Data Science Laboratory 1 Lecture Notes

J.D.P. Evans (j.evans8@herts.ac.uk

January 21, 2023

Contents

Part I

Essential Tools

Chapter 1

Overview and Foundational Concepts

1.1 What will we be seeing?

This course aims to introduce you to key topics in data science, such as clustering, classification and regression. In particular, it aims to do so in a practical way which will encourage you to work with python (and potentially other languages) and develop good programming practices that will benefit you throughout your degree. There will be three key stages 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.

The module 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. A vague roadmap is seen in Table 1.1.

The assessment will take the following form:

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%

You will need to achieve at least 40% to pass this module. This is computed once all assessment components are combined.

Road map		
Week	Lecture (1 hour)	Lab (2 hours)
1	Coding standards and Git	Getting started
2	Input/Output	Python Basics
3	Data visualisation	NumPy
4	Statistics	Pandas
5	Data exploration	Data Visualisation
6	Data manipulation	Data Pre-processing
7	Data modelling and fitting	Time Series
8	Regression	Regression
9	Clustering	Classification
10	Clustering continued	Clustering
11	Summary	Summary
12	Presentations	

Table 1.1: A roadmap for the course.

1.2 Further Reading

The following is a list of texts that may be useful throughout the course. It will be periodically updated so do keep an eye on this.

- 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.

1.3 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[1], 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.

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

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, the following are some guidelines that will help write clear code from the start.

1.3.1 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. For example,

```
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 ∼.

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

[1]The PEP (Python Enhancement Proposal) documents are contributed by the Python development community and cover a range of topics, seeking to provide proposals for the improvement of Python in general.

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

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`

Another consideration is 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.

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.

N.B. 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`

In any case, 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)
```

Here, the arguments are indented by one level, i.e. four spaces. 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. Therefore it is recommended to either add an extra indentation to the continuation of the conditional statement, or to add commenting, e.g.

```
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. 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'
]
```

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. 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.[3] As ever, a balance needs to be struck and this is usually a matter of experience. As a simple example, if we have five lines of code that builds a list, and then another three lines that do something with this list, it is appropriate to place a blank line between the two sections. However, we should not place three or four blank lines between the sections.

1.3.2 Imports

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.[4] 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.

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
```

[3] Blank lines will not affect how our code runs, but they will affect the readability of our code. The Python interpreter uses horizontal indentation to interpret the meaning of our code, but it disregards vertical spacing.

[4] Two essential Python modules for data science are NumPy (`http://numpy.org`), short for *Numerical Python*, and pandas (`http://pandas.pydata.org`).

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
```

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
```

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
```

1.3.3 Whitespace

We should keep whitespace to a minimum so there are no superfluous spaces in expressions and statements. 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, i.e. we should end the line with a carriage return where a statement actually ends.

One place where whitespace is particularly handy is when using binary operators. In this case, these should be padded by (one space of) whitespace:

```
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)
```

1.3.4 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)[5]
- `A` (single uppercase letter)
- `lowercase`
- `lowercase_with_underscores` (my preferred)
- `UPPERCASE`
- `UPPERCASE_WITH_UNDERSCORES`
- `CamelCase`[6]
- `mixedCase` (note initial lowercase character)

[5] When using single characters, avoid letters like `l` or `o` or `I` that can be confused.
[6] CamelCase because of the bumpy nature of the words, like camels' humps.

- `Capitalised_With_Underscores`

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.

1.3.5 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.[7] When using a hash to start our comment, we should put a single space before the text.

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. 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)
```

We 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)
```

[7] Use capitalisation at the start of a comment sentence unless it is a variable or name that starts with a lower-case letter.

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).

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. For example:

```
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)
```

N.B. Refer to PEP-257 'Docstring Conventions' for more guidance on docstring good practice.

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 – information that for all intents and purposes is like a comment since it encodes useful information but does not really participate in program flow. 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**[8] which should appear after the imports.

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

[8]Dunder = double underscore.

```
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"
```

These identifiers, denoted by leading and trailing double underscores, are essentially metadata for the module. If we save the example above as a module `my_module.py` and then import that module, we could call `help()` on it:

```
import my_module

help(my_module)
```

This would yield the following message:

```
Help on module my_module:

NAME
    my_module

FILE
    /Users/jdpev/Scripts/my_module.py

DESCRIPTION
    My module does
    something really cool.

DATA
    __author__ = 'John Evans'
    __copyright__ = 'Copyright 2022, UH"
    __credits__ = ['Guido van Rossum', 'Jo Bloggs']
    __date__ = '2022-12-14'
    __email__ = 'j.evans8@herts.ac.uk'
    __license__ = 'GPL'
```

```
    __maintainer__ = 'John Evans'
    __status__ = 'Development'
    __version__ = '1.0.1'
```

Here, GPL stands for GNU General Public License `https://www.gnu.org/licenses/gpl-3.0.en.html`. Note that we can see how the docstring and metadata is included. We can also access the metadata in program flow if we want:

```
import my_module

print(my_module.__version__)
#'1.0.1'
```

Hopefully it should be obvious that including such metadata is useful for the legacy of our code, for working within teams and professional development environments, and for version control, which we will explore in Section 1.5

1.4 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. 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')[9].

Now let us look at a slightly more complicated example. We have the following script:

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

[9]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(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 have 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¿). So we have 'traced back' through the code to where the bug is located.

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)[10]. Pdb allows us to interactively step through our code line by line, working through the execution flow. We can invoke Pdb when executing Python on the command line, e.g.

```
>python -m pdb my_script.py
```

If there is a bug in `my_script.py` that causes the execution to break abnormally, then Pdb will go into immediate 'post mortem' mode. We can use the Pdb commands to work back and forth through execution flow. For example, taking the buggy script example above:

```
>python3 -m pdb test.py
> /Users/jdpev/Scripts/test.py(2)<module>()
-> def example(x):
(Pdb) s
> /Users/jdpev/Scripts/test.py(5)<module>()
-> example(1)
(Pdb) s
--Call--
> /Users/jdpev/Scripts/test.py(2)example()
-> def example(x):
(Pdb) s
> /Users/jdpev/Scripts/test.py(3)example()
-> return x + z
```

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

```
(Pdb) s
NameError: name 'z' is not defined
> /Users/jdpev/Scripts/test.py(3)example()
-> return x + z
```

Here we execute the script using the `-m pdb` switch. We enter the interactive Pdb environment. We use the command '`s`' to (s)tep through the execution sequence, seeing actually what is happening at each step before we meet the error. This is a very simple example, but in more complicated scripts bugs may be very well hidden, or at least obscure, and the traceback might not give you enough information. The control offered by Pdb may help us here, and it is important to be aware of the different control features available when using this facility. When using the Spyder IDE, there is an interactive version of Pdb built in to the IPython interactive console.

1.5 Version control

Nearly every piece of code development will involve updates and refinements that result in different versions of the code, and this is true of software development in general. A prime example is Python itself: we are now on version 3 of the Python release (actually, v3.11.1 at the time of writing). Version 3 was a major update of version 2, which was around for many years, and included several significant updates to the language. Having a systematic way of recording updates to code, keeping a repository of different versions, and – importantly – having the ability to go back to previous versions is called version control, which we will now explore.[11]

1.5.1 Principles of version control

The central concept of version control is the idea of a flow of development, and that this flow can be described by a graph that has a tree-like structure. We start with a piece of code, or **baseline**. This is **committed** to a **repository** (or 'repo'). The repo contains the files in a project *and* the record of changes that have occurred.

When a piece of code (suppose we have a single script for simplicity) is first committed, it adds to the **trunk** (or **master**) of the tree. So, we have a flow of different versions that progress along the trunk. However, a user can decide to create a **branch** in the trunk. Branching is used when one would like to perform sub-development or testing (for example, some new functionality) that is separate from the main development trunk without interfering with 'what works'. Branches can either continue as an independent pathway of development (effectively evolving away from the original trunk), discontinue (maybe the development didn't work, or turned out to be unnecessary), or can **merge** back into the main trunk. Note that it is possible (and perhaps usual) that a single repo representing a project might be updated by multiple users.

There are several version control systems available, but perhaps the most popular is called *Git*.

[11] Of course, version control does not just apply to code and software, but anything else where updates occur – for example, a document.

1.5.2 Git

Git is simply one of a number of version control systems. Git was invented by Linus Torvalds, the inventor of Linux. It provides a set of tools for creating repos and performing tasks like commits, merges, and so-on. The best way to see how it works in practice is with a simple walk-through.

First, we write a simple Hello World script, `hello.py`. This file resides somewhere on our file system, say in the directory:

`/Users/jdpev/Data_Science_Laboratory1`

This is going to be our repo, but it is not actually a repo yet – it is just a directory. To make it into a repo, we go into the directory and invoke git:

```
>cd /Users/jdpev/Data_Science_Laboratory1
>git init
Initialized empty Git repository in /Users/jdpev/Data_Science_Laboratory1/.
   git/
```

We have created the repo and a hidden sub-directory has been created called `.git`. This stores all the information about the project. The script `hello.py` is present in the top-level directory, but is not yet part of the project. We can ask the status of the repo:

```
>git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        hello.py

nothing added to commit but untracked files present (use "git add" to track)
```

The status says that `hello.py` is present, but not actually tracked in the repo (we are also told we are on the master branch). So, first we need to add the file to the control system:

```
>git add hello.py
```

What this does is add the file to a 'staging area' which contains all the files that are to be committed (or, 'added to a commit'). After this step we can perform the commit:

```
>git commit -m "Added hello.py"
[master (root-commit) a15c335] Added hello.py
 1 file changed, 1 insertion(+)
 create mode 100644 hello.py
```

Note the `-m "Added hello.py"` part of the commit statement: this is a text message that describes something about what we have done, which is added to the record. In this case,

we just want to say that we have added the script to the repo. Git then reports the success and some information about the commit. Now if we look at the status:

```
>git status
On branch master
nothing to commit, working directory clean
```

Congratulations, your code is now under version control. Next, we edit `hello.py` by altering the print statement to add an exclamation mark, 'Hello World!'. After saving this change, the status now returns the following:

```
>git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   hello.py

no changes added to commit (use "git add" and/or "git commit -a")
```

We are told that `hello.py` has been modified and that these changes have not been added to the commit. Since `hello.py` is already tracked, we can use the `-a` switch to add to the commit:

```
>git commit -a -m "Bang!"
[master 9dea7dc] Bang!
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Okay, now we are going to add a branch. To do so, we run a **checkout** command with a switch to create a new branch called `new_branch`:

```
>git checkout -b new_branch
Switched to a new branch 'new_branch'
>git status
On branch new_branch
nothing to commit, working directory clean
```

The status tells us that we are now on the new branch we have created. When we checkout a branch we update the files in the working directory to match those on the branch, and any new commits while checked out are recorded on the branch. We will edit `hello.py` to change the language to Esperanto 'Saluton Mondo!'. After doing that we must add to the commit on the branch:

```
>git commit -a -m "Esperanto update"
[new_branch 8667d86] Esperanto update
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Note that we can see what branches exist, and what branch we are currently on (*):

```
>git branch
  master
* new_branch
```

So now there are two versions of the code, one on the main branch which is our original Hello World, and one on the new branch which is our Esperanto translation. We can switch back to the master branch:

```
>git checkout master
Switched to branch 'master'
```

If we inspect `hello.py` we will see the original 'Hello World!'. If we continue working on this file and add another line that prints 'Hello Again!', then we can once again add to the commit:

```
>git commit -a -m "New line"
[master f8fd493] New line
 1 file changed, 1 insertion(+)
```

So we have continued the development of the master branch.

Question: What happens if we try to merge the branches?

While checked out in the master branch, we can merge the new branch back in:

```
>git merge new_branch
Auto-merging hello.py
CONFLICT (content): Merge conflict in hello.py
Automatic merge failed; fix conflicts and then commit the result.
```

Git tries to automatically merge the two versions of `hello.py` but the merge failed. We have a **conflict**. This is where git encounters changes to the same part of the same file in the two versions and does not know which to use. It is left to the user to resolve the conflict and then commit the result. If we now examine `hello.py` we can see that git has annotated the conflict:

```
<<<<<<< HEAD
print("Hello World!")
print("Hello Again")
=======
print("Saluton Mondo!")
>>>>>>> new_branch
```

Everything before `=======` represents code in the receiving branch, and everything after represents the merging branch. We could resolve this issue by replacing the first `print("Hello World!")` by `print("Saluton Mondo!")`. If we do that and try to commit in the usual way, we are successful.

