Lecture 4: 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*

July 29, 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 arguements.
    """
    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 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):
    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):
    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`.