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:
 - $\circ~$ This function takes $~\mathbf{x}$, 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

```
1 def kelvin_to_celsius(temp_k):
2 return temp_k - 273.15
3 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:

```
1 temp_f = 212.0
2 temp_c = kelvin_to_celsius(fahr_to_kelvin(temp_f))
3 print(temp_c)
```

SLIDE NEW FUNCTIONS FROM OLD

• We can wrap this composed function inside a new function: fahr to celsius :

• Demo code

```
1 def fahr_to_celsius(temp_f):
2     return kelvin_to_celsius(fahr_to_kelvin(temp_f))
3     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

1	def outer(s)
2	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

1	a = "Hello"
2	
3	<pre>def my_fn(a):</pre>
4	a = "Goodbye"
5	
6	<pre>my_fn(a)</pre>
7	<pre>print(a)</pre>

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



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!



SLIDE ANALYSE()

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

```
def analyze(data):
1
2
        fig = matplotlib.pyplot.figure(figsize=(10.0, 3.0))
3
4
        axes1 = fig.add_subplot(1, 3, 1)
5
         axes2 = fig.add_subplot(1, 3, 2)
6
        axes3 = fig.add_subplot(1, 3, 3)
7
8
         axes1.set ylabel('average')
9
         axes1.plot(numpy.mean(data, axis=0))
10
         axes2.set_ylabel('max')
11
        axes2.plot(numpy.max(data, axis=0))
12
13
        axes3.set_ylabel('min')
14
         axes3.plot(numpy.min(data, axis=0))
15
16
        fig.tight_layout()
17
18
        matplotlib.pyplot.show()
```

SLIDE DETECT_PROBLEMS()

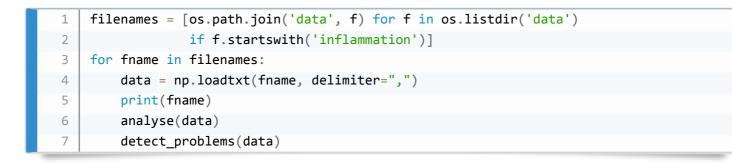
- · We noticed before that some data was questionable
- This function spots problems with the data:
 - The first datapoint is 0, and the 20th is 20
 - The sum of all minima is zero
- Demo code

1	<pre>def detect_problems(data):</pre>
2	<pre>if numpy.max(data, axis=0)[0] == 0 and numpy.max(data, axis=0)[20] == 20:</pre>
3	<pre>print('Suspicious looking maxima!')</pre>
4	<pre>elif numpy.sum(numpy.min(data, axis=0)) == 0:</pre>
5	<pre>print('Minima add up to zero!')</pre>
6	else:
7	<pre>print('Seems OK!')</pre>

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
```



- 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

```
1 def centre(data, desired):
2 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

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

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

SLIDE REAL DATA

Demo code

```
1 data = numpy.loadtxt(fname='data/inflammation-01.csv', delimiter=',')
2 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

```
1 centred = centre(data, 0)
2 print('original min, mean, and max are:', np.min(data), np.mean(data), np.max(data))
3 print('min, mean, and max of centered data are:', np.min(centred),
4 np.mean(centred), np.max(centred))
5 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

```
1 def centre(data, desired):
2 """Returns the array in data, recentered around the desired value."""
3 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)

1

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

```
1 def centre(data, desired):
2 """Returns the array in data, recentered around the desired value.
3
4 Example: centre([1, 2, 3], 0) => [-1, 0, 1]
5 """
6 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

1

```
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:

```
1 def centre(data, desired=0.0):
2 """Returns the array in data, recentered around the desired value.
3
4 Example: centre([1, 2, 3], 0) => [-1, 0, 1]
5 """
6 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:

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

SLIDE EXERCISE 03

1	<pre>def rescale(data):</pre>
2	"""Returns input array rescaled to [0.0, 0.1]."""
3	l = np.min(data)
4	h = np.max(data)
5	return (data - L) / (H - L)

