2019-05-02 Python lesson tutor notes

These notes are intended for the tutor as they work through the material, but may also be useful for independent learning.

Start the slides

TITLE: Building Programs With Python (Part 1)

SLIDE: Etherpad

- Please use the Etherpad for the course **DEMONSTRATE LINK**
  - note that **slides and tutor notes are available online**

SLIDE: Why Are We Here?

- We're here to learn **how to program**
  - this lesson just happens to be in Python, but the **principles** are relevant to all languages
- Programming is a way to **solve problems in your research** through making a computer do work **quickly** and **accurately**
- You'll build **functions** that do specific, defined tasks
- You'll **automate** those functions to perform tasks over and over again (in various combinations)

- You'll **manipulate data**, which is at the heart of all academia
- You'll learn some **file input/output** to make the computer read and write useful information
- You'll learn some **data structures**, which are ways to organise data so that the computer can deal with it efficiently

---

SLIDE: XKCD: The best use of your time

- The XKCD comic is tongue-in cheek, but there's a **lot of truth in this**
  - The less time you spend messing with Excel or manually processing data files, the more time you have for getting your research done

---

SLIDE How are we doing this?

- We'll be learning how to program **using Python**
- **Why Python?**
- We need to use *some* language
- Python is **free**, with **good documentation** and lots of books and online courses.
- Python is **widely-used** in academia, and there's lots of support online
- It can be **easier for novices** to pick up than other languages
- **We won't be covering the entire language in detail**
- For those with a bit more experience, note: **we will be using some long-handed ways of doing things to keep them clear for novices**

SLIDE: XKCD: Python

- This XKCD comic highlights two of those key points:
  - Programming in Python is **FUN**
  - There are **libraries for pretty much anything** you might want to do

---

SLIDE No, I mean "*How* are we doing this?"

- We'll use **two tools to write Python**
- The bulk of the course will be in a **text editor**
  - Text editors are part of the **edit-save-execute** cycle, which is how much code is written
- We'll also spend a little bit of time writing code in the `Jupyter` notebook**
  - `Jupyter` is **good for exploring data, prototyping code, data-wrangling, and teaching**
  - However, it's **not so good for writing "production code"** in a general sense
- There are also specialist **integrated development environments (IDEs)** for Python that are extremely useful for developers, but we'll not be using them

---

SLIDE Do I need to use `Python` afterwards?

- **No.**
- The lesson and principles are general, we're just teaching in Python

- What you learn here will be relevant in other languages
- If your field or colleagues use another language in preference, **there may be very good reasons for that**, and they **may be able to offer detailed, relevant support and help to you in that language**. This is valuable.
    - My advice is usually: when learning, choose the language that you can get local help in.
- Language Wars waste everyone's time.

SLIDE What are we doing?

- We're using **a motivating example of data analysis**
- We've got some data relating to a new treatment for arthritis, and we're going to **explore it.**
- Data represents patients and their **daily measurements of inflammation**
- We need to **analyse** the data
- We need to **visualise** the data
- We're going to get the computer to do this for us
    - It's a small dataset, so we could do this by hand (it would take us a day, maybe)
    - (Excel anecdote from lab?)
- **Automation** is key:
    - **fewer human mistakes**
    - easier to **apply to other future datasets**
    - easier to share with others (**transparency**)
- We can also **share our code and results** *via* sites such as GitHub and BitBucket
    - publish as supplementary information
    - greater impact
    - encourage collaboration

SECTION 01: Setup

SLIDE Setting Up - 1 - **DEMO**

- We want a **neat (clean) working environment**: always a good idea when starting a new project - it helps for when you might want to use `git` to put it under version control, later.
- **Change directory to desktop** (in terminal or Explorer)
- **Create directory** `python-novice-inflammation`
- **Change your working directory** to that directory

```
cd ~/Desktop
mkdir pni
cd pni
```

SLIDE Setting Up - 2 - **DEMO**

- We need to **download our data** (and also a little code that can help us)

  - This is just like grabbing data from an analytical machine's output, or being given data by a collaborator

- **Go to Etherpad in browser** http://pad.software-carpentry.org/2019-05-02-standrews

- **Point out file links** http://swcarpentry.github.io/python-novice-inflammation/data/python-novice-inflammation-data.zip

- **Click on file links to download**

- **Move files** to `pni` directory

- **Extract files** - this will create a subdirectory called `data` in that folder

- **CHECK WHETHER EVERYONE HAS EXTRACTED THE DATA**

- **SHOW FILE STRUCTURE IN TERMINAL AND FILE EXPLORER**

🟥 Red sticky for a question or issue 🟩 Green sticky if complete

---

SECTION 02: Getting Started

SLIDE `Python` in the terminal

- We start the **`Python` console** with the command `python`
  - This should bring up the interactive console
- **Explain** header information
- **Explain** the prompt

```
$ python
Python 3.6.3 |Anaconda custom (64-bit)| (default, Oct  6 2017, 12:04:38)
[GCC 4.2.1 Compatible Clang 4.0.1 (tags/RELEASE_401/final)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

- **CHECK WHETHER EVERYONE HAS STARTED THE CONSOLE**

🟥 Red sticky for a question or issue 🟩 Green sticky if complete

---

SLIDE `Python` REPL

- You learned about the REPL (read-evaluate-print-loop) in the shell lesson

    - `Python`'s console implements the REPL
- We can use `Python` like a complex calculator
- **Note the spaces around operators** - good `Python` style

```
>>> 3 + 5
8
>>> 12 / 7
1.7142857142857142
>>> 2 ** 16
65536
>>> 15 % 4
3
>>> (2 + 4) * (3 - 7)
-24
```

🟥 Red sticky for a question or issue 🟩 Green sticky if complete

---

SLIDE My first variable

- We've seen how to use the REPL
    - To build interesting things, we'll need to store values
    - We need to work with **variables**
- Variables are like **named boxes**
    - An item of data goes into the box
    - When we refer to the box/variable name, we get the **contents of the box**
- We need a **variable name**
- We need **variable contents**
- **Use a real-life example to hand if possible**
- You can think of a variable as a labelled box, containing a data item
    - Here, we have a box labelled `Name` - this is the variable name
    - We've put the value `Samia` into the box

SLIDE: Creating a variable

- We **assign** a value to a variable with the equals sign: `=`
- Variable **name goes on the left**, **value on the right**
    - Character strings (words etc.) are enclosed in quotes

- **EXPLAIN DOUBLE- AND SINGLE-QUOTES**
  - Python accepts either double- or single-quotes, but you can't mix them.
- After assignment, if we refer to the variable `Name`, we get the value that's in the box, which is: `Samia`
- The **`print()`** function shows the value of a variable
  - **KEY POINTS: We refer to the name of the variable, but get its contents**

```
>>> name = "Samia"
>>> name
'Samia'
>>> print(name)
Samia
```

- **CHECK THAT EVERYONE GETS THE CONCEPT/SEES THE NAME**
- **Variable names** can include
  - letters, digits, and underscores
  - must not start with a number
  - are case sensitive

```
>>> myvar_1 = "Michael"
>>> myvar_1
'Michael'
>>> myvar-1 = "Michael"
  File "<stdin>", line 1
SyntaxError: can't assign to operator
>>> 1myvar = "Michael"
  File "<stdin>", line 1
    1myvar = "Michael"
         ^
SyntaxError: invalid syntax
>>> Myvar_1 = "Alex"
>>> Myvar_1
'Alex'
>>> myvar_1
'Michael'
```

🟥 Red sticky for a question or issue 🟩 Green sticky if complete

---

SLIDE: Working with variables

- **Lead the students** through the code:

```
>>> weight_kg_text = "weight in kilograms:"
>>> weight_kg = 55
```

- Note, we're assigning an integer now (no quotes), but **assignment is the same for all data items**
- **Python knows about several types of data**, including
  - integer numbers
  - floating point numbers
  - strings
- We can print `weight_kg` to see its value
- The `print()` function will also **take more than one argument**, separated by commas, and print them:

```
>>> print(weight_kg)
55
>>> print(weight_kg_text, weight_kg)
weight in kilograms: 55
```

- **Variables can be substituted by name wherever a value would go**, in calculations for example

```
>>> 2.2 * weight_kg
121.00000000000001
```

- We can **mix strings and variables** and even **do calculations with variables** inside the `print()` function:

```
>>> print("Weight in pounds:", 2.2 * weight_kg)
Weight in pounds: 121.00000000000001
```

- People may ask about floating point representations here - an introduction is at https://docs.python.org/3/tutorial/floatingpoint.html - this is on the Etherpad.
  - Most decimal fractions can't be represented exactly as binary fractions
- **Reassigning** to the same variable overwrites the old value

```
>>> weight_kg = 65.0
>>> print("Weight in kilograms is now:", weight_kg)
Weight in kilograms is now: 65.0
```

- Changing the value of one variable **does not automatically change** the values of other variables calculated using the original value

```
>>> weight_lb = 2.2 * weight_kg
>>> print('weight in kilograms:', weight_kg, 'and in pounds:', weight_lb)
weight in kilograms: 65.0 and in pounds: 143.0
>>> weight_kg = 100
>>> print('weight in kilograms:', weight_kg, 'and in pounds:', weight_lb)
weight in kilograms: 100 and in pounds: 143.0
```

- Although we changed the value in `weight_kg`, **`weight_lb` did not change** when we did so

---

SLIDE Exercise 01 (1min)

- **PUT THE EXERCISE SLIDE ON SCREEN**
- **MCQ: put up four colours of sticky notes**
- The solution is `2`

SLIDE Exercise 03 (1min)

- **MCQ: put up four colours of sticky notes**
- The code prints `Hopper Grace`

SECTION 03: Data Analysis

SLIDE Examine the data

- **SHOW THE TERMINAL ON SCREEN**
- In the terminal (`head` was used this morning)
- **Exit `Python` first!**
  - `Ctrl-D`
  - `quit()`

```
$ head data/inflammation-01.csv
0,0,1,3,1,2,4,7,8,3,3,3,10,5,7,4,7,7,12,18,6,13,11,11,7,7,4,6,8,8,4,4,5,7,
3,4,2,3,0,0
0,1,2,1,2,1,3,2,2,6,10,11,5,9,4,4,7,16,8,6,18,4,12,5,12,7,11,5,11,3,3,5,4,
4,5,5,1,1,0,1
0,1,1,3,3,2,6,2,5,9,5,7,4,5,4,15,5,11,9,10,19,14,12,17,7,12,11,7,4,2,10,5,
4,2,2,3,2,2,1,1
0,0,2,0,4,2,2,1,6,7,10,7,9,13,8,8,15,10,10,7,17,4,4,7,6,15,6,4,9,11,3,5,6,
3,3,4,2,3,2,1
0,1,1,3,3,1,3,5,2,4,4,7,6,5,3,10,8,10,6,17,9,14,9,7,13,9,12,6,7,7,9,6,3,2,
2,4,2,0,1,1
0,0,1,2,2,4,2,1,6,4,7,6,6,9,9,15,4,16,18,12,12,5,18,9,5,3,10,3,12,7,8,4,7,
3,5,4,4,3,2,1
0,0,2,2,4,2,2,5,5,8,6,5,11,9,4,13,5,12,10,6,9,17,15,8,9,3,13,7,8,2,8,8,4,2
,3,5,4,1,1,1
0,0,1,2,3,1,2,3,5,3,7,8,8,5,10,9,15,11,18,19,20,8,5,13,15,10,6,10,6,7,4,9,
3,5,2,5,3,2,2,1
0,0,0,3,1,5,6,5,5,8,2,4,11,12,10,11,9,10,17,11,6,16,12,6,8,14,6,13,10,11,4
,6,4,7,6,3,2,1,0,0
0,1,1,2,1,3,5,3,5,8,6,8,12,5,13,6,13,8,16,8,18,15,16,14,12,7,3,8,9,11,2,5,
4,5,1,4,1,2,0,0
```

- **Describe the data**: plain text, csv format

- **Can you tell what the data is?** (i.e. is this good practice for sharing data?)

    - One row per patient
    - One column per day
    - Values separated by commas

- **WE WANT TO FIND SUMMARY INFORMATION ABOUT INFLAMMATION BY PATIENT AND BY DAY**

SLIDE `Python` libraries

- Most programming languages have **libraries** (also known as **modules**, or **packages**).

    - **Libraries contain code that's not in the main language** but is useful for something specific - they can define **functions**, **data types**, and whole programs
        - Libraries are like **TOOLBOXES** - they provide extra equipment that helps solve specific problems
    - **Libraries add specific functionality to the language** - you can import as many as you need

- `Python` has libraries for many types of work and operations

    - **we'll use the `numpy` library** to work with this in Python
        - `numpy` is a good library to use for numerical work
    - **In `Python`, we call on libraries with the `import` statement**, when we need them
    - Importing a library is like getting a new piece of equipment out of the locker and onto the lab bench

- There are several repositories that host `Python` packages

    - `PyPI`
    - `conda`

- **Import and describe libraries**

