Lecture 5: Good coding practice and testing

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*

October 7, 2019

Aim

This lecture will introduce several aspects of *good coding practice* before going on to cover how to write tests to ensure your code it doing what you think it is.

1 Writing readable code

One of the benefits of using Python over other programming language is the clear syntax. However, we can improve the readablity and comprehensibility of our code further but *writing readable code*. Now, this isn't a comment on handwriting or grammar but rather a discussion into aspects such as:

- Using sensible variable names
- Being consistent
- Being concise, but clear
- Use conventions

If we consider the following example, where we are trying to determine which elements in the list below are liquid at room temperature. The code below is not readable (at least not easily) but it is trying to print the elements that are liquid at room temperature,

```
# Ugly, hard to read code

s = ['H', 'Ca', 'Fe', 'Hg', 'Br', 'Xe']
m = [13.99, 1115, 1811, 234.321, 265.8, 161.4]
b = [20.271, 1757, 3134, 629.88, 332, 165.051]
for k, d in enumerate(range(0, len(s))):
    if m[d] < 273.15 and b[d] >= 273.15:
        print(s[d])
        print("This element is a liquid and not a solid or a gas
            ↪ at standard temperature and pressure.")
```

It is not immediately clear what this code is aiming to achieve. For example, the three lists are simply giving the names `s`, `m`, and `b`, which the author (currently) knows are short for chemical `s`ymbols, `m`elting point, and `b`oiling point. However, if someone else was to try and understand this it is not clear, or it the original author comes back to the code many months (or years) later. Additionally, on the third line the code isn't very concise (or clear) as the loop enumerates through a range function, such that the variables `k` and `d` (again poorly named and not in keeping with conventions) always have the same value. Finally, the final line is very long, running to over 100 characters (when is widely considered that the maximum readbility is found when there is less than 80 characters per line).

Some of the guidelines regarding the readability of Python code are outlined in the PEP 8 documentation (`https://www.python.org/dev/peps/pep-0008/`). However, many of them are not possible to legislate on, but the mantra of the Python is to write clear, simple code to the best of your ability.

Exercise

Rewrite the code to be more readable and clear, remembering to have descriptive variable names, to be concise and to consider iterations conventions. You may also test you code in an online PEP 8 checker (`http://pep8online.com`) to see if it passes the guidelines.

2 Comments and docstrings

One of the easiest ways to make it more clear what your code is trying to do is to add comments. These are lines in the code that are ignored by the Python interperator. You will have seen these in most of the code blocks in these handouts, where the line starts with a `#` symbol. This means that the line will be ignored, and these can be a great way to give the reason for a particular decision in your code. For example, before the fifth line of the code block above the following comment could be included,

```
# This line determines if room temperature is between the
# melting and boiling point of the given element
```

A comment like this makes the purpose of the `if` statement much clearer.

A subset of the comments are *docstrings*, these are comments that describe the purpose of a function, often including the expected parameters. There are a variety of styles that are used in docstrings; the most common are Sphinx, Google, and NumPy. However, we will focus on using the NumPy style as it is the most human readable, however you should be aware that these other styles exist. An example of a docstring is shown below the function definition below,

```
# An example of a docstring
import numpy as np

def pH(H):
    """Determine the pH for a given H+ concentration
```

```
    Parameters
    ----------
    H: float
        Concentration of H+ (or H3O+) in solution

    Returns
    -------
    float
        The pH value
    """
    return np.log10(H)
```

As can be seen, the typical docstring consists of three parts, the function description, the input parameters, and the returned values. Note that the input parameters and returns statements include information about the variable types for each.

The utility of the docstrings can be leveraged in the Jupyter notebook, as the docstrings for a given function are easily accessible. For example, if you defined the `pH` function above, it would be possible to investigate the docstring using the following command in a Jupyter Notebook cell,

```
pH?
```

This will return the docstring for the `pH` function in a pop up window at the bottom of the page. This helper functionality is not limited to custom functions, as most library functions include detailed docstrings aswell.

Exercise

Use the docstring helper functionality to investigate some of the NumPy functions that you have encounters so far and the following:

- `scipy.optimize.curve_fit`
- `numpy.fft.fft`
- `matplotlib.pyplot.bar`

Note for some of these you will need to `import` the appropriate libraries.

3 Testing

The final topic to cover in this lecture is testing, that is tests for your code to ensure that they do what you think they do. It was mentioned in the debugging lecture that people aren't perfect and therefore cannot write perfect code. Therefore it is important that we write tests to make sure that our code is doing the right thing under all circumstances.

Typically tests are applied at a function level, where there is, at least, a test for each function. However, in order to achieve full test coverage for a

given function more than one test may be necessary. Consider the function below, which will convert either degrees Fahrenheit or degress Celsius to Kelvin (similar to the Problem from the first week),

```
# A temperature conversion

def temp_conv(value, unit='C'):
    """This can convert either Celsius or Fahrenheit to Kelvin

    Parameters
    ----------
    value: float
        Temperature to be converted
    unit: str
        Unit of original temperuture (either 'C' or 'F', defaults
            ↪ to 'C')

    Returns
    -------
    float
        Converted temperature (in K)
    """
    if unit == 'C':
        return value + 273.15
    elif unit == 'F':
        return (value + 459.67) * 5 / 9
    else:
        return -1
```

In this example we have returned `-1` if the unit is neither `'C'` or `'F'`, it would be more useful to raise a custom exception in this case, however that is beyond the scope of this course.

We can now write some tests for this function, since we know, for example, the melting point of water in all three units (273.15 K, 0 °C, and 32 °F),

```
c = temp_conv(0, 'C')
np.testing.assert_almost_equal(c, 273.15)
f = temp_conv(32, 'F')
np.testing.assert_almost_equal(f, 273.15)
```

This `assert_almost_equal` function is an element of the NumPy testing library which acts on floating point number (which may not be **exactly** equal doing to arithmetic precision). The first argument in this function is the result of the function we are testing, while the second argument is the expected result.

Exercise

Change the values of the expected result to see the output when the test fails, then write the test that has been missed out.

3.1 Test driven development

Test driven development (TDD) is a methodology for computer programming where the tests are written first for a particular used case, and then the operational code is written such that it will satisfy the tests. This methodology is different from that applied to this point in the course, however, it is very popular and powerful when used appriopriately. In the problem below, you will be able to try your hand at some test driven development.

4 Problem

Download the folder entitled `testing` from moodle, and open the Jupyter Notebook included (`testing.ipynb`). This Notebook includes a series of empty functions (the purpose of each is described above), you need to fill in the function, including a docstring,s uch that the tests, that immediately follow each function, pass (they should produce no output). Note that this is an attempt to practice for the first assessment which is modeled after this problem.