As another example, we create another branch and this time add a different file that contains a simple function definition. First, create the branch and checkout:

```
>git branch another_branch
>git checkout another_branch
```

If we create a file in this branch, `my_function.py`, remember that we first have to add it to the index:

```
>git add my_function.py
>git commit -m 'new function'
[another_branch 92f656b] new function
 1 file changed, 2 insertions(+)
 create mode 100644 my_function.py
```

Suppose we developed this code a bit, then wanted to merge it back into the master. We first checkout the master, then attempt to merge:

```
>git checkout master
Switched to branch 'master'
>git merge another_branch
Updating c395682..92f656b
Fast-forward
 my_function.py | 2 ++
 1 file changed, 2 insertions(+)
 create mode 100644 my_function.py
```

That worked just fine because there were no conflicts and `another_branch`'s commit history is merged with the master.[12]

So far we have been working on a local repo – one that only exists on our own machine. Next we see how we can use a remote git repo for developing projects.

1.5.3 Github

Github (`https://github.com`) is an online platform for hosting git repositories. Anyone can register for an account with a username and password for free. Once we have an account, and create a repo using the web interface, we can interact with it remotely.

Suppose we have just created a repo on github and want to commit to it from the command line. We start from scratch and initialise a local git repo, where we have written a simple README, which we will add and commit:

```
>git init
Initialized empty Git repository in /Users/jdpev/example/.git/
>git add README.md
>git commit -m "first commit"
[master (root-commit) 809e0a9] first commit
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 README.md
```

[12] A 'fast-forward' merge is one where there is a simple linear path from the current branch to the target branch – in this example we just added a new file.

Now we will add it to our remote repo:

```
>git remote add origin https://github.com/jdpev/example.git
```

Here, `origin` is a shortcut name for the remote repo described by the URL. Having added the remote, we can **push** to it:

```
>git push -u origin master
Username for 'https://github.com': jdpe
Password for 'https://j.evans8@github.com':
Counting objects: 3, done.
Writing objects: 100% (3/3), 207 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/jdpev/example.git
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
```

We have to enter our login credentials for github and then the objects are uploaded. If we now go to the github browser interface, we will see our README. If we want local changes to be recorded in the remote repo, we need to do a push after each commit.

The opposite of push is **pull**. We can make a pull request from a remote repo, and what this provides is a way for others to see what changes have been made to a project. Suppose someone else added a file `new.txt` to the github repo. We would do the following pull request to get those changes:

```
>git pull origin master
From https://github.com/jdpev/example
 * branch            master     -> FETCH_HEAD
Updating d731603..8d9aaa0
Fast-forward
 new.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 new.txt
```

We will now see `new.txt` locally. Note that if we just wanted to see what has changed in the remote repo, we use **fetch** instead of pull, which just tells us what has changed without updating the file system (and is therefore 'harmless'). To read what has changed, we can do the following:[13]

```
>git log -p ..FETCH_HEAD
```

This reports what has changed.

Another thing we might want to do is make a copy, or **clone** of a particular repo. Taking the example we just made, we can do the following:

```
>git clone https://github.com/jdpe/example.git
Cloning into 'example'...
```

[13]Git pull actually just combines a fetch with a merge.

```
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 5 (delta 0), reused 5 (delta 0), pack-reused 0
Unpacking objects: 100% (5/5), done.
Checking connectivity... done.
```

This creates a local directory called `example`, which contains a clone of the repo stored at the corresponding github URL. This is useful if someone has written some nice software that we would like to make use of (and interrogate the source code) and is maintaining it on github.

In the examples above, we have used git from the command line, which is a great way to understand what is going on at a nuts-and-bolts level. There exist GUI-based 'desktop' applications for working with git repositories that might be useful, for example GitHub Desktop: `https://desktop.github.com`.

Week 1 practical challenges

1. Get familiar with Anaconda and Spyder for writing code.
    - Start by creating a simple .py file called `hello_world.py` which prints "Hello, world!".
    - Try using variables in `hello_world.py`.
    - Create a .py file called `name.py` which prints out a name, then prints out that same name in uppercase, lowercase and with just the first letters capitalised. Try doing this with Python methods, rather than manually.
    - For the same file as in the previous challenge, find out what `rstrip()` does. What about `\t` and `\n`?
    - Create a .py file called `numbers` which adds, subtracts, multiplies and divides several integers and floats of your choosing.
    - The Python community's philosophy is contained in 'The Zen of Python' by Tim Peters. You can access this brief set of principles for writing good Python code by entering `import this` into your interpreter. Do this and try to understand what each line is saying.
2. Create your own GitHub account and repo.
    - Create a local repo containing a simple script that calculates the Fibonacci sequence (`https://en.wikipedia.org/wiki/Fibonacci_number`). Practice committing and pushing changes to your GitHub repo (you are recommended to practice using the command line, but also feel free to use a GUI interface). Remember to use messages when you commit so you can keep track of what you have done.
3. Make sure your code is PEP-8 compliant, including proper comments and header detail.

Chapter 2

Input and Output

The fundamental component of data science is, unsurprisingly, data. In general, we have some data stored in a file (or other resource) and we want to load this data into a Python script so that we can clean up the data, manipulate the data, analyse the data, and so on. We then need to store the result. For this reason, reading and writing data (input and output, or I/O) is key to being able to work with data science.

2.1 Binary and ASCII data

Binary data is essentially any data that is not text. Data represented in a file points to the memory address on the system where the data actually resides, byte by byte. For this reason, using binary formats are more efficient (though less user friendly): they take a smaller memory footprint and have the benefit that to access a particular part of the data does not require reading the entire file. The NumPy[1] `save` and `load` helper functions will write and read NumPy objects such as arrays in binary format for us. Without worrying too much about NumPy just yet, we demonstrate how these work.

In NumPy, arrays[2] are much like Python's in-built `list` type which will be seen in Lab 1, however they provide much more efficient storage and data operations once the arrays grow large. In general, NumPy arrays form the core of almost the entire ecosystem of data science tools in Python, so it is important that we understand them. One way to create an aray is to use NumPy's `arange` function. This creates a linear sequence of numbers. For example, if we wrote `np.arange(0, 20, 2)` then we would start at 0, end at 20 with step-size 2 (note the `np.` which just indicates this comes from NumPy which we have imported with alias `np`). If instead we wrote `np.arange(10)`, then we would produce a list with 10 elements, starting at 0 and with step-size 1.

```
import numpy as np

arr = np.arange(10)    # create our array
```

[1] Coming in Lab 2!

[2] Recall, an array is just a series of rows and columns of numbers (or other objects).

```
np.save('some_array', arr)    # save our array as the string 'some_array'
```

If the file path does not end with *.npy* then the extension will be appended.

To read the data, we need to unpack it. We can do this with NumPy's `load` function

```
np.load('some_array.npy')
```

If we want to save multiple arrays in an uncompressed archive, we use `np.savez` and then pass arrays as keyword arguments:

```
np.savez('array_archive.npz',a=arr, b=arr)
```

To load a *.npz* file, we again use `np.load`. Finally, if we want to compress our data as well, then we use `np.savez_compressed`.

Moving on, ASCII stands for American Standard Code for Information Interchange. It is an established encoding scheme for representing alphanumeric characters on computers. For all intents and purposes, data stored as human-readable text or numbers can be classed as ASCII data, but note that in practice ASCII represents a special encoding on the computer. Actually, most text is now encoded using UTF-8.[3]

In Python, to read a file containing text we can simply open a file object

```
# Use the open() built-in function. 'r' means 'read'
myfile = open('my_file.txt','r')
```

We have not actually read the data yet, simply created the object.[4] We can read all the data in one go:

```
data = myfile.read()
```

Normally, it is useful to read the data into a list where each item is a single line:

```
lines = myfile.readlines()
```

After we have read the data, we should always close the file:

```
myfile.close()
```

If we are reading in data this way, it is likely that we will be dealing with some form of columnar data. In order to get it into a usable format, we need to **parse** the data. At the moment each item in `lines` is a long string. So for example, if each line contained, say a date and an average temperature, the first item might look like

```
lines = myfile.readlines()
print(lines[0])
#2022-12-17 0.2
```

[3] `https://en.wikipedia.org/wiki/UTF-8`

[4] An important characteristic of Python is the consistency of its *object model*. Every number, strong, data structure, function, class, module, and so on exists in the Python interpreter in its own 'box', which is referred to as a *Python object*. Each object has an associated *type*, such as a *string* or *function*, and internal data. In practice, this makes the language very flexible, as even functions can be treated like any other object.

We can see that there is whitespace **delimiting** the two columns. We can use string operations to get the data into the format we want. First, the `split()` method will split a string into a list of components. We could unpack these into two variables called `thedate` and `temperature`

```
lines = myfile.readlines()
thedate, temperature = lines[0].split()
print(thedate, temperature)
```

By default it will split the string according to whitespace, but we could also supply an argument like `split(',')` that will split a string at every instance of (in this case) a comma. This is useful for when dealing with comma separated values (CSV). At the moment `thedate` and `temperature` are still strings, so we need to cast them into an appropriate data type. We can use the `datetime` module to turn the date into a `date` object, and we can make the temperature a float:

```
from datetime import date

thedate = date.fromisoformat(thedate)
temperature = float(temperature)
```

The `fromisoformat()` method takes a string of the form YYYY-MM-DD and turns it into a `date` object. What we would like to do is loop over all lines and create two sequences storing all the dates and temperatures. One way to do this is through list appending:

```
# Declare two empty lists
dates, temperatures =  [], []

# Loop over all lines
for line in lines:
    thedate, temperature = line.split()
    dates.append(date.fromisoformat(thedate))
    temperatures.append(float(temperature))
```

Now we have two sequences representing our two columns. We could cast them as NumPy arrays if we wanted to:

```
import numpy as np

dates = np.array(dates, dtype='datetime64')
temperatures = np.array(temperatures, dtype='float')
```

This allows us to now manipulate the data using NumPy functionality.

Question: What about writing data to a file?

In the following example we create some dummy data: a sequence of dates generated using the NumPy `arange` function, and a random number representing some 'data'. The idea is that we want to write this data as two columns to an output file:

```
import numpy as np

# Generate sequence of dates for all days in 2022
dates = np.arange('2022-01',
                  '2022-12',
                  dtype='datetime64[D]')

# Create output file for writing ('w')
outfile = open('out.txt', 'w')

# Loop over dates
for d in dates:
    #Generate random number
    v = np.random.uniform()
    # Use an f-string to write out a string to the file
    outfile.write(f'{d} {v:.5f}\n')

# Close the file
outfile.close()
```

The most important line in the code above is the write statement itself. It describes how the data is going to appear in the file. We use an 'f-string' which means 'formatted string'. Note the syntax: the string in the argument to the `write()` method is contained within quotation marks but preceded by '`f`'. Inside we have two sets of curly brackets. The first references `d`, which in the loop we have written is pointing to each date in the sequence. That date will be printed in the familiar YYYY-MM-DD format, so we do not need to do anything fancy to it. The next set of curly brackets contain a reference to `v` which is our dummy data – just a random number. But we want to format this in a specific way, and so we supply some more information. The `.5f` bit after the colon says that we want to write out the number to five decimal places. This makes for a nice, cleanly formatted output file. Note how we use whitespace between the columns just by putting a space, and finally a newline return at the end to make sure the next write goes on the following line.