```
>>> import numpy
```

- `numpy` is a library that provides functions and methods to **work with arrays and matrices**, such as those in your dataset

---

SLIDE Load data from file

- The `numpy` library gives us a function called **`loadtxt()` that loads tabular data from a file**

```
>>> numpy.loadtxt(fname='data/inflammation-01.csv', delimiter=',')
array([[ 0.,  0.,  1., ...,  3.,  0.,  0.],
       [ 0.,  1.,  2., ...,  1.,  0.,  1.],
       [ 0.,  1.,  1., ...,  2.,  1.,  1.],
       ...,
       [ 0.,  1.,  1., ...,  1.,  1.,  1.],
       [ 0.,  0.,  0., ...,  0.,  2.,  0.],
       [ 0.,  0.,  1., ...,  1.,  1.,  0.]])
```

- **`loadtxt()` expects two *arguments* or *parameters*** - values it needs to know to work

- The parameter `fname` takes the **path to the file we want to load**

- The parameter `delimiter` takes the **character that we think separates columns** in that file

- To use a `function` from a `library`, **the format is usually `library.function()`: *dotted notation***

- Here, our function is `numpy.loadtxt()`, and *Dotted notation* tells us that `loadtxt()` belongs to `numpy`

- `Python` will accept **double- *or* single-quotes** around strings (not both)

SLIDE Loaded data

- If we don't ask `Python` to do anything with the data, it just loads the data, then shows the data to us.
- The data display is truncated by default - *ellipses* (`...`) show rows and columns that were excluded for space
- Significant digits are not shown
- **NOTE that integers in the file have been converted to floating point numbers**
- **Ask the learners to assign the matrix to a variable called `data`

```
>>> data = numpy.loadtxt(fname="data/inflammation-01.csv", delimiter=",")
```

- Now when we execute the cell **we see no output**, but `data` now contains the array, which we can see by **printing the variable**

```
>>> print(data)
[[ 0.  0.  1. ...,  3.  0.  0.]
 [ 0.  1.  2. ...,  1.  0.  1.]
 [ 0.  1.  1. ...,  2.  1.  1.]
 ...,
 [ 0.  1.  1. ...,  1.  1.  1.]
 [ 0.  0.  0. ...,  0.  2.  0.]
 [ 0.  0.  1. ...,  1.  1.  0.]]
```

- Now that **the data is in memory** we can **manipulate it**.

SLIDE What is our data?

- We've loaded some data, but **what is it?**

```
>>> type(data)
<class 'numpy.ndarray'>
```

- You can always find out what *kind* of data is in a variable with the `type()` function
- **`Python` sees our data as a special `type`: `numpy.ndarray`**
- From *dotted notation* we see that `ndarray` belongs to (was defined in) the `numpy` library
- `ndarray` stands for "n-dimensional array" - so this is **an n-dimensional array from the `numpy` library**

SLIDE Data attributes

- **Creating our `data` array created a lot of information, too**
- We created **information about the array** called *attributes* (or *members*)
- This information belongs to `data` so is **accessed in the same way as a module function**, through *dotted notation*
    - The *attribute* belongs to `data` the way that a function belongs to a library

```
>>> print(data.dtype)
float64
>>> print(data.shape)
(60, 40)
```

- `print(data.dtype)` tells us that the **data type for values in the array** is: 64-bit floating point numbers
- `print(data.shape)` tells us that there are **60 rows and 40 columns** in the dataset

---

SLIDE Indexing arrays

- We often want to work with subsets of data
  - individual rows and columns
  - subgroups of rows and columns
  - **individual patients** (rows)
  - **individual days** (columns)
- Arrays are indexed by *row* and *column*, using *square bracket* notation
- To get a single element from the array, **index using *square bracket* notation** - row first, then column

```
>>> data[10, 10]
5.0
```

- Most of us **naturally count starting at one** (1, 2, 3, 4)
  - In **`Python`** **we count from zero**, so the first element in the array is `data[0, 0]`

```
>>> print('first value in data:', data[0, 0])
first value in data: 0.0
>>> print('middle value in data:', data[30, 20])
middle value in data: 13.0
```

- **REFER TO IMAGE**
  - If we have a 3x3 array in Python, the rows are numbered 0, 1, 2 - and the columns are numbered 0, 1, 2
  - **In general** for a `M x N` array, the rows are numbered 0, .., M-1, and the columns are numbered 0, .., N-1

🟥 Red sticky for a question or issue 🟩 Green sticky if complete

---

SLIDE Slicing arrays

- We've seen how to get a single *element* from an array, but we often want to work with a whole row or column, or a **slice out of the array**
- To get a section from the array, index using *square bracket* notation - but specify start and end points, separated by a colon

```
>>> print(data[0:4, 0:10])
[[ 0.  0.  1.  3.  1.  2.  4.  7.  8.  3.]
 [ 0.  1.  2.  1.  2.  1.  3.  2.  2.  6.]
 [ 0.  1.  1.  3.  3.  2.  6.  2.  5.  9.]
 [ 0.  0.  2.  0.  4.  2.  2.  1.  6.  7.]]
>>> print(data[5:10, 0:10])
[[ 0.  0.  1.  2.  2.  4.  2.  1.  6.  4.]
 [ 0.  0.  2.  2.  4.  2.  2.  5.  5.  8.]
 [ 0.  0.  1.  2.  3.  1.  2.  3.  5.  3.]
 [ 0.  0.  0.  3.  1.  5.  6.  5.  5.  8.]
 [ 0.  1.  1.  2.  1.  3.  5.  3.  5.  8.]]
>>> print(data[2:4, 2:4])
[[ 1.  3.]
 [ 2.  0.]]
```

- The slice `0:4` means start at index zero and go up to, but not including, index 4. So it takes elements `0, 1, 2, 3` (four elements)

🟥 Red sticky for a question or issue 🟩 Green sticky if complete

---

SLIDE Another slice, please!

- If we don't specify a start for the slice, `Python` assumes the first element is meant (element zero)
- If we don't specify an end for the slice, `Python` assumes the last element is meant
- To get the top-right corner of the array, we can specify `data[:3, 36:]`

```
>>> small = data[:3, 36:]
>>> print(small)
[[2. 3. 0. 0.]
 [1. 1. 0. 1.]
 [2. 2. 1. 1.]]
```

- **QUESTION: What does `:` on its own mean?**

```
>>> data[:,0]
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0.])
>>> print(data[0:2, :])
[[  0.   0.   1.   3.   1.   2.   4.   7.   8.   3.   3.   3.  10.   5.
    7.   4.   7.   7.  12.  18.   6.  13.  11.  11.   7.   7.   4.   6.
    8.   8.   4.   4.   5.   7.   3.   4.   2.   3.   0.   0.]
```

```
 [  0.   1.   2.   1.   2.   1.   3.   2.   2.   6.  10.  11.   5.   9.
    4.   4.   7.  16.   8.   6.  18.   4.  12.   5.  12.   7.  11.   5.
   11.   3.   3.   5.   4.   4.   5.   5.   1.   1.   0.   1.]]
```

- `data[:,0]` returns the first column
- `data[0:2, :]` returns the first two rows
- **`:` represents all values on an array axis**

SLIDE Exercise 03 (1min)

- **MCQ: put up four colours of sticky notes**

- The value is `oxyg`, number `1`

SLIDE Array operations

- Arithmetic operations on `array`s are **performed elementwise.**
  - `numpy` is particularly good at this sort of calculation

```
>>> doubledata = data * 2.0
```

- This operation multiplies every array element by 2.0.
- **Look at the top right corner of the original array**

```
>>> print("original:\n", data[:3, 36:])
original:
 [[ 2.  3.  0.  0.]
 [ 1.  1.  0.  1.]
 [ 2.  2.  1.  1.]]
```

- **Look at the top right corner of the doubled array**

```
>>> print("doubledata:\n", doubledata[:3, 36:])
doubledata:
 [[ 4.  6.  0.  0.]
 [ 2.  2.  0.  2.]
 [ 4.  4.  2.  2.]]
```

- We can also do arithmetic with two arrays of the same shape.

```
>>> tripledata = doubledata + data
>>> print("tripledata[:3, 36:])
 [[ 6.  9.  0.  0.]
 [ 3.  3.  0.  3.]
 [ 6.  6.  3.  3.]]
```

SLIDE `numpy` functions

- `numpy` provides functions that can perform *more complex* operations on arrays
- Some of the **`numpy` operations include statistical summaries: `.mean()`, `.min()`, `.max()` etc.**

```
>>> print(numpy.mean(data))
6.14875
```

- We can asssign the output from these functions to variables
- **By default, these functions give summaries of the whole array**
- **Introduce multiple assignment on one line, or use three lines**

```
>>> maxval, minval, stdval = numpy.max(data), numpy.min(data),
numpy.std(data)
>>> print('maximum inflammation:', maxval)
maximum inflammation: 20.0
>>> print('minimum inflammation:', minval)
minimum inflammation: 0.0
>>> print('standard deviation:', stdval)
standard deviation: 4.61383319712
```

🟥 Red sticky for a question or issue 🟩 Green sticky if complete

---

SLIDE Summary for one patient

- **We want to get summaries patient-by-patient (row-by-row)?**
- We can extract a single row into a *temporary variable*, and calculate the mean for that variable

```
>>> patient_0 = data[0, :] # temporary variable
>>> print('maximum inflammation for patient 0:', numpy.max(patient_0))
maximum inflammation for patient 0: 18.0
```

- **NOTE: that comments are preceded with a hash `#` and can be placed after a line of code**

- **EXPLAIN: why leaving comments is good (can do that in all code - not just Jupyter notebooks)**
- We can also **apply the `numpy` function directly**, without creating a variable

```
>>> print('maximum inflammation for patient 2:', numpy.max(data[2, :]))
maximum inflammation for patient 2: 19.0
```

SLIDE Summary for all patients

- But **what if we want to know about all patients?**
- Or **what if we want an average inflammation per day?**
- Writing one line per row, or per column, is likely to lead to mistakes and typos
  - Use `numpy` functions
  - **We can specify which axis a function applies to**
- **`numpy` functions take an `axis=` parameter** which controls the axis for summary statistic calculations.
  - Specifying `axis=0` makes the function work on columns (days)
  - **working on values 'by' row/row-wise**
  - Specifying `axis=1` makes the function work on rows (patients)
  - **working on values 'by' column/column-wise**

```
>>> print(numpy.mean(data, axis=0))
[  0.           0.45         1.11666667   1.75         2.43333333   3.15
   3.8          3.88333333   5.23333333   5.51666667   5.95         5.9
   8.35         7.73333333   8.36666667   9.5          9.58333333
  10.63333333  11.56666667  12.35        13.25        11.96666667
  11.03333333  10.16666667  10.           8.66666667   9.15         7.25
   7.33333333   6.58333333   6.06666667   5.95         5.11666667   3.6
   3.3          3.56666667   2.48333333   1.5          1.13333333
   0.56666667]
>>> print(numpy.max(data, axis=1))
[ 18.  18.  19.  17.  17.  18.  17.  20.  17.  18.  18.  18.  17.  16.
17.
  18.  19.  19.  17.  19.  19.  16.  17.  15.  17.  17.  18.  17.  20.
17.
  16.  19.  15.  15.  19.  17.  16.  17.  19.  16.  18.  19.  16.  19.
18.
  16.  19.  15.  16.  18.  14.  20.  17.  15.  17.  16.  17.  19.  18.
18.]
```

- `numpy.mean(data, axis=0)` gives us the mean inflammation each day
- `numpy.max(data, axis=1)` gives us the maximum inflammation per patient

🟥 Red sticky for a question or issue 🟩 Green sticky if complete

---

SECTION 04: Visualisation

SLIDE Visualisation

> "The purpose of computing is insight, not numbers" - Richard Hamming

- The best way to gain insight is often to visualise data.
- Visualisation is a large topic that deserves more attention
  - **We're just scratching the surface, here**

SLIDE Graphics package `matplotlib`

- `matplotlib` is the *de facto* standard/base plotting library in `Python`

