LESSON 02 - 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 **DEMO**

- We want a neat (clean) working environment
- **IF NECESSARY!**
- Change directory to desktop (in terminal or Explorer)
- Change your working directory to `python-novice-inflammation` (from yesterday/earlier)

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**

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 need to package our code for reuse.
- **We do this by writing functions**

SLIDE 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` - defined by `f()` .
- **Not all functions in code take an input, or produce a usable output, but the principle is generally the same.**

SLIDE MY FIRST FUNCTION

- We'll write a function to convert Fahrenheit to Kelvin, called `fahr_to_kelvin()`
- The mathematical function is described:
  - This function takes `x`, subtracts 32, multiplies by 5/9, and adds 273.15
- In `Python` this translates to the code below
  - Functions are *defined* by the `def` keyword
  - The name of the function follows the `def` keyword (equivalent to `f` in the mathematical example)
  - 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`.
  - The function performs a calculation, which is *returned* by the `return` statement.
  - The value of `temp` is taken through the same calculation as in the mathematical function, and is then *return*ed.
- **Demo code**

SLIDE Calling the function

- We call `fahr_to_kelvin` in exactly the same way we call any other function we've seen so far

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

SLIDE Composing functions

- Composing `Python` functions works just like mathematical functions: `y = f(g(x))`
- Suppose we have a function that converts Kelvin to Celsius, called `kelvin_to_celsius()`
- **Demo code**

```
def kelvin_to_celsius(temp_k):
    return temp_k - 273.15
print('absolute zero in Celsius:', kelvin_to_celsius(0.0))
```

- 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

- We can wrap this composed function inside a new function: `fahr_to_celsius` :

- **Demo code**

```
def fahr_to_celsius(temp_f):
    return kelvin_to_celsius(fahr_to_kelvin(temp_f))
print('freezing point of water in Celsius:', fahr_to_celsius(32.0))
```

- **This is how programs are built: combining small bits into larger bits until the function we want is obtained**

SLIDE EXERCISE 01

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

SLIDE SCOPE

- Variables defined within a function, including parameters, are not 'visible' outside the function
- This is called *function scope* **Demo code**

```
a = "Hello"

def my_fn(a):
  a = "Goodbye"

my_fn(a)
print(a)
```

- To move values to and from functions, you should generally `return` them from the function
- **Demo code**

```
a = "Hello"

def my_fn(a):
  a = "Goodbye"

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

SLIDE EXERCISE 02

- Solution: `1: 7 3` (this differs from that on the SWC page)

SLIDE ANALYSIS

SLIDE TIDYING UP

- Now we can write functions
- Let's make the inflammation analysis easier to reuse
- **Do the imports!**

```
%pylab inline

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

SLIDE `ANALYSE()`

- We'll write a function called `analyse()` that plots the data
- **Demo code**

```
def analyze(data):
    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(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()
    matplotlib.pyplot.show()
```

SLIDE `DETECT_PROBLEMS()`

- We noticed before that some data was questionable
- This function spots problems with the data:
  - The first datapoint is 0, and the 20th is 20
  - The sum of all minima is zero
- **Demo code**

```
def detect_problems(data):
    if numpy.max(data, axis=0)[0] == 0 and numpy.max(data, axis=0)[20] == 20:
        print('Suspicious looking maxima!')
    elif numpy.sum(numpy.min(data, axis=0)) == 0:
        print('Minima add up to zero!')
    else:
        print('Seems OK!')
```

SLIDE CODE REUSE

- Now we can identify the input files, then apply one function per action in a loop:
  - Load the data with `np.loadtxt()`
  - Print the filename
  - `analyse()` the data
  - `detect_problems()` in the data
- **Demo code**

```
filenames = [os.path.join('data', f) for f in os.listdir('data')
             if f.startswith('inflammation')]
for fname in filenames:
    data = np.loadtxt(fname, delimiter=",")
    print(fname)
    analyse(data)
    detect_problems(data)
```

- **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**

SLIDE 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
- **Demo code**

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

SLIDE TEST DATASETS

- 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
- **Demo code**

```
z = np.zeros((2, 2))
print(centre(z, 3.0))
```

- If this works, we'll try it on real data

SLIDE REAL DATA

- **Demo code**

```
data = numpy.loadtxt(fname='data/inflammation-01.csv', delimiter=',')
print(centre(data, 0))
```

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

SLIDE CHECK PROPERTIES

- We can check properties of the original and centred data
  - `mean` , `min` , `max` , `std`
- We'd expect the mean of the new dataset to be approximately `0.0`
- The variance of the dataset should be unchanged.
- Also, the range (`max` - `min`) should be unchanged.
- **Demo code**

```
centred = centre(data, 0)
print('original min, mean, and max are:', np.min(data), np.mean(data), np.max(data))
print('min, mean, and max of centered data are:', np.min(centred),
      np.mean(centred), np.max(centred))
print('std dev before and after:', np.std(data), np.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
- **Demo code**

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

- 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:

```
help(centre)
```

- Using the triple quotes (""") allows us to use a multi-line string to describe the function:

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

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

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
- **Demo code**

```
centre([1, 2, 3], 0)
```

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

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

    Example: centre([1, 2, 3], 0) => [-1, 0, 1]
    """
    return (data - np.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 EXERCISE 03

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

SLIDE ERRORS AND EXCEPTIONS

SLIDE CREATE A NEW NOTEBOOK

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 one, so we can look at it
- **Demo code**

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

favourite_ice_cream()
```

SLIDE PARTS OF A TRACEBACK

- **Talk through the traceback on 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
- The second step is 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'
- *Syntax* errors occur when the code is not interpretable as valid `Python`
- **Demo code**

```
1  def some_function()
2      msg = "hello, world!"
3      print(msg)
4       return msg
```

SLIDE SYNTAX TRACEBACK

- `Python` tells us there's a `SyntaxError` - the code isn't written correctly
- 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**

SLIDE FIXED?

- Show fixed code
- **Demo code**

```
1  def some_function():
2      msg = "hello, world!"
3      print(msg)
4       return msg
```

SLIDE NOT QUITE

- `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
- **Demo code**

```
print(a)
```

```
---------------------------------------------------------------------------
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**

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

```
 ---------------------------------------------------------------------------
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`

```
letters = ['a', 'b', 'c']
print("Letter #1 is", letters[0])
print("Letter #2 is", letters[1])
print("Letter #3 is", letters[2])
print("Letter #4 is", letters[3])

Letter #1 is a
Letter #2 is b
Letter #3 is c
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-7-656a22fa6ec5> in <module>()
      3 print("Letter #2 is", letters[1])
      4 print("Letter #3 is", letters[2])
----> 5 print("Letter #4 is", letters[3])

IndexError: list index out of range
```

SLIDE EXERCISE 04

```
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)
```

SLIDE DEFENSIVE PROGRAMMING

SLIDE CREATE A NEW NOTEBOOK

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 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, that speeds up software development and helps ensure that your code is doing what you intend.

SLIDE ASSERTIONS

- Assertions are a `Pythonic` way to see if code runs correctly
    - 80-90% of the `Firefox` source code is assertions!
- We `assert` that a *condition* is `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.

```
assert <condition>, "Some text describing 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)
```

- **Demo code**

```
---------------------------------------------------------------------------
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 which *assertion* failed.