There are other formatting options and it is left as an exercise to explore them. Another common formatting operation is 'zero padding'. Suppose that instead of a random float, we had an integer. If we want to make sure our data column containing that integer is of a fixed width we could do something like the following:

```
import numpy as np

# Generate sequence of dates for all days in 2020
dates = np.arange('2022-01',
                  '2022-12',
                  dtype='datetime64[D]')

# Create output file for writing ('w')
```

```
outfile = open('out.txt', 'w')

# Loop over dates
for d in dates:
    #Generate random integer between 0-1,000,000
    v = np.random.randint(low=0,high=1.e6)
    # Use an f-string to write out a string to the file
    outfile.write(f'{d} {v:9}\n')

# Close the file
outfile.close()
```

This would output something like

```
2022-01-01    329059
2022-01-02    535272
2022-01-03    863713
2022-01-04    429343
2022-01-05     18213
2022-01-06    414018
2022-01-07    720761
2022-01-08    666152
2022-01-09    921780
2022-01-10    159584
```

Note the alignment of the second column – the '9' in `v:9` specifies that the width of the string that represents `v` should be exactly nine characters wide. We could make this more obvious by zero-padding the number, with the small tweak to the format

```
outfile.write(f'{d} {v:09}\n')
```

And the output would look like

```
2022-01-01 000007366
2022-01-02 000213080
2022-01-03 000810772
2022-01-04 000544033
2022-01-05 000226956
2022-01-06 000286032
2022-01-07 000309208
2022-01-08 000591712
2022-01-09 000048268
2022-01-10 000196558
```

When it comes to reading the data in and casting as an integer, it does not matter that we have the leading zeros. Of course, we can make the format statement more complicated, building in more columns, having different delimiters, and so-on.

It is good practice to have some form of header information in the files we write. This can take the form of some 'comment'-like indicator, such as a hash symbol which we can ignore when the file is read. Modifying the read example above yields the following:

```
# Declare two empty lists
dates, temperatures =  [], []

# What symbol identifies a line to ignore in the input?
comment = '#'

# Loop over all lines
for line in lines:
    # Check if first character matches the comment symbol.
    # Read the data if it doesn't.
    if not line.startswith(comment):
        thedate, temperature = line.split()
        dates.append(date.fromisoformat(thedate))
        temperatures.append(float(temperature))
```

Knowing how to read and write data files in this way will give us maximum flexibility when it comes to interacting with data stored on a file system. Note however that there also exist helper functions in NumPy that allow us to achieve similar results. For example, if `my_data` represents a 2d array containing three rows of data that we want to save as three columns in CSV format then we could use the following:

```
np.savetxt('my_data.csv',my_data,delimiter=',',fmt=['%d','%.1f','%03d'])
```

Here we specify the output format of each column using the `fmt` keyword argument, in this case saying we want column one to be an integer, column two to be a float to one decimal place and column three to be a 3-zero padded integer. Similarly, we can load data, say columns 1, 3 and 5 of a CSV file with # comments:

```
d = np.loadtxt('my_data.csv', dtype='float', comments='#', delimiter=',',
    usecols=(0,2,4))
```

Noting that column numbers are zero-indexed. The resulting `d` array will have a shape corresponding to the number of rows and columns to be read. In the next section we will explore some more advanced usage of NumPyhy helper functions when reading structured data.

2.2 Structured and unstructured data

Most data in the world is unstructured – that is, it does not follow a single set of rules governing formatting. The text of a book is an example: its contents could represent anything, and we do not know in advance how many words it contains, what language it is in, and so-on. A website is another. Structured data on the other hand does follow a fixed format.

An example would be a table of data with a set number of columns and a description of what each column contains.

First let us see how we could deal with some unstructured data: the contents of the BBC news website. For this, we will use the requests module which provides methods to 'get' or 'post' data via HTTP[5]. We can use requests[6] to interact with web content.

```
import requests

# We use requests to perform a get request on the URL.
r = requests.get('https://www.bbc.co.uk/news')

# The actual data is the content attribute.
# We want to decode it as UTF-8 formatted text
# representing the HTML of the page.
# We can use the text() method to do this,
# equivalent to data = (r.content).decode('utf-8')
data = r.text

# Define a counter.
n_coronavirus = 0

# Split the data based on whitespace and loop over it.
for d in data.split():
    # Look for instances of 'coronavirus' or 'Covid-19'
    # strings in each d.
    if ('coronavirus' in d) or ('Covid-19' in d):
            # Add to counter.
            n_coronavirus += 1

# Report using an f-string
print(f'There are {n_coronavirus} mentions of coronavirus or Covid-19 today.
    ')
```

In this simple example we got the HTML of the current version of `https://www.bbc.co.uk/news` as text. Then, after splitting the contents based on whitespace, we simply looped over each item in the HTML and checked for instances of mentions of 'coronavirus' or 'Covid-19', adding to the counter when we found one. Finally, the information is reported back.

Now let us see an example of dealing with structured data on the web. We will use the record of weather data measured at the Heathrow station, accessed via the MET office. This is just a simple record of key meteorological data on a month-by-month basis since 1948, represented by a table (the structure of which we can inspect): `https://www.metoffice.gov.uk/pub/data/weather/uk/climate/stationdata/heathrowdata.txt`. If we actually look at that URL in a browser, we will see a table of text, with columns for year, month,

[5]HTTP = Hypertext Transfer Protocol

[6]See also, the urllib modules.

temperature etc., along with a short header describing the contents. We now write a script that pulls out the average rainfall for the month of August, and compares it to the latest value:

```
import requests

import numpy as np

# Define URL.
heathrow = 'https://www.metoffice.gov.uk/pub/data/weather/uk/climate/
   stationdata/heathrowdata.txt'

# Use requests.
r = requests.get(heathrow)

# Get the text.
data = r.text

# Split it into lines based on newline character.
lines = data.split('\n')

# List to store rainfall.
mm_of_rain = []

# Loop over lines - starting at index 7
# since there are 7 header lines to ignore
for line in lines[7:]:
        # Split the line into columns.
        entries = line.split()
        # Column 2 is the month ID - we want August = 8.
        if entries[1]=='8':
                # The 6th column is the rainfall
                # which we cast as a float and append.
                mm_of_rain.append(float(entries[5]))

# Turn it into an array.
mm_of_rain = np.array(mm_of_rain)

# Calculate the mean
average_amount_of_rain = np.mean(mm_of_rain)

# The latest value is at the [-1] index, which we compare to the mean
# Normalised by the standard deviation.
significance = (mm_of_rain[-1]-average_amount_of_rain) / np.std(mm_of_rain)
```

```
# Print the information to the screen
print(f'On average we expect {average_amount_of_rain:.1f} mm of rain in
    August at Heathrow.')
print(f'This August {mm_of_rain[-1]:.1f} mm of rain were recorded,
    equivalent to {significance:.2f} standard deviations from the mean.')
```

We have now seen how to use the requests module for scraping data from the Web, which is obviously a rich source of data in general. The general principles of parsing and manipulating the structured and unstructured data are just as applicable if the data is read from a local file system.

2.3 Images

Having covered binary and text data, the other main data type we will encounter is imagery. Images are stored in various formats: Portable Network Graphics (PNG), Tagged Image File Format (TIF or TIFF), Joint Photographic Experts Group (JPEG) and Graphics Interchange Format (GIF) being the most common. Each format has its advantages and disadvantages. For example, JPEGs do not take up too much memory, but they use a compression algorithm which makes them 'lossy' – i.e. some of the information is thrown away when the file is written. This is why JPEGs often seem blocky or pixellated. TIFFs on the other hand can be lossless, which allows one to store high quality imaging at the expense of memory footprint. Which format we use really depends upon the application.

An image file generally contains the data itself in some binary format and a header describing what sort of format the data is in. When we read an image, it is generally **rasterized**, which means that the data is represented by 2D grid, representing pixels. It should be no surprise therefore that there is a synergy between images and NumPy arrays, and there are well-established Python tools for reading and writing images. A greyscale image can be represented by one 2D array. A colour image can be represented by three 2D arrays, representing the red, green and blue channels. Sometimes there is a fourth channel called alpha, which can handle image transparency. Not all imaging formats support transparency.[7] Let us now see how to interact with images in Python using the skimage package. Reading an image is simple. We take the example, `goodgirl.jpg`:

```
from skimage import io
import numpy as np

# Read an image
image = io.imread('goodgirl.jpg')

print(image.shape)
# (300, 200, 3)
```

[7]The Cyan Magenta Yellow Black or CMYK format is another system for representing colour images.

Note that `imread` automatically knows what format the image is in and applies the relevant decoding for us. It returns the data as a 3D Numpy array, with a shape (300, 200, 3). Note the ordering of the axes: the vertical direction is the *first* axis in the array. We could play around a bit, and just pull out the red channel, and then save that as a new image:

```
from skimage import io
import numpy as np

# Read an image
image = io.imread('goodgirl.jpg')

# Get all the pixels, but just the red channel (RGB)
red = image[:,:,0]

# Write it out again
io.imsave('goodgirl_red.jpg', red)
```

The result is viewed as a single channel greyscale representing the luminance in the red channel of the original image:

This functionality allows us to manipulate imaging data in the same way we would any NumPy array, which is quite powerful. For example, it now becomes trivial to flip, rotate or trim an image. Of course, if we have NumPy data that could be visualised by a raster map, we can create new images from scratch. We will later use skimage to explore more image processing.

Week 2 practical challenges

1. Read the data at `https://www.metoffice.gov.uk/pub/data/weather/uk/climate/datasets/Tmean/date/UK.txt` representing gridded mean air temperature data for the UK since 1884. Calculate:
   (a) The month and year of the most extreme temperatures on the record.
   (b) The temperature for a given month and year. Print this out upon request.
2. Parse the HTML of the 'Astronomy Picture of the Day' (APOD) website `https://apod.nasa.gov/apod/astropix.html` to automatically download the most recent picture. Crop out the central 256×256 pixels and save as a local JPEG.

Extension: explore the use of visualisation using matplotlib (or other package) to display the UK weather data linked above.

Remember: use GitHub to maintain your code!

Chapter 3

Data structures

3.1 Object orientation

Object orientation is a programming concept that essentially relates to the definition of 'classes' of object that have their own attributes and methods. The ability to define our own classes is exceptionally powerful, especially when it comes to describing complex data. Here we will quickly see how this works.

Suppose we want to define a general class to describe a minibeast. It might be a spider, an insect, a worm etc. In Python, we start by defining the class as follows:

```
class minibeast:
    """
    This is a class to represent minibeasts
    Define a minibeast by the number of legs and wings
    """

    #Set some default attributes
    number_of_legs = 0
    number_of_wings = 0

    def __init__(self,legs,wings):
        """Requires integer arguments for legs and wings"""

        #Check that we have integer legs and wings
        if not isinstance(legs,int):
            raise ValueError("Non-integer legs!")
        if not isinstance(wings,int):
            raise ValueError("Non-integer wings!")

        #Set attributes
        self.number_of_legs = legs
        self.number_of_wings = wings
```

```
    def can_it_fly(self):
        """Simple test to see if beast can fly"""
        #Check number of wings
        if self.number_of_wings>0:
            return True
        else:
            return False
```

Note the familiar colon and indent structure. The first line starts the class definition and we are calling the object `minibeast`. The comment below is called a docstring, containing information about the class – we will return to that later. Next, we set some attributes – these are data types of our own choosing that describe the object. Here, we just define the number of legs and wings and set these to be zero by default. Next is a function definition. Function definitions within a class are called 'methods' of the class. In particular, `__init__` is a special 'reserved method'. It is the method called when we create a new object from this class, otherwise known as a constructor. The first argument `self` refers to the instance of the class itself, and should always be present. Here we have added two more arguments to the constructor: `legs` and `wings`. These will become 'attributes' of the class. Note that we do some checking to test that the values passed are integers using the `isinstance(<object>,<type>)` built in function. If either `legs` or `wings` is not an integer, a `ValueError` is raised and message passed back to the user. If all is well, then we set two attributes called `number_of_legs` and `number_of_wings`. Note again the use of `self` here.

After the constructor we can add any number of other methods we want. Here we have added a method that tests whether the minibeast can fly, based on the number of wings. It has no arguments, apart from `self`, and the method returns a bool type accordingly. Of course, this is just a trivial example.