```
>>> import matplotlib.pyplot
```

- We use `matplotlib.pyplot` to make the interaction a bit more like `matlab`
- **NOTE: THERE IS CURRENTLY AN ISSUE WITH MATPLOTLIB ON macOS**
  - There may be an `Abort Trap: 6` message
  - See this issue
  - A workaround is below

```
import matplotlib
matplotlib.use('TkAgg')
import matplotlib.pyplot
```

🟥 Red sticky for a question or issue 🟩 Green sticky if complete

---

SLIDE `matplotlib.pyplot.imshow()`

- The `.imshow()` function **converts matrix values into an image**

```
>>> image = matplotlib.pyplot.imshow(data)
>>> matplotlib.pyplot.show()
```

- Here, small values are blue, and large values are yellow (**you can change this colour scheme with other settings…**)
- From the image, we can see **inflammation rising and falling** over a 40-day period for all patients.
- **QUESTION: does this look reasonable?**

SLIDE `matplotlib.pyplot.plot()`

- **`.plot()` shows a conventional line graph**
- We're going to use it to **plot the average inflammation level on each day of the study**
- We'll create the variable `ave_inflammation` and use `numpy.mean()` on axis `0` of the data

```
>>> ave_inflammation = numpy.mean(data, axis=0)  # mean inflammation per day
>>> ave_plot = matplotlib.pyplot.plot(ave_inflammation)
>>> matplotlib.pyplot.show()
```

- **NOTE: ask students if the data looks reasonable?**
  - The **data does not look reasonable**. Biologically, we expect a sharp rise in inflammation, followed by a slow tail-off
- **NOTE:** Demonstrate closing the window (cross/red dot in top left)

SLIDE Investigating data

- Since **our plot of `.mean()` values looks artificial, let's check on other statistics** to see if we can see what's going on.
- We'll plot the maximum value by day

```
>>> max_plot = matplotlib.pyplot.plot(numpy.max(data, axis=0))
>>> matplotlib.pyplot.show()
```

- **NOTE we're not defining a temporary variable to hold our calculation, this time**

```
>>> min_plot = matplotlib.pyplot.plot(numpy.min(data, axis=0))
>>> matplotlib.pyplot.show()
```

- **Ask students if the data looks reasonable?**

  - The data looks very artificial. The maxima are completely smooth, but the minima are a step function.

- **NOTE: we would not have noticed this without visualisation**

SLIDE Exercise 04 (5min)

```
>>> std_plot = matplotlib.pyplot.plot(numpy.std(data, axis=0))
>>> matplotlib.pyplot.show()
```

🟥 Red sticky for a question or issue 🟩 Green sticky if complete

---

SLIDE FIGURES AND SUBPLOTS

- **NOTE: WE'RE WORKING IN A SCRIPT FOR THE FIRST TIME**
- **Exit `Python` (`Ctrl-D` or `quit()`)**
- **Open a new script called `subplots.py`**

```
nano subplots.py
```

- **You're now in the `nano` editor**
- We write the `Python` code here, then save it, exit, and run it with `python subplots.py`.
- **DESCRIBE SCRIPT**
- First we **import packages**

```
import numpy
import matplotlib.pyplot
```

- Next we **load data**
  - Comments tell us why/what we are doing

```
# Load inflammation data
data = numpy.loadtxt(fname='data/inflammation-01.csv', delimiter=',')
```

- We can put all three plots we just drew into a single figure

  - **DRAW THIS OUT ON THE WHITEBOARD AS WE GO**
- To do this, we use `matplotlib` to **create a figure**, and put it in a variable called `fig`
  - the `figsize` argument sets the `(width, height)` of the figure in inches

```
# Create a figure
fig = matplotlib.pyplot.figure(figsize=(10.0, 3.0))
```

- We then **create three *axes***
  - these are the variables that hold the individual plots
  - one axis per plot
- Using the `.add_subplot()` function, we need to specify three things: -* number of rows, number of columns, which cell this figure goes into
  - **THIS NEEDS TO BE DRAWN OUT ON THE BOARD**

```
# Add subplots for statistical summaries
axes1 = fig.add_subplot(1, 3, 1)
axes2 = fig.add_subplot(1, 3, 2)
axes3 = fig.add_subplot(1, 3, 3)
```

- Once we've created our plot axes, we can add labels and plots to each of them in turn

```
# Add y-axis label to each subplot
axes1.set_ylabel('average')
axes2.set_ylabel('max')
axes3.set_ylabel('min')
```

- Plot axes properties are usually changed using the `.set_<something>()` syntax

  - Here we're changing only the label on the *y*-axis

- We can plot on an axis by using its `.plot()` function

  - As before, we can pass the output from the `numpy.max()` function directly

```
# Plot the summary data
axes1.plot(numpy.mean(data, axis=0))
axes2.plot(numpy.max(data, axis=0))
axes3.plot(numpy.min(data, axis=0))
```

- We'll tighten up the presentation by using `fig.tight_layout()`
  - this is a function that moves the axes until they are visually pleasing.

```
# Tidy the figure
fig.tight_layout()
```

- Finally, we'll show the figure in the interactive window

```
# Show figure in the interactive window
matplotlib.pyplot.show()
```

- **Write the file**
- **Save the file**
- **Exit to terminal**
- Now we run the script

```
python subplots.py
```

- The command `python subplots.py` tells Python to execute the code in the file `subplots.py`
  - We see the interactive display window showing the three plots we expect
- **This is the most demanding code you have written, so far! ROUND OF APPLAUSE FOR YOURSELVES!**

SLIDE Exercise 05 (5min)

- Note that it helps to change `figsize`
- Otherwise the only change is in `add_subplot()`
- `cp` the file to prepare for a new script

```
cp subplots.py exercise_05.py
nano exercise_05.py
```

- New script:
  - **NOTE:** Best to edit in-place rather than type everything out again

```
import numpy
import matplotlib.pyplot

# Load inflammation data
```

```
data = numpy.loadtxt(fname='data/inflammation-01.csv', delimiter=',')

# Create a figure
fig = matplotlib.pyplot.figure(figsize=(3.0, 10.0))

# Add subplots for statistical summaries
axes1 = fig.add_subplot(3, 1, 1)
axes2 = fig.add_subplot(3, 1, 2)
axes3 = fig.add_subplot(3, 1, 3)

# Add y-axis label to each subplot
axes1.set_ylabel('average')
axes2.set_ylabel('max')
axes3.set_ylabel('min')

# Plot the summary data
axes1.plot(numpy.mean(data, axis=0))
axes2.plot(numpy.max(data, axis=0))
axes3.plot(numpy.min(data, axis=0))

# Tidy the figure
fig.tight_layout()

# Show figure in the interactive window
matplotlib.pyplot.show()
```

- Execute script

```
python subplots.py
```

🟥 Red sticky for a question or issue 🟩 Green sticky if complete

- You've already learned how to:
  - read in a numerical dataset
  - extract and work with specific subsets of the data
  - calculate statistics on a dataset
  - plot those statistics
  - write and run a script to generate a multipart graph
- **NOW, TO DO *EVEN MORE* INTERESTING THINGS, WE NEED TO LEARN A LITTLE BIT MORE ABOUT PROGRAMMING**

SECTION 05: `for` loops

SLIDE Motivation

- We wrote code that plots values of interest from our dataset
- **BUT** soon we're going to get **dozens of datasets** to analyse
- So, we need to tell the computer to **repeat our plots and analysis on each dataset**
- We're going to do this with `for` loops
- **NOTE: `for` loops are a fundamental method for program control across nearly every programming language**
- **NOTE: `for` loops in python work just like those the learners saw in `bash`, but are syntactically different**

SLIDE Spelling bee

- What if we wanted to process a word, like 'lead', one letter at a time?
    - We might be passing each letter to a display, for instance
    - Here we're going to put each letter on its own line
- How could we do it?
- **START UP A PYTHON CONSOLE**

```
python
```

```
>>> word = "lead"
```

- We could *index* each letter in turn (just like elements of an array)

```
>>> print(word[0])
l
>>> print(word[1])
e
>>> print(word[2])
a
>>> print(word[3])
d
```

- But this has some problems - **ASK LEARNERS WHAT PROBLEMS THEY SEE**
    - The commands interrupt the letters, but we could write this as a script and avoid that problem.
    - The **approach doesn't scale** - what if our word is hundreds of letters long? Or we want to do it for a book?

  - **What if our word is longer than the indices?** We don't get all the data; if it's shorter, we get an error.
  - The code is **hard to modify** - if we wanted to add an asterisk to each letter when we print it, we'd have to make four changes to the code.
  - **demonstrate with `oxygen` and `tin` - MODIFY THE WORD IN PLACE**

```
>>> word = "tin"
>>> print(word[3])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

SLIDE `for` loops

- `for` loops perform an operation ***for* every item *in* a collection**

- **NOTE: importance of the `tab` character and syntactic whitespace**

```
>>> word = "lead"
>>> for char in word:
...     print(char)
...
l
e
a
d
```

- Why is this better? **ASK THE LEARNERS**
  - **It's shorter code** (less opportunity for error)
  - **It's more flexible and robust** - it doesn't matter how long `word` is, the code will still spell it out one letter at a time

```
>>> word = "oxygen"
>>> for char in word:
...     print(char)
...
o
x
y
g
e
n
```

SLIDE `for` loop syntax

- The general loop syntax is defined by a `for` statement, and a *code block*
    - The `for` loop **statement ends in a colon, `:`**
    - The *code block* is **indented** with a `tab` (`\t`)
- **Everything indented immediately below the `for` statement is part of the `for` loop**
    - this code is executed on each cycle of the loop
    - it's called the **CODE BLOCK**
- **There is no command or statement to signify the end of the loop body - only a change in indentation**
- This is quite different from most other languages (and some people hate `Python` because of it)
- **If further example needed, put the code below in a script**

```python
for char in word:
    print(char)
    print("I'm in the loop")
    # This is a comment
    print("Still in the loop")

    print("I'm in the loop as well")
print("Not in the loop")
```

SLIDE Counting with a `for` loop

- Code in a `for` loop can see and modify variables defined outside the loop

```python
>>> length = 0
>>> for vowel in 'aeiou':
...     length = length + 1
...
>>> print("There are", length, "vowels")
There are 5 vowels
```

- The `length = length + 1` line adds 1 to the value in the variable `length`, and then assigns the new number to the variable `length`
- The variable `length` is available inside and outside the loop
    - (The variable `vowel` still exists outside the loop and contains the final value in the string `"aeiou"`: `"u"`)
- **Ask the learners what output they expect**
- Talk through the operations of the loop, if necessary

SLIDE `for` loop variables

- The *loop variable* gets updated once per cycle of the loop
- The *loop variable* also still exists once the loop is finished

```
>>> letter = "z"
>>> print(letter)
z
>>> for letter in "abc":
...     print(letter)
...
a
b
c
>>> print("after the loop, letter is:", letter)
after the loop, letter is: c
```

- **ASK THE LEARNERS WHAT OUTPUT THEY EXPECT (RED/GREEN STICKY FOR z or c)**
    - The value of `letter` is `c`, the last updated value in the loop - not `z`, which would be the case if the loop variable only had scope within the loop
- We don't *need* to use a loop to find the length of a string (or other variable)
    - Python provides the **`len()`** function to do this

```
len("aeiou")
```

SLIDE `range()`

- The `range()` function creates a **sequence of numbers.**
- The sequence depends on the number and value of arguments given

```
>>> seq = range(3)
>>> print("Range is:", seq)
Range is: range(0, 3)
>>> for val in seq:
...     print(val)
...
0
1
2
```

- **Substitute other ranges and run again**

```
>>> seq = range(2, 5)
>>> print("Range is:", seq)
```

```
Range is: range(2, 5)
>>> for val in seq:
...     print(val)
...
2
3
4
>>> seq = range(3, 10, 3)
>>> print("Range is:", seq)
Range is: range(3, 10, 3)
>>> for val in seq:
...     print(val)
...
3
6
9
>>> seq = range(10, 0, -1)
>>> print("Range is:", seq)
Range is: range(10, 0, -1)
>>> for val in seq:
...     print(val)
...
10
9
8
7
6
5
4
3
2
1
```

