy, E, then a step towards the energy minimum is given by the following,

$$r_1 = r_0 - \frac{E'}{E''}, \tag{2}$$

where, r_1 is the new distance between the two atoms. We can then reevaluated E' and E'' for the new r, and this process is repeated until E reaches some minimum value.

For the purpose of this exercise, the interatomic interaction energy, E, and its derivatives are given by the followign reformulation of the Lennard-Jones

equation,

$$\begin{aligned} E &= \frac{A}{r^{12}} - \frac{B}{r^6} \\ E' &= -12\frac{A}{r^{13}} + 6\frac{B}{r^7} \\ E'' &= 156\frac{A}{r^{14}} - 42\frac{B}{r^8} \end{aligned} \tag{3}$$

Write a program that, from values of A, B and an initial guess of r, uses a 10 iteration loop to move towards the distance at which the energy minimum is present. In each iteration: print a line containing the initial value of r (i.e. r_0), E, E', E'', and the final r (i.e. r_1), and replace r_0 with r_1. You will need to have three functions (one for each of E, E', and E'') which will take A, B, and r as their arguments. *Remember to plan the algorithm first!*

Given the following values for A and B, what is the energy-minimum distance r (in Å)?

1. $A = 1 \times 10^5\,\mathrm{eV\AA}^{12}$ and $B = 40\,\mathrm{eV\AA}^{6}$
2. $A = 4 \times 10^5\,\mathrm{eV\AA}^{12}$ and $B = 1\,\mathrm{eV\AA}^{6}$

Adjust the program so that it will do as many iterations as are necessary to achieve energy minimisation, rather than only a fixed number. Compare cases 1. and 2. to see how stable and/or quick the minimisation is and how the *convergence* depends on the initial guess of r.

Next, create a new function in your module to calculate the energy, and first and second derivatives of a harmonic oscillator (a common model for a covalent bond),

$$\begin{aligned} E &= \frac{1}{2}k(r - r_{\mathrm{eq}})^2 \\ E' &= k(r - r_{\mathrm{eq}}) \\ E'' &= k \end{aligned} \tag{4}$$

Verify that the method developed above (known as the Newton-Raphson method) converges to the energy minimum in a single step, regardless of the initial value of r, within this harmonic potential. For this test, use $k = 32.2\,\mathrm{eV\AA}^{-2}$, which is the force constant for a HCl molecule, and $r_{\mathrm{eq}} = 1.27\,\mathrm{\AA}$, where r_{eq} is the equilibrium bond distance.