LESSON 01 - Building Programs With Python

These notes are a guide to the speaker, as they present the material.

Before you start

- Test your `Jupyter` installation and make sure you can connect to the kernel.

Slides

SLIDE Building Programs With Python (1)

SLIDE INTRODUCTION

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 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

- This cartoon is a *little* flippant, but only a bit
- The principles of a programming language like Perl are universal
- Many concepts are **universal across programming languages**
- **Learning one programming language will speed up the process of learning others**
- **Q: HOW MANY PEOPLE HERE HAVE EXPERIENCE OF AT LEAST ONE PROGRAMMING LANGUAGE?**
- What the more experienced here encounter should be recognisable to them

---

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**
- **We will be using some long-handed ways of doing things to keep them clear for novices**

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 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
- So, we'll also spend a little bit of time writing code in a **text editor**
- Text editors are part of the **edit-save-execute** cycle, which is how much code is written
- 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.
- 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 **daily measurements of inflammation**
- We're going to **analyse** the data
- We're going to **visualise** the data
- We're going to get the computer to do this for us
- **Automation** is key: **fewer human mistakes**, easier to **apply to other future datasets**, and share with

others (**transparency**)
- We can also **share our code and results** *via* sites such as GitHub and BitBucket (supplementary information, impact)

SLIDE SETUP

AT THIS POINT, PUT THE `TERMINAL` ON-SCREEN IN A SINGLE PROJECTOR SETUP, AND MOVE THE SLIDES TO THE DESKTOP

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 python-novice-inflammation
cd python-novice-inflammation
```

SLIDE SETTING UP - 2 - **DEMO**

- We need to **download our data** (and also a little code that can help us)
- **Go to Etherpad in browser** http://pad.software-carpentry.org/2017-05-18-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 `python-novice-inflammation` directory
- **Extract files** - this will create a subdirectory called `data` in that folder
- **CHECK WHETHER EVERYONE HAS EXTRACTED THE DATA**

SLIDE GETTING STARTED

SLIDE STARTING `JUPYTER` **DEMO**

- 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 **DEMO**

- **`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 **DEMO**

- 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 **DEMO**

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

SLIDE CELL TYPES **DEMO**

- `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 **DEMO**

- `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

```
# Variables in Python

## Python as a calculator

We can use `Python` as a calculator by typing mathematical statements
into a code cell, and *executing* that cell by pressing `Shift + Enter`.

We will enter the statement $1 + 2$ to see the result.
```

SLIDE ENTERING CODE **DEMO**

- **Mathematical statements can be entered directly** into a code cell

  - **ENTER** `1+2`

- **Before the cell is executed, note that the `In [ ]` prompt has no value in it**
- Note that the **code is colour syntax-highlighted**

  - **EXECUTE THE CELL** `Shift + Enter`

- Note that **after execution, the `In [ ]` prompt now has a number in it** to indicate the order in which cells were executed
- Note also that because there is no place to put the output, **a value has been returned as** `OUT [1]`, showing the result of the calculation

- **A new code cell is created** beneath the executed cell.

Red sticky for a question or issue   Green sticky if complete

---

SLIDE EXERCISE 01

- **PUT THE EXERCISE SLIDE ON SCREEN**
- **Ask the learners to try some calculator calculations, and demo some of your own**

```
2 ** 4
48 / 2 * (9 + 3)  # AMBIGUOUS!
48 / (2 * (9 + 3))
(48 / 2) * (9 + 3)
1e3 + 1e4
6 % 2
7 / 2
7 % 2
```

Red sticky for a question or issue   Green sticky if complete

WHEN FINISHED, GO BACK TO THE NOTEBOOK AND PUT THE SLIDES ON THE DESKTOP

SLIDE MY FIRST VARIABLE

- **TYPE THE MARKDOWN IN A CELL AND EXECUTE**
    - This is to keep the notebook as an example of literate programming (and a handy reference for the students)

```
## 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
```

- **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

```
name = "Samia"
```

- **LET'S DO THIS FOR REAL IN PYTHON - follow on from the physical example if possible**
    - To *assign* a value we use the *equals sign*
    - The variable name/box label goes on the left, and the data item goes on the right
    - *Character strings*, or *strings*, are enclosed in quotes
    - Executing the cell assigns the variable
- So now, if we refer to the variable `Name`, we get the value that's in the box: `Samia`

Red sticky for a question or issue

Green sticky if complete

SLIDE INSPECTING A VARIABLE

- The `print()` function shows contents of a variable
- **We refer to the name of the variable, and get its contents**

```
print(name)
```

Red sticky for a question or issue

Green sticky if complete

SLIDE WORKING WITH VARIABLES

- **Lead the students** through the code:
- Note, we're assigning an integer now (no quotes), but **assignment is the same for all data items**
- Print `weight_kg` to see its value

```
weight_kg = 55
print(weight_kg)
```

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

```
2.2 * weight_kg
```

- People may ask about floating point representations here - an introduction is at https://docs.python.org/3/tutorial/floatingpoint.html - put this on the Etherpad.

- **The `print()` function will take more than one argument, separated by commas, and print them**

```
print("weight in pounds", 2.2 * weight_kg)
```

- **Reassigning to the same variable overwrites the old value**

```
weight_kg = 57.5
print("weight in kilograms is now:", weight_kg)
```

- **Changing the value of one variable does not automatically change the values of other defined variables**

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

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

Red sticky for a question or issue

Green sticky if complete

SLIDE EXERCISE 02 (5MIN)

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

SLIDE EXERCISE 03 (5MIN)

MCQ: put up four colours of sticky notes

- The code prints `Hopper Grace`

WHEN FINISHED, GO BACK TO THE NOTEBOOK AND PUT THE SLIDES ON THE DESKTOP

SLIDE DATA ANALYSIS

SLIDE START A NEW NOTEBOOK

- **Create a new notebook, and give it the name `analysis`**
- For this, you can introduce `File -> New Notebook -> Python 3` as a way to create a new notebook

```
# Data analysis

This notebook introduces the use of `Jupyter` and `Python` for data analysis
```

Red sticky for a question or issue

Green sticky if complete

SLIDE EXAMINE THE DATA

- **SHOW THE TERMINAL ON SCREEN**
- Use the terminal (`head` from this morning)

```
head data/inflammation-01.csv
```

- **Describe the data**: plain text, csv format
  - One row per patient
  - One column per day
  - Values separated by commas
- **State that we'll use the `numpy` library** to work with this in Python

---

SLIDE `PYTHON` LIBRARIES

- Most programming languages have **libraries** (or **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 add specific functionality to the language** - you import as many as you neeed
- `Python` has libraries for many types of work and operations
- **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
- **Import and describe libraries**

```
import numpy
```

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

---

SLIDE LOAD DATA

```
## Load data

Load comma-separated data from a file
```

- The `numpy` library gives us a function called **`loadtxt()` that loads tabular data from a file**
- To use a `function` from a `library`, **the format is usually `library.function()` : *dotted notation***
- **`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

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

- **NOTE: This can be a good place to introduce tab-completion!**
- 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
- **EXECUTE THE CELL**

SLIDE LOADED DATA

- We didn't ask `Python` to do anything with the data, so it it just 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` : MAKE THIS CHANGE IN-PLACE**

```
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)
```

SLIDE WHAT IS OUR DATA? **LIVE DEMO**

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

```
type(data)
```

- **`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 MEMBERS AND ATTRIBUTES

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

```
print(data.dtype)
print(data.shape)
```

- `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

- **Take learners through making notes in the notebook: fence blocks**

```
# Indexing arrays

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[30, 20] # get entry at row 30, column 20 of the array
```

- **Execute the cell**
- In **`Python`** **we index from zero**, so the first element is `data[0, 0]`

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

Red sticky for a question or issue

Green sticky if complete

SLIDE SLICING ARRAYS

- **Take learners through making notes in the notebook**

```
## Slicing arrays

We select sections of an array by *slicing* it - defining the start and end points of the

The *slice* `0:4` means "start at index 0 and go up to, but not including, index 4"
```

- To get a section from the array, index using *square bracket* notation - but specify start and end points, separated by a colon
- 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)
- **Do the two `print()` examples**

```
print(data[0:4, 0:10])
print(data[5:10, 0:10])
print(data[2:4, 2:4])
```

Red sticky for a question or issue

Green sticky if complete

SLIDE MORE SLICES, 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:]`
- **Demo the code**

```
small = data[:3, 36:]
print('small is:')
print(small)
```

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

```
print(data[0:2, :])
```

SLIDE EXERCISE 04

- **PUT THE EXERCISE SLIDE ON SCREEN MCQ: put up four colours of sticky notes**

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

WHEN FINISHED, GO BACK TO THE NOTEBOOK AND PUT THE SLIDES ON THE DESKTOP

SLIDE ARRAY OPERATIONS

```
## Array operations

Arithmetic operations on arrays are performed *elementwise*

The `numpy` package provides functions that perform more complex operations on arrays.
```

- **Arithmetic operations on `array` s are performed elementwise.**

```
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:')
print(data[:3, 36:])
```

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

```
print('doubledata:')
print(doubledata[:3, 36:])
```

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))
```

- We can asssign the output from these functions to variables
- **By default, these functions give summaries of the whole array**

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

Red sticky for a question or issue

Green sticky if complete

SLIDE SUMMARY BY PATIENT

- **What if we want to get summaries patient-by-patient (row-by-row)?**

- We can extract a single row into a variable, and calculate the mean

```
patient_0 = data[0, :] # Row zero only, all columns
print('maximum inflammation for patient 0:', patient_0.max())
```

- **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 0:', numpy.max(data[0, :]))
print('maximum inflammation for patient 2:', numpy.max(data[2, :]))
```

SLIDE SUMMARY OF ALL PATIENTS

- But **what if we want to know about all patients at once?**
- 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
- **We can specify which axis a function applies to**

- **MOVE SLIDE TO SCREEN TO DEMONSTRATE AXES 0 AND 1**

-
- Specifying `axis=0` makes the function work on columns (days)

- Specifying `axis=1` makes the function work on rows (patients)

- **RETURN NOTEBOOK TO SCREEN**

SLIDE `NUMPY` OPERATIONS ON AXES

- **`numpy` functions take an `axis=` parameter** which controls the axis for summary statistic calculations.

```
print(numpy.max(data, axis=1))  # max value for each patient
print(numpy.mean(data, axis=0)) # mean value on each day
```

Red sticky for a question or issue

Green sticky if complete

SLIDE VISUALISATION

SLIDE VISUALISATION

- **Start a new `markdown` notebook**

```
# 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**

SLIDE `JUPYTER` MAGIC

- **`Jupyter` provides another way to control libraries, through *magics***
- `matplotlib` is the *de facto* standard plotting library in `Python`
- **Do the `matplotlib` magic**
- Note that warnings about fonts may be normal.

```
%matplotlib inline
import matplotlib.pyplot
```

- **Import `numpy` and `seaborn`**
- `seaborn` is a library that enables attractive graphs and statistical summaries

```
import numpy
import seaborn
```

SLIDE Load data

- We want to visualise our data, and so we need to load it into a variable in the notebook again
- **Load the data again**

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

SLIDE `MATPLOTLIB` `.IMSHOW()`

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

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

- Here, small values are white, and large values are black (**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` `.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)
ave_plot = matplotlib.pyplot.plot(ave_inflammation)
```

- **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

---

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))
```

- **NOTE we're not defining a variable, this time**

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

- **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 05

- **PUT THE EXERCISE SLIDE ON SCREEN**

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

Red sticky for a question or issue

Green sticky if complete

WHEN FINISHED, GO BACK TO THE NOTEBOOK AND PUT THE SLIDES ON THE DESKTOP

SLIDE FIGURES AND SUBPLOTS

- **THE CODE ALL NEEDS TO GO IN ONE CELL, BUT WE CAN EXECUTE AFTER EACH SECTION TO SHOW BUILD-UP**
- We can put all three plots we just drew into a single figure
- To do this, we use `matplotlib` to **create a figure**, and put it in a variable called `fig`

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

- The `figsize` argument specifies the *width*, then the *height* of the figure being produced, in inches
- We then **create three *axes*** - these are the variables that hold the individual plots
- 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**

```
axes1 = fig.add_subplot(1, 3, 1)                     # Add three subplots
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
- Plot axes properties are usually changed using the `.set_<something>()` syntax
    - Here we're changing only the label on the *y*-axis

```
axes1.set_ylabel('average')                          # Label the graphs
axes2.set_ylabel('max')
axes3.set_ylabel('min')
```

- We can plot on an axis by using its `.plot()` function
    - As before, we can pass the output from the `numpy.max()` function directly

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

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

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

- **This is the most demanding code you have written, so far! ROUND OF APPLAUSE FOR YOURSELVES!**

SLIDE EXERCISE 06

- **PUT THE EXERCISE SLIDE ON SCREEN**

- Note that it helps to change `figsize`
- Otherwise the only change is in `add_subplot()`

```
fig = matplotlib.pyplot.figure(figsize=(3.0, 10.0))  # Create a figure object
axes1 = fig.add_subplot(3, 1, 1)                     # Add three subplots
axes2 = fig.add_subplot(3, 1, 2)
axes3 = fig.add_subplot(3, 1, 3)
axes1.set_ylabel('average')                          # Label and plot the graphs
axes1.plot(numpy.mean(data, axis=0))
axes2.set_ylabel('max')
axes2.plot(numpy.max(data, axis=0))
axes3.set_ylabel('min')
axes3.plot(numpy.min(data, axis=0))
fig.tight_layout()                                   # tidy the figure
```

Red sticky for a question or issue

Green sticky if complete

WHEN FINISHED, GO BACK TO THE NOTEBOOK AND PUT THE SLIDES ON THE DESKTOP

- **NOW, TO DO MORE INTERESTING THINGS, WE NEED TO LEARN A LITTLE MORE ABOUT PROGRAMMING**

SLIDE LOOPS

SLIDE START A NEW NOTEBOOK

- Create a new notebook, and give it the name `loops`

```
## Loops

*Loops* allow us to repeat operations on a series of items.
```

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

- If we want to spell a word, like 'lead' one letter at a time

```
word = "lead"
```

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

```
print(word[0])
print(word[1])
print(word[2])
print(word[3])
```

- But this has some problems - **ASK LEARNERS WHAT PROBLEMS THEY SEE**
- The **approach doesn't scale** - what if our word is hundreds of letters long?
- **What if our word is longer than the indices?** We don't get all the data; if it's shorter, we get an error.
- **demonstrate with `oxygen` and `tin` - MODIFY THE WORD IN PLACE**

SLIDE `FOR` LOOPS

- **`for` loops perform an operation *for* every item *in* a collection**
- **REPLACE THE INDEXING AND DEMO FOR `oxygen` , `lead` , and `tin`**

```
for char in word:
    print(char)
```

- 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

---

SLIDE BUILDING `FOR` LOOPS

- The general loop syntax is defined by a `for` statement, and a *code block*

```
The general loop syntax is

for element in collection:
    <do something with element>
```

- 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**
- **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)

- **DEMO THE CODE BELOW**

```
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 THINGS

- Code in a `for` loop can still see variables defined outside the loop
- **PUT THE CODE INTO A CELL:**

```
length = 0
for vowel in 'aeiou':
    length = length + 1
print('There are', length, 'vowels')
```

- **Ask the learners what output they expect**
- Talk through the operations of the loop, if necessary

SLIDE LOOP VARIABLES

- The *loop variable* alsp still exists once the loop is finished
- **PUT CODE IN A CELL**

```
letter = 'z'
print(letter)
for letter in 'abc':
    print(letter)
print('after the loop, letter is', letter)
```

- **ASK THE LEARNERS WHAT OUTPUT THEY EXPECT**
- 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

---

SLIDE `RANGE()`

- **Make a markdown cell**

```
## `range()`

The `range()` function creates a sequence of numbers
```

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

```
seq = range(3)
print("Range is:", seq)
for val in seq:
    print(val)
```

- **Substitute other ranges and run again**

```
seq = range(3)
seq = range(2, 5)
seq = range(3, 10, 3)
seq = range(10, 0, -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 07

- **PUT THE EXERCISE SLIDE ON SCREEN**

```
result = 1
for val in range(3):
    result = result * 5
print(result)
```

SLIDE EXERCISE 08

```
instr = "Newton"
outstr = ""
for char in instr:
    outstr =  char + outstr
print(outstr)
```

WHEN FINISHED, GO BACK TO THE NOTEBOOK AND PUT THE SLIDES ON THE DESKTOP

SLIDE `enumerate()`

```
## `enumerate()`

The `enumerate()` function creates paired indices and values for elements of a sequence
```

- **DEMO CODE BELOW**

```
seq = enumerate('aeiou')
print("Sequence is:", seq)
for idx, val in seq:
    print(idx, val)
 ```

SLIDE USING `enumerate()`

* We can use enumerate to index lists in order, which can be very useful in a variety of c
* **PUT THE MARKDOWN IN THE NOTEBOOK**

```markdown
Calculate $y$ when $x=5$ when the coefficients are `coeffs = [2, 4, 3, 2, 1]`:

$$y = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4$$
```

We can solve this with a little `Python`

```
x = 5
coeffs = [2, 4, 3, 2, 1]

y = 0
for idx, val in enumerate(coeffs):
    y = y + val * (x ** idx)

print(y)
```

SLIDE LISTS

SLIDE START A NEW NOTEBOOK

```
# Lists

Lists are a built-in `Python` datatype, describing ordered collections of elements.

Lists are defined by comma-separated values, in square brackets.
```

SLIDE LISTS

- Lists are defined as ordered lists of values, in square brackets, separated by commas

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

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

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

- They can be iterated over, in loops

```
for number in odds:
    print(number)
```

SLIDE MUTABILITY

```
## Mutability

`Python` has a concept of mutability. Items that can be changed in-place are *mutable*. Th

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

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

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

- `string` s are **NOT** mutable

```
name = 'Darwin'
name[0] = 'd'
```

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

- **DEMO CODE**

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

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

```
print("your list:", your_list)
```

- 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, you can make a *copy* of a `list` by *slicing* it**, or using the `list()` function that returns a new list
- **DEMO CODE - MODIFY THE CODE ABOVE IN-PLACE IN THE NOTEBOOK **

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

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

Red sticky for a question or issue

Green sticky if complete

SLIDE NESTED LISTS

- **ADD MARKDOWN CELL**

```
## Nested `list`s and `list` functions

A `list` can contain any other datatype - even another `list`!
```

- **`list`s can contain any datatype, even other lists**
- Imagine we have a grocery store with three shelves, and the items on the shelves are arranged with {pepper, zucchini, onion} on the top shelf, {cabbage, lettuce, garlic} on the middle shelf, and {apple, pear, banana} on the lower shelf.
- **We can represent this in a *nested list***: one list per shelf, and a list that contains the three lists, to represent the grocery store.
- **Demo code**

```
shelves = [['pepper', 'zucchini', 'onion'],
           ['cabbage', 'lettuce', 'garlic'],
           ['apple', 'pear', 'banana']]
```

NOTE: This should remind you of the `numpy` array you loaded earlier! Work through the code below

```
print(shelves[0])
print([shelves[0]])
print(shelves[0][0])
```

SLIDE `LIST` FUNCTIONS

- `list`s are `Python` objects and have a number of useful functions to modify their contents
- `.append()` adds a value to the end of the list

```
odds.append(9)
print("odds after adding a value:", odds)
```

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

```
odds.reverse()
print("odds after reversing:", odds)
```

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

```
print(odds.pop())
print("odds after popping:", odds)
```

SLIDE OVERLOADING

- We can add (`+`) and multiply (`*`) lists, even though they're not really arithmetic operations
- *Overloading* refers to an *operator* (e.g. `+`) having more than one meaning, depending on the thing it operates on.

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

- **NOTE: multiplication of lists does not work like multiplication of `numpy` arrays**

```python
counts = [2, 4, 6, 8, 10]
repeats = counts * 2
print(repeats)
```

- **Ask the learners what 'addition' (`+`) and 'multiplication' (`*`) do for lists**

SLIDE MAKING CHOICES

SLIDE START A NEW NOTEBOOK

- **Call it `choices`**
- **Add an introduction cell**

```
# Making Choices

We often want to make the computer perform one task if some condition is true, but a diffe
```

- **Add the Python code to the markdown**

```
if <condition>:
  <executed if condition is True>
```

SLIDE CONDITIONALS

- We often want the computer to 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
- **EXECUTE CODE**

```python
num = 37
if num > 100:
    print('greater')
print('done')
```

- **CHANGE NUMBER TO VARIOUS VALUES IN THE SAME CELL**

```
num = 137
num = 100
```

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

```
if 'atlas' == 'atlas':
    print("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`**

- **MAKE CHANGES AND EXECUTE CODE IN EXISTING CELLS**

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

```
if 'atlas' == 'atlash':
    print("the same")
else:
    print('different')
```

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.

- **EXECUTE DEMO CODE IN EXISTING CELL**

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

- **NOTE: the test for equality is a double-equals!**

Red sticky for a question or issue   Green sticky if complete

---

SLIDE COMBINING CONDITIONS

- We can **combine conditions using *Boolean Logic***
- Operators include `and` , `or` and `not`
- **EXECUTE CODE IN NEW CELL**

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

- **VARY THE CODE IN PLACE**

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

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

SLIDE EXERCISE 09

- **PUT THE EXERCISE SLIDE ON SCREEN**
- **MCQ: Put up four stickies**

- Solution: `c`

WHEN FINISHED, GO BACK TO THE NOTEBOOK AND PUT THE SLIDES ON THE DESKTOP

SLIDE MORE OPERATORS

- **ADD THE MARKDOWN**

```
## Operators

* `==` (equals)
* `in`
```

- 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
- **DEMO CODE**

```
print(1 == 1)
print(1 == 2)
```

- `in` is the **membership operator**, and returns `True` if the left-hand-side value is in the right-hand-side value
- **DEMO CODE**

```
print('a' in 'toast')
print('b' in 'toast')
print(1 in [1, 2, 3])
print(1 in range(3))
print(1 in range(2, 10))
```

SLIDE ANALYSING MULTIPLE FILES

SLIDE START A NEW NOTEBOOK

- **Call it `files`**
- **ADD NEW HEADER CELL**

```
# Analysing Multiple Files

We're now almost ready to start analysing multiple files of inflammation data.
```

- **ADD IMPORTS**

```
%matplotlib inline

import matplotlib.pyplot
import numpy
import seaborn
```

SLIDE ANALYSING MULTIPLE FILES

- We have **received several files of data** from the inflammation studies, and 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.

SLIDE THE `os` MODULE

- **New Markdown cell**

```
## The `os` module

Allows us to interact with the computer's filesystem
```

- 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**
- **IMPORT THE MODULE**

```
import os
```

SLIDE `OS.LISTDIR`

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

```
os.listdir('.')
```

- Our data is in the `'data'` directory
- **Reuse the cell**

```
os.listdir('data')
```

- **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*

```python
for file in os.listdir('data'):
    if 'inflammation' in file:
        print(file)
```

- 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 populate it one file at a time, using `.append()`
- **ADAPT THE EXISTING CELL**

```python
files = []
for file in os.listdir('data'):
    if 'inflammation' in file:
        files.append(file)
print(files)
```

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**
- **EXAMPLE CODE IN NEW CELL**

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

- **MODIFY PREVIOUS CELL TO GET**

```python
files = []
for file in os.listdir('data'):
    if 'inflammation' in file:
        files.append(os.path.join('data', file))
print(files)
```

SLIDE VISUALISING THE DATA

- **Add markdown**

```
## Visualising data

We can now load data from each file in turn, and visualise the mean, minimum and maximum v
```

- 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**
- **1 - show that we see each filename in turn** `python for file in files: print(file)`
- **2 - show the data in each file**

```
for file in files:
    print(file)

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

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

```
for file in files:
    print(file)

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

    # create figure and 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**

```
for file in files:
    print(file)

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

    # create figure and 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 axes
    axes1.set_ylabel('average')
    axes2.set_ylabel('maximum')
    axes3.set_ylabel('minimum')
```

- **5 - plot the data**

```
for file in files:
    print(file)

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

    # create figure and 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 axes
    axes1.set_ylabel('average')
    axes2.set_ylabel('maximum')
    axes3.set_ylabel('minimum')

    # plot 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**

```
for file in files:
    print(file)

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

    # create figure and 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 axes
    axes1.set_ylabel('average')
    axes2.set_ylabel('maximum')
    axes3.set_ylabel('minimum')

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

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

- **Show the collapse/expand click option in the notebook**

Red sticky for a question or issue   Green sticky if complete

---

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?
- **ADD TO EXISTING CODE BEFORE PLOT**

```
if numpy.max(data, axis=0)[0] == 0 and numpy.max(data, axis=0)[20] == 20:
    print('Suspicious looking maxima!')
```

SLIDE SUSPICIOUS MINIMA

- Are all the minima zero? (do they sum to zero?)
- **ADD TO EXISTING CODE BEFORE PLOT - AS ELIF**

```
elif numpy.sum(numpy.min(data, axis=0)) == 0:
    print('Minima sum to zero!')
```

SLIDE BEING TIDY

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

```
else:
    print('Seems OK!')
```

Red sticky for a question or issue

Green sticky if complete

SLIDE MAKING A SCRIPT

SLIDE RUN THE NOTEBOOK

- **Run the notebook:** `Kernel -> Restart & Run All`
- Interactive output appears in the notebook
- `Jupyter` **is good for this kind of work** - prototyping, interactive, teaching
- But it's not how most people write `Python` day-to-day
- **We'll use our notebook as the basis for a script**

SLIDE DOWNLOAD `PYTHON` CODE

- **Download the notebook** as a script: `File -> Download As -> Python`
- It will download as `files.py`

PUT THE TERMINAL ON SCREEN

- **Move the file to your working directory**
- **OPEN THE FILE WITH AN EDITOR**

- `files.py` is a plain text file, containing `Python` code and comments, from your notebook
- **All the Markdown has been converted to comments**
- **All the `In[]` and `Out[]` markers are also now comments**

SLIDE RUN `PYTHON` CODE

- **In the editor**
  - **COMMENT OUT `get_ipython().magic('matplotlib inline')`** magic
- **In the terminal**
  - **RUN `python files.py`**
- **OUTPUT MAY DIFFER DEPENDING ON INDIVIDUALS' SETUPS - ASK WHAT THEY SEE**

SLIDE Edit `PYTHON` CODE

- Seeing each image in turn is not convenient
- We'll **write each image to file instead of viewing it**
- `EDIT - SAVE - EXECUTE` cycle
- **EDIT THE FILE AS SHOWN BELOW**

```
# matplotlib.pyplot.show()
outfile = file + '.png'
print("Writing PNG to", outfile)
matplotlib.pyplot.savefig(outfile)
```

- The files are placed in the `data` directory

SLIDE CONCLUSIONS

PUT THE SLIDES ON SCREEN

SLIDE LEARNING OUTCOMES

- `Jupyter` notebooks
- 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!**