- A single value *n* gives the sequence `[0, ..., n-1]`
- Two values: *m, n* gives the sequence `[m, ..., n-1]`
- Three values: *m, n, p* gives the sequence `[m, m+p, ..., n-1]` and skips `n-1` if it's not in the sequence.
- **NOTE: `range()` returns a `range` type that can be iterated over.**

SLIDE Exercise 06 (5min)

- Tell learners that you can **add strings**

```
>>> instr = "Newton"
>>> outstr = ""
>>> for char in instr:
...     outstr = char + outstr
...
```

```
>>> print(outstr)
notweN
```

- There's a slice solution to the problem:

```
>>> instr[::-1]
'notweN'
```

SECTION 06: `list`s

SLIDE Lists

- Lists are used to **store multiple values**
  - Unlike `numpy` arrays, they are **built in to the language**
- Lists are defined as ordered lists of values
  - enclosed in **square brackets**
  - separated by commas

```
>>> odds = [1, 3, 5, 7]
>>> print("odds are:", odds)
odds are: [1, 3, 5, 7]
```

- They can be **indexed and sliced**, as seen for arrays

```
>>> print('first and last:', odds[0], odds[-1])
first and last: 1 7
>>> print(odds[2:])
[5, 7]
```

- They can be iterated over in `for' loops, just like strings:

```
>>> for number in odds:
...     print(number)
...
1
3
5
7
```

SLIDE Mutability

- `Python` has a concept of **mutability**.
    - Items that can be modified in-place are *mutable*.
    - Those that can't are *immutable*.
- Lists are *mutable*, strings are *immutable*.
    - `list`s and `string`s are both sequences, **BUT** you can change the elements in a list, after it is created: **lists are mutable**

```
>>> names = ["Curie", "Darwing", "Turing"] # typo in Darwin's name
>>> print("names is originally:", names)
names is originally: ['Curie', 'Darwing', 'Turing']
```

- **We have a typo - let's correct it**

```
>>> names[1]
'Darwing'
>>> names[1] = 'Darwin'     # correct the name
>>> print('final value of names:', names)
final value of names: ['Curie', 'Darwin', 'Turing']
```

- We've successfully modified this list *in-place*
- `string`s however are **NOT** mutable

```
>>> name = names[1]
>>> name
'Darwin'
>>> name[0] = "B"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

- We can't change individual characters in a string - we can only replace the old string with a new string.

SLIDE Changer danger

- **There are risks associated with modifying lists in-place**
- Rather than make copies of lists, when assigned to more than one variable, `Python` will make *reference* to the original list

```
>>> my_list = [1, 2, 3, 4]
>>> your_list = my_list
>>> print("my_list:", my_list)
my_list: [1, 2, 3, 4]
>>> my_list[1] = 0
>>> print("my_list:", my_list)
my_list: [1, 0, 3, 4]
```

- **ASK LEARNERS WHAT THEY THINK `your_list` contains**

```
>>> print("your list:", your_list)
your list: [1, 0, 3, 4]
```

- **If two variables refer to the same list, any changes to that list are reflected in both variables.**

SLIDE List copies

- To avoid this kind of effect:
  - make a *copy* of a `list` by *slicing* it, or by using the `list()` function that returns a new list

```
>>> my_list = [1, 2, 3, 4]           # original list
>>> your_list = my_list[:]           # copy 1
>>> your_other_list = list(my_list)  # copy 2
>>> print("my_list:", my_list)
my_list: [1, 2, 3, 4]
>>> my_list[1] = 0
>>> print("my_list:", my_list)
my_list: [1, 0, 3, 4]
>>> print("your_list:", your_list)
your_list: [1, 2, 3, 4]
>>> print("your_other_list:", your_list)
your_other_list: [1, 2, 3, 4]
```

🟥 Red sticky for a question or issue 🟩 Green sticky if complete

---

SLIDE `list` functions

- **`list`s are `Python` objects and have a number of useful functions (called *methods*) to modify their contents**

- `.append()` adds a value to the end of the list

```
>>> print(odds)
[1, 3, 5, 7]
>>> odds.append(9)
>>> print("odds after adding a value:", odds)
odds after adding a value: [1, 3, 5, 7, 9]
```

- `.reverse()` reverses the order of list items **in place**

```
>>> odds.reverse()
>>> print("odds after reversing the list:", odds)
odds after reversing the list: [9, 7, 5, 3, 1]
```

- `.pop()` returns the last item in the list, and removes it from the list

```
>>> odds.pop()
1
>>> print("odds after popping:", odds)
odds after popping: [9, 7, 5, 3]
```

SLIDE Overloading

- *Overloading* refers to an *operator* (e.g. `+`) having more than one meaning, depending on the thing it operates on.

```
>>> vowels = ['a', 'e', 'i', 'o', 'u']
>>> vowels_welsh = ['a', 'e', 'i', 'o', 'u', 'w', 'y']
>>> print(vowels + vowels_welsh)
['a', 'e', 'i', 'o', 'u', 'a', 'e', 'i', 'o', 'u', 'w', 'y']
```

- We can add (`+`) and **even multiply** (`*`) lists, even though they're not really arithmetic operations **NOTE: multiplication of lists does not work like multiplication of `numpy` arrays**

```
>>> counts = [2, 4, 6, 8, 10]
>>> counts + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate list (not "int") to list
>>> repeats = counts * 2
>>> print(repeats)
[2, 4, 6, 8, 10, 2, 4, 6, 8, 10]
```

- **Ask the learners what 'multiplication' (`*`) does for lists**

SECTION 07: Making choices

SLIDE Conditionals

- We often want the computer to perform different actions depending on some value in the data
  - this often turns out to be: do `<something>` **`if`** some condition is **true**
- To do this, we can use an `if` **statement**
  - **`if` statements end in a colon (`:`)**
  - **they also have a *condition*** - the *condition* is evaluated and, if found to be `true`, the code block is executed
  - The **code block** is *indented* as was the case with the `for` loop
- **This is an almost universal construct in programming languages**

```
>>> num = 37
>>> if num > 100:
...     print('greater')
...
>>> num = 149
>>> if num > 100:
...     print('greater')
...
```

- **Any condition** that might evaluate to `True` or `False` can be used:
- **SHOW A DIFFERENT TEST**

```
>>> if 'atlas' == 'atlas':
...     print("The same")
...
The same
```

SLIDE `if-else` statements

- **An `if` statement executes code if the condition evaluates as `true`**
  - But **what if the condition evaluates as `false`?**
- The **`else` structure** is like the `if` structure
  - it **ends in a colon (`:`)**

  - the **indented code block beneath it executes if the condition is `false`**

```
>>> num = 37
>>> if num > 100:
...     print('greater')
... else:
...     print('not greater')
...
not greater
```

- Where, in the last example, there was no output because `num > 100` evaluated as `False`, now that is caught by the `else:` block.

SLIDE Conditional logic

- **OPTIONALLY SHOW THIS SLIDE**

- Describe flowchart

SLIDE `if-elif-else` conditionals

- We can **chain conditional tests together with `elif` (short for `else if`)**
- The `elif` statement structure is the same as the `if` statement structure
  - the indented code block is executed if the condition is true, and **no previous conditions have been met**.

```
>>> num = -3
>>> if num > 0:
...     print(num, "is positive")
... elif num == 0:
...     print(num, "is zero")
... else:
...     print(num, "is negative")
...
-3 is negative
```

- **NOTE: the test for equality is a double-equals!**
  - Choosing the wrong one can be a bug!

```
>>> i = 10
>>> i == 10
True
>>> i = 9
>>> i == 10
False
```

SLIDE COMBINING CONDITIONS

- We can **combine conditions using *Boolean Logic***
  - Operators include `and`, `or` and `not`

```
>>> if (1 > 0) and (-1 > 0):
...     print('both parts are true')
... else:
...     print('at least one part is false')
...
at least one part is false
>>> if (4 > 0) and (2 > 0):
...     print('both parts are true')
... else:
...     print('at least one part is false')
...
both parts are true
>>> if (1 > 0) or (-1 > 0):
...     print("at least one part is true")
... else:
...     print("both parts are false")
...
at least one part is true
```

🟥 Red sticky for a question or issue 🟩 Green sticky if complete

---

SLIDE Exercise 07 (1min)

- **MCQ: Put up four stickies**

- Solution: `C`

- **NOTE: There are two `elif`s and no `else`**

SLIDE More about operators

- These are **two operators** you will meet and use frequently
  - `==` **(double-equals) is the equality operator**, and returns `True` if the left-hand-side value is equal to the right-hand-side value
  - **we've already been using this**

```
>>> print(1 == 1)
True
```

```
>>> print(1 ==2)
False
```

- `in` is the **membership operator**, and returns `True` if the left-hand-side value is in the right-hand-side value (which should be a collection)

```
>>> print('a' in 'toast')
True
>>> print('b' in 'toast')
False
>>> print(1 in [1, 2, 3])
True
>>> 1 in 123
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: argument of type 'int' is not iterable
>>> print(1 in range(3))
True
>>> 1 in range(0, 3, 2)
False
>>> 1 in range(2, 10)
False
```

- **WE'RE NOW ALMOST READY TO WRITE A MORE COMPLEX SCRIPT TO ANALYSE MULTIPLE DATA FILES**

SECTION 08: Analysing multiple files

SLIDE Analysing multiple files

- We have **received several files of data** from the inflammation studies
  - we would like to **perform the same operations on each of them**.
- We have **learned how to open files, read data, visualise data, loop over data, and make decisions** based on that content.
- Now we need to know how to **interact with the *filesystem*** to get our data files.
- To interact with the filesystem, **we need to import the `os` module**
- This allows us to interact with the filesystem in the same way, regardless of the operating system we work on! **INTEROPERABILITY AND REPRODUCIBILITY**

SLIDE `os.listdir()`

- The `os.listdir()` function lists the contents of a directory

- **SAVE AND EXIT SCRIPT**

```
python
```

```
>>> import os
>>> os.listdir('.')
['subplots.py', 'code', 'exercise_05.py', 'analyse_files.py', 'data']
```

Our data is in the `'data'` directory

```
>>> os.listdir('data')
['inflammation-05.csv', 'inflammation-11.csv', 'inflammation-10.csv',
'inflammation-04.csv', 'inflammation-12.csv', 'inflammation-06.csv',
'inflammation-07.csv', 'inflammation-03.csv', 'small-02.csv', 'small-
03.csv', 'inflammation-02.csv', 'small-01.csv', 'inflammation-01.csv',
'inflammation-09.csv', 'inflammation-08.csv']
```

- This gives us a **list** and we know how to deal with lists
- **We only want `inflammation` data** so we would like to ignore the `small` files
  - We want to turn the list from `os.listdir()` into a list that contains only `inflammation*` files: **use `for` loop and `if` to filter**
  - The list can be filtered with a `for` loop or *list comprehension*

```
>>> for file in os.listdir('data'):
...     if 'inflammation' in file:
...             print(file)
...
inflammation-05.csv
inflammation-11.csv
inflammation-10.csv
inflammation-04.csv
inflammation-12.csv
inflammation-06.csv
inflammation-07.csv
inflammation-03.csv
inflammation-02.csv
inflammation-01.csv
inflammation-09.csv
inflammation-08.csv
```

- We'd like to work with this set of files, so we store it in a variable, called `files`.
  - A suitable data type here is a `list`, and we can create an empty list then populate it one file at a time, using `.append()`

```
>>> files = []
>>> for file in os.listdir('data'):
...     if 'inflammation' in file:
...             files.append(file)
...
>>> print(files)
['inflammation-05.csv', 'inflammation-11.csv', 'inflammation-10.csv',
'inflammation-04.csv', 'inflammation-12.csv', 'inflammation-06.csv',
'inflammation-07.csv', 'inflammation-03.csv', 'inflammation-02.csv',
'inflammation-01.csv', 'inflammation-09.csv', 'inflammation-08.csv']
```

SLIDE `analyse_files.py`

- **Let's start to write our new script**
  - **EXIT THE CONSOLE**
  - **OPEN THE SCRIPT**
- We're going to write a new script to do this
  - **Exit `Python` console**
  - **Start new script**

```
nano analyse_files.py
```

- **Put imports at the top of the script**

```
# import packages
import matplotlib.pyplot
import numpy as np
import os

# Get a list of inflammation data files
files = []
for fname in os.listdir('data'):
  if 'inflammation' in fname:
    files.append(fname)
print("Inflammation data files:", files)
```

- **EXIT EDITOR AND RUN THE SCRIPT**

