PHYS20161 Style guide

Written by Dr. Lloyd Cawthorne

Updated by Dr. Charanjit Kaur
October 2023

Contents

Housekeeping

These notes have been developed to assist you in understanding what aspects of style we are looking for and why. In the past we have received many *queries* where a student has been marked down for code that works and produces the correct answer, though is overcomplicated, difficult to read and not versatile. We have produced these in an attempt to make these deductions more transparent. We want you to write code that not only works, but is well structured, easy to follow and would easily allow you to collaborate.

> "Code is read much more often than it is written."
>
> (Guido van Rossum — Author of Python)

These are not a complete set of notes on style and merely try to outline guidance for this course and PEP8. Feel free to look at other resources, many of which cover more advanced aspects. For instance, Google's Python style guide can be found here: `http://google.github.io/styleguide/pyguide.html`[1]. There are many recognised styles for coding; this is just one of them. We want you all to adhere to the same style as it should prevent you developing any bad habits early. Think of it as if you were working for a company, you will need to share work with many people, some of which you will have never met and you will need to understand each other's code quickly. Writing to the same standards makes this much easier.

We expect all of you to work in the same style outlined in this document.

We mark style by deducting marks for errors rather than awarding marks for good practice. Given style accounts for 37.5 marks, you can only be deducted a maximum of 37.5 for style related issues. This ensures that style is marked independently to calculation and gives us flexibility for the many different submissions that are presented.

[1] Our guide is essentially a subset of Google's.

Chapter 1

Specific aspects

These are aspects that will be checked by a person. They are necessary to understand what the code does line-by-line.

1.1 Structure

We expect your code to presented in a particular order. This makes your code easier to follow and encourages practices which make modifications easier. The order we expect is:

1. Header
2. Import statements
3. Constant definitions
4. Function definitions
5. Main code

If some of these items are unfamiliar at this point do not worry they will become clear in a couple of weeks.

If you say you *have* to deviate from this structure or your code will not work it suggests that your code is poorly constructed and needs modifying. You might also be marked down for whichever part of the code is at fault.

1.2 Headers

Every programme should have an opening comment that explains what the programme does, who wrote it (or uni ID) and when. See listing 1.1 for an example.

```
# -*- coding: utf-8 -*-
"""
Title: An example of a header comment.

Headers should contain a title, description, author and date. They allow a user
to quickly understand the purpose of the code without having to read every
line. Including the author serves slightly as some form of copyright, but also
indicates who should be contacted if there is a query. The date lets you know
if there have been any updates since it was written (either in the software or
the project [e.g. new data released]).

Lloyd Cawthorne 30/01/2020
"""
```

Listing 1.1: Example of basic header with all four key parts.

Note the first line contains an automatically generated comment that specifies the character encoding. It is useful to keep this as many compatibility issues can be related to the encoding; especially when working with international colleagues.

There are many personal styles that are allowed with different ways to break up the various parts, we include another example below, listing 1.2. As long as all parts are included you will not be deducted any marks. You are free to vary the order in which they appear.

```
# -*- coding: utf-8 -*-
"""
_________________TITLE_________________
PHYS20161 - Assignment 3 - Nuclear Decay
-------------------------------------
This python script reads in data from .csv files. It then
validates the imported data by:

1) checking for first condition
2) checking for second condition
3) checking for third condition

If an element is found that breaks one of these rules
then something is done.
The script then performs a variety of things to understand beta decay.

Last Updated: 12/12/19
@author: E. Fermi UID: 9999999
"""
```

Listing 1.2: Different example of basic header with all four key parts, details have been ammended for brevity.

1.3 Constants

Constants are an accepted form of global variable. They are accepted because their value is fixed (although you cannot enforce this in Python) and is not going to be changed when running the programme. They are useful because it allows us to set values in the programme

characteristic of the problem we are solving. They should be defined in `UPPER_CASE` before the function definitions. If the problem changes, then we can just update these values rather than having to trawl through the code and update every occurrence. See listings 1.3 for an example.

```
# -*- coding: utf-8 -*-
"""
Title: Basic example that calculates kinetic energy of object given velocity

This code defines a function that returns the classical kinetic energy given
the velocity.

Note: NumPy is imported but mot used merely to show where constants should
reside.

Lloyd Cawthorne 31/01/2020
"""

import numpy as np

# SI units

MASS = 5.0


def kinetic_energy(velocity, mass=MASS):
    """
    Returns the classical kinetic energy (float) given the velocity and
    mass of the object. Assumes mass is constant unless key word argument
    is given otherwise.
    #
    velocity: float
    mass:    float
    """

    return 0.5 * velocity * mass**2
```

Listing 1.3: Example of basic use of constants. Note NumPy is unusued in this example, but illustrates where the constant, `MASS`, should be defined.

