Week 1

Data Science Laboratory 1

Dr John Evans
`j.evans8@herts.ac.uk`

University of Hertfordshire UH

Plan for today

Overview for the course

PEP-8 Coding Standards

- Code Layout
- Imports
- Whitespace
- Naming Conventions
- Comments

Debugging

Goal for this course

Introduce you to key topics in data science, such as clustering, classification and regression. In particular, we shall do so in a practical way which will encourage you to work with python (and potentially other languages), and develop good programming practices.

Goal for this course

continuous data

discrete data

Introduce you to key topics in data science, such as clustering, classification and regression. In particular, we shall do so in a practical way which will encourage you to work with python (and potentially other languages), and develop good programming practices. The following are three key stages that we will consider:

1. Data preparation - this could involve handling missing or noisy data.
2. Data exploration - this could involve implementing algorithms which will perform clustering or classification on data sets.
3. Data communication - this could involve analysing and interpreting results, as well as ways of communicating these results to others.

Dr John Evans will handle the 'lectures', while Matthew Doherty will handle the 'labs'.

1 hr 2hrs

Rough roadmap for this course

The course will have three parts:

- In Part 1, we will explore good practices, as well as tools such as git.
- Next, Part 2 will explore data in more detail and discuss how to prepare it for analysis.
- Part 3 will then discuss various algorithms to perform this analysis, as well as how we can evaluate what we have done.

Throughout this module, the emphasis will be in all parts to apply Python (and other languages, as required) to accomplish our goals.

More detailed roadmap for this course

Each week there will be one 1-hour ‘lecture’ and one 2-hour ‘lab’. This is to be treated fairly roughly as we will sometimes spend more time on the ‘lecture’, and other weeks will spend more time on the ‘lab’.

More detailed roadmap for this course

Each week there will be one 1-hour 'lecture' and one 2-hour 'lab'. This is to be treated fairly roughly as we will sometimes spend more time on the 'lecture', and other weeks will spend more time on the 'lab'.

- Week 1
  - Lecture - Coding standards and Git
  - Lab - Getting started with Python.
- Week 2
  - Lecture - Input/Output
  - Lab - Python basics
- Week 3
  - Lecture - Data visualisation
  - Lab - NumPy

Road map continued

- Week 4
  - Lecture - Summary statistics
  - Lab - pandas
- Week 5
  - Lecture - Data exploration
  - Lab - Data visualisation
- Week 6
  - Lecture - Data manipulation
  - Lab - Data pre-processing
- Week 7
  - Lecture - Data modelling and fitting
  - Lab - Time series

Road map completed

- Week 8
  - Lecture - Regression
  - Lab - Regression
- Week 9
  - Lecture - Clustering
  - Lab - Classification
- Week 10
  - Lecture - Clustering continued
  - Lab - Clustering
- Week 11
  - Lecture - Summary
  - Lab - Summary

Assessment

The assessment will have four components[1]:

1. On-going coding quality assessment (10%). Coding quality will be assessed according to PEP8 standards or equivalent. This will also include version control.;
2. 1 page report (Due Week 8) - 30%;
3. 1-page report (Due Week 10) - 30%;
4. 1 page report (Due Week 11, presented Week 12) - 30%.

[1]Exact dates may change.

Further Reading

- Coding standards
  - PEP 8 – Style Guide for Python Code, by van Rossum et al. (https://peps.python.org/pep-0008/)
- Python
  - Python Data Science Handbook, VanderPlas.
  - Python for Data Analysis: Data Wrangling with Pandas, Numpy and iPython, by McKinney.
- General Data Science
  - Introduction to Data Mining, by Tan, Steinbach, Kumar & Karpatne.
  - Data Mining: Concepts and Techniques, by Han & Kamber.
  - Mathematics for Machine Learning, by Deisenroth, Faisal & Ong.

PEP-8 coding standards

- Writing ‘clean’ and consistent code is just good practice. Not only will it help keep track of the programmes that we write, but it will also help others understand our code as well.
- This latter point is especially important when communicating with others and for when we work on projects within a team.

[2]PEP-8: `https://www.python.org/dev/peps/pep-0008` (van Rossum et al.)