```
$ python analyse_files.py
Inflammation data files: ['inflammation-05.csv', 'inflammation-11.csv',
'inflammation-10.csv', 'inflammation-04.csv', 'inflammation-12.csv',
'inflammation-06.csv', 'inflammation-07.csv', 'inflammation-03.csv',
'inflammation-02.csv', 'inflammation-01.csv', 'inflammation-09.csv',
'inflammation-08.csv']
```

- **QUESTION: what's wrong with the filenames?**
  - The files are actually in the `data` directory

SLIDE `os.path.join()`

- The **`os.listdir()` function only returns filenames**, not the *path* (relative or absolute) to those files.

  - **WE NEED THE FULL PATH TO A FILE TO BE ABLE TO USE IT**

- To **construct a path**, we can use the `os.path.join()` function.

  - `os.path.join()` takes directory and file names, and returns a path built from them as a string, suitable for the underlying operating system.
  - **This is useful for making code shareable and usable on all OS/computers**

- **START PYTHON CONSOLE**

```
python
```

```
>>> os.path.join('parent', 'child', 'file.txt')
'parent/child/file.txt'
>>> os.path.join('data', 'inflammation-01.csv')
'data/inflammation-01.csv'
```

- **CLOSE CONSOLE AND OPEN EDITOR**

```
nano analyse_files.py
```

```
# Get a list of inflammation data files
files = []
for fname in os.listdir('data'):
  if 'inflammation' in fname:
    files.append(os.path.join('data', fname))
print("Inflammation data files:", files)
```

- **SAVE AND EXIT SCRIPT, THEN RUN**

```
$ python analyse_files.py
Inflammation data files: ['data/inflammation-05.csv', 'data/inflammation-
11.csv', 'data/inflammation-10.csv', 'data/inflammation-04.csv',
'data/inflammation-12.csv', 'data/inflammation-06.csv',
'data/inflammation-07.csv', 'data/inflammation-03.csv',
'data/inflammation-02.csv', 'data/inflammation-01.csv',
'data/inflammation-09.csv', 'data/inflammation-08.csv']
```

SLIDE Visualising the data

- Now **we have all the tools we need** to load all the inflammation data files, and visualise the mean, minimum and maximum values in an array of plots.
  - We can get a **list of paths to the data files** with `os` and a *list comprehension*
  - We can **load data from a file** with `numpy.loadtxt()`
  - We can **calculate summary statistics** with `numpy.mean()`, `numpy.max()`, etc.
  - We can **create figures** with `matplotlib`, and arrays of figures with `.add_subplot()`

SLIDE Visualisation code

- **BUILD THE CODE IN STAGES**
  - **OPEN THE SCRIPT IN THE EDITOR**

```
nano analyse_files.py
```

- **1 - show that we see each filename in turn**

```
# Analyse each file in turn
for fname in files:
    print("Analysing", fname)
```

- **2 - load the data in each file**

```
# Analyse each file in turn
for fname in files:
    print("Analysing", fname)

    # load data
    data = numpy.loadtxt(fname=fname, delimiter=',')
```

- **3 - create a figure for each file**

```
# Analyse each file in turn
for fname in files:
    print("Analysing", fname)

    # load data
    data = numpy.loadtxt(fname=fname, delimiter=',')

    # create figure and three axes
    fig = matplotlib.pyplot.figure(figsize=(10.0, 3.0))
    axes1 = fig.add_subplot(1, 3, 1)
    axes2 = fig.add_subplot(1, 3, 2)
    axes3 = fig.add_subplot(1, 3, 3)
```

- **4 - decorate the axes**

```
# Analyse each file in turn
for fname in files:
    print("Analysing", fname)

    # load data
    data = numpy.loadtxt(fname=fname, delimiter=',')

    # create figure and three axes
    fig = matplotlib.pyplot.figure(figsize=(10.0, 3.0))
    axes1 = fig.add_subplot(1, 3, 1)
    axes2 = fig.add_subplot(1, 3, 2)
    axes3 = fig.add_subplot(1, 3, 3)

    # decorate the axes
    axes1.set_ylabel('average')
    axes2.set_ylabel('maximum')
    axes3.set_ylabel('minimum')
```

- **5 - plot the data**

```
# Analyse each file in turn
for fname in files:
    print("Analysing", fname)

    # load data
    data = numpy.loadtxt(fname=fname, delimiter=',')

    # create figure and three axes
    fig = matplotlib.pyplot.figure(figsize=(10.0, 3.0))
    axes1 = fig.add_subplot(1, 3, 1)
    axes2 = fig.add_subplot(1, 3, 2)
    axes3 = fig.add_subplot(1, 3, 3)

    # decorate the axes
    axes1.set_ylabel('average')
    axes2.set_ylabel('maximum')
    axes3.set_ylabel('minimum')

    # plot the data
    axes1.plot(numpy.mean(data, axis=0))
    axes2.plot(numpy.max(data, axis=0))
    axes3.plot(numpy.min(data, axis=0))
```

- **6 - tidy and show plot**

```
# Analyse each file in turn
for fname in files:
    print("Analysing", fname)

    # load data
    data = numpy.loadtxt(fname=fname, delimiter=',')

    # create figure and three axes
    fig = matplotlib.pyplot.figure(figsize=(10.0, 3.0))
    axes1 = fig.add_subplot(1, 3, 1)
    axes2 = fig.add_subplot(1, 3, 2)
    axes3 = fig.add_subplot(1, 3, 3)

    # decorate the axes
    axes1.set_ylabel('average')
    axes2.set_ylabel('maximum')
    axes3.set_ylabel('minimum')

    # plot the data
    axes1.plot(numpy.mean(data, axis=0))
    axes2.plot(numpy.max(data, axis=0))
    axes3.plot(numpy.min(data, axis=0))

    # tidy plot and render
    fig.tight_layout()
    matplotlib.pyplot.show()
```

- **EXIT THE EDITOR AND RUN SCRIPT**

```
python analyse_files.py
```

🟥 Red sticky for a question or issue 🟩 Green sticky if complete

- **THIS IS INTERACTIVE. WHAT IF WE WANT TO SAVE IMAGES?**

  - **OPEN THE EDITOR AND CHANGE THE SCRIPT**

```
[...]

# Get a list of inflammation data files
files = []
for fname in os.listdir('data'):
  if 'inflammation' in fname and fname[-4:] == '.csv':
    files.append(os.path.join('data', fname))

[...]

    # tidy plot and render
    fig.tight_layout()
    # plt.show()

    # save image to file
    imgname = fname[:-4] + '.png'
    print('Writing image to', imgname)
    plt.savefig(imgname)
```

- **EXIT THE EDITOR AND RUN SCRIPT**

```
python analyse_files.py
```

- **CHECK CONTENTS OF `data` DIRECTORY, AND VIEW `.png` FILES**

SLIDE Checking data

- There are **two suspicious features** to some of the datasets

1. The **maximum values rose and fell as straight lines**
2. The **minimum values are consistently zero**

- We'll use `if` statements to **test for these conditions and give a warning**

SLIDE Test for suspicious maxima

- Is day zero value 0, and day 20 value 20?

```
$ nano analyse_files.py
```

- **ADD TO EXISTING CODE BEFORE CREATING FIGURE**

```
    # Test for suspicious maxima
    if np.max(data, axis=0)[0] == 0 and np.max(data, axis=0)[20] == 20:
        print("Suspicious-looking maxima!")
```

- **RUN SCRIPT**

```
$ python analyse_files.py
```

SLIDE SUSPICIOUS MINIMA

- Are all the minima zero? (do they sum to zero?)

```
$ nano analyse_files.py
```

- **ADD TO EXISTING CODE BEFORE CREATING FIGURE**

```
    # Test for suspicious maxima
    if np.max(data, axis=0)[0] == 0 and np.max(data, axis=0)[20] == 20:
        print("Suspicious-looking maxima!")
    elif np.sum(data.min(axis=0)) == 0:
        print('Minima sum to zero!')
```

- **RUN SCRIPT**

```
$ python analyse_files.py
```

SLIDE BEING TIDY

- If everything's OK, **let's be reassuring**
- **ADD TO EXISTING CODE BEFORE PLOT**
- **ADD TO EXISTING CODE BEFORE CREATING FIGURE**

```
    # Test for suspicious maxima
    if np.max(data, axis=0)[0] == 0 and np.max(data, axis=0)[20] == 20:
        print("Suspicious-looking maxima!")
    elif np.sum(data.min(axis=0)) == 0:
        print('Minima sum to zero!')
    else:
        print('Seems OK!')
```

- **RUN SCRIPT**

```
$ python analyse_files.py
```

🟥 Red sticky for a question or issue 🟩 Green sticky if complete

---

SECTION 09: Conclusions (Part 1)

SLIDE Learning outcomes

Some things you might not have known about at lunchtime:

- variables
- data types: arrays, lists, strings, numbers
- file IO: loading data, listing files, manipulating filenames
- calculating statistics
- plotting data: plots and subplots
- program flow: loops and conditionals
- automating multiple analyses
- `Python` scripts: edit-save-execute

SLIDE WELL DONE!

- **SEND THEM HOME HAPPY!**

TITLE: Building Programs With Python (Part 2)

SLIDE: Etherpad

- Please use the Etherpad for the course **DEMONSTRATE LINK**

SLIDE Why are we here?

- We're here to learn **how to program**
- This is a way to **solve problems in your research** through making a computer do work **quickly** and **accurately**
  - You'll be continuing to **build your data analysis** script from yesterday
  - You'll **automate** functions to perform tasks over and over again (in various combinations)

- You'll **manipulate data**, which is at the heart of all academia
  - You'll learn how to build **functions** that do specific, defined tasks and encapsulate code, making it reusable and readable
  - You'll learn some **defensive programming**, so that you automatically catch problems in your code/data handling

---

SLIDE XKCD

- Again, this slide is only a little bit flippant
- *No-one* writes perfect code, first time
  - It's all about revision, and good practice: **defensive programming**
  - These principles will make your life, and other people's lives, much easier

---

SLIDE Setting up

- We want a neat (clean) working environment

- Change directory to desktop (in terminal or Explorer)

- Change your working directory to `python-novice-inflammation` (from yesterday/earlier)

```
$ cd ~/Desktop
$ cd python-novice-inflammation
```

🟥 Red sticky for a question or issue 🟩 Green sticky if complete

---

SECTION 10: `Jupyter` notebooks

SLIDE Starting `Jupyter`

- Make sure you're in the project directory `python-novice-inflammation`
- **Start `Jupyter`** from the command-line
- **CHECK WHETHER EVERYONE SEES A WORKING JUPYTER NOTEBOOK**

```
$ jupyter notebook
```

🟥 Red sticky for a question or issue 🟩 Green sticky if complete

---

SLIDE `Jupyter` landing page

- **`Jupyter` landing page is a file browser**, like Explorer/Finder
  - Point out `Python` (`.py`) files, `.zip` files, and directories)
  - Point out directory (`data`), and how the file symbols are different. (*the triangle by the check box gives a key*)
- **Point out `New` button.**

SLIDE Create a new notebook

- Click on `New -> Python 3`
- Point out that there may or may not be other options in the student's installation
- Indicate the new features on the empty notebook:
  - The **notebook name**: `Untitled`
  - **Checkpoint** information (the last time the notebook was saved, for safety)
  - The **menu bar** (`File Edit etc.`) - just like `Word` or `Excel`
  - An indication of **which kernel you're using/language you're in**
  - **Icon view** (just like `Word` or `Excel`)
  - An empty cell with `In [ ]:`
- Point out the **box around the cell**, and that it **changes colour** when you start to edit

---

SLIDE My first notebook

- **Give the notebook the name `functions`**
- Click on `Untitled` and enter the name `functions`

SLIDE Cell types

- `Jupyter` documents are comprised of `cells`
- A `cell` can be one of **several types** - we'll focus on two:
  - `Code`: **code** in the current kernel/language
  - `Markdown`: **text**, with the opportunity for formatting
- **Change the first cell type to `Markdown`**
  - The box **colour changes** from green to blue
  - The `In []` **prompt disappears**

SLIDE Markdown text

- `Markdown` lets us **enter formatted text**
  - **Headers** are preceded by a hash: `#`
  - The **level of header** is determined by the number of hashes: `#`
  - **Typewriter text/code** is shown by enclosing in backticks: ```

  - **Italics** are shown by enclosing text in single asterisks: `*italic*`
  - **LaTeX** can be entered within dollar signs `$`

- Press `Shift + Enter` to execute a cell

- The cell is rendered, and a new cell appears beneath the executed cell

```
# Functions

Functions are pieces of code that take an input and return an output. They
enable us to break our code into logical chunks that are easier to
understand and maintain.