To call the function with a given velocity we simply write the function with one argument as displayed below.

```
In [5]: kinetic_energy(10)
Out[5]: 125.0
```

Note that in this example we bypass calling `MASS` as a global variable within the function by setting it as a default value. To use this function with a different value we have to specify the *key word argument*, or kwarg, e.g. `kinetic_energy(5.0, mass=1.0)`.

1.4 Functions

1.4.1 Function documentation

We expect each function that you define to include a comment with a brief description of what the function does and what variable type the input and output arguments are. This not only helps other users better understand your code, but prepares you for learning other languages where the type declaration is mandatory. An example can be found above in 1.3. Another way of presenting this information can be found in 1.4, this particular example follows a style more similar to Google's.

```
def square_root(x_squared_coefficient, x_coefficient, constant):
    """
    Returns the two roots of a second order polymial.

    Coefficients should be given to conform to

    x_squared_coefficient x^2 + x_coefficient x + constant = 0.

    x = (-x_coefficient +/- sqrt[x_coefficient^2 - 4 x_squared_ coeffiecient
                                * constant]) / (2
                                                * x_squared_coefficient)

    Args:
        x_squared_coefficient: float
        x_coefficient: float
        constant: float
    Returns:
        Two solutions in a list: [float, float]
    Raises:
        ZeroDivisionError: If x_squared_coefficient = 0
        ValueError: Math domain error, imaginary solution

    L. Cawthorne 05/02/20
    """

    try:
        square_root_term = math.sqrt(x_coefficient**2 - 4
                                     * x_squared_coefficient
                                     * constant)
        solution_1 = ((-x_coefficient + square_root_term)
                      / (2 * x_squared_coefficient))
        solution_2 = ((-x_coefficient - square_root_term)
                      / (2 * x_squared_coefficient))
        return [solution_1, solution_2]
    except ZeroDivisionError:
        print('x_squared_coefficient cannot be 0.')
        return None
    except ValueError:
        print('No real solutions.')
        return None
```

Listing 1.4: Example of a function where the arguments, returns and exceptions clearly defined

in the docstring. Note we won't expect exceptions to be listed here, but other styles do.

Some people also include their name and date in this comment, though we do not enforce this.

1.4.2 Defining functions

Each function defined should be accessible to all parts of the code. Furthermore, they should only depend on their input arguments. By all means constants can be called within them, but they must be just that: constant. If your function *has* to be defined part way through your code because it depends on other variables being defined first, you have structured your code incorrectly and will be marked down. Let's consider an example associated with applying Gauss' law.

In this example, the charge is defined as a constant and the user inputs the distance away from the charge. The code then calculates the area of the Gaussian surface and then the electric field at that point. We might think that it would be best to define the function for the area within the function for the electric field, as it is used there, see listings 1.5. However that would be bad style as the code becomes more difficult to read. The function `area_sphere` will still be accessible to `electric_field` if it were defined globally. This example, where a function is defined within another, is known as *currying*[1]. The code will still run correctly and the linter (see chapter 2) might not flag it, but it is still poor practice.

```
1
2  VACUUM_PERMITIVITY = 8.854 * 10**-12  # SI Units, F/m or C/(Vm)
3  CHARGE = 1 # Coulomb
4
5
6  def electric_field(distance, charge=CHARGE):
7      """
8      Calculates the electric field some distance away from a point
9      charge.
10
11     Parameters
12     ----------
13     distance : float
14     charge : float, optional
15              The default is CHARGE.
16     Returns
17     -------
18     float
19     """
20
21     def area_sphere(radius):
22         """
23         Calculates area of a sphere given the radius.
24
25         Parameters
26         ----------
```

[1]There is a place for this style within functional programming, typically when we want to reduce the number of input arguments. However within Python there really isn't any need for it at all. Especially given there are far more elegant and succinct methods to approach these issues as will be given in the course.

```
        radius : float.

        Returns
        -------
        float
        """

        return 4 * np.pi * radius**2


    area = area_sphere(distance)

    return charge / (VACUUM_PERMITIVITY * area)


distance_string = input('How far away is the test charge in m? ')
distance_float = float(distance_string)

print('E(r, q) = E({0:.2f} m, {1:.2f} C) = {2:.3g} V/m'.format(
    distance_float, CHARGE, electric_field(distance_float)))
```

Listing 1.5: Example where function definitions are done incorrectly as one is definined within the scope of another.

