LESSON 02 - 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

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

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 refactor our code from yesterday
- We're going to **document** what the code does
- · We're going to catch errors in our code, and respond sensibly

......

SLIDE SETUP

SLIDE SETTING UP

- 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)

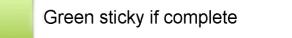
SLIDE STARTING JUPYTER DEMO

- Start Jupyter from the command-line

SLIDE JUPYTER LANDING PAGE DEMO

- Landing page is a file browser, like Explorer/Finder
- MAKE SURE EVERYONE IS IN THE CORRECT LOCATION

Red sticky for a question or issue



SLIDE FUNCTIONS

SLIDE MOTIVATION

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

......

SLIDE WHAT IS A FUNCTION?

- Functions in code work like mathematical functions, like y=f(x)
- f() is the function
- x is an input (or inputs)
- y is the returned value, or output(s)
- The function's output y depends in some way on the value of x defined by f().
- Not all functions in code take an input, or produce a usable output, but the principle is generally the same.
- You've already been using functions in this course: print(), numpy.max(), etc.

.....

SLIDE MY FIRST FUNCTION

- TALK ABOUT THE FUNCTION AND ITS PARTS BEFORE CREATING IT
- We'll write a function to convert Fahrenheit to Kelvin, called fahr_to_kelvin()
- Describe the mathematical function:
 - $\circ~$ This function takes \fbox{x} , subtracts 32, multiplies by 5/9, and adds 273.15
- In Python this translates to the code below:
 - The function performs a calculation, which is *returned* by the **return** statement.
 - The value of the variable temp is taken through the same calculation as in the mathematical function, and is then *return*ed.
 - Functions are *defined* by the def keyword
 - The name of the function follows the def keyword (equivalent to f in the mathematical example)
 - The first line ends in a colon, just like a for loop or if statement.
 - The code, or *body* of the function is indented, just like a for loop or if statement.
 - The *parameters* or *inputs* to the function are then defined in parentheses. These get a variable name **which only exists within the function**. Here, there is one parameter, called temp.

......

SLIDE CREATE A NEW NOTEBOOK DEMO

• PUT THE NOTEBOOK ON SCREEN

- · We'll create a new notebook to play with some functions
- Call the notebook functions
- Add a header

1	# Functions
2	
3	Functions are pieces of code that take an input and return an output. They enable us

SLIDE CREATE THE FUNCTION

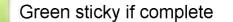
• WRITE THE FUNCTION IN THE NOTEBOOK

SLIDE CALLING THE FUNCTION

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

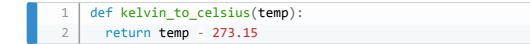
```
print('freezing point of water:', fahr_to_kelvin(32))
print('boiling point of water:', fahr_to_kelvin(212))
```

Red sticky for a question or issue



SLIDE CREATE A NEW FUNCTION

- ASK THE LEARNERS HOW WE WOULD CREATE A NEW FUNCTION TO CONVERT KELVIN TO CELSIUS
- Walk through the process, being prompted

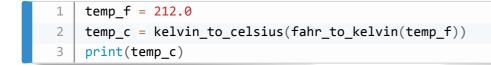


ASK THE LEARNERS HOW TO CALL THE FUNCTION

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

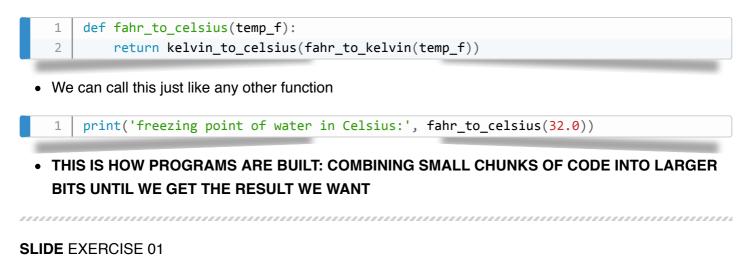
SLIDE COMPOSING FUNCTIONS

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

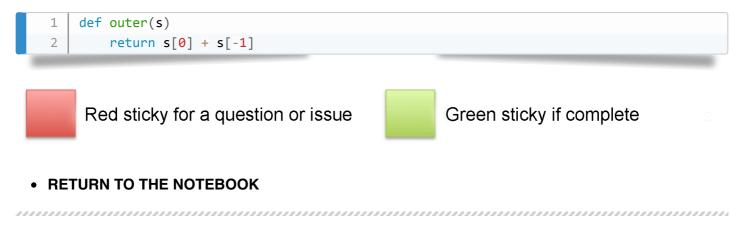


SLIDE NEW FUNCTIONS FROM OLD

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



SHOW THE SLIDES FOR THE EXERCISE



SLIDE SCOPE

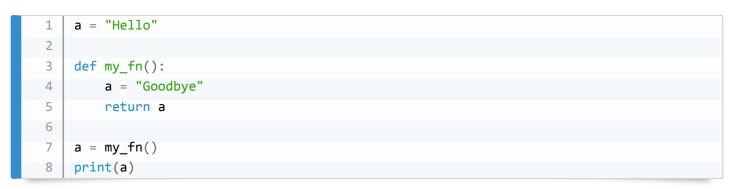
• Make a Markdown note

1	## Scope
2	
3	Variables defined within a function (including parameters) are not available outside