## Temperature conversion

As an example in `Python`, we will create a function that converts
temperature between *Fahrenheit* and *Kelvin* scales.
```

## SECTION 11: Functions

SLIDE Motivation

- We wrote some code that plots values of interest from multiple datasets
  - **but that code is long and complicated**
- The code is also not very flexible if we want to deal with thousands of files, and we can't modify it to plot only a subset of files very easily
  - Cutting and pasting is slow and error-prone
- **SO** we will package our code for reuse.
  - **We do this by writing functions**

SLIDE What is a function?

- Functions in code work **like mathematical functions**, like $y = f(x)$
  - $f()$ is the function
  - $x$ is an **input** (or inputs)
  - $y$ is the **returned value**, or output(s)
- The function's output $y$ depends in some way on the value of $x$ - the dependency is defined by $f()$.
- **Not all functions in code take an input, or produce a usable output, but the principle is generally the same.**

- **You've already been using functions in this course: `print()`, `numpy.max()`, etc.**

SLIDE My first function

- **REFER TO THE CODE IN THE NOTEBOOK**
  - We've written a function to convert Fahrenheit to Kelvin, called `fahr_to_kelvin()`
- **Describe the mathematical function**:
  - This function takes `x`, subtracts 32, multiplies by 5/9, and adds 273.15
- In `Python` this **translates to the code below**:
  - The function **performs a calculation, which is *returned* by the `return` statement.**
  - The value of **the variable `temp` is taken through the same calculation as in the mathematical function**, and is then *return*ed.
  - Functions are *defined* by the `def` keyword
  - The name of the function follows the `def` keyword (equivalent to `f` in the mathematical example)
  - The first line ends in a colon, just like a `for` loop or `if` statement.
  - The code, or *body* of the function is indented, just like a `for` loop or `if` statement.
  - The *parameters* or *inputs* to the function are then defined in parentheses. These get a variable name **which only exists within the function**. Here, there is one parameter, called `temp`.

SLIDE Calling the function

- We **call `fahr_to_kelvin` in exactly the same way we call any other function we've seen** so far
  - e.g. `print()` or `numpy.mean()`

```
print('freezing point of water:', fahr_to_kelvin(32))
print('boiling point of water:', fahr_to_kelvin(212))
```

- **NOTE: that the returned values from executing code show up in the notebook below the cell**

🟥 Red sticky for a question or issue 🟩 Green sticky if complete

---

SLIDE Create a new function

- **ASK THE LEARNERS HOW WE WOULD CREATE A NEW FUNCTION TO CONVERT KELVIN TO CELSIUS**

- **Walk through the process, being prompted**

```
def kelvin_to_celsius(temp):
  return temp - 273.15
```

- **ASK THE LEARNERS HOW TO CALL THE FUNCTION**

```
print('freezing point of water', kelvin_to_celsius(273.15))
```

SLIDE Composing functions

- **Composing `Python` functions works just like mathematical functions**: `y = f(g(x))`
- **ASK HOW WE CAN CONVERT FAHRENHEIT TO CELSIUS WITH OUR EXISTING FUNCTIONS**
    - We could convert a temperature in fahrenheit (`temp_f`) to a temperature in celsius (`temp_c`) by **executing the code**:

```
temp_f = 212.0
temp_c = kelvin_to_celsius(fahr_to_kelvin(temp_f))
print(temp_c)
```

SLIDE New functions from old

- **ASK LEARNERS HOW WE CAN TURN THIS INTO A NEW FUNCTION: `fahr_to_celsius()`:

```
def fahr_to_celsius(temp_f):
    return kelvin_to_celsius(fahr_to_kelvin(temp_f))
```

- We can call this just like any other function

```
print('freezing point of water in Celsius:', fahr_to_celsius(32.0))
```

- **THIS IS HOW PROGRAMS ARE BUILT: COMBINING SMALL CHUNKS OF CODE INTO LARGER BITS UNTIL WE GET THE RESULT WE WANT**

SLIDE Exercise 08 (10min)

- **SHOW THE SLIDES FOR THE EXERCISE**

```
def outer(s)
    return s[0] + s[-1]
```

🟥 Red sticky for a question or issue 🟩 Green sticky if complete

- **RETURN TO THE NOTEBOOK**

SLIDE Function scope

- **Make a Markdown note**

```
## Function scope

Variables defined within a function (including parameters) are not
available outside the function unless they are returned.
```

- **This is called *function scope***
- **DEMO THE CODE BELOW**

```
a = "Hello"

print(a)
```

- This code defines a variable `a` and gives it a value "Hello"
- **NOW DECLARE A FUNCTION (IN THE SAME CELL) AND CALL IT**

```
a = "Hello"

def my_fn(a):
    a = "Goodbye"

my_fn(a)
print(a)
```

- Returning `a` doesn't - by itself - change anything

```
a = "Hello"
```

```
def my_fn():
    a = "Goodbye"
    return a

my_fn(a)
print(a)
```

- To move values to and from functions, you should generally `return` them from the function, *and* catch them in a variable
- **COMPLETE THE CODE EXAMPLE IN THE CELL**

```
a = "Hello"

def my_fn(a):
  a = "Goodbye"
  return a

a = my_fn(a)
print(a)
```

SLIDE Exercise 09 (5min)

- **PUT THE SLIDES ON SCREEN**
- **MCQ: put coloured stickies up**
- Solution: `1: 7 3` (this differs from that on the SWC page)

SECTION 12: Refactoring

SLIDE Tidying up

- Now we can write functions, **let's make the inflammation analysis easier to reuse**
  - **ONE FUNCTION PER OPERATION**
- **CLOSE THE NOTEBOOKS**
- **OPEN UP THE `ANALYSE_FILES.PY` SCRIPT
- **TALK THE STUDENTS THROUGH THE CODE LOGIC: TWO SECTIONS - ANALYSE AND DETECT PROBLEMS**
- The code can be divided into two main sections, which could be functions:
  1. check the data for problems
  2. plot the data

SLIDE `detect_problems()`

- We noticed that some data was questionable
- This function spots problems with the data
    - Call the function after loading, before plotting
- **OPEN EDITOR AND CHANGE CODE**

```
$ nano analyse_files.py
```

```
# Detect problems with a dataset
def detect_problems(data):
    if np.max(data, axis=0)[0] == 0 and np.max(data, axis=0)[20] == 20:
        print("Suspicious-looking maxima!")
    elif np.sum(data.min(axis=0)) == 0:
        print('Minima sum to zero!')
    else:
        print('Seems OK!')

# Analyse each file in turn
for fname in files:
    print("Analysing", fname)

    # load data
    data = np.loadtxt(fname=fname, delimiter=',')

    # identify problems in the data
    detect_problems(data)
```

- **SAVE AND RUN SCRIPT**

```
$ python analyse_files.py
```

SLIDE `plot_data()`

- We'll write **a function that plots the data**

```
# Plot passed data in the specified file
def plot_data(data, fname):
    # create figure and three axes
    fig = plt.figure(figsize=(10.0, 3.0))
    axes1 = fig.add_subplot(1, 3, 1)
    axes2 = fig.add_subplot(1, 3, 2)
    axes3 = fig.add_subplot(1, 3, 3)
    # decorate the axes
    axes1.set_ylabel('average')
    axes2.set_ylabel('maximum')
    axes3.set_ylabel('minimum')

    # plot the data
    axes1.plot(data.mean(axis=0))
    axes2.plot(data.max(axis=0))
    axes3.plot(data.min(axis=0))

    # tidy plot and render
    fig.tight_layout()
    print('Writing image to', imgname)
    plt.savefig(imgname)

# Analyse each file in turn
for fname in files:
    print("Analysing", fname)

    # load data
    data = np.loadtxt(fname=fname, delimiter=',')

    # identify problems in the data
    detect_problems(data)

    # plot image in file
    imgname = fname[:-4] + '.png'
    plot_data(data, imgname)
```

SLIDE Code reuse

- **The logic of the code is now easier to understand**
- We identify the input files, then apply one function per action in a loop:
    - Print the filename
    - Load the data with `np.loadtxt()`
    - `detect_problems()` in the data
    - `plot_data()` the data

```
# Analyse each file in turn
for fname in files:
    print("Analysing", fname)

    # load data
    data = np.loadtxt(fname=fname, delimiter=',')

    # identify problems in the data
    detect_problems(data)

    # plot image in file
    imgname = fname[:-4] + '.png'
    plot_data(data, imgname)
```

- **THIS HAS ADVANTAGES**
  - **The code is much shorter (as we read it, here)**
  - **The function names are human-readable and descriptive**
  - **It is much easier to see what the code is doing**

🟥 Red sticky for a question or issue 🟩 Green sticky if complete

---

SLIDE Good code pays off

- **YOU MAY BE ASKING YOURSELF WHY YOU WANT TO BOTHER WITH THIS**
  - After 6 months, the referee report arrives and you need to rerun experiments
  - Another student is continuing the project
  - Some random person reads your article and asks for the code
  - Helps spot errors quickly
  - Clarifies structure in your mind as well as in the code
  - Saves you time in the long run! ("Future You" will back this up)

SECTION 13: Command-line programs

SLIDE Learning objectives

- **How can I write Python programs that will work like Unix command-line tools?**
- Use the values of **command-line arguments** in a program.
- Handle **flags and files** separately in a command-line program.
- **Read data from standard input** in a program so that it can be used in a pipeline (with *pipes*: `|`)

SLIDE The `sys` module

- The `sys` module is the main way `Python` lets you interact with the operating system. You can:
  - **run programs**
  - **parse commands**
  - **get information about the system**
- We're going to use it in some new scripts
  - **Create a new file called `sys_version.py`**

```
$ nano sys_version.py
```

- **Enter the code below**

```python
import sys
print('version is', sys.version)
```

- **Run the script**

```
$ python sys_version.py
version is 3.6.3 |Anaconda custom (64-bit)| (default, Oct  6 2017,
12:04:38)
[GCC 4.2.1 Compatible Clang 4.0.1 (tags/RELEASE_401/final)]
```

SLIDE `sys.argv`

- `sys.argv` is a variable that contains the command-line arguments used to call our script
  - The variable is a **list of arguments**
- **Open a new file called `sys_argv.py` in the editor**

```
$ nano sys_argv.py
```

- **Enter the code below**

```python
import sys
print('sys.argv is', sys.argv)
```

- **Run the script with some options**

```
$ python sys_argv.py
sys.argv is ['sys_argv.py']
$ python sys_argv.py item1 item2 somefile.txt
sys.argv is ['sys_argv.py', 'item1', 'item2', 'somefile.txt']
```

- The **name of the script is always the first element: `sys.argv[0]`**

SLIDE Building a new script

- We're going to build a script that **reports readings from data files**

```
$ python readings.py mydata.csv
```

- We will make it **take options** `--min`, `--max`, `--mean`
  - The script will report *one* of these

```
$ python readings.py --min mydata.csv
```

- We will make it **handle multiple files**

```
$ python readings.py --min mydata.csv myotherdata.csv
```

- We will make it take `STDIN` so we can **use it with *pipes***

```
$ python readings.py --min < mydata.csv
```

SLIDE Starting the framework

- We start with a script that doesn't do all that
  - We'll **build features in one-by-one**
- **Create a new file called `readings.py` in the editor**

```
$ nano readings.py
```

- **Add the code below and explain**
  - **imports at the top**
  - **define a `main()` function** to hold code that does the work of the script
  - We **catch the script name**
  - We **catch the first argument** (filename)
  - We **load the data**
  - For each patient, we print the mean inflammation

```
import sys
import numpy

def main():
    script = sys.argv[0]
    filename = sys.argv[1]
    data = numpy.loadtxt(filename, delimiter=',')
    for m in numpy.mean(data, axis=1):
        print(m)
```

- **Run the script**