SLIDE ERRORS AND EXCEPTIONS

SLIDE CREATE A NEW NOTEBOOK

SLIDE ERRORS

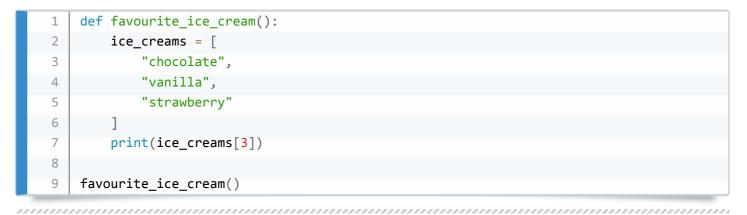
• Programming is essentially just making errors over and over again until the code works ;)

- 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



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

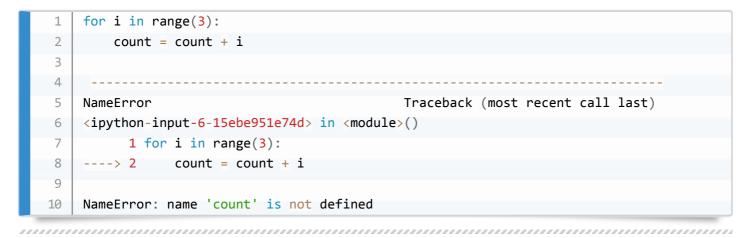
- Python now tells us that there's an IndentationError
- We don't learn about all the syntax errors at one time Python gives up after the first one it finds
- (fixing the first error in a file might correct all subsequent errors)

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

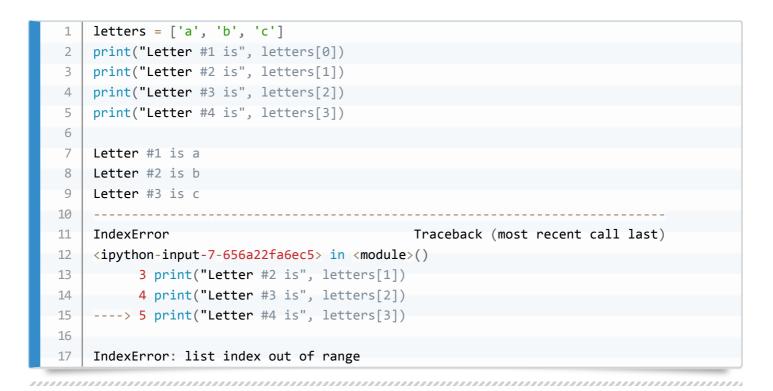
```
1 print(a)
2 
3 
------
4 NameError Traceback (most recent call last)
5 <ipython-input-5-c5a4f3535135> in <module>()
6 
----> 1 print(a)
7
8 NameError: name 'a' is not defined
```

• This is true in functions/loops, too



SLIDE INDEX ERRORS

• If you try to access an element of a collection that does not exist, you'll get an IndexError



SLIDE EXERCISE 04

1	message = ""
2	<pre>for number in range(10):</pre>
3	# use a if the number is a multiple of 3, otherwise use b
4	if (number % 3) == 0:
5	message = message + "a"
6	else:
7	message = message + "b"
8	<pre>print(message)</pre>

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.

1 | assert <condition>, "Some text describing the problem"

......

SLIDE EXAMPLE ASSERTION

• Type code then ask learners what it does

```
1 numbers = [1.5, 2.3, 0.7, -0.001, 4.4]
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)
```

Demo code

```
1
2
    AssertionError
                                                Traceback (most recent call last)
3
    <ipython-input-1-985f50018947> in <module>()
          2 \text{ total} = 0.0
4
5
          3 for n in numbers:
               assert n > 0.0, 'Data should only contain positive values'
6
    ----> 4
7
          5
                 total += n
8
       6 print('total is:', total)
9
10
    AssertionError: Data should only contain positive values
```

• The traceback tells us which assertion failed.