Now, we save this in a file called `bug.py`. We can then use it as a module and import it into the namespace. So, in a new script where we want to use our minibeast class, we would import it and then create a couple of objects:

```
from bug import minibeast

worm = minibeast(0,0)
moth = minibeast(6,2)

print(worm.number_of_legs)
#echos "0"
print(moth.can_it_fly())
#echos "True"
```

We have created two 'instances' of the class called `worm` and `moth`. Note that we do not actually specify the 'self' first argument to the constructor – that is implicit. However, we do supply the number of legs and wings. We can then access the attributes using `<object>.<attribute>` and call methods using `<object>.<method>(<args>)`, again noting that we do not explicitly supply the `self` argument to the method. We can see that by defining our own classes of object we have immense flexibility when it comes to describing

(and handling) complex data, and it allows us to write our code in a very modular way.

Finally, return to that 'docstring' idea; we encountered these in Chapter 1. The docstrings are the triple quoted comments that come immediately after the main class definition and the method definitions (i.e. not the # line comments). Comments can obviously be read by someone looking at the code, but the placement and format of comments within classes such as this can also be interrogated with the Python built-in function `help()`. This is useful mainly if we are working with the Python interpreter interactively. The idea is that we can get help on a Python object (e.g. `help(worm)`, such that we will be informed of the class structure, attributes and methods available – and our docstrings will be included. For example:

```
from bug import minibeast

worm = minibeast(0,0)
help(worm)
```

This will report the following:

```
Help on instance of minibeast in module __main__:

class minibeast
 |  This is a class to represent minibeasts
 |  Define a minibeast by the number of legs and wings
 |
 |  Methods defined here:
 |
 |  __init__(self, legs, wings)
 |      Set the number of legs and wings
 |
 |  can_it_fly(self)
 |      Simple test to see if beast can fly
 |
 |  ----------------------------------------------
 |  Data and other attributes defined here:
 |
 |  number_of_legs = 0
 |
 |  number_of_wings = 0
(END)
```

We can do this if we are unsure of the properties of a given object in Python, and to find out what methods and attributes are available, etc. This can be useful for debugging (ahem) our code. The concept of object orientation is a much broader subject than we can cover here, and there are many other principles that go beyond this very brief introduction. To learn more,look at https://realpython.com/python3-object-oriented-programming.

3.2 HDF5

HDF5 stands for Hierarchical Data Format 5. It is a binary data format that is designed for very large datasets (e.g. up to TBs). The key concept in HDF5 is that the data in the file is organised hierarchically, like a file system. Importantly, because it is a binary format, you can access parts of the data without reading the entire set. This is very useful when dealing with big data. To interact with the main HDF5 libraries, there is a Python module h5py. It is easy to create files and data sets with a nice interaction with NumPy:

```
import h5py
import numpy as np

# Open a file for writing.
f = h5py.File("test.hdf5" ,"w")

# Create a dataset called 'test' and give it some data direct from a Numpy
    object.
d = f.create_dataset("test", data = np.linspace(0,1,10))

#Close the file.
f.close()

# Re-open it in read mode
f = h5py.File('test.hdf5','r')

# Access the data:
print(f['test'])
print(f['test'].shape)
print(np.array(f['test']))

"""
<HDF5 dataset "test": shape (10,), type "<f8">
(10,)
[0.         0.11111111 0.22222222 0.33333333 0.44444444 0.55555556
 0.66666667 0.77777778 0.88888889 1.        ]
 """

f.close()
```

Note how we can use NumPy indexing tricks directly when we read the data:

```
my_data = np.array(f['test'][0:2])
print(my_data)
# [0.         0.11111111]
```

This is useful when dealing with large datasets. Datasets can be organised in an HDF5 file in 'groups' which operate a bit like pathnames. They allow you to structure your data in

meaningful and efficient ways. For example:

```
import h5py
import numpy as np

# New file.
f = h5py.File("test2.hdf5" ,"w")

# First create a group (note that it is called as a method of f)
g1 = f.create_group("Level1")

# Now create a sub-group (note that it is called as a method of g1)
g2 = g1.create_group("Level2")

# Now create a dataset in the sub-group (called as a method of g2)
g2.create_dataset("test", data = np.linspace(0,1,10))

# Close file.
f.close()

# Open for reading.
f = h5py.File('test2.hdf5','r')

# We can access dataset using the path hierarchy defined above.
print(f['Level1/Level2/test'])
#<HDF5 dataset "test": shape (10,), type "<f8">
# Close file.
f.close()
```

In this example we created the dataset `test` as a method of the group object `g2`. Note that we could also create the dataset as a method of `f` by specifying the full path:

```
d = f.create_dataset('Level1/Level2/test',
                     data = np.linspace(0,1,10))
```

Another important concept is that of attributes, which are like metadata for datasets and groups. They can be used to provide additional information that describe the data:

```
import h5py
import numpy as np

f = h5py.File("test2.hdf5" ,"w")
g1 = f.create_group("Level1")
g2 = g1.create_group("Level2")
g2.create_dataset("test", data = np.linspace(0,1,10))

# Create an attribute of the Level2 sub-group
```

```
g2.attrs['date'] = '2020-09-23'
f.close()

f = h5py.File('test2.hdf5','r')

# Retrieve attribute by name.
print(f['Level1/Level2'].attrs['date'])
f.close()
```

HDF5 datasets behave in a similar manner to NumPy arrays in many respects, but note that there is additional functionality in HDF5 when it comes to storing the data, such as compression and chunking – again, useful for very large datasets. Further reading about the full functionality of h5py for interfacing with HDF5 files can be found at `https://docs.h5py.org`.

3.3 Pandas

Pandas is a package that 'is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language.' The core concept of pandas is the idea of a **dataframe**, which is really just a representation of tabular data. In essence, a `DataFrame` object is a Numpy array with some additional information:

```
import numpy as np
import pandas

# Some data with 10 rows and 3 columns
some_data = np.zeros((10,3))

# Make it a data frame with column headers
df = pandas.DataFrame(data=some_data,
                      columns=('col1',
                               'col2',
                               'col3'))

# Look at the full table
print(df)
"""
   col1  col2  col3
0   0.0   0.0   0.0
1   0.0   0.0   0.0
2   0.0   0.0   0.0
3   0.0   0.0   0.0
4   0.0   0.0   0.0
5   0.0   0.0   0.0
```

```
6   0.0   0.0   0.0
7   0.0   0.0   0.0
8   0.0   0.0   0.0
9   0.0   0.0   0.0
"""

# Or we can access individual columns
# using dictionary-like keys:
print(df['col1'])
"""
0    0.0
1    0.0
2    0.0
3    0.0
4    0.0
5    0.0
6    0.0
7    0.0
8    0.0
9    0.0
Name: col1, dtype: float64
"""
```

Note that we could achieve similar results straight from a dictionary object:

```
df = pandas.DataFrame(data={'col1': [1, 2],
                            'col2': [3, 4]})
```

One of the most powerful aspects of pandas is in the way it interfaces with a wide variety of data formats, from simple CSV files to Excel spreadsheets. To read about the full range of functionality see `https://pandas.pydata.org/docs/reference/io.html`. Let us see a simple example using Excel. Consider the spreadsheet (I the data is just made up), saved as the file ExcelExample.xlsx

	A	B
1	Date	Temperature
2	22/09/2020	18
3	23/09/2020	20
4	24/09/2020	23
5	25/09/2020	23
6	26/09/2020	21
7	27/09/2020	15
8	28/09/2020	21
9	29/09/2020	17
10	30/09/2020	19
11	01/10/2020	21
12	02/10/2020	20
13	03/10/2020	20
14	04/10/2020	18

We can read this directly into a dataframe as follows

```
import pandas

df = pandas.read_excel('ExcelExample.xlsx')

print(df)
"""
         Date  Temperature
0  2020-09-22           18
1  2020-09-23           20
2  2020-09-24           23
3  2020-09-25           23
4  2020-09-26           21
5  2020-09-27           15
6  2020-09-28           21
7  2020-09-29           17
8  2020-09-30           19
9  2020-10-01           21
10 2020-10-02           20
11 2020-10-03           20
12 2020-10-04           18
"""
```

What makes pandas powerful is in its data manipulation facility, and here we will explore some key functionality. First, suppose we wanted a quick statistical summary of the dataframe. Then we can use the `describe()` method:

```
print(excel_data.describe())
"""
```

```
       Temperature
count    13.000000
mean     19.692308
std       2.287087
min      15.000000
25%      18.000000
50%      20.000000
75%      21.000000
max      23.000000
"""
```

Similarly, we can perform specific statistical measurements on the full table or individual columns, for example:

```
# Median of full table
excel_data.median()

# Median of Temperature column
excel_data['Temperature'].median()
```

See `https://pandas.pydata.org/docs/user_guide/computation.html` for a list of possible computations you can perform on dataframes.

Another common operation when working with tabular data is sorting. We do this using the `.sort_values` method.

```
df_sort = excel_data.sort_values(by='Temperature',
                                 ascending=False)
print(df_sort)
"""
         Date  Temperature
2  2020-09-24           23
3  2020-09-25           23
4  2020-09-26           21
6  2020-09-28           21
9  2020-10-01           21
1  2020-09-23           20
10 2020-10-02           20
11 2020-10-03           20
8  2020-09-30           19
0  2020-09-22           18
12 2020-10-04           18
7  2020-09-29           17
5  2020-09-27           15
"""
```

Here we have specified that we want to sort according to the values in the Temperature column, in descending order. Note how the index of each row is retained, such that the first

row has index 2. If we wanted to reset the index in the sorted dataframe, we could call `df_sort.reset_index(drop=True)`. The `drop` argument says that we do not want to retain the sequence of sorted indices as a new column.[1]

Similar to regular arrays, we can use Boolean indexing to select out certain rows, for example:

```
print(sorted_df[sorted_df['Temperature']<20])
"""
         Date  Temperature
8  2020-09-30           19
9  2020-09-22           18
10 2020-10-04           18
11 2020-09-29           17
12 2020-09-27           15
"""
```

If there was a missing value in the input Excel spreadsheet – say, an empty cell – then this would be interpreted as a NaN, which is what pandas assumes for missing data. In panda-based calculations, NaNs are automatically ignored (although this can be overridden if desired). To replace NaN values in the dataframe we can 'fill' them with another value:

```
# Fill all missing data with the value -99
df.fillna(-99)

# Fill missing data in the Temperature column with -99
df['Temperature'].fillna(-99)
```

We can also create a boolean mask to identify missing data in a dataframe using the `pandas.isna()` function.

It is also easy to combine tables. Consider the following example:

```
import numpy as np
import pandas


df1 = pandas.DataFrame(data=np.zeros((3,4)),
                       columns=('A','B','C','D'))

print(df1)
"""
     A    B    C    D
0  0.0  0.0  0.0  0.0
1  0.0  0.0  0.0  0.0
2  0.0  0.0  0.0  0.0
```

[1]Note that by default, row indexing is by integer sequence, but we can also supply a list of index labels, effectively naming rows in the table.

```
"""

df2 = pandas.DataFrame(data=np.ones((4,4)),
                       columns=('A','B','C','D'))

print(df2)

"""
  A    B    C    D
0  1.0  1.0  1.0  1.0
1  1.0  1.0  1.0  1.0
2  1.0  1.0  1.0  1.0
3  1.0  1.0  1.0  1.0
"""

df = pandas.concat([df1,df2], ignore_index=True)

print(df)
"""
     A    B    C    D
0  0.0  0.0  0.0  0.0
1  0.0  0.0  0.0  0.0
2  0.0  0.0  0.0  0.0
3  1.0  1.0  1.0  1.0
4  1.0  1.0  1.0  1.0
5  1.0  1.0  1.0  1.0
6  1.0  1.0  1.0  1.0
"""
```

Here we have concatenated two dataframes column-wise to create a new, single dataframe. Note the use of `ignore_index` which resets the index of each row in the new object.

Question: What if the two dataframes had different columns?