Another problem we could encounter is if we calculated the area as an intermediate, global step. If `electric_field` then relies on this as a global variable it requires us to request user input and calculate the area before all functions have been defined. This again is poor practice as function definitions are interspersed with aspects of main code, see listings 1.6. Again, the code will still run correctly, but the code is difficult to read and the functions lack versatility; `electrc_field` could not be used in a different script unless `AREA` was also defined somewhere.

```

VACUUM_PERMITIVITY = 8.854 * 10**-12  # SI Units, F/m or C/(Vm)
CHARGE = 1 # Coulomb


def area_sphere(radius):
    """
    Calculates area of a sphere given the radius.

    Parameters
    ----------
    radius : float.

    Returns
    -------
    float
    """

    return 4 * np.pi * radius**2

distance_string = input('How far away is the test charge in m? ')
distance_float = float(distance_string)
```

```
area = area_sphere(distance_float)

def electric_field(charge=CHARGE):
    """
    Calculates the electric field some distance away from a point
    charge.

    Parameters
    ----------
    distance : float
    charge : float, optional
             The default is CHARGE.
    Returns
    -------
    float
    """

    return charge / (VACUUM_PERMITIVITY * area)




print('E(r, q) = E({0:.2f} m, {1:.2f} C) = {2:.3g} V/m'.format(
    distance_float, CHARGE, electric_field(distance_float)))
```

Listing 1.6: Example where function definitions are done incorrectly as parts of the main code lie between definitions.

The correct style for this code is presented in listings 1.7. Here, we could even define `electric_field` *before* `area_sphere` as when `electric_field` is called in the main code, `area_sphere` is defined.

```
VACUUM_PERMITIVITY = 8.854 * 10**-12  # SI Units, F/m or C/(Vm)
CHARGE = 1 # Coulomb


def area_sphere(radius):
    """
    Calculates area of a sphere given the radius.

    Parameters
    ----------
    radius : float.

    Returns
    -------
    float
    """

    return 4 * np.pi * radius**2


def electric_field(distance, charge=CHARGE):
```

```
    """
    Calculates the electric field some distance away from a point
    charge.

    Parameters
    ----------
    distance : float
    charge : float, optional
             The default is CHARGE.
    Returns
    -------
    float
    """

    area = area_sphere(distance)

    return charge / (VACUUM_PERMITIVITY * area)


distance_string = input('How far away is the test charge in m? ')
distance_float = float(distance_string)

print('E(r, q) = E({0:.2f} m, {1:.2f} C) = {2:.3g} V/m'.format(
    distance_float, CHARGE, electric_field(distance_string)))
```

Listing 1.7: Example where function definitions are done correctly.

1.5 Variable and function names

When declaring variables and defining functions we get to choose their name. In the past few years there has been a strong drive to ensure all names are self-explanatory to reduce the amount of comments. Furthermore, to avoid any ambiguity these should be written in full English. If your variable or function needs a comment to clarify what it is, then it has a poor name.

We use `snake_case` for both variable and function names; all lower case with spaces given by underscores. The linter (see chapter 2) will catch names not adhering to the case, but the relevance of the name will be checked by person.

As physicists we work in the language of mathematics where single character variable names are king. So much so that we have adopted additional alphabets to increase our naming possibilities. However this approach is very much frowned upon when coding. As an example, consider how we define the area, A, of a triangle,

$$A = \frac{bh}{2}, \tag{1.1}$$

where b is the base and h is the height. We define all the symbols in the text. Following the same philosophy in Python, we would write something like the below.

```
A = b * h / 2
```

This alone is meaningless. For someone else to understand this there would at least need to be some comments.

```
# A is area
# b is base
# h is height
A = b * h / 2
```

This *is* better, but ideally we do not need the comments and can just understand what this operation is doing simply by reading it.

```
area = base * height / 2
```

Say we are developing a code that computes the area of multiple shapes; we would want to define this operation as a function. Non-surprisingly a minimalist approach is also bad; no matter if the header clarifies the names.

```
def a(b, h):
    """
    Returns the area of a triangle, float
    Args:
        b, base (float)
        h, height (float)
    """
    return b * h / 2
```

Defining these variables with full names is considerably better.

```
def area(base, height):
    """
    Returns the area of a triangle, float
    Args:
        base (float)
        height (float)
    """
    return base * height / 2
```

Furthermore, in most IDEs when you start writing a function a pop-up box will appear with what arguments the function requires. If you just saw `a(b, h)` this is pretty useless, but `area(base, height)` is much more helpful. If we had multiple functions for different shapes, the function name would also have to specify that it concerns areas of triangles. You might be tempted to make this as brief as possible.

```
def areat(base, height):
    """
    Returns the area of a triangle, float
    Args:
        base (float)
        height (float)
    """
    return base * height / 2
```

However the meaning of the `t` could soon be forgotten. We call a triangle a `triangle` not `t`, or `t1`, or `tri` or `tangl`. Hence, this function should be called `area_triangle`. On a subjective note, I think the two words this way round is better. You are computing an area so would start typing `area` then by pressing tab the IDE will either auto-complete or show a list of