```
$ python readings.py
```

- **NOTHING HAPPENS - WHY?**
  - We've defined a function, but it hasn't been called

SLIDE Calling a script

- **There's a way to tell if a `Python` file is being run as a script**

- If we use this, we can use the same file as:

  - a module (`import readings`)
  - a script (`$ python readings.py`)

- The `Python` code has `__name__ == '__main__'` only when run as a script

- **We want to run `main()` only if the file is run as a script**

  - **Add this code to the bottom of `readings.py`**

```
if __name__ == '__main__':
   main()
```

- **Run the script**
  - `small-01.csv` is a reduced dataset, created for testing

```
$ python readings.py data/small-01.csv
0.333333333333
1.0
```

SLIDE Handling multiple files

- We want to be able to analyse multiple files with one command
  - **NOTE:** wildcards are expanded by the operating system
  - **DEMO the code**

```
$ ls data/small-*
data/small-01.csv  data/small-02.csv  data/small-03.csv
$ python sys_argv.py data/small-*
sys.argv is ['sys_argv.py', 'data/small-01.csv', 'data/small-02.csv',
'data/small-03.csv']
```

- **All arguments from index `1` onwards are filenames**
  - So loop over everything in `sys.argv[1:]`
  - **Change the `main()` function**

```
def main():
    script = sys.argv[0]
    for filename in sys.argv[1:]:
        print(filename)
        data = numpy.loadtxt(filename, delimiter=',')
        for m in numpy.mean(data, axis=1):
            print(m)
```

- **Demo the code**

```
$ python readings.py data/small-*
data/small-01.csv
0.333333333333
1.0
data/small-02.csv
13.6666666667
11.0
data/small-03.csv
0.666666666667
0.666666666667
$ python readings.py data/small-01.csv
data/small-01.csv
0.333333333333
1.0
```

SLIDES Handling flags

- **We want to use `--min`, `--max`, `--mean` to tell the script what to calculate**

```
$ python readings.py --max myfile.csv
```

- The flag will be `sys.argv[1]`, so filenames are `sys.argv[2:]`
  - We'll need to modify the code to handle this

- We should check that flags are valid
  - Check this with an `if` statement
  - **Use `sys.exit()` to quit the script if the action is wrong**
- **MODIFY THE SCRIPT AS BELOW**

```
def main():
    script = sys.argv[0]
    action = sys.argv[1]
    filenames = sys.argv[2:]
    if action not in ['--min', '--mean', '--max']:
        print('Action is not one of --min, --mean, or --max: ' + action)
        sys.exit(1)
    for f in filenames:
        process(f, action)
```

- **TRY THE SCRIPT**

```
$ python readings.py --min data/small-01.csv
Traceback (most recent call last):
  File "readings.py", line 15, in <module>
    main()
  File "readings.py", line 12, in main
    process(f, action)
NameError: name 'process' is not defined
```

- We'll add a `process()` function shortly
- **TEST A BAD ACTION**

```
$ python readings.py --std data/small-01.csv
Action is not one of --min, --mean, or --max: --std
```

- **We have a useful error message**

SLIDE Add `process()`

- We split the script into two functions for **readability**
  - The `main()` function clearly handles the command-line
  - The `process()` function handles the data
- **Add the code to `readings.py`

```
def process(filename, action):
    data = numpy.loadtxt(filename, delimiter=',')

    if action == '--min':
        values = numpy.min(data, axis=1)
    elif action == '--mean':
        values = numpy.mean(data, axis=1)
    elif action == '--max':
        values = numpy.max(data, axis=1)

    for m in values:
        print(m)
```

- **TRY THE SCRIPT**

```
$ python readings.py --min data/small-01.csv
0.0
0.0
$ python readings.py --mean data/small-01.csv
0.333333333333
1.0
$ python readings.py --mean data/small-0*
0.333333333333
1.0
13.6666666667
11.0
0.666666666667
0.666666666667
```

SLIDE Using `STDIN`

- The final change will let us **use `STDIN` if no file is specified**
  - `sys.stdin` catches `STDIN` from the operating system
- **MODIFY THE SCRIPT AS BELOW**

```
    if len(filenames) == 0:
        process(sys.stdin, action)
    else:
        for f in filenames:
            process(f, action)
```

- **TEST THE SCRIPT**

```
$ python readings.py --mean data/small-*
0.333333333333
1.0
13.6666666667
11.0
0.666666666667
0.666666666667
$ python readings.py --mean < data/small-01.csv
0.333333333333
1.0
```

- **AND WE'RE DONE!!!**

SECTION 14: Testing and documentation

SLIDE Motivation

- Once a useful function is written, it gets reused over and over, often without further checking
- When you write a function you should:
  - **Test output for correctness**
  - **Document the expected function**
- We'll demonstrate this with a function to centre a numerical array

---

SLIDE Create a new notebook

- **New notebook called `testing`**

- **ADD AN INTRO IN MARKDOWN**

```markdown
# Testing and Documentation

When writing a function, we should

- test output for correctness
- document the expected function
```

- **ADD IMPORTS**

```python
import numpy as np
```

SLIDE `centre()`

- **Write the test function**

- When doing some analyses, such as PCA, we might want to recentre and normalise our dataset.
- Let's write a function to recentre an array of data, like the inflammation data.
- **EXPLAIN THE MATHS IF NECESSARY**

```
def centre(data, desired):
    return (data - np.mean(data)) + desired
```

SLIDE Test datasets

- **ASK THE LEARNERS HOW WE CAN CHECK THAT THE FUNCTION WORKS IN THE WAY WE INTEND**
- We could try `centre()` on our real data, but we *don't know what the answer should be!**
    - We'll use `numpy`'s `zeros()` function to generate an **input set where we know the answer**
- **SHOW THE TEST DATA**

```
z = np.zeros((2, 2))
z
```

- **Let's recentre the data at the value 2**

```
centre(z, 3.0)
```

- **This works, so we'll try it on real data**

SLIDE Real data

- **LOAD THE DATA**

```
data = np.loadtxt(fname='data/inflammation-01.csv', delimiter=',')
```

- **Let's recentre the data to zero**

```
centre(data, 0))
```

- This looks OK, but **how would we know it worked?**

SLIDE Check properties

- **ASK LEARNERS HOW THEY COULD VERIFY THE FUNCTION WORKED AS INTENDED**

- We can **check properties of the original and centred data**

    - `mean`, `min`, `max`, `std`

```
print('original min, mean, and max are:', numpy.min(data),
numpy.mean(data), numpy.max(data))
```

- We'd expect the **mean of the new dataset to be approximately `0.0`**
- Also, the **range (`max` - `min`) should be unchanged.**

```
centred = centre(data, 0)
print('min, mean, and max of centered data are:', numpy.min(centred),
      numpy.mean(centred), numpy.max(centred))
```

- The limits seem OK, but has the *shape* of the data distribution changed?
- The **variance of the dataset should be unchanged.**

```
print('std dev before and after:', numpy.std(data), numpy.std(centred))
```

- The range and variance are as expected, but the mean is not quite `0.0`

- **The function is probably OK, as-is**

SLIDE Documenting functions

- We can document what our function does by **writing comments in the code**, and this is a good thing.

- But Python allows us to **document what a function does directly in the function** using a *docstring*.

- This is a string that is put in a **specific place in the function definition, and it has special properties that are useful**.

- To add a docstring to our `centre()` function, we add a string immediately after the function declaration

- **ADD DOCSTRING TO EXISTING FUNCTION AND RUN CELL**

```
def centre(data, desired):
    """Returns the array in data, recentered around the desired value."""
    return (data - numpy.mean(data)) + desired
```

- **RESTART KERNEL AND RUN ALL**
- This documents the function directly in the source code, and it also **hooks that documentation into `Python`'s `help` system.**
    - We can ask for help on any function using the `help()` function:
- **built-in** functions

```
help(print)
```

- **functions from modules**

```
help(numpy.mean)
```

- and **if you write it** your own functions

```
help(centre)
```

- **SHOW LEARNERS HOW DETAILED THE BUILTIN AND NUMPY HELP IS**
- Using the triple quotes (""") allows us to use a multi-line string to describe the function:
- **ADD EXTRA DOCUMENTATION**

```
def centre(data, desired):
    """Returns the array in data, recentred around the desired value.

    Example
    -------
    >>> centre([1, 2, 3], 0)
    [-1, 0, 1]
    """
    return (data - numpy.mean(data)) + desired
```

- **DEMONSTRATE THE CHANGE**

SLIDE Default arguments

- So far we have named the two arguments in our `centre()` function
  - We need to specify both of them when we call the function

```
centre([1, 2, 3], 0)
array([-1.,  0.,  1.])

centre([1, 2, 3])
---------------------------------------------------------------------------
-
TypeError                                 Traceback (most recent call
last)
<ipython-input-13-9131fef8e3dc> in <module>()
----> 1 centre([1, 2, 3])

TypeError: centre() missing 1 required positional argument: 'desired'
```

- We **can set a *default* value for function arguments** when we define the function
- Set defaults **by assigning a value in the function declaration**, as follows:

```
def centre(data, desired=0.0):
    """Returns the array in data, recentred around the desired value.

    Example
    -------
    >>> centre([1, 2, 3], 0)
    [-1, 0, 1]
    """
    return (data - numpy.mean(data)) + desired
```

- The change we've made is to set `desired=0.0` in the function *prototype*.
- Now, by default, the function will recentre the passed data to zero, without us having to specify that:

```
centre([1, 2, 3])
```

SLIDE Create a new notebook

- **New notebook called `testing`**
- **ADD AN INTRO IN MARKDOWN**

```
# Testing and Documentation

When writing a function, we should
```

```
- test output for correctness
- document the expected function
```

- **ADD IMPORTS**

```
import numpy
```

- **Write the test function**
- When doing some analyses, such as PCA, we might want to recentre and normalise our dataset.
- Let's write a function to recentre an array of data, like the inflammation data.

```
def centre(data, desired):
    return (data - np.mean(data)) + desired
```

SLIDE Exercise 10 (10min)

- **MOVE SLIDES TO THE SCREEN**

```
def rescale(data):
    """Returns input array rescaled to [0.0, 0.1]."""
    l = numpy.min(data)
    h = numpy.max(data)
    return (data - l) / (h - l)
```

🟥 Red sticky for a question or issue 🟩 Green sticky if complete

---

SLIDE Errors and exceptions

- **MOVE NOTEBOOK TO THE SCREEN**

SLIDE Create a new notebook

- **Call the notebook `errors`**
- **ADD AN INTRO**

```
# Errors and Exceptions
```

```
`Python` provides useful error reports of what has gone wrong, which can
help with debugging.
```

SECTION 15: Errors

SLIDE Create a new notebook

- **Call the notebook `errors`**
- **ADD AN INTRO**

```
# Errors and Exceptions

`Python` provides useful error reports of what has gone wrong, which can
help with debugging.
```

SLIDE Errors

- **Programming is essentially just making errors over and over again until the code works** 😉
    - The key skill is **learning how to identify, and then fix, the errors** when they are reported.
    - **All programmers** make errors.

SLIDE Traceback

- `Python` tries to be helpful, and provides extensive information about errors
    - These are called *tracebacks*
- **We'll induce a traceback, so we can look at it**
- **ENTER CODE IN A CELL**

```
def favourite_ice_cream():
    ice_creams = ["chocolate",
                  "vanilla",
                  "strawberry"]
    print(ice_creams[3])
```

- **NEW CELL**

```
favourite_ice_cream()
```

SLIDE Anatomy of a traceback

```
--------------------------------------------------------------------------
-
IndexError                                Traceback (most recent call
last)
<ipython-input-4-8f18c934933f> in <module>()
----> 1 favourite_ice_cream()

<ipython-input-3-3f8910a0f7ad> in favourite_ice_cream()
      3                    "vanilla",
      4                    "strawberry"]
----> 5     print(ice_creams[3])

IndexError: list index out of range
```

- **TALK THROUGH THE TRACEBACK IN THE NOTEBOOK**
    - The *stack* of all steps leading to the error is shown
    - The steps are separated by lines starting `<ipython-input-1…`
    - The steps run in order from top to bottom
- The first step has an arrow, showing where we were when the error happened. We were calling the `favourite_ice_cream()` function
- The second step tells us that we were *in* the `favourite_ice_cream()` function
    - The second step also points to the line `print(ice_creams[3])`, which is where the error occurs
    - This is also the last step, and the precise error is shown on the final line: `IndexError: list index out of range`
- Together, this tells us that we have made an index error in the line `print(ice_creams[3])`, and by looking we can see that we've tried to use an index outside the length of the list.

SLIDE Syntax errors

- **The error you saw just now was a *logic error*** - the code was valid `Python`, but it did something 'illegal' when it ran
    - **We have to run the code to see the error**
- ***Syntax* errors occur when the code is not interpretable as valid `Python`**
    - **The error is reported before the code runs**
- **ENTER CODE IN A NEW CELL - NOTE THE EXTRA SPACE AND LACK OF COLON!**