If we concatenate in the same way, we get the following result:

```
df1 = pandas.DataFrame(data=np.zeros((3,4)),
                       columns=('A','B','C','D'))

print(df1)
"""
     A    B    C    D
0  0.0  0.0  0.0  0.0
1  0.0  0.0  0.0  0.0
2  0.0  0.0  0.0  0.0
"""
```

```
df2 = pandas.DataFrame(data=np.ones((4,4)),
                       columns=('E','F','G','H'))

print(df2)
"""
     E    F    G    H
0  1.0  1.0  1.0  1.0
1  1.0  1.0  1.0  1.0
2  1.0  1.0  1.0  1.0
3  1.0  1.0  1.0  1.0
"""

df = pandas.concat([df1,df2], ignore_index=True)

print(df)
"""
     A    B    C    D    E    F    G    H
0  0.0  0.0  0.0  0.0  NaN  NaN  NaN  NaN
1  0.0  0.0  0.0  0.0  NaN  NaN  NaN  NaN
2  0.0  0.0  0.0  0.0  NaN  NaN  NaN  NaN
3  NaN  NaN  NaN  NaN  1.0  1.0  1.0  1.0
4  NaN  NaN  NaN  NaN  1.0  1.0  1.0  1.0
5  NaN  NaN  NaN  NaN  1.0  1.0  1.0  1.0
6  NaN  NaN  NaN  NaN  1.0  1.0  1.0  1.0
"""
```

Here the behaviour is similar to before – effectively stacking the two tables column-wise and adding the additional columns E–H. Where there is no 'overlap' between the tables, the dataframe is padded by NaNs. This may be desired behaviour, but more likely we are interested in aligning the rows. We can do this by specifying the concatenation axis:

```
df = pandas.concat([df1,df2], axis=1)

print(df)
"""
     A    B    C    D    E    F    G    H
0  0.0  0.0  0.0  0.0  1.0  1.0  1.0  1.0
1  0.0  0.0  0.0  0.0  1.0  1.0  1.0  1.0
2  0.0  0.0  0.0  0.0  1.0  1.0  1.0  1.0
3  NaN  NaN  NaN  NaN  1.0  1.0  1.0  1.0
"""
```

Note that this time we do not want to set the `ignore_axis` argument to `True` because it refers to the index along the concatenation index, which in this case refers to column names. So, the first, second and third rows in `df1` and `df2` line up, but because `df2` had an extra row that does not exist in `df1`, values A–D in row four are set to NaN. There are more options for

the exact method of concatenation and the reader is directed to the pandas documentation for further examples: `https://pandas.pydata.org/docs/user_guide/merging.html`.[2]

Hopefully it is clear that by organising data into pandas dataframes, it is possible to perform manipulation and analysis of tabular datasets very easily. We will close with an illustration of a technique called pivoting. This is a task that allows you to 'pivot' a table, turning columns into rows and allowing you to explore and analyse large datasets with ease. Take the following spreadsheet:

	A	B	C	D	E	F	G
1	yyyy	mm	tmax	tmin	af	rain	sun
2	2000	1	8.6	2.4	7	16.5	78.6
3	2000	2	10.4	3.8	5	62.2	102.5
4	2000	3	12.1	4.9	1	16	120.4
5	2000	4	12.9	5.4	0	99.6	135.8
6	2000	5	18	9.6	0	87.2	202.9
7	2000	6	21	12.8	0	19.2	169.5
8	2000	7	21	12.8	0	52.8	174.7
9	2000	8	23.2	14	0	32.3	211.4
10	2000	9	20.1	12.5	0	105.8	132.1
11	2000	10	14.7	8.5	0	155.4	98
12	2000	11	11	4.5	1	99.5	74.9
13	2000	12	9	4.6	5	73.9	50
14	2001	1	7.1	1.8	9	75.2	87
15	2001	2	9.2	2.8	5	69.6	92.3
16	2001	3	9.5	3.8	3	95.3	78.2
17	2001	4	12.8	5.4	0	66.5	138.5
18	2001	5	18.8	9.1	0	26	228
19	2001	6	20.8	11.3	0	39.1	248
20	2001	7	24.3	14.5	0	35.2	229.1
21	2001	8	23.5	14.5	0	79	210.9
22	2001	9	18.1	10.8	0	59.6	143.3
23	2001	10	17.6	11.3	0	108.2	109.9

It contains some of the meterological data from the Heathrow monitoring station that we encountered in Lecture 2. Here we have just stripped out the data for 2000–2004 for demonstration purposes. Let's use pivoting to answer the question 'what was the month-by-month difference in rainfall between 2000 and 2004?'. We can do it in a few lines of code:

```
import pandas

# Read the data
df = pandas.read_excel('heathrow.xlsx')

# Create a pivot table
# Set the year column to be the rows
# Just keep the rainfall values
piv = pandas.pivot(df,columns='yyyy',values='rain')

# Inspect head and tail
print(piv.head())
print(piv.tail())
"""
yyyy  2000  2001  2002  2003  2004
0     16.5   NaN   NaN   NaN   NaN
1     62.2   NaN   NaN   NaN   NaN
```

[2] Also note that an `append()` method exists that allows for incremental additions to dataframe objects, similar in methodology to `concat()`.

```
2     16.0   NaN   NaN   NaN   NaN
3     99.6   NaN   NaN   NaN   NaN
4     87.2   NaN   NaN   NaN   NaN
yyyy  2000  2001  2002  2003   2004
55     NaN   NaN   NaN   NaN  107.8
56     NaN   NaN   NaN   NaN   18.8
57     NaN   NaN   NaN   NaN   96.6
58     NaN   NaN   NaN   NaN   29.6
59     NaN   NaN   NaN   NaN   46.4
"""

# Now separate into rainfall in 2000 and 2004
# and reset the indices, which correspond to months
rain_in_2000 = (piv[2000][~pandas.isna(piv[2000])])
rain_in_2000.reset_index(inplace=True,drop=True)

rain_in_2004 = (piv[2004][~pandas.isna(piv[2004])])
rain_in_2004.reset_index(inplace=True,drop=True)

# Calculate the difference and cast as new dataframe
diff = pandas.DataFrame(rain_in_2004-rain_in_2000)

# Do some naming to make things easy to understand
diff.columns = ['difference']
diff.index = ['Jan','Feb','Mar','Apr','May','Jun',
              'Jul','Aug','Sep','Oct','Nov','Dec']

# Inspect the result
print(diff)
"""
     difference
Jan        55.4
Feb       -36.0
Mar        13.5
Apr       -30.8
May       -42.2
Jun        21.2
Jul       -15.2
Aug        75.5
Sep       -87.0
Oct       -58.8
Nov       -69.9
Dec       -27.5
"""
```

Note what has happened when we actually do the pivot. Each column now corresponds to the five years 2000–2004, and the rows are the total number of data entries: 5×12=60 in total. We have only kept the original values corresponding to the rainfall. The reason all those NaNs appear is that the pivot operation does not know that we want to treat those 60 values as five sets of twelve months, so they just appear in sequence, with NaNs filling the gaps. However, we can easily extract the rain in 2000 and 2004 using the column names and a boolean index that says 'only return the values that are not missing' for the two years. We then end up with two series `rain_in_2000` and `rain_in_2004`. We do a bit of housekeeping to reset the indices to be aligned (0–11 in both cases). We can then easily do the arithmetic to calculate the difference between them, which we make into a new dataframe object. Finally, we do a bit more housekeeping to rename the indices to actual month names, and to give the new column a descriptor – but that is just for presentation. We have now calculated the difference in rainfall between 2004 and 2000, month by month... in just a few lines of code.

A more general use of pivot tables is to summarise and inspect large data tables. A subtly different function `pivot_table()` will allow one to aggregate data (according to some function) for duplicate index/column pairs. To illustrate, we can immediately get a summary of the mean values of each original column on a year-by-year basis:

```
import pandas

# Read the data
df = pandas.read_excel('heathrow.xlsx')

# Create a pivot table.
# Use the default aggfunc=mean method
# and turn years into columns
piv = pandas.pivot_table(df,columns='yyyy')
print(piv)
"""
yyyy   2000   2001   2002   2003   2004
af     1.58   2.58   0.83   2.25   2.08
mm     6.50   6.50   6.50   6.50   6.50
rain  68.37  58.77  65.56  39.62  51.55
sun  129.23 145.57 136.02 166.96 136.58
tmax  15.17  15.03  15.74  16.30  15.51
tmin   7.98   7.59   8.40   7.88   8.19
"""
```

Hopefully this section has illustrated how pandas can be used to quickly manipulate and explore tabular data, and is a powerful tool. We will return to pandas throughout the module, and you are encouraged to experiment with its features.

Week 2 practical challenges continued

- Use the website `https://www.metoffice.gov.uk/research/climate/maps-and-data/uk-and-regional-series` to explore the MET Office UK and regional series data. Access all the parameters available (e.g. max temp, Sunshine, etc.), year-ordered, read directly from the URL. Create your own class to represent and work with the data, using attributes and methods.
- Assemble all the meteorological data into a single HDF5 file, organised appropriately using groups and attributes. Ensure you can read the resulting HDF5 file and access specific content.
- Read the HDF5 file above into a pandas dataframe. Use pandas functions such as pivot tables to produce your own statistical summaries of meteorological data for the UK over the past century. Use your initiative – there are many different approaches to this problem. Consider using a raster image format to visualise data.

Extension: Experiment with saving your pandas summary analysis to different data formats, such as CSV or Excel. Find other data sets to experiment with, and to hone your data manipulation skills.

Part II

Data

Chapter 4

Data Visualisation

Visualisation is at the heart of data science – it allows us to communicate efficiently with others, be it a simple description of a single dataset, showing the relationship between different datasets, or revealing trends and correlations. It can enable us to identify outliers and generate ideas for models. At its best, data visualisation is beautiful[1]. At its worst, it is simply appalling[2].

The main motivation for using visualisation is so that people can quickly absorb large amounts of visual information and find patterns in it. For example, in Figure 4.1 we have the Sea Surface Temperature (SST) in degrees Celsius for July, 2982. This figure quickly summarises the information from approximately 250,000 numbers and can be easily interpreted in a matter of seconds. For example, it is easy to see that the ocean temperature is highest at the equator and lowest at the poles. This demonstrates a key guiding principle in visualisations: simplicity is king.

4.1 Good practice

There are many different tools we can use to visualise data, and we will explore just some of these here. However, no matter what tool we use, there are some general principles of good practice when it comes to data vizualisation:

1. Less is more: a 'busy' or crowded visualisation is hard to read. Think of the main message we want our graphics to convey, and then use as few components as possible to achieve that.

2. Be consistent: our plots and figures should conform to a single style. This can be our own design, or following the convention or style-guide of a particular project or organisation. A good example can be seen in the look-and-feel of graphical data presented in *The Economist*.

3. Attention to detail: ensure we take the time to make our visualisations professional. Do not mix font styles, employ adequate spacing between labels, make sure tick marks are visible, margins are sensible, etc.

[1] `https://flowingdata.com`
[2] `https://viz.wtf`

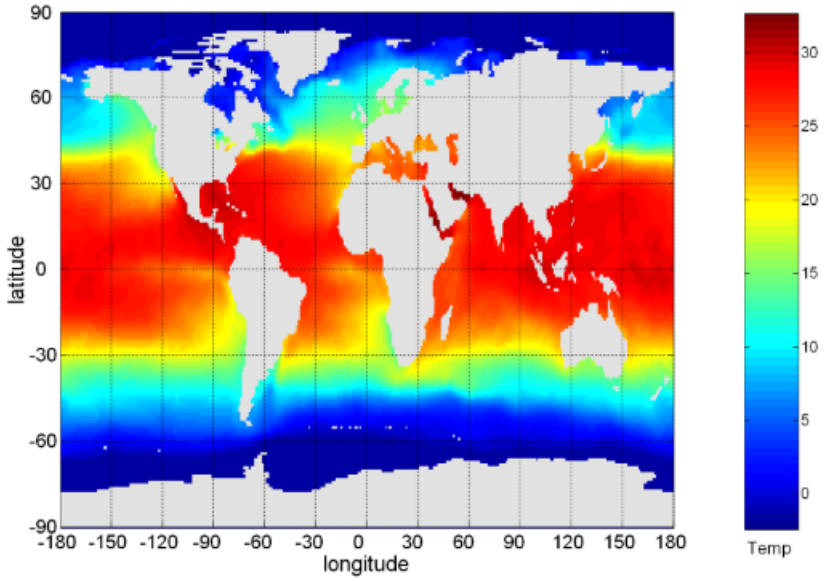


Figure 4.1: Sea Surface Temperature (SST) for July, 1982. [1]

4. Think of our audience: make our visualisations accessible to all. Carefully consider our colour schemes (make sure our palette is colour-blind friendly for example), line styles, marker sizes, and so-on.[3]

5. Be self-contained: often our visualisation will be accompanied by some explanatory text or other description. But imagine if someone *only* had the graphical part – say a single graph. We should make sure that the visualisation includes everything necessary for someone to understand what is being shown. This includes descriptive axis labels, a title, a legend and possibly annotation (as long as this does not clash with Rule 1 above).