PEP-8 coding standards

- Writing 'clean' and consistent code is just good practice. Not only will it help keep track of the programmes that we write, but it will also help others understand our code as well.
- This latter point is especially important when communicating with others and for when we work on projects within a team.
- Throughout this module the quality of coding will be assessed and we will follow the 'PEP-8' recommendations set out in document 8 of the Python Enhancement Proposals (PEP), concerning the 'Style Guide for Python Code'.
- In principle, if we have followed these guidelines, then we should be able to understand the overall structure of anyone else's Python code, as long as they have also followed these guidelines.

Exercise: Read the full PEP-8 document[2] and take on board its recommendations in all work during this module (and others).

[2]PEP-8: `https://www.python.org/dev/peps/pep-0008` (van Rossum et al.)

Differences between OS and Kernel

Differences between OS and Kernel	
OS	Kernel
OS is a system software	Kernel is a system software which is part of the OS
OS provides interface between user and hardware	Kernel provides interface between applications and hardware
OS provides protection and security	Main purpose is memory management, process management and task management
All systems need an OS to run	All OS need a kernel to run
It is the first program to load when a computer boots up	It is the first program to load when an OS loads

General ethod of writing code

- The Python style guide was written with the understanding that code is read more often than it is written.
- As a rough rule, we write code once and then start reading as we start the debugging process (more on this shortly).

General ethod of writing code

- The Python style guide was written with the understanding that code is read more often than it is written.
- As a rough rule, we write code once and then start reading as we start the debugging process (more on this shortly).
- When we want to add features to a programme, we will again spend more time reading the code so that we ensure our additions are well-placed and run correctly.
- Then, once we are happy with our programme, we share it with others and then other programmers will again read the code.

General ethod of writing code

- The Python style guide was written with the understanding that code is read more often than it is written.
- As a rough rule, we write code once and then start reading as we start the debugging process (more on this shortly).
- When we want to add features to a programme, we will again spend more time reading the code so that we ensure our additions are well-placed and run correctly.
- Then, once we are happy with our programme, we share it with others and then other programmers will again read the code.
- As such, Python programmers will almost always encourage the writing of code that is easier to read, even at the expense of code that is easier to write. With this in mind, on the following slides are some guidelines that will help write clear code from the start.

Code layout

One of the attractive properties of Python is the fact that its syntax is intuitive to read. This ‘readibility’ is enhanced by ensuring we lay out our code in a consistent manner, following the recommendations of PEP-8.

In Python, **indentation** is used to define blocks of code, for example:

- In a loop
- Following a conditional statement or a series of logic blocks
- Function and class definitions

Code layout

One of the attractive properties of Python is the fact that its syntax is intuitive to read. This ‘readibility’ is enhanced by ensuring we lay out our code in a consistent manner, following the recommendations of PEP-8.

In Python, **indentation** is used to define blocks of code, for example:

- In a loop
- Following a conditional statement or a series of logic blocks
- Function and class definitions

Typically indentation happens from the line following a colon `:` until the end of the block.

- We can indent text using tabs or spaces but the PEP-8 recommendation is to **use spaces** for indentation, with **four** spaces per indent level. This improves readability while leaving room for multiple levels of indentation on each line.

Example

Loops:

```
for i in iterator:
~~~~print(i)    # where ~ represents a single space
~~~~for j in another_iterator:
~~~~~~~~print(j)
```

Note that we have illustrated the position and number of space with $\sim$.

Spaces and tabs

- While spaces are the preferred method of indentation, it is also possible to use tabs.
- While tabs work well for word processing documents, the Python interpreter will get confused when tabs and spaces are mixed.
- As such, if we are presented with a piece of code that has been tab-indented, then we should continue using tab indentation to remain consistent.

Spaces and tabs

- While spaces are the preferred method of indentation, it is also possible to use tabs.
- While tabs work well for word processing documents, the Python interpreter will get confused when tabs and spaces are mixed.
- As such, if we are presented with a piece of code that has been tab-indented, then we should continue using tab indentation to remain consistent.
- Nevertheless, in many cases, the text editor being used will provide a setting that lets us use the TAB key but then converts each tab to a set number of spaces. For example, in Spyder v. 4.0 or higher we proceed as follows:

```
Tools » Preferences » Editor » Source code » Indentation characters
```

Line length

- Many Python programmers recommend that each line should be less than 80 characters.
- Historically, this is because most computers could fit only 79 characters on a single line in a terminal window. While this is no longer the case in modern computers with larger screens, there are other reasons to still adhere to the 79-character standard line length.

Line length

- Many Python programmers recommend that each line should be less than 80 characters.
- Historically, this is because most computers could fit only 79 characters on a single line in a terminal window. While this is no longer the case in modern computers with larger screens, there are other reasons to still adhere to the 79-character standard line length.
- Professional programmers often have several files open on the same screen. Thus, using the standard line length allows programmers to see entire lines in two or three files that are open side by side on-screen.
- PEP-8 also recommends that we limit all comments (see later) to 72 characters per line because some of the tools that generate automatic documentation for larger projects add formatting characters at the beginning of each commented line.

Line length continued

- While the PEP-8 guidelines for line length are not set in stone (some teams prefer a 99-character limit, for example), it is important to keep these guidelines in mind, even if the code itself is of primary concern in the beginning. In doing so, it will become a more natural part of the programming process.

Line length continued

- While the PEP-8 guidelines for line length are not set in stone (some teams prefer a 99-character limit, for example), it is important to keep these guidelines in mind, even if the code itself is of primary concern in the beginning. In doing so, it will become a more natural part of the programming process.
- Most editors can set up a visual cue (usually a vertical line on the screen) that shows where the character limits are. For example, in Spyder v. 4.0 or higher we proceed as follows:

`Tools » Preferences » Completion and lintin » Code style`

Continuation

- Limiting the line length in this way means we will have to use **continuation** lines. For example, suppose we are calling a function that has lots of arguments:

```
result = some_function(arg1, arg2, arg3, arg4, arg5)
```

Continuation

- Limiting the line length in this way means we will have to use **continuation** lines. For example, suppose we are calling a function that has lots of arguments:

```
result = some_function(arg1, arg2, arg3, arg4, arg5)
```

- We can split these over multiple lines, as follows:

```
result = some_function(arg1, arg2,
                       arg3, arg4, arg5)
```

- Note how the start of the second line is **vertically aligned** with the character following the opening parenthesis.
- Another approach is to use **hanging indentation**, where a new line is started immediately after an opening parenthesis, i.e. there are no arguments on the first line:

```
result = some_function(
    arg1, arg2,
    arg3, arg4, arg5)
```

Hanging indentation and functions

- Care must therefore be taken when using hanging indentation in function definitions, such that we should distinguish arguments from the body of the function:

```
def my_function(
        arg1, arg2, arg3):
    return arg1+arg2+arg3
```

- We achieved this distinction by simply using an extra indent level for the arguments.

Hanging indentation and functions

- Care must therefore be taken when using hanging indentation in function definitions, such that we should distinguish arguments from the body of the function:

```
def my_function(
        arg1, arg2, arg3):
    return arg1+arg2+arg3
```

- We achieved this distinction by simply using an extra indent level for the arguments.
- A subtlety is when the condition following an `if` statement requires line continuation, since the set of characters '`if (`' is itself equivalent to four spaces.
- It is recommended to either add an extra indentation to the continuation of the conditional statement, or to add commenting.

Example

```
if (condition1 and
        condition2):
    do_this()

#or

if (condition1 and
    condition2):
    #A comment to break up the code block
    do_this()
```

Example

```
if (condition1 and
        condition2):
    do_this()

#or

if (condition1 and
    condition2):
    #A comment to break up the code block
    do_this()
```

- Note that the latter is generally only sensible if our editor is performing syntax highlighting.

Example

- If we want to put our closing parenthesis (or brace or square bracket) on its own continuation line, then it should either align with the first non-whitespace character of the previous line, or align with the first character of the block of code that starts the multiline construct:

Example

- If we want to put our closing parenthesis (or brace or square bracket) on its own continuation line, then it should either align with the first non-whitespace character of the previous line, or align with the first character of the block of code that starts the multiline construct:

```
a_list = [
    'a', 'b', 'c',
    'e', 'f'
    ]

#or

a_list = [
    'a', 'b', 'c',
    'e', 'f'
]
```

Continuation...continued...

- Any code within parentheses, square brackets or braces that is split over multiple lines, like in the examples above, is implicitly continued.
- In other words, Python knows that, despite the break in the line, the code in the continuation lines are part of the same statement.

Continuation...continued...

- Any code within parentheses, square brackets or braces that is split over multiple lines, like in the examples above, is implicitly continued.
- In other words, Python knows that, despite the break in the line, the code in the continuation lines are part of the same statement.
- If we want to use line breaks for continuation, then putting the relevant bit of code in parentheses is the preferred PEP-8 method. However, there is another way, and that is using a backslash:

```
while something_is_true and \
        something_else_is_false:
    do_this()
```

- Note how in this example we used two levels of indentation on the continuation line to distinguish the conditions from the actions of the while loop.

Continuation and binary operators

- Finally, when it comes to continuation lines that include binary operators (e.g. ‘+’) we should ensure that the line breaks *before* the operator, for example:

```
some_result = (a + b + c + d)

#can be split
some_result = (a
               + b
               + c
               + d)
```

- This improves mathematical readability.

Continuation and binary operators

- Finally, when it comes to continuation lines that include binary operators (e.g. ‘+’) we should ensure that the line breaks *before* the operator, for example:

```
some_result = (a + b + c + d)

#can be split
some_result = (a
               + b
               + c
               + d)
```

- This improves mathematical readability.
- There is a word of caution here. We should always use blank lines sparingly in our code. Blank lines should be used to distinguish logically coherent sections, when defining functions and classes (put two blank lines before and after the definition) and within a class (separate methods by a single blank line). However, if we overuse blank lines, then our code takes up too much space.

Python modules

- As our programming skills develop, we will start creating functions.
- These allow us to separate chunks of our code which can be called in a neat and, hopefully, intuitive fashion. They also allow us to make our main programmes more readable.

[3] Specifically, NumPy (`http://numpy.org`) and pandas (`http://pandas.pydata.org`).

University of Hertfordshire UH

Python modules

- As our programming skills develop, we will start creating functions.
- These allow us to separate chunks of our code which can be called in a neat and, hopefully, intuitive fashion. They also allow us to make our main programmes more readable.
- We can also go a step further and store multiple functions in a separate file called a *module*.
- This is simply a file ending in *.py* that contains code we want to import into our programme.
- We import this module using `import` and can even import specific functions from that module.[3]
- While these import commands can be placed anywhere, it is generally recommended to place the import commands at the top of our code. PEP-8 also recommends some further good practice considerations when it comes to imports.

[3]Specifically, NumPy (`http://numpy.org`) and pandas (`http://pandas.pydata.org`).

Importing modules

- Firstly, we should avoid ‘wildcard’ imports such as

```
from os import *
```

- Not only is this inefficient, but it also makes it confusing what names are in the namespace. Instead, it is better to import the entirety of the module:

```
import os
```

Importing modules

- Firstly, we should avoid ‘wildcard’ imports such as

```
from os import *
```

- Not only is this inefficient, but it also makes it confusing what names are in the namespace. Instead, it is better to import the entirety of the module:

```
import os
```

- Second, place imports on separate lines:

```
import os
import sys
```

- If we did want to ignore the first recommendation, then we can import multiple names from a module on the same line:

```
from numpy import pi, sin, cos, tan
```

Grouping imports

- Third, we should group imports together in the order: standard library, third party, local/custom modules. We then separate the groups by a blank line:

```
import os
import sys

import numpy

import my_class
```

Aliases

- If the name of a function we are importing might conflict with an existing name in our programme, or if a function/module name is long, we can use a short, unique *alias*. This is simply a nickname given to the function/module and is given when we import. Popular aliases for NumPy and pandas are np and pandas:

```
import numpy as np
import pandas as pd
```

Aliases

- If the name of a function we are importing might conflict with an existing name in our programme, or if a function/module name is long, we can use a short, unique *alias*. This is simply a nickname given to the function/module and is given when we import. Popular aliases for NumPy and pandas are np and pandas:

  ```
  import numpy as np
  import pandas as pd
  ```

- If we wanted to use a function from numpy, we could then use `np.function` and it would work as normal. We can also provide aliases to functions. The general syntax is as follows:

  ```
  from module_name import function_name as fn
  ```

Whitespace

- We should keep whitespace to a minimum. When done right, this should aid readability without becoming excessive. For example:

  ```
  a = [1, 2, 3]
  #not
  a = [ 1 , 2 , 3 ]
  ```

- Note how there is a single space after a comma.

Whitespace

- We should keep whitespace to a minimum. When done right, this should aid readability without becoming excessive. For example:

```
a = [1, 2, 3]
#not
a = [ 1 , 2 , 3 ]
```

- Note how there is a single space after a comma. The same goes for colons:

```
if condition:  do_this()
#not
if condition :  do_this()
```

Whitespace

- We should keep whitespace to a minimum. When done right, this should aid readability without becoming excessive. For example:

```
a = [1, 2, 3]
#not
a = [ 1 , 2 , 3 ]
```

- Note how there is a single space after a comma. The same goes for colons:

```
if condition:  do_this()
#not
if condition :  do_this()
```

- In other words, whitespace should follow, not precede, a colon. The exception is when using colons in slicing:

```
my_sub_list = my_list[2:5]
```

- We should also avoid any trailing whitespace.

Whitespace and binary operators

- Whitespace is particularly useful when using binary operators:

```
z = x * y
#not
z=x*y
```

Whitespace and binary operators

- Whitespace is particularly useful when using binary operators:

```
z = x * y
#not
z=x*y
```

- However, when using = signs for keyword arguments in functions, we should not use padded whitespace:

```
result = my_function(arg1=1, arg2=2)
#not
result = my_function(arg1 = 1, arg2 = 2)
```

Naming conventions

- In our code we will name several variables, functions, classes and whole modules. How we name these things is important and, as a general rule, they should be intuitive (not random letters and numbers), without being overly lengthy.
- For example, `rand_num_gen` instead of `random_number_generator`, or `velocity` instead of `v`.

[4]When using single characters, avoid letters like `l` or `o` or `I` that can be confused.

Naming conventions

- In our code we will name several variables, functions, classes and whole modules. How we name these things is important and, as a general rule, they should be intuitive (not random letters and numbers), without being overly lengthy.
- For example, `rand_num_gen` instead of `random_number_generator`, or `velocity` instead of `v`.
- The most common naming styles are as follows:
  - `a` (single lowercase letter)[4]
  - `A` (single uppercase letter)
  - `lowercase`
  - `lowercase_with_underscores` (my preferred)
  - `UPPERCASE`
  - `UPPERCASE_WITH_UNDERSCORES`
  - `CamelCase`
  - `mixedCase` (note initial lowercase character)
  - `Capitalised_With_Underscores`

[4]When using single characters, avoid letters like `l` or `o` or `I` that can be confused.

Naming conventions

- Within a project, we should aim to maintain a consistent style using one of the above.
- The exceptions are module names, which should be short, all lowercase words, as should function names. Class names should use CamelCase as is Python convention.

Comments

- Comments are an essential part of our code and should be kept up to date as the code evolves.
- As our programmes become longer and more complicated, it is imperative that we describe our overall approach to the problem we are solving. This is not just for other programmers, but also for ourselves as it is likely we will forget what we had in mind when we revisit our code.
- We should write comments in English, using full sentences including capitalised first letters.
- When using a hash to start our comment, we should put a single space before the text.

Comments

- The main idea when writing comments is to explain what our code is supposed to do and how we are making it work.
- When determining whether or not to write a comment, it is often helpful to ask whether we had to consider several approaches before coming up with a reasonable way to make something work. If we did, then write a comment about our solution.
- It is also useful to keep in mind that it is always easier to go back and delete comments, than it is to write comments for a sparsely commented programme.

Comments

- The main idea when writing comments is to explain what our code is supposed to do and how we are making it work.
- When determining whether or not to write a comment, it is often helpful to ask whether we had to consider several approaches before coming up with a reasonable way to make something work. If we did, then write a comment about our solution.
- It is also useful to keep in mind that it is always easier to go back and delete comments, than it is to write comments for a sparsely commented programme.
- There are three main classes of comment. First, ‘inline’ comments:

```
print("Hello World")    # An inline comment
```

- Be careful! Inline comments can clutter the code, and should be used sparingly. We should make sure they are well-separated from the inline code.

Block comments

In general, block comments are more useful:

```
for i in range(100):
    # This is a block comment.
    # Notice how it is at the same
    # level of indentation as the code it refers to.
    print(i)
```

You could also use triple quotes for block comments:

```
for i in range(100):
    """This is a block comment.
    Notice how it is at the same
    level of indentation as the code it refers to.
    """
    print(i)
```

Note how the text starts immediately after the opening set of quotes, and the closing quote marks have their own line (unless the entire comment is on one line).

Documentation string

Finally, we have a special type of comment called a *documentation string*, or *docstring*. These are comments which should appear for all modules, functions and classes, and are picked up by Python's object `help()` function. We use triple quotes for these.

Documentation string

Finally, we have a special type of comment called a *documentation string*, or *docstring*. These are comments which should appear for all modules, functions and classes, and are picked up by Python's object `help()` function. We use triple quotes for these.