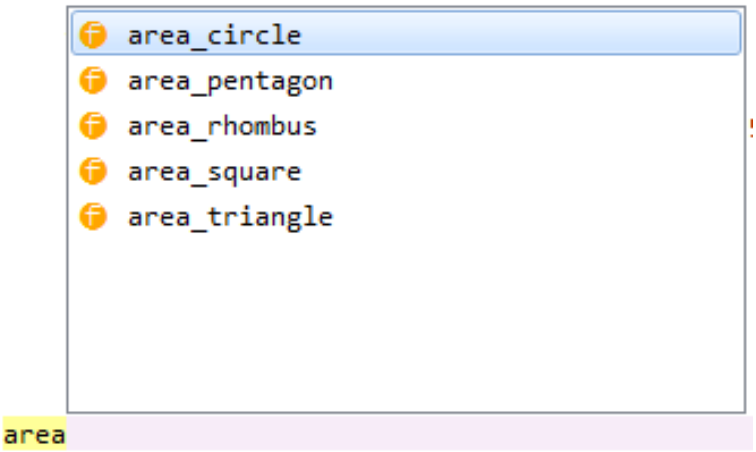

Figure 1.1: If we start writing a function and press tab, a list of all defined functions whose names start the same appears.

options available. If you were to define the area of many shapes this way then all of them would appear together which would be nicer to work with, see figure 1.1.

```
def area_triangle(base, height):
    """
    Returns the area of a triangle, float
    #
    This is the naming convention you should adopt.
    #
    Args:
        base (float)
        height (float)
    """
    return base * height / 2
```

You could also go too far the other way have a variable name that is too verbose, say `area_of_a_plane_figure_with_three_sides_and_three_angles`. This, although precise, removes the readability of the code and what it is doing with this function.

In summary, both variable and function names should be written in full English in `snake_case`. Their names should be meaningful, to remove the need for comments, and concise.

1.6 Opening and closing files

In your assignments you will be asked to read in data from external files. To do this you must open and close the files within Python. We will also expect you to validate if the input file exists and stop the code accordingly if it is not there. Failure to do these tasks results in severe deductions.

Opening files never tends to be an issue, however we often see cases where people have omitted the closing operation. This is very poor as Python will keep the file open until instructed otherwise. This prevents anything else (including us!) from being able to edit the file. Closing the file should be done at the earliest opportunity. Do not wait until the end of

the code to do this. We recognise (and will teach) routines that do not require the file to be closed explicitly.

When reading a file it is important to check the file exists before continuing. Then, if it is no there, exiting nicely. This can be done with an `if` statement or using `sys.exit()`. When marking, we simply change the name of the file and see what the code does, then inspect more thoroughly if needed.

Chapter 2

PEP8 and PyLint

Python Enhancement Proposal-8, PEP8, is the universal style for Python. Pretty much everything you will find online will be written to this standard. As new programmers, being able to write to this style will make you stand out (especially if you want a job in development). There are many little rules you must abide by in PEP8, so it will feel frustrating at first. However, once you get the hang of it, you will see your code becomes far clearer and will look professional.

The full set of guidelines and their reason can be found in the documentation: `https://www.python.org/dev/peps/pep-0008/` .

We will first outline how you can check your code's compliance with PEP8 before describing some of the key criteria. We won't describe everything as that is all explained fully in the documentation.

2.1 Static code analysis

It is very tricky to simply write in accordance with this style, or even proof read and amend everything. To help with this, Spyder has a static code analyser built in: PyLint[1]. This is called a *linter* and checks through your code flagging any aspects which are not inline with the configuration file, the default of which is PEP8. This will allow you to quickly find and amend all of these minor aspects that when combined create very well written code.

When running the analysis, the main output is a score which is determined by

$$\text{errors, warnings, refactors, conventions and statements} \tag{2.1}$$

where 'errors' will typically prevent the code from running, 'warnings' are issues that could potentially go wrong, these are flagged automatically by Spyder; 'refactors' are issues with duplicated or surplus code, 'conventions' are aspects of style regarding spacing and numbers of arguments and 'statements' are the number of instructions that Python executes in the code (assignment: =, `for`, `if`, `while`, etc.).

We strongly encourage you to run the analysis on your code. You can do this by selecting "Run code analysis" from the "Source" drop down menu, as shown in figure 2.1. Once run

[1] Please use the same Pylint version as the one on lab PCs (or check it on those PCs).

Please note for these examples Pylint 2.6.9 was used. For your code please use the same Pylint version as the one on lab PCs (or check it on those PCs).