4.2 Basic techniques with Matplotlib

The most common visualisation package in Python is Matplotlib. It is a desktop plotting package designed for creating (mostly two-dimensional) publication-quality plots. The project itself was started by John Hunter in 2002 with the goal of enabling a MATLAB-like[4] plotting interface in Python. Over time, it has spawned a number of add-on toolkits for data visualisation that use matplotlib for their underlying plotting. One of these is seaborn[5] which is explored in the lab.

[3] `https://thenode.biologists.com/data-visualization-with-flying-colors/research`

[4] MATLAB is a (proprietary) programming language often used in scientific research and data science, designed specifically for mathematical/numerical operations. We won't cover MATLAB in this course, but you should be aware of its existence.

[5] See `http://seaborn.pydata.org`

The matplotlib and IPython communities have collaborated to simplify interactive plotting from the IPython shell (and now, Jupyter Notebook). It now supports various GUI backends on all operating systems and can export visualisations to all of the common vector and raster graphics formats (pdf, svg, jpg, png, bmp, gif etc.).

If we wish to use matplotlib from within a script, then the function `plt.show` is our friend. This starts an event loop that looks for all currently active `Figure` objects and opens one or more interactive windows that displays our figure(s). We can then run this script from the command-line prompt if we wanted (e.g. `$ python myplot.py`). This will result in a window opening with our figure. Alternatively, if we run the script in Spyder then it will appear in our window. Note however that the `plt.show` command should only be used once per Python session, and is most often seen at the very end of the script. Multiple `show` commands can lead to unpredictable backend-dependent behaviour, and should mostly be avoided.

If instead we want to plot from Jupyter notebook, then we can do this just as easily. IPython is built to work well with matplotlib, so long as we specify matplotlib mode. To enable this mode, we use the following magic command after starting notebook:

```
%matplotlib notebook
```

This will lead to interactive plots embedded within the notebook. If we instead used the magic command `%matplotlib inline` then this will lead to static images of our plot embedded in the notebook. Note that `plt.show` is not required in IPython's matplotlib mode.

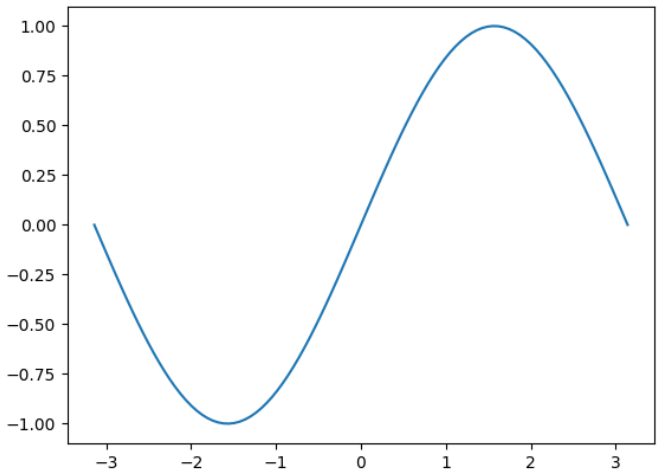
4.2.1 Basic plotting

We start with building up a simple plot, where we make a set of values `theta` and calculate `sin(theta)`:

```
import numpy as np
import matplotlib.pyplot as plt

#a flag to determine if we show the plot on the screen
show = False

#set up a set of 100 values over +/- pi
x = np.linspace(-np.pi,np.pi,100)

#plot sin(x) versus x
plt.plot(x, np.sin(x),label='sin(x)')

#save the plot as a .png file
plt.savefig('fig1.png')

#show it on the screen if show==True
if show: plt.show()
```

Note the strange way we are importing Matplotlib. The 'pyplot' bit refers to a set of functions within the Matplotlib package that allow users to make visualisations in a style similar to MATLAB.

We can call the `plot` function, providing the two arrays that represent the variables we want to appear on the x and y axes, as well as a string label for the data we are plotting using the keyword argument `label`. We save the plot as an image (PNG format) on disk, and also display to the screen if we want. This is the result:



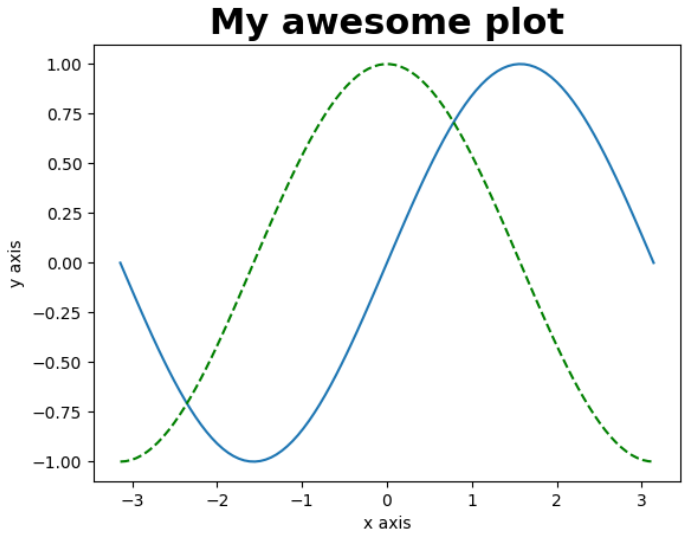
We can now add some titles for the plot, and the axes (this assumes a continuation of the code block above):

```
plt.xlabel('x axis')
plt.ylabel('y axis')
plt.title('My awesome plot', size=22, weight='bold')
plt.savefig('fig2.png')
```

Note that we can control the font style using keyword arguments when we use text in matplotlib. The result is as follows:

Next we add another line representing `cos(theta)`. This time, we want to control the look of the line:

```
plt.plot(x, np.cos(x), linestyle='dashed', color='green',label='cos(x)')
plt.savefig('fig3.png')
```

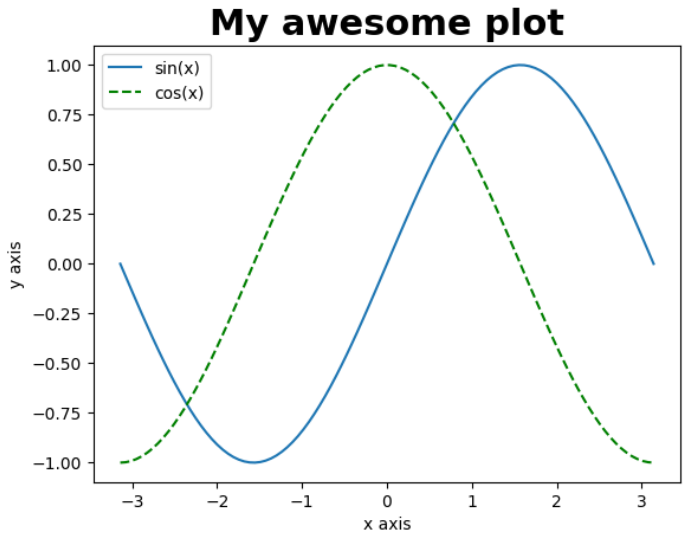


Here we have used keyword arguments to control the colour and style of the line. Now we should add a legend, explaining what the lines mean:

```
plt.legend(loc='best')
plt.savefig('fig4.png')
```

The labels supplied when we call each `plot` function is used to associate with each of the lines – note that matplotlib can automatically detect what the different components are (in

this case a solid blue and dashed green line), but we can also explicitly specify what elements appear in the legend. The `loc` keyword argument sets the position of the legend, which in this case is automatically determined as the 'best' position that does not interfere with the content of the plot. Other options are, for example 'bottom right' or 'top left'. We can also supply a tuple of two numbers representing the coordinates of the lower left corner of the legend.



We can display data as a scatter plot as well. To show this, we can simulate some 'noisy' data where we simply perturb the pure sinusoidal signals using Gaussian noise – i.e. using the NumPy `normal` generator:

```
#make some noisy data by adding Gaussian noise to the analytic functions
some_random_sin_points = np.sin(x) + np.random.normal(0,0.2,size=100)
some_random_cos_points = np.cos(x) + np.random.normal(0,0.1,size=100)

#plot noisy sin(x)
plt.plot(x,some_random_sin_points,marker='o',linestyle='None',
    markerfacecolor='lightblue',markeredgecolor='black',markersize=4,label='
    noisy sin(x)')

#plot noise cos(x)
plt.plot(x,some_random_cos_points,marker='s',linestyle='None',
    markerfacecolor='lightgreen',markeredgecolor='black',markersize=4,label=
    'noisy cos(x)')

#let's add a grid for the hell of it
plt.grid()

#update the legend
plt.legend(loc='best')
```

```
plt.savefig('fig5.png')
```

The result:

To display the data as points only, we set the `linestyle='none'` and specify a marker ('o'=circle, 's'=square)[6]. We can control the edge and face colour of the markers using keyword arguments, as well as their size. Again, we supply a label for the data to be used in the legend, that we replot. Finally, just to show off, we overlay a grid to make reading off the values easier.

It should be obvious at this point that we can control the appearance of markers, lines and text through the use of keyword arguments. We have not covered all the possible options here, so it is recommended to look at the relevant documentation to explore the possibilities: `https://matplotlib.org/contents.html`. It may also be worth looking at Part IV of [3] or Chapter 9 of [2].

4.2.2 Histograms

In data science, and statistics in general, we often refer to distributions. This could be some probability distribution, or the distribution of values in some sample. We can display distributions easily using the `hist` function. In the following we use the NumPy random module to generate two samples of 1000 elements. Sample 1 is selected from a Normal distribution[7] with mean -1 and standard deviation 1, Sample 2 is sampled from a Normal distribution with mean 1 and standard deviation 0.5.[8] We want to plot and compare these distributions as histograms:

[6]See a full list of marker style codes here `https://matplotlib.org/api/markers_api.html#module-matplotlib.markers`

[7]Also called a Gaussian distribution or probability bell curve.

[8]Recall, a Normal distribution is symmetric about the mean μ with width determined by the standard deviation σ. More accurately, the range covered by $\mu \pm \sigma$ includes about 68% of the distribution, $\mu \pm 2\sigma$ includes about 95% of the distribition, while $\mu \pm 3\sigma$ includes about 99.7% of the distribution. This is telling us that data near the mean are more frequent, while those further away from the mean are less frequent. One

```
import numpy as np
import matplotlib.pyplot as plt

#define the samples
sample1 = np.random.normal(-1,1,size=1000)
sample2 = np.random.normal(1,0.5,size=1000)

#manually set the plot range [xmin, xmax, ymin, ymax]
plt.axis([-5, 5, 0, 150])

#plot the histograms
plt.hist(sample1,bins=20,color='red',alpha=0.8,label='Sample 1')
plt.hist(sample2,bins=20,color='blue',alpha=0.8,label='Sample 2')

#set out labels
plt.xlabel('Some parameter')
plt.ylabel('Number')

#add a legend
plt.legend(loc='upper right')

#save the plot
plt.savefig('fig6.png')
```

Here we demonstrate how we can manually control the axis ranges using `axis()`. In the `hist` calls, we supply the arrays representing our samples, and define how many evenly spaced bins we want to break them up into. We can explicitly set the bins ourselves or, like we do here, set an integer number that means that the bin edges are automatically calculated.