```
def print_my_name():
    """This function simply
    prints my name.
    """


    print("John")
```

If we define this function and then call `help(print_my_name)` we get the following docstring:

```
Help on function print_my_name in module __main__:
print_my_name()
    This function simply
    prints my name.
(END)
```

Headers

In addition to imports, at the top of our code we should also put the module docstring. This should precede the import statements themselves:

```
"""My module does
something really cool.
"""


import os
import math

import numpy as np
```

There is yet more information that we can add at the header of the script. We need to document things like who wrote the code, when, what version it is, and so on. We do this using special identifiers called **dunders**[5] which should appear after the imports.

[5]Dunder = double underscore.

Dunders

```
"""My module does
something really cool.
"""

import os
import math
import numpy as np

__author__ = "John Evans"
__copyright__ = "Copyright 2022, UH"
__credits__ = ["Guido van Rossum", "Jo Bloggs"]

__date__ = "2022-12-14"
__license__ = "GPL"
__version__ = "1.0.1"
__maintainer__ = "John Evans"
__email__ = "j.evans8@herts.ac.uk"
__status__ = "Development"
```

Debugging

- Debugging is the process of identifying and correcting errors in our code.
- We will inevitably have to do this, and debugging will actually teach us a lot about programming.
- As Python is an interpreted language, our code will normally fail at the first error the interpreter encounters.
- The simplest way of debugging errors is using the Python **traceback**. This is the name given to the report representing the function call or expression that raised an error. In other words, Python will have encountered a problem and is trying to help us figure out what that problem is.

Example

To demonstrate, we can open up Python in interactive mode and try to print the value of a variable we have not yet defined:

```
Python 3.6.2 (default, Aug 10 2017, 13:39:08)
>>>print(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
>>>
```

- The traceback report clearly tells us what has gone wrong: a `NameError` has been raised, with a message that `a` has not been defined.
- In addition, we are told where in the code the offending call happened (in this case the equivalent of line 1 on the 'standard input').
- In coding, the standard input (e.g. an active terminal) is called `stdin`. Standard output is called `stdout` (also the active terminal). These can be accessed via read and write operation in a similar manner to files.

Example

```
def example(x):
    return x + z

example(1)
```

Example

```
def example(x):
    return x + z

example(1)
```

If we execute this code, we get a traceback:

```
Traceback (most recent call last):
  File "test.py", line 5, in <module>
    example(1)
  File "test.py", line 3, in example
    return x + z
NameError: name 'z' is not defined
```

- This time the bug is in the function – we used an undefined variable `z` in the definition.
- Note how the traceback tells us first where the offending call happened, and the error that was thrown (when we called `example(1)` on line 5), and then where the bug actually is (on line 3, in the function `example`, rather than the general <module>).

Python Debugger (Pdb)

- It is important to really take note of what the traceback is telling us when our code fails (it is amazing how many novice Python coders ignore the message!).
- If there are multiple bugs in our code, we may solve one bug (the first one encountered) but keep getting traceback messages as we work our way through the other errors – again, that is why it is essential to pay attention to the contents of the report.

[6] Documented here: `https://docs.python.org/3/library/pdb.html`

Python Debugger (Pdb)

- It is important to really take note of what the traceback is telling us when our code fails (it is amazing how many novice Python coders ignore the message!).
- If there are multiple bugs in our code, we may solve one bug (the first one encountered) but keep getting traceback messages as we work our way through the other errors – again, that is why it is essential to pay attention to the contents of the report.
- For most errors, the simple process of inspecting the traceback and correcting the code accordingly should be enough.
- A more interactive approach to debugging is offered through the Python Debugger (Pdb)[6].
- Pdb allows us to interactively step through our code line by line, working through the execution flow.

[6] Documented here: `https://docs.python.org/3/library/pdb.html`

Summary

- We have seen the PEP-8 Coding Standards. This helps guide us to writing clean, readable code that is consistent among programmers. It covers things such as:
  - Code layout
  - Imports
  - Whitespace
  - Naming conventions
  - Comments
- We also saw ways we can debug our code.

Exercise: Read the notes on version control (pp. 17–23) by next week.