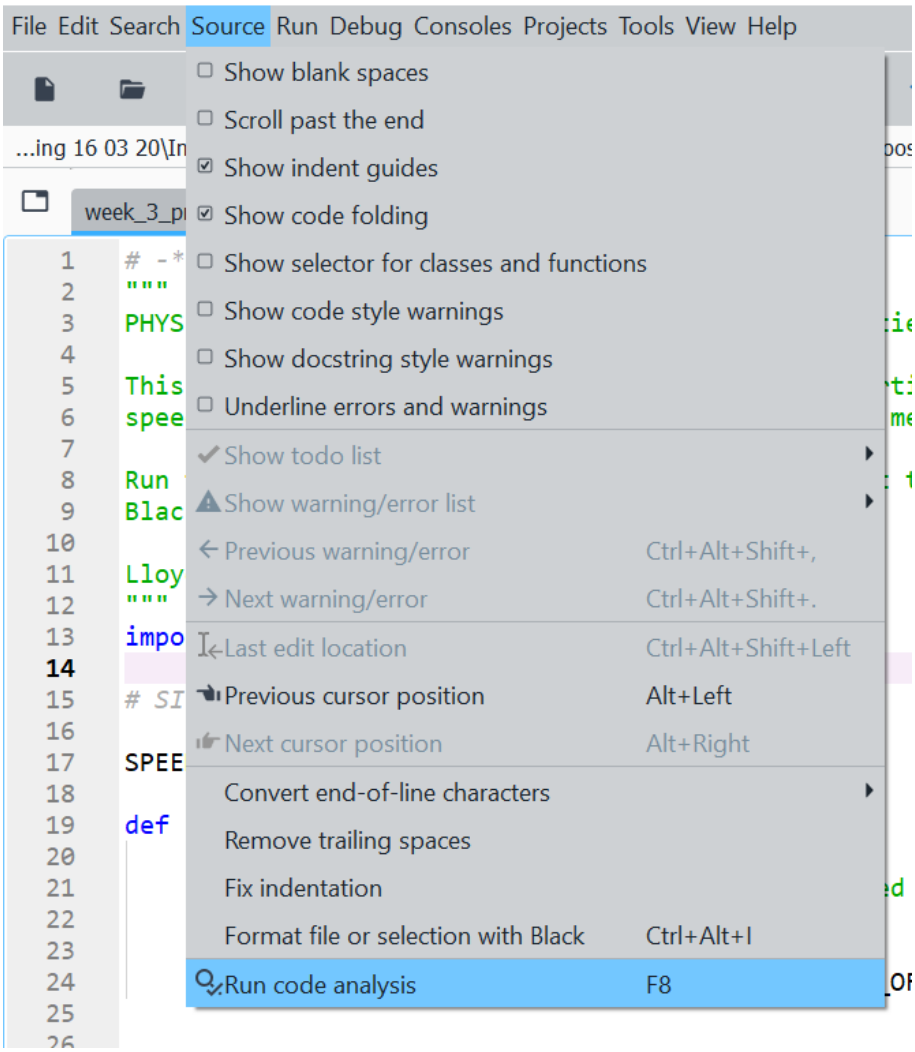

Figure 2.1: How to access the static code analysis in Spyder.

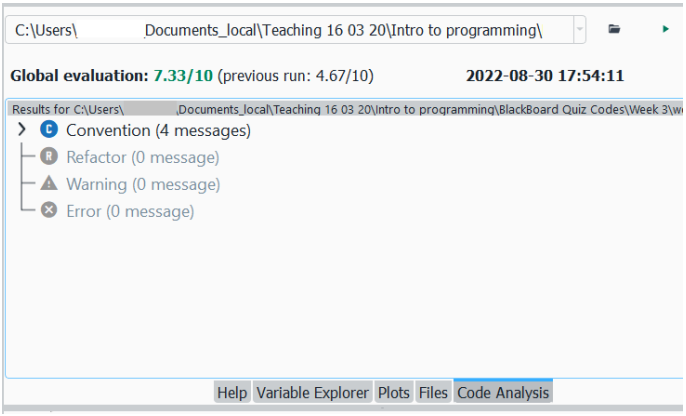

Figure 2.2: Static code analysis pane after initial run.