```
def some_function()
    msg = "hello, world!"
    print(msg)
     return msg
```

SLIDE Syntax traceback

```
  File "<ipython-input-6-bef8c18baffa>", line 1
    def some_function()
                       ^
SyntaxError: invalid syntax
```

- `Python` tells us there's a `SyntaxError` - the code isn't written correctly
    - **We don't get the chance to run the code**
- It points to the approximate location of the problem with a caret/hat (`^`)
    - We can see that we need to put a colon at the end of the function declaration
- **FIX THE CODE IN PLACE**

SLIDE Fixed?

- **SHOW AND RUN FIXED CODE**

```
def some_function():
    msg = "hello, world!"
    print(msg)
     return msg
```

SLIDE Not quite

```
  File "<ipython-input-7-b32ba7f38b6b>", line 4
    return msg
    ^
IndentationError: unexpected indent
```

- `Python` now tells us that there's an `IndentationError`
- We don't learn about all the syntax errors at one time - `Python` gives up after the first one it finds

  - **(fixing the first error in a file might correct all subsequent errors)**

SLIDE Name errors

- If you try to use a variable that is not defined in *scope*, you will get a `NameError`
  - This often happens with typos
- **ENTER CODE IN A NEW CELL**

```
print(a)
```

- We have a **NAME ERROR**

```
--------------------------------------------------------------------------
-
NameError                                 Traceback (most recent call
last)
<ipython-input-5-c5a4f3535135> in <module>()
----> 1 print(a)

NameError: name 'a' is not defined
```

- **This is true in functions/loops, too**
  - **ENTER CODE IN A NEW CELL**

```
for i in range(3):
    count = count + i
```

- **This still gives us a name error**

```
 -------------------------------------------------------------------------
--
NameError                                 Traceback (most recent call
last)
<ipython-input-6-15ebe951e74d> in <module>()
      1 for i in range(3):
----> 2     count = count + i

NameError: name 'count' is not defined
```

SLIDE Index errors

- If you try to access an element of a collection that does not exist, you'll get an `IndexError`
- **ENTER CODE IN NEW CELL**

```
letters = ['a', 'b']
print("Letter #1 is", letters[0])
print("Letter #2 is", letters[1])
print("Letter #3 is", letters[2])

Letter #1 is a
Letter #2 is b
--------------------------------------------------------------------------
-
IndexError                                Traceback (most recent call
last)
<ipython-input-9-62bced7460d2> in <module>()
      2 print("Letter #1 is", letters[0])
      3 print("Letter #2 is", letters[1])
----> 4 print("Letter #3 is", letters[2])

IndexError: list index out of range
```

SLIDE Exercise 11 (10min)

- **PUT SLIDES ON SCREEN**

```
message = ""
for number in range(10):
    # use a if the number is a multiple of 3, otherwise use b
    if (number % 3) == 0:
        message = message + "a"
    else:
        message = message + "b"
print(message)
```

🟥 Red sticky for a question or issue 🟩 Green sticky if complete

SECTION 16: Defensive programming

SLIDE (Un)readable code

- What does this function do?

- **GIVE LEARNERS A COUPLE OF MINUTES TO TRY TO WORK IT OUT**

```python
def s(p):
    a = 0
    for v in p:
        a += v
    m = a / len(p)
    d = 0
    for v in p:
        d += (v - m) * (v - m)
    return numpy.sqrt(d / (len(p) - 1))
```

SLIDE Readable code

- What does this function do?

- **GIVE LEARNERS A COUPLE OF MINUTES TO TRY TO WORK IT OUT**

```python
def std_dev(sample):
    sample_sum = 0
    for value in sample:
        sample_sum += value

    sample_mean = sample_sum / len(sample)

    sum_squared_devs = 0
    for value in sample:
        sum_squared_devs += (value - sample_mean) * (value - sample_mean)

    return numpy.sqrt(sum_squared_devs / (len(sample) - 1))
```

- **This is the same code as in the previous slide**
  - sensible function name
  - sensible variable names
  - blank lines to separate code blocks

- **Even without comments/documentation it's readable**

- **FIRST LINE OF DEFENCE: sensible names, and documentation**
  - But that's not all you can do to make your life easier.

SLIDE Create a new notebook

- **Call it `defensive`**

- **ADD INTRO IN MARKDOWN**

````
# Defensive Programming

*Defensive programming* is the practice of expecting your code to have
mistakes, and guarding against them.
````

SLIDE Defensive programming

- So far **we have focused on the basic tools** of writing a program: variables, lists, loops, conditionals, and functions.
  - We haven't looked very much at whether a program is getting the right answer (and whether it continues to get the right answer as we change it).
  - **It's all very well having some code, but if it doesn't give the right answer it can be damaging, or worse than useless**
- **Defensive programming** is the practice of expecting your code to have mistakes, and guarding against them.
  - To do this, we will write some **code that *checks its own operation*.**
  - This is generally good practice, speeds up software development, and helps ensure that your code is doing what you intend.

SLIDE Assertions

- **ADD INTRODUCTORY TEXT**

````
## Assertions

Assertions are a pythonic way to see if a program's state is correct.

``python
assert <condition>, "Some text describing the problem"
``
````

- **Assertions** are a `Pythonic` way to see if code runs correctly
  - 10-20% of the `Firefox` source code is assertions/checks on the rest of the code!
- **We *assert* that a *condition* should be `True`**
  - If it's `True`, the code may be correct
  - If it's `False`, the code is **not** correct
- The syntax for an assertion is that we `assert` some `<condition>` is `True`, and if it's not, an error is thrown (`AssertionError`), with some text explaining the problem.

SLIDE Example assertion

- Type code **then ask learners what it does**

```
numbers = [1.5, 2.3, 0.7, -0.001, 4.4]
total = 0.0
for n in numbers:
    assert n > 0.0, 'Data should only contain positive values'
    total += n
print('total is:', total)
```

- **EXECUTE CELL**

```
---------------------------------------------------------------------------
-
AssertionError                            Traceback (most recent call
last)
<ipython-input-1-985f50018947> in <module>()
      2 total = 0.0
      3 for n in numbers:
----> 4     assert n > 0.0, 'Data should only contain positive values'
      5     total += n
      6 print('total is:', total)

AssertionError: Data should only contain positive values
```

- The **traceback tells us there is an `AssertionError` and highlights which *assertion* failed.**
  - The assertion is a check that the code **behaves how *we* expect**
  - It can be valid Python, and not throw an error, but it might not be what we want

---

SLIDE When do we use assertions?

- **Assertions are useful in three circumstances:**

  1. *preconditions* - must be true at the start of an operation
  2. *postcondition* - something guaranteed to be true when an operation completes
  3. *invariant* - something always true at a particular point in code

- **PUT EXAMPLE CODE IN NEW CELL**

```
def normalise_rectangle(rect):
    """Normalises a rectangle to the origin, longest axis 1.0 units."""
    x0, y0, x1, y1 = rect

    dx = x1 - x0
    dy = y1 - y0
    if dx > dy:
        scaled = float(dy) / dx
        upper_x, upper_y = 1.0, scaled
    else:
        scaled = float(dx) / dy
        upper_x, upper_y = scaled, 1.0

    return (0, 0, upper_x, upper_y)
```

- **Test with some values - in the same cell**

```
# Test function
normalise_rectangle((1.0, 1.0, 4.0, 4.0))
normalise_rectangle((1.0, 1.0, 4.0, 6.0))
```

- **DO ALL INPUTS MAKE SENSE?**

```
normalise_rectangle((6.0, 4.0, 1.0, 1.0))
normalise_rectangle((6.0, 4.0, 1.0))
```

- **ASK LEARNERS WHAT SORT OF CHECKS WE NEED TO MAKE**
- Examples:
  - **Input type** - 4 values, all numbers
  - **x0 < x1; y0 < y1** - lower left corner is identified first
  - **output values less than or equal to 1** - correct result returned

---

SLIDE Preconditions

- **Preconditions** must be true at the start of an operation or function
  - Here, we want to ensure that `rect` has four values
- **MAKE CHANGE IN CELL**

```
def normalise_rectangle(rect):
    """Normalises a rectangle to the origin, longest axis 1.0 units."""
    assert len(rect) == 4, "Rectangle must have four co-ordinates"
    x0, y0, x1, y1 = rect

    dx = x1 - x0
    dy = y1 - y0

    if dx > dy:
        scaled = float(dy) / dx
        upper_x, upper_y = 1.0, scaled
    else:
        scaled = float(dx) / dy
        upper_x, upper_y = scaled, 1.0

    return (0, 0, upper_x, upper_y)
```

- **TEST FAILING INPUT AND SHOW ASSERTIONERROR**

```
normalise_rectangle((6.0, 4.0, 1.0))

---------------------------------------------------------------------------
-
AssertionError                            Traceback (most recent call
last)
<ipython-input-10-6da0caef5016> in <module>()
      1 # Test function
----> 2 normalise_rectangle((6.0, 4.0, 1.0))

<ipython-input-9-7b5bc166ed3d> in normalise_rectangle(rect)
      1 def normalise_rectangle(rect):
      2     """Normalises a rectangle to the origin, longest axis 1.0
units."""
----> 3     assert len(rect) == 4, "Rectangle must have four co-ordinates"
      4     x0, y0, x1, y1 = rect
      5

AssertionError: Rectangle must have four co-ordinates
```

- **SHOW ANOTHER PROBLEM IN NEW CELL**

```
normalise_rectangle((6.0, 4.0, 1.0, -0.5))
```

- **Ask learners what's going on**

SLIDE Postconditions

- **Postconditions** must be true at the end of an operation or function.
  - Here, we want to assert that the upper x and y values are in the range [0, 1]
- **MAKE CHANGE IN CELL**

```
def normalise_rectangle(rect):
    """Normalises a rectangle to the origin, longest axis 1.0 units."""
    assert len(rect) == 4, "Rectangle must have four co-ordinates"
    x0, y0, x1, y1 = rect
    dx = x1 - x0
    dy = y1 - y0

    if dx > dy:
        scaled = float(dy) / dx
        upper_x, upper_y = 1.0, scaled
    else:
        scaled = float(dx) / dy
        upper_x, upper_y = scaled, 1.0

    assert 0 < upper_x <= 1.0, "Calculated upper x-coordinate invalid"
    assert 0 < upper_y <= 1.0, "Calculated upper y-coordinate invalid"

    return (0, 0, upper_x, upper_y)
```

- **TEST FAILING INPUT TO SHOW ASSERTIONERROR**

```
normalise_rectangle((6.0, 4.0, 1.0, -0.5))
```

- **This isn't our code's fault!**
  - The problem is that the input values have the upper-right corner below the lower left corner
  - **We actually need to add another *precondition***

```
def normalise_rectangle(rect):
    """Normalises a rectangle to the origin, longest axis 1.0 units."""
    assert len(rect) == 4, "Rectangle must have four co-ordinates"
    x0, y0, x1, y1 = rect

    assert x0 < x1, "Invalid x-coordinates"
    assert y0 < y1, "Invalid y-coordinates"

    dx = x1 - x0
    dy = y1 - y0

    if dx > dy:
        scaled = float(dy) / dx
        upper_x, upper_y = 1.0, scaled
    else:
        scaled = float(dx) / dy
        upper_x, upper_y = scaled, 1.0

    assert 0 < upper_x <= 1.0, "Calculated upper x-coordinate invalid"
    assert 0 < upper_y <= 1.0, "Calculated upper y-coordinate invalid"

    return (0, 0, upper_x, upper_y)
```

- **DEMONSTRATE THE ERROR THAT'S RAISED**

SLIDE Notes on assertions

- **PUT SLIDES ON SCREEN**
- Assertions help understand programs: they **declare what the program should be doing**
- Assertions help the person reading the program match their understanding of the code to what the code expects
- *Fail early, fail often*
- **Turn bugs into assertions or tests**: if you've made the mistake once, you might make it again