To better understand binning, we can consider an example. Consider the following example of data which has been sorted into increasing order.

$$4,\ 8,\ 15,\ 21,\ 21,\ 24,\ 25.$$

We can partition this data into *equal-frequency* bins of size 3:

$$\begin{array}{ll} Bin\ 1 & 4,\ 8,\ 15 \\ Bin\ 2 & 21,\ 21,\ 24 \\ Bin\ 3 & 25,\ 28,\ 34. \end{array}$$

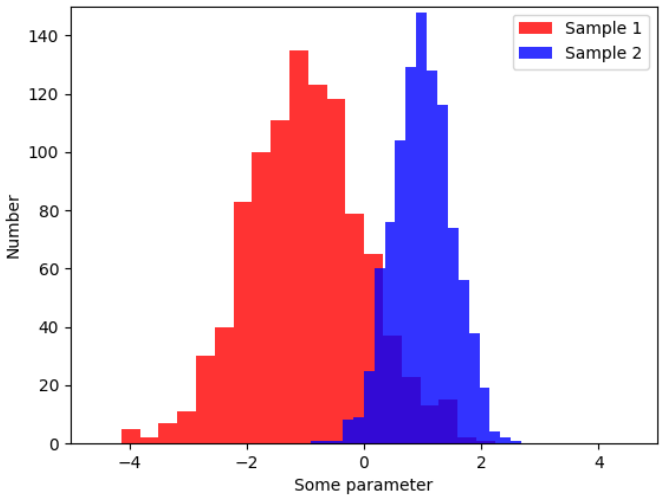
Alternatively, we could partition data into *equal-width* bins. Here, we choose a width (typically, we set $width = \frac{max - min}{no.\ of\ bins}$) and then each bin has range $min + width$. Take another data set:

$$[5, 10, 11, 13, 15, 35, 50, 55, 72, 92, 204, 215].$$

reason for its importance is the Central Limit Theorem which states that if we have a data set/population (again with mean μ and standard deviation σ) and take sufficiently large random samples from the population with replacement, then the distribution of the sample means will be approximately normally distributed.

Suppose we once again want 3 bins, then $width = 70$. In this case, we have the following:

Bin 1	5, 10, 11, 13, 15, 35, 50, 55, 72
Bin 2	92
Bin 3	204, 215.



Note that the `alpha` keyword argument in the `hist` call sets a transparency for the colour, so we can see where the distributions overlap (on a 0–1 scale where `alpha=0` is totally transparent). Again, there are a number of other keyword arguments to `hist` that control its behaviour, and to understand what they do the reader is directed to `https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.hist.html`.

4.2.3 Controlling figure size

In the examples above we let matplotlib decide what size the plot should be, but we can control this easily using the `figure()` function. For example, the following will set the size of the plot to 6 inches by 6 inches.

```
plt.figure(figsize=(6,6))
```

This is useful if we are preparing a figure to appear in a document or presentation.

4.2.4 Sub-plots

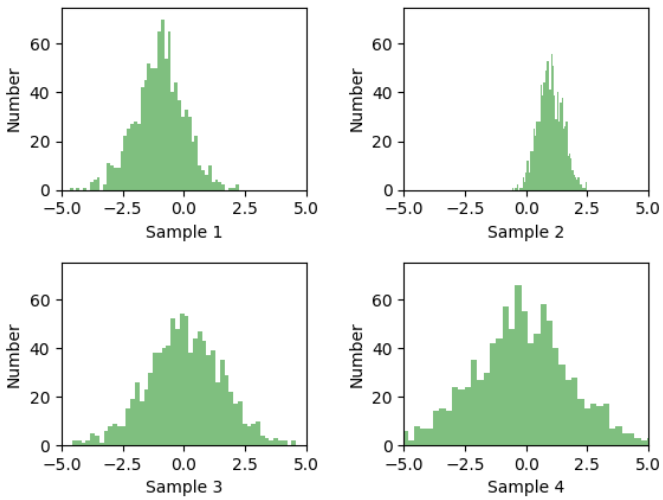
Often, it is useful to compare several plots side-by-side, or arranged in a grid, all in one figure. We do this using sub-plots. For example, suppose we want to show four separate distributions as a 2×2 grid of histograms:

```
import numpy as np
import matplotlib.pyplot as plt
```

```
#four random samples
sample1 = np.random.normal(-1,1,size=1000)
sample2 = np.random.normal(1,0.5,size=1000)
sample3 = np.random.normal(0,1.5,size=1000)
sample4 = np.random.normal(-0.2,2,size=1000)

#put these in an iterable list
data = [sample1,sample2,sample3,sample4]

#use the subplots_adjust function to make a bit more space between the
   plots
plt.subplots_adjust(hspace=0.4, wspace=0.4)

#now loop over the data list
for i,d in enumerate(data):
    plt.subplot(2,2,i+1)
    plt.axis([-5, 5, 0, 75])
    plt.hist(d,bins=50,color='green',alpha=0.5)
    plt.xlabel('Sample '+str((i+1)))
    plt.ylabel('Number')

plt.savefig('fig7.png')
```

We use the `subplots_adjust` function to increase the spacing between the individual plots – this is just one of the ways to improve presentation. Here, we also introduce the built in `enumerate` function. The argument to this function is a list (in this case holding our samples), and what is returned is essentially a pair of values representing a counter (called `i` here) and the corresponding item in the list (which we call `d` here). It is a very handy function when it comes to iterating over iterable objects in Python. The reason we want the value `i` is to pass as an argument to `subplot`.

The arguments to `subplot` are the total number of sub-plots in the horizontal and vertical direction (in this case 2 and 2), and the final argument is the counter of the sub-plot we are working on. Irritatingly, this is indexed from 1, unlike most Pythonic indexing, which starts at 0 by convention. As a result, we have to do `i+1` here, as the first counter returned by `enumerate` is zero. After that, we define our axes ranges, then plot the corresponding histogram of samples represented by `d`. We add a label to tell us what each plot represents, and then come out of the loop and save the figure. The result is as follows:



Sub-plots allow for very flexible layouts and we are not limited to a regular grid as above. We can define almost arbitrary arrangements of sub-plots, including having a plot *inside* another one. The reader can find out more through experimentation, and through the documentation: `https://matplotlib.org/3.3.1/api/_as_gen/matplotlib.pyplot.subplot.html`.

4.2.5 Controlling style using custom configurations

Every plot element in a matplotlib plot has a specific style. The default style is okay, but not great. The good news is that we can customize our plots using custom styles which change the look and feel of the plot (e.g. font, sizes, colours, etc.). There are some in-built styles to choose from, and we can see what is available using the following:

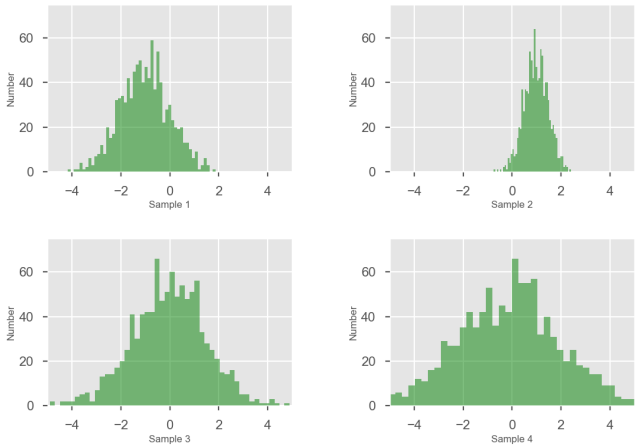
```
import matplotlib.pyplot as plt

print(plt.style.available)
"""
['Solarize_Light2', '_classic_test_patch', 'bmh', 'classic', '
    dark_background', 'fast', 'fivethirtyeight', 'ggplot', 'grayscale', '
    seaborn', 'seaborn-bright', 'seaborn-colorblind', 'seaborn-dark', '
    seaborn-dark-palette', 'seaborn-darkgrid', 'seaborn-deep', 'seaborn-
    muted', 'seaborn-notebook', 'seaborn-paper', 'seaborn-pastel', 'seaborn
    -poster', 'seaborn-talk', 'seaborn-ticks', 'seaborn-white', 'seaborn-
    whitegrid', 'tableau-colorblind10']
"""
```

We have a list of style names and can set the style using one of those names, e.g.

```
import matplotlib.pyplot as plt
```

```
plt.style.use('ggplot')
```

The effect of applying this to the last figure is the following:



To inspect the look-and-feel of the individual in-built styles the reader is directed to `https://matplotlib.org/3.1.1/gallery/style_sheets/style_sheets_reference.html`. Furthermore, we can also create our own style sheet by creating a file (e.g. `mystyle`) and supplying this using something like `plt.style.use('/path/to/mystyle')`. The full list of options we can control can be found by inspecting the `matplotlibrc` file, which is the main configuration file for matplotlib. We can pick and choose a subset of the potential line items to customise in our own style sheet (the remainder will be set according to `matplotlibrc`. To locate this file, do the following:

```
import matplotlib as mpl
print(mpl.matplotlib_fname())
```

This will tell us the path to the file on our system, which we can then inspect on the terminal or using a text editor.

Exercise: Experiment with making your own custom style sheet.

4.3 Visualising distributions

One of the most important visualisation techniques is to (simply) display the distribution of some data. It could be the properties of a sample of observations, or the probability distribution representing the confidence of some measurement. Here, we will examine a few different techniques for displaying this information.

4.3.1 Kernel Density Estimate

A Kernel Density Estimate, or KDE, is an approximation of the probability distribution of a random variable. In the examples above we saw how to plot the distribution of a random

variable using histograms. Histograms present discrete 'counts' or frequency within intervals spanning the range of values of the data. A KDE provides a continuous, smooth estimate of the probability density of drawing a given value from a distribution.

Suppose we were drawing samples $(x_1, x_2, ..., x_n)$ from a distribution. Let the probability density of that distribution be $p(x)$. We do not know what $p(x)$ is, but we can estimate it as follows:

$$p(x) \approx \frac{1}{nh} \sum_{i=1}^{n} K\left(\frac{x - x_i}{h}\right) \tag{4.3.1}$$

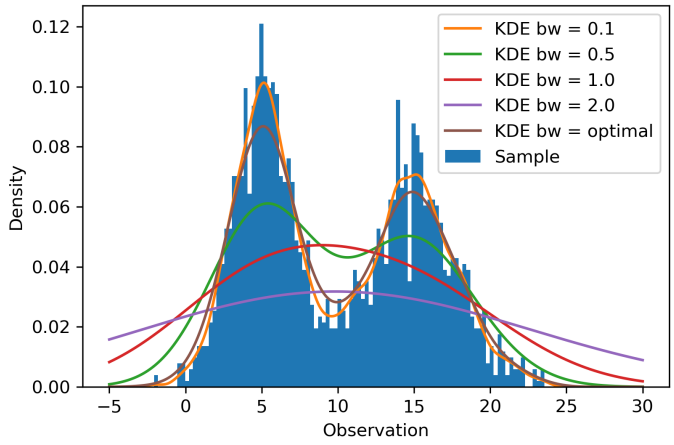
Here h is called the 'bandwidth', where $h > 0$, and K is the kernel, which is a function. Different kernels can be used, but a common one is the Normal distribution. The bandwidth can be tuned as a free parameter: too small and it will not sample the data well enough, but too large and the KDE will be over-smoothed. A rule of thumb for univariate data is to set $h = 1.06\sigma/n^{0.2}$, where σ is the standard deviation of the sample.

A simple implementation of applying a Normal (or Gaussian) KDE in Python is via the `scipy` module:

```
import matplotlib.pyplot as plt
import numpy as np
from scipy import stats

# Generate two random samples from a Normal distribution
sample1 = np.random.normal(5,2,1000)
sample2 = np.random.normal(15,3,1000)

# Make into one big sample
sample = np.concatenate((sample1,sample2))

# Plot the histogram
plt.hist(sample, bins=100, density=True, label='Sample')

# Define a continuous set of x values
xs = np.linspace(-5,30,1000)

# Loop over some example bandwidths
for bw in [0.1,0.5,1,2]:
    # Calculate the KDE with a fixed bw
    kde = stats.gaussian_kde(sample, bw_method=bw)
    # Plot the KDE
    plt.plot(xs,kde(xs),label='KDE bw = %.1f'%bw)

# Also show the default optimal bw
kde = stats.gaussian_kde(sample)
plt.plot(xs,kde(xs),label='KDE bw = optimal')
```

```
plt.xlabel('Observation')
plt.ylabel('Density')
plt.legend()
```

In this example we create a sample of data with two peaks in the distribution by drawing from two independent Normal distributions. We plot the histogram and KDE. Note that we use the 'density' option of the `hist()` function:



We calcuate the KDE using different bandwidths – notice the effect on the resultant curves. Unless we specifically want to set the bandwidth, it is probably best to go with the default, which aims to optimize the bandwidth based on the sample.

4.3.2 Box and Whisker

Box plots can illustrate the quartile range, median and total range of a sample of data. The quartiles can be labelled: Q_1 (25% percentile, or lower quartile), Q_2 (50% percentile, or median), Q_3 (75% percentile, or upper quartile). The interquartile range (IQR) is defined by $Q_3 - Q1$.

To construct our boxplot we draw it spanning Q_1 to Q_3, and indicate Q_2 with a line through the box[9]. 'Whiskers' extend either side of the box, and can be used to indicate the maximum and minimum values, or alternatively some percentile range beyond Q_1 and Q_3. For example, the 16% and 84% percentiles representing the '1σ' range.

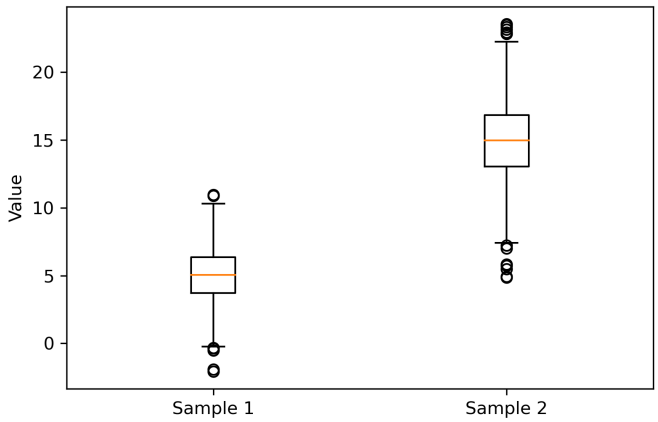
An important point to note is that box and whisker plots show distributions with 'outliers' removed. The outliers are usually shown as separate points. There are different ways to define a statistical outlier in a data set, but often one can use a 'σ-clip', whereby an outlier is defined as any point with a value some multiple of standard deviations away from the mean.

To demonstrate the above, we can use `matplotlib`. Taking the two individual samples from the KDE example above, we do the following:

[9]Box and whisker plots can be drawn horizontally or vertically.

```
import matplotlib.pyplot as plt
import numpy as np

# Two samples
sample1 = np.random.normal(5,2,1000)
sample2 = np.random.normal(15,3,1000)

# Supply samples as a sequence to boxplot, along with labels
plt.boxplot([sample1, sample2],
            labels=['Sample 1', 'Sample 2'])

# A little bit of decoration
plt.ylabel('Value')
```

Here is the result:



We have not customised the plot much here, but there are various options for defining the length of the whiskers, what defines an outlier, whether to show vertical or horizontal boxes and so-on. It is left as an exercise to experiment with playing around with the settings to see the effect on the box and whisker plot.

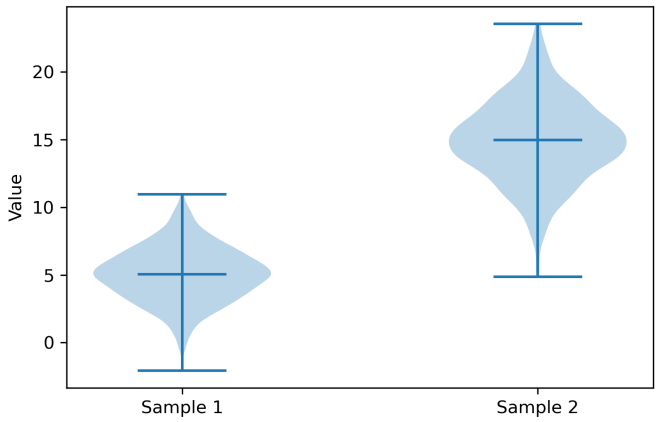
4.3.3 Violin plots

A violin plot can be thought of as a combination of a box plot and KDE plot. Box plots significantly reduce the amount of information shown about a distribution by only representing the range and quartiles (or percentiles) – i.e. summary statistics (coming soon!). Violin plots show the KDE, which is of course a representation of the full probability distribution of the data. Again, we can plot these easily with `matplotlib`. Using the same two samples as above, we do the following:

```
import matplotlib.pyplot as plt
import numpy as np
```

```
# Two samples again
sample1 = np.random.normal(5,2,1000)
sample2 = np.random.normal(15,3,1000)

# Use subplots to return an axis object for
# better control
fig, ax = plt.subplots(1,1)

# Like boxplot, supply samples as sequence and show medians
ax.violinplot([sample1,sample2],
              showmedians=True)

# Modify the default axis labelling
ax.set_xticks([1,2])
ax.set_xticklabels(['Sample 1', 'Sample 2'])
plt.ylabel('Value')
```

The result:



Okay, they don't look that much like violins, but you get the idea. Here, we are also showing the position of the median, and the extremes of the distributions. The shaded regions are the KDEs of the distributions, reflected about the vertical axis. We can control how the KDE is calculated in a similar manner to the example above. For example, by supplying a given bandwidth as an argument to `violin()`. Of course, we can also modify the fine detail of how the plots look in terms of colour, line styles, etc.

4.4 Plotting with Pandas

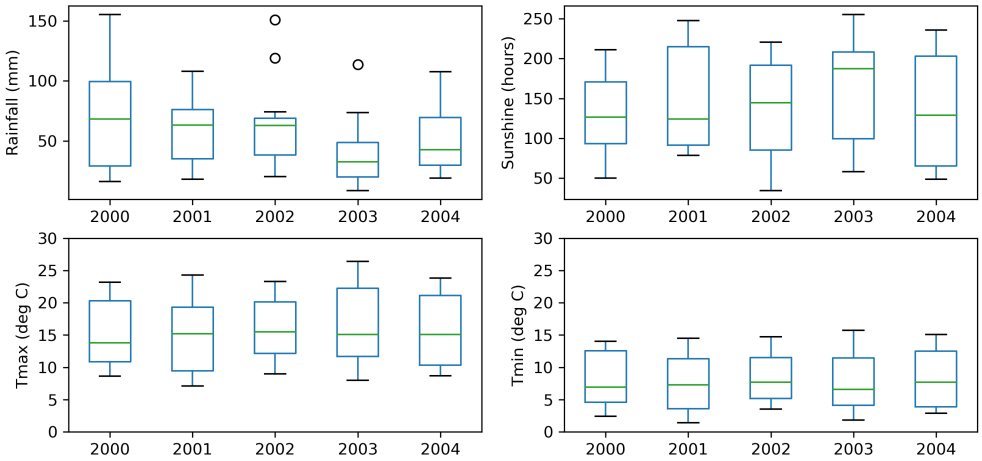
We have seen how to use *pandas* to wrangle data, by way of the concept of 'dataframes'. Luckily for us, pandas can also 'talk' to matplotlib, so it is easy to construct plots and

visualisations of data represented by dataframes. Taking the example of meteorological data from earlier, we can create a visualisation using a DataFrame. In this example we will make a set of box and whisker plots of the sun hours, rainfall and minimum and maximum temperatures on a year-by-year basis:

```
import matplotlib.pyplot as plt
import pandas

# Read the data into a dataframe
df = pandas.read_excel('heathrow.xlsx')

# Set up a 2x2 grid of subfigures
fig,ax = plt.subplots(nrows=2,ncols=2,figsize=(10,5))

# Plot boxplots for each parameter, grouped by year
df.boxplot(column='rain',by='yyyy',
           ax=ax[0,0],
           grid=False)

# Set some labels
ax[0,0].set_title('')
ax[0,0].set_ylabel('Rainfall (mm)')
ax[0,0].set_xlabel('')

df.boxplot(column='sun',by='yyyy',
           ax=ax[0,1],
           grid=False)

ax[0,1].set_title('')
ax[0,1].set_ylabel('Sunshine (hours)')
ax[0,1].set_xlabel('')

df.boxplot(column='tmax', by='yyyy',
           ax=ax[1,0],
           grid=False)

ax[1,0].set_title('')
ax[1,0].set_ylabel('Tmax (deg C)')
ax[1,0].set_ylim(bottom=0,top=30)
ax[1,0].set_xlabel('')

df.boxplot(column='tmin', by='yyyy',
           ax=ax[1,1],
           grid=False)
```

```
ax[1,1].set_title('')
ax[1,1].set_ylabel('Tmin (deg C)')
ax[1,1].set_ylim(bottom=0, top=30)
ax[1,1].set_xlabel('')

plt.title('')
plt.suptitle('')
```

The result:



Note that here `boxplot()` is a method of a DataFrame object, but we can supply a predefined 'axis' object made in `matplotlib`, and this is useful if we want to control the fine detail of the layout. Of course, we could also simply extract data from the DataFrame as we wish and supply these to the regular `matplotlib` plotting commands. The reader can read more about visualisation of pandas DataFrames here: `https://pandas.pydata.org/pandas-docs/stable/user_guide/visualization.html`.

Exercise: Experiment with generating different types of plot, either using the DataFrame methods, or via `matplotlib` directly.

Week 5 Practical Challenges

1. Start simple and create a simple line plot which is a straight line starting at 0, ending at 8.

2. Use `pd.read_csv()` to load the US presidential heights data set. Then, print the maximum height, minimum height, mean height and standard deviation. Next print the 25^{th} percentile, median and 75^{th} percentile. Finally, create a histogram of presidential heights.

3. While we have seen how to use matplotlib, it is a fairly low level tool. Better options include using pandas (as we have seen) and seaborn. In particular, the latter is a statistical graphics library created by Michael Easkom (`https://seaborn.pydata.org/`) which simplifies many common visualisation types. Importing seaborn modifies the default matplotlib colour schemes and plot styles to improve the readability and aesthetics. Even if we do not use the seaborn API, it is sometimes preferable to import seaborn (we often use the alias `sb`) as a simple way to improve the visual aesthetics of general matplotlib plots.

   (a) Use DataFrame's `plot` method to plot five different lines (labelled A, B, C and D) on the same subplot (note the creation of of a legend automatically). The data should consist of 10 points and be randomly generated between 0 and 100. As an extra step, try cumulatively summing along the columns.

   (b) Use `plot.bar` and `plot.barh()` to create vertical and horizontal bar plots. In particular, create a Series with 16 randomly generated values of float type from a uniform distribution over [0,1] (label them a, b, $c \ldots$) to use as the data. Ensure both are in the same figure, one on top of the other.

   (c) Use `pd.read_csv()` to load the restaurant tipping data set on canvas. Create a stacked bar plot showing the percentage of data points for each party size on each day.

   (d) Take the tipping data set again and use the seaborn package to create a bar plot which has the tipping percentage by day, along with error bars which represent the 95% confidence interval. The x-axis should have the tip percentage, while the y-axis should have the day.

   (e) By once again using the tipping data set, use seaborn to create a histogram of tip percentages of the total bill. On the same plot, add a density plot (using seaborn). A density plot is formed by computing an estimate of a continuous probability distribition that might have generated the observed data. The usual procedure is to approximate this distribution as a mixture of "kernels" (i.e. simpler distributions like the Normal distribution). Thus, density plots are also known as kernel density estimate (KDE) plots.

Bibliography

[1] P. Tan et al. *Introduction to Data Mining*. Global Edition. Pearson, Second edition, 2020.

[2] W. McKinney. *Python for Data Analysis, 2nd Edition*. O'Reilly Media, Inc., 2017.

[3] J. VanderPlas. *Python Data Science Handbook*. O'Reilly Media, Inc., 2016.