LESSON 01 - Building Programs With Python

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

SLIDE Building Programs With Python (1)

SLIDE INTRODUCTION

SLIDE GOAL 1

- We are teaching programming, not `Python` *per se*
- We need to use *some* language, though
- Python is free, and likely to be usable on your machine
- Python is widely-used, and there's lots of support online
- It can be easier for novices to pick up than other languages
- You should use what is common in your area/with your colleagues
- The principles of programming are the same in other languages

---

SLIDE GOAL 2

- We're using a motivating example of data analysis
- Data is in plain text, tabular (CSV)
- Data represents patients and daily measurements
- 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 datasets, and share with others (transparency)

SLIDE SETUP

SLIDE SETTING UP - 1 - **DEMO**

- We want a neat (clean) working environment
- Change directory to desktop (in terminal or Explorer)
- Create directory `python-novice-inflammation`
- Change your working directory to that directory

SLIDE SETTING UP - 2 - **DEMO**

- We need to acquire our data (and also a little code that can help us)
- Copy `.zip` files from repository (or online!) **PUT IN ETHERPAD**
- Extract files (command-line or in Explorer)

SLIDE GETTING STARTED

SLIDE STARTING `JUPYTER` **DEMO**

- Start `Jupyter` from the command-line

SLIDE `JUPYTER` LANDING PAGE **DEMO**

- 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.
- 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 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 [ ]:`

SLIDE MY FIRST NOTEBOOK **DEMO**

- Give the notebook 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: `#`
    - Italics are shown by enclosing text in single asterisks: `*italic*`
    - Typewriter text/code is shown by enclosing in backticks: ```
    - 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

---

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**

- Note that after execution, the `In [ ]` prompt now has a number in it to indicate the order in which cells are 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.
- **ASK THE LEARNERS GO THROUGH THE EXERCISE**

SLIDE EXERCISE 01

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)

- 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
    - **LET'S DO THIS FOR REAL IN PYTHON**
    - 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`

SLIDE WORKING WITH VARIABLES

- Lead the students through the code:

```python
weight_kg = 55
print(weight_kg)
2.2 * weight_kg
print("weight in pounds", 2.2 * weight_kg)
weight_kg = 57.5
print("weight in kilograms is now:", weight_kg)
```

- Assign an integer (assignment is the same for all data items)
- Print `weight_kg` to see its value
- Variables can be substituted by name wherever a value would go
- The `print()` function will take more than one argument, separated by commas, and print them
- Reassigning to the same variable overwrites the old value

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

- Lead students through the code

```python
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`, we did not change `weight_lb` when we did so**

SLIDE EXERCISE 02 (5MIN)

- The solution is `3`

SLIDE EXERCISE 03 (5MIN)

- The code prints `Hopper Grace`

SLIDE WHO'S WHO IN MEMORY?

- In a `Jupyter` notebook or `iPython` terminal, `%whos` will list the variables that are in memory, and their contents.

SLIDE DATA ANALYSIS

SLIDE START A NEW NOTEBOOK

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

SLIDE EXAMINE THE DATA

- Use the terminal (`head` from this morning)

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

- Describe plain text, csv format
- State that we'll use the `numpy` library

SLIDE `PYTHON` LIBRARIES

- `Python` contains many basic and general functions and tools
- Specialised tools are packaged in *libraries*
- We can 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
- Libraries add functionality to your current `Python` instance

SLIDE `JUPYTER` MAGIC

- `Jupyter` provides another way to load libraries, through *magics*
- **Do the `pylab` magic**
- **Import `numpy` and `seaborn`**
- Note that warnings about fonts may be normal.

SLIDE `NUMPY`, `SEABORN`, `PYLAB`

- `numpy` is a library that provides functions and pethods to work with arrays and matrices, such as those in your dataset
- `seaborn` is a library that enables attractive graphs and statistical summaries
- `pylab` is a library that mimics `MatLab` in `Python`, providing a number of useful tools for numerical operations and visualisation

---

SLIDE LOAD DATA

- The `numpy` library gives us a function called `loadtxt()` that loads tabular data from a file
- It's used as `numpy.loadtxt()`
- *Dotted notation* tells us that `loadtxt()` belongs to `numpy`
- `loadtxt()` expects two *arguments* or *parameters*
- 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
- `Python` will accept double- or single-quotes around strings
- **Execute the line in a cell**

SLIDE LOADED DATA

- Since we didn't ask `Python` to do anything with the data, it just shows it 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`**

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

- Take the learners through the code:

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

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

---

SLIDE MEMBERS AND ATTRIBUTES

- When we created `data` we didn't just create the array, we also created information about the array, called *members* or *attributes*
- This information belongs to `data` so is accessed in the same way as a module function, through *dotted notation*

SLIDE INDEXING ARRAYS

- **Take learners through making notes in the notebook**
- To get a single element from the array, index using *square bracket* notation - row first, then column
- In `Python` we index from zero, so the first element is `data[0, 0]`
- **Do the two `print()` examples**

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

SLIDE SLICING ARRAYS

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

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

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**

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

SLIDE EXERCISE 04

- The value is `oxy`, number `3`

SLIDE ARRAY OPERATIONS

- Arithmetic operations on `array` s are performed elementwise.
- **Demo the code**
- Operations with scalars act *elementwise*
- Operations with two arrays are *elementwise*

```
doubledata = data * 2.0
print('original:')
print(data[:3, 36:])
print('doubledata:')
print(doubledata[:3, 36:])
```

SLIDE `NUMPY` FUNCTIONS

- `numpy` provides functions that can perform *more complex* operations on arrays
- Some of these operations include statistical summaries: `.mean()` , `.min()` , `.max()` etc.
- **Demo code**
- These operations give summaries of the whole array
- The `data` array also has these summary functions

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

SLIDE SUMMARY BY PATIENT

- What if we want to get summaries patient-by-patient (row-by-row)?
- **Demo code**
- We can extract a single row into a variable, and calculate the mean
- We can also apply the `numpy` function directly, without creating a variable
- **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)**

```
patient_0 = data[0, :] # Row zero only, all columns
print('maximum inflammation for patient 0:', patient_0.max())
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 instead specify which axis a function applies to
- Specifying `axis=0` makes the function work on columns (days)
- Specifying `axis=1` makes the function work on rows (patients)

SLIDE `NUMPY` OPERATIONS ON AXES

- **Demo the code**

```python
print(numpy.max(data, axis=1))
print(data.mean(axis=0))
```

SLIDE VISUALISATION

SLIDE VISUALISATION

- **Start a new `markdown` cell**
- Outline how visualisation is a large topic that deserves more attention
- Show off the SSI course materials

SLIDE `MATPLOTLIB`

- There's no "official" plotting library or graphics library in `Python`
- `matplotlib` is the `de facto` standard
- **lots of tools are built on `matplotlib`**
- We imported `seaborn`, which makes `matploltlib` output a bit more publication-ready
- We used `%pylab inline`, which puts `matplotlib` output in the notebook
- **Demo code**

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

SLIDE `MATPLOTLIB` `.IMSHOW()`

- The `.imshow()` function converts matrix values into an image
- Here, small values are white, and large values are black (*8you can change this colour scheme…*)
- From the image, we cna see inflammation rising and falling over a 40-day period for all patients.

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
- We plot the data with `matplotlib`
- **Demo the code**

```
ave_inflammation = numpy.mean(data, axis=0)
ave_plot = matplotlib.pyplot.plot(ave_inflammation)
```

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

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

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

SLIDE FIGURES AND SUBPLOTS

- **Demo code**
- The code needs to all go in a single cell

```python
fig = matplotlib.pyplot.figure(figsize=(10.0, 3.0))  # Create a figure object
axes1 = fig.add_subplot(1, 3, 1)                      # Add three subplots
axes2 = fig.add_subplot(1, 3, 2)
axes3 = fig.add_subplot(1, 3, 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
```

- **This is the most demanding code you will write, so far**
- 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`
    - 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 might need to be drawn out on the board**
- 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
- We can plot on an axis by using its `.plot()` function
    - As before, we can pass the output from the `numpy.max()` function directly
- Finally, we'll tighten up the presentation by using `fig.tight_layout()` - a function that moves the axes until they are visually pleasing.

SLIDE EXERCISE 06

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

```python
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
```

SLIDE LOOPS

SLIDE START A NEW NOTEBOOK

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

- If we want to spell a word one letter at a time, we can *index* each letter in turn
- **Demo code**

```python
word = "lead"
print(word[0])
print(word[1])
print(word[2])
print(word[3])
```

- But this is bad - **Why?**
- The approach doesn't scale - what if our word is hundreds of letters long?
- 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`**

SLIDE `FOR` LOOPS

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

```
word = "lead"
for char in word:
    print(char)
```

- This has two advantages - it's shorter code (less opportunity for error), and 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
  - **demonstrate with `oxygen` and `tin`**

SLIDE BUILDING `FOR` LOOPS

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

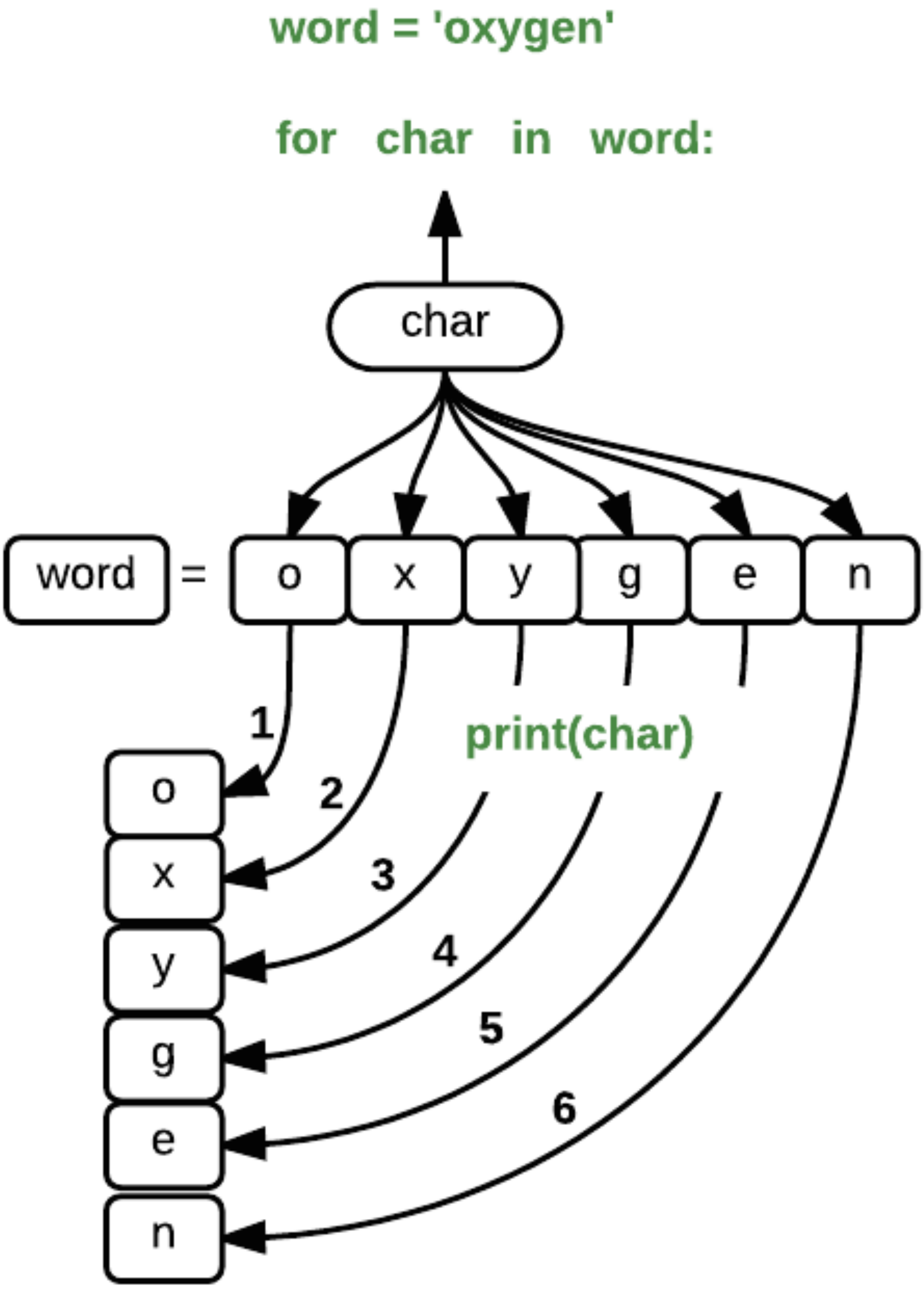
```
for element in collection:
    <do things with element>
```

- 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)
- **Demonstrate**

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

    print("Still in the loop!")
print("This line is not part of the loop")
```

SLIDE `FOR` LOOP CYCLES



- In this image, each successive character of `word` is placed, one at a time, in the variable `char`.
    - In each cycle of the loop, the contents of the variable `char` are printed
    - Once the code block is complete, the next cycle of the loop starts

---

SLIDE COUNTING THINGS

- **Ask the learners what output they expect from the code below**

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

- **Demo the code**
- Talk through the operations of the loop, if necessary

SLIDE LOOP VARIABLES

- The *loop variable* still exists once the loop is finished
- **Demo code**

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

- 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()`

- The `range()` function creates a sequence of numbers.
- The sequence depends on the number and value of arguments given
- **Demo code - loop over all three options**

```
range(3)
range(2, 5)
range(3, 10, 3)
for val in range(3, 10, 3):
    print(val)
```

- 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

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

SLIDE EXERCISE 08

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

SLIDE EXERCISE 09

```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

SLIDE LISTS

- `list` s are a built-in `Python` datatype, denoting ordered collections of elements
- `list` s are defined by square brackets, and comma-separated values
- **Demo code**

```python
odds = [1, 3, 5, 7]
print('odds are:', odds)
print('first and last:', odds[0], odds[-1])
for number in odds:
    print(number)
```

- They can be indexed and sliced, as seen for arrays
- They can be iterated over, in loops

SLIDE MUTABILITY

- `list`s and `string`s are both sequences
- **BUT** you can change the elements in a list, after it is created: **lists are mutable**
- `string`s are **NOT** mutable
- **Demo code**

```
names = ['Newton', 'Darwing', 'Turing'] # typo in Darwin's name
print('names is originally:', names)
names[1] = 'Darwin' # correct the name
print('final value of names:', names)
name = 'Darwin'
name[0] = 'd'
```

SLIDE CHANGER DANGER

- There are risks associated with modifying lists in-place
- **Demo code**

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

```
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)
```

SLIDE NESTED LISTS

- `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**

```
x = [['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([x[0]])
print(x[0])
print(x[0][0])
```

SLIDE `LIST` FUNCTIONS

- `list` s are `Python` objects and have a number of useful functions to modify their contents
- **Demo code**

```
odds.append(9)
print("odds after adding a value:", odds)
odds.reverse()
print("odds after reversing:", odds)
print(odds.pop())
print("odds after popping:", odds)
```

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)
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

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 (`:`) and 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
- **Demo code**

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

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 (`:`) and the indented code block beneath it executes if the condition is `false`
- **Demo code**

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

SLIDE CONDITIONAL LOGIC

- 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**.
- **Demo code**

```
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!**

SLIDE COMBINING CONDITIONS

- Conditions can be combined using *Boolean Logic*
- Operators include `and` , `or` and `not`
- **Demo code**

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

SLIDE EXERCISE 10

- Solution: `C`

SLIDE MORE OPERATORS

- There are two operators you will meet and use frequently
- `==` (double-equals) is the quality operator, and returns `True` if the left-hand-side value is equal to the right-hand-side value
- `in` is the membership operator, and returns `True` if the left-hand-side value is in the right-hand-side value
- **Demo code**

```python
print(1 == 1)
print(1 == 2)
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 LIST COMPREHENSIONS

- We often want to loop over a list of elements and make a decision on the basis of whether the element meets some condition.
- *List comprehensions* offer a concise way to do this
- **Demo code**
- We can write a loop that checks letters for whether they're a vowel and, if they are, convert them to upper case and add them to a list.

```python
letters = 'abcdefghijklmnopqrstuvwxyz'
vowels = 'aeiou'

result = []
for l in letters:
    if l in vowels:
        result.append(l.upper())
print(result)
```

- We can do this in a single line with a *list comprehension*
- Firstly, we rewrite the loop as a *list comprehension*
- By enclosing the for l in letters part of the loop in square brackets, we are asking the loop to return a list.
- By asking directly for l in the brackets, we get the loop to return each element l to us as we go round the loop

```python
result = [l for l in letters]
print(result)
```

- In addition to asking for each element with each cycle around the loop, we can do things with or to that element, such as convert it to upper case by calling its .upper() method/function.

```python
result = [l.upper() for l in letters]
print(result)
```

- Finally, we can add a condition to the list comprehension so that the loop only returns values when the condition evaluates to True.
- *Here, we require that the letter in l can be found in the vowels string.

```python
result = [l.upper() for l in letters if l in vowels]
print(result)
```

SLIDE ANALYSING MULTIPLE FILES

SLIDE START A NEW NOTEBOOK

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 in the data, visualise the data, loop over contents, 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

- 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
- **Do imports in notebook**
- **NOTE: it's usual to abbreviate imported modules, e.g. `numpy` to `np`, if they are used frequently**

```python
%pylab inline

import matplotlib.pyplot
import numpy as np
import os
import seaborn
```

SLIDE `OS.LISTDIR`

- The `.listdir()` function lists the contents of a directory
- Our data is in the `'data'` directory
- **Demo code**

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

- The list can be filtered with a `for` loop or *list comprehension*

```python
files = [f for f in os.listdir('data')]
print(files)
```

- We can use the .startswith() function of the `string` object (all the filenames are strings) as the conditional
- We keep only filenames that start with `inflammation`.

```python
files = [f for f in os.listdir('data') if f.startswith('inflammation')]
print(files)
```

SLIDE `OS.PATH.JOIN`

- The `os.listdir()` function only returns filenames, not the *path* (relative or absolute) to those files.
- To construct a path, we can use the `os.path.join()` function. This 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**
- **Demo code**

```python
print(os.path.join('data', 'inflammation-01.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 `np.loadtxt()`
    - We can calculate summary statistics with `mp.mean()`, `np.max()`, etc.
    - We can create figures with `matplotlib`, and arrays of figures with `.add_subplot()`

SLIDE VISUALISATION CODE

```python
filenames = [os.path.join('data', f) for f in os.listdir('data')
             if f.startswith('inflammation')]

for f in filenames:
    print(f)

    data = np.loadtxt(fname=f, delimiter=',')

    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)

    axes1.set_ylabel('average')
    axes1.plot(np.mean(data, axis=0))

    axes2.set_ylabel('max')
    axes2.plot(np.max(data, axis=0))

    axes3.set_ylabel('min')
    axes3.plot(np.min(data, axis=0))

    fig.tight_layout()
    matplotlib.pyplot.show()
```

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