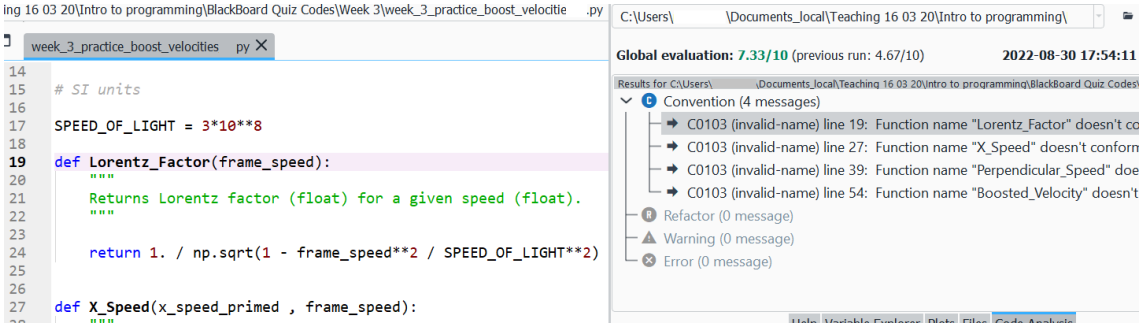

Figure 2.3: List of flagged issues from static code analysis with space before bracket convention highlighted.

(check the path for the code matches the code you are interested in), you will see a score and the number of issues flagged in each category, see fig.2.2. Clicking on each category opens up the list of flagged issues. Clicking on each will direct to you which line in the code needs attention. In figure 2.3 there is an issue with the name of the function `Lorentz_Factor` as it does not conform to `snake_case`. As you can see many of these issues appear very minor as they do not affect the code's ability to run. However being consistent with these aspects is good practice.

2.2 Common convention warnings

Some of the warnings that the linter flag will appear pointless as the code will still work. However, when combined they make code far easier to read and look professional. Often, it will alert you to poor variable and function names. If you write a function that has too many input or internal variables, it will flag those too. In these cases, it is best to think about how to split the function into simpler tasks.

You will also likely come across refactor warnings. These relate to the code logic and are described in more detail below.

2.3 Refactors

Refactoring issues are more common in more elaborate pieces of code that have inefficient logic. There are many posts online about these as some of the flags are controversial and generate debate. Where it might be seen is if you unnecessarily increase the complexity of the code, typically `else` after `return`. An example of such a warning is found in listings 2.1, the associated message is shown in figure 2.4.

```
1  def function_1(number_1, number_2):
2      """
3      Prints messages based on which of the inputs is greater.
4      This version generates a refactor warning.
5      Args:
6          number_1 (float)
7          number_2 (float)
8      """
9
10     if number_1 > number_2:
11         print('{0:.1f} is greater than {1:.1f}.'.format(number_1,
12                                                         number_2))
13         return None
14
15     else:
16         print('{0:.1f} is not greater than {1:.1f}.'.format(number_1,
17                                                             number_2))
18         return None
19
20
21 def function_2(number_1, number_2):
22     """
23     Prints messages based on which of the inputs is greater.
24     This version does not generate a refactor warning.
25     Args:
26         number_1 (float)
27         number_2 (float)
28     """
29
30     if number_1 > number_2:
31         print('{0:.1f} is greater than {1:.1f}.'.format(number_1,
32                                                         number_2))
33         return None
```

```
34
35      print('{0:.1f} is not greater than {1:.1f}.'.format(number_1,
36                                                          number_2))
37      return None
```

Listing 2.1: Two versions of the same function. The first case has a refactor flag whilst the second does not.

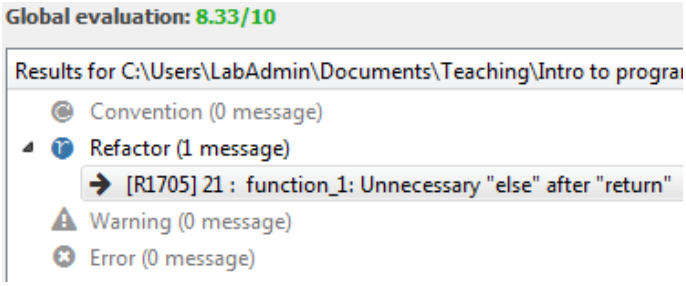

Figure 2.4: Message generated by PyLint for refactor flag in the first function defined in listings 2.1.

In this example the `else` statement is redundant and requires an extra level of unnecessary indentation, thus increasing the complexity.

2.4 Whitespace and autoformatting

Many of PEP8s standards refer to correct spacing; that is, when there should be a space and when there should not. Previously the user had to adjust all of these by hand, which was very tedious. Thankfully autoformatters have not only been developed but also integrated into Spyder.

We expect your code to have appropriate spacing. The simplest way to do this is to enable the autoformatter. For consistency, we will use *autopep8*[2] To enable the autoformatter, go to Tools → Preferences → Completion and linting → Code style and formatting. Scroll down and under 'Code Formatting' select 'autopep8' as the provider and click the box that autoformats on save. At this point, it is also worth setting the maximum line length to 79 characters (see the next subsection for details). **Then click Apply.** These settings are shown in figure 2.5. If you do not like your code being automatically formatted on save, you can simply highlight the code you want to format in the editor, right click and select autoformat.

These changes might appear random at first, but they are following a set of rules. These are briefly described in the below subsections.

2.4.1 Maximum line length

There is growing divided opinion on how long a line of code should be. Historically it was 79 characters[3]. Fortunately this is also autopep8's default.

[2]Documentation: `https://pypi.org/project/autopep8/`
[3]I personally prefer this and write most of my code to this standard.

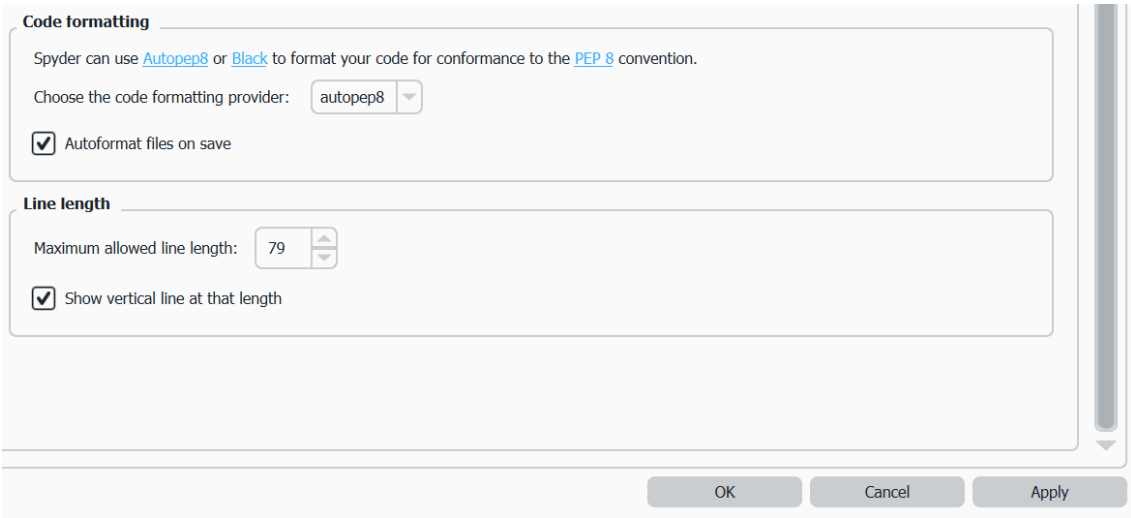

Figure 2.5: Prefence settings to enable the autoformatter autopep8 with correct line length.

All lines should be a maximum of 79 characters long. This ensures that all code is visible without having to scroll horizontally. As a consequence of this, the number of levels of indentation that is acceptable is restricted. This also leads to a series of conventions of how to deal with different types of line breaks as outlined further below.

2.4.2 Line breaks and indentation

Indentation is key to Python as it specifies the variable scope. **Each level of indentation should be specified with four spaces.** Though in most IDEs you can specify tab to return this to make typing easier.

Where this becomes tricky is if you have to line break. The convention when defining functions is to include an extra level of indentation for the variable names, see listings 2.2. If you need more than one line to list the variables, then subsequent lines should start at this level.

```
1  def chi_squared_function_good(
2          observed_data, predicted_data, uncertainty_on_observed_data):
3      """
4      Returns the chi_squared (float) for the data given.
5      #
6      observed_data [float]
7      predicted_data [float]
8      uncertainty_on_observed_data [float]
9      """
10
11     return np.sum(((observed_data - predicted_data)
12                    / uncertainty_on_observed_data)**2)
```

Listing 2.2: Example of defining a function where line breaks are necessary and are done correctly.

In this example the line is broken at the opening parenthesis. If we were to ignore this

convention, but adhere to the maximum line length, then it would be unclear where the function starts as different aspects are on the same level, see listings 2.3.

```
def chi_squared_function_bad(observed_data, predicted_data,
    uncertainty_on_observed_data):
    """
    Returns the chi_squared (float) for the data given.

    observed_data [float]
    predicted_data [float]
    uncertainty_on_observed_data [float]
    """

    return np.sum(((observed_data - predicted_data)
    / uncertainty_on_observed_data)**2)
```

Listing 2.3: Example of defining a function where line breaks are necessary but different parts occupy the same level of indentation.

Similarly, the output is broken before the mathematical operator and aligned to the innermost parenthesis. Spyder does this alignment automatically when pressing the return key (or at least it should).

When calling the function, break the line at the comma and start the next line at the level of the opening parenthesis, see listing 2.4.

```
chi_squared_function(observed_data, predicted_data,
                     uncertainty_on_observed_data)
```

Listing 2.4: Example indenting when calling a function. Note, the variable names should be different to those in the function definition to prevent confusion.

Mathematical operations should be broken up such that the next line starts with the operator at the same level as the previous term. To ensure Python understand what we mean, the terms should be wrapped in parenthesis, see listings 2.5 for a couple of examples.

```
def leonard_jones_potential(distance, equilibrium_distance, potential_depth):
    """
    Returns the value of the Leonard-Jones potential (float).

    Args:
        distance (float)
        equilibrium_distance (float)
        potential_depth (float)
    """

    return potential_depth * ((equilibrium_distance / distance)**12
                              - 2 * (equilibrium_distance / distance)**6)


def distance(initial_distance, initial_velocity, acceleration, time):
    """
    Returns the position of an object subject to uniform accelereation (float).
    Args:
        initial_distance (float)
```

```
        initial_velocity (float)
        acceleration (float)
        time(float)
    """

    return (initial_distance + initial_velocity * time
                             + 0.5 * acceleration * time**2)
```

Listing 2.5: Further examples of breaking long mathematical expressions. Note parenthesis are used in the unifrom acceleration example to ensure Python reads multiple lines.

Breaking long strings should be done with either a backslash, \, or parentheses; see listings 2.6.

```
# Using a backslash
long_string = "In computer programming, a string is traditionally a " \
              "sequence of characters, either as a literal constant or as " \
              "some kind of variable. The latter may allow its elements to" \
              "be mutated and the length changed, or it may be fixed."

# Using parenthesis
another_long_string = ("A parenthesis is a punctuation mark used to enclose "
                       "information, similar to a bracket. The open "
                       "parenthesis, which looks like (, is used to begin "
                       "parenthetical text. The close parenthesis, ), denotes "
                       "the end of parenthetical text.")
```

Listing 2.6: Examples of two acceptable methods to line break long strings.

2.4.3 Vertical spacing or blank lines

Blank lines allow you to separate different aspects of the code into different blocks. If you don't have any blank lines it becomes hard to see where one item ends and another starts. If you have too much then the code looks sparse.

Function definitions should be surrounded by two blank lines above and below, see listing 2.5. Otherwise, single blank line should be used to separate the logic of the code. So you might have a series of lines making a calculation; you might want a blank line between that and the next calculation.

Note the last line of the programme should be blank.

2.4.4 Horizontal spacing or whitespace

There are a multitude of conventions regarding spaces between characters in PEP8. These will all be fixed by black. We have summarised them here so you can try to write in accordance with the standards and see what the autoformatter is aiming for.

When to include a space:

- Assignment operations: `=`, `+=`, `*=`, etc.

```
# Do this:
number = 1

# Don't do this:
number=1
```

- Around comparisons: ==, >, <=, etc.

```
# Do this:
if number >= 1:
    print('Number is greater than or equal to one.')

# Don't do this:
if number>=1:
    print('Number is greater than or equal to one.')
```

- After hash mark, #, when writing a comment.

```
# Do this

#Don't do this
```

- After a comma.

```
# Do this:
value = function(variable_1, variable_2):

# Don't do this:
value = function(variable_1,variable_2):
```

- Around mathematical operations to show priority.

```
# Do this:
value = variable_1**2 - variable_2**2

# Don't do this:
value = variable_1**2-variable_2**2
```

When not to include a space:

- At the end of a line; trailing whitespace. This is the most difficult to spot as it is invisible. It might seem pointless but it can stop code compiling if a statement is written over multiple lines. Furthermore, it requires more memory. Finally, when writing you want to be able to press 'end' and start typing, not have to delete a few spaces first.

- Before a comma, semicolon or colon.

```
# Do this:
value = function(variable_1, variable_2)

# Don't do this:
value = function(variable_1 , variable_2)
```

- Before and after a parenthesis, bracket or brace.

```
# Do this:
value = function(variable_1, variable_2)

# Don't do this:
value = function( variable_1, variable_2 )
```

- Between a function/list and it's opening parenthesis/bracket.

```
# Do this:
value = function(variable_1, variable_2)

# Don't do this:
value = function (variable_1, variable_2)
```

- When passing keyword arguments.

```
# Do this:
def function(variable_1, variable_2, g_constant=9.81):

# Don't do this:
def function(variable_1, variable_2, g_constant = 9.81):
```

- To align.

```
# Do this:
raw_value = 5.6
propagated_value = 7.4

# Don't do this:
raw_value        = 5.6
propagated_value = 7.4
```

There are many more rules and caveats for more advanced routines. If you are curious then look at the PEP8 documentation. If you have written a long code that has some instances where you feel breaking these rules is necessary for clarity, this should be OK as the linter score is weighted by the number of statements and you have 0.10 of leeway. We leave it to you to judge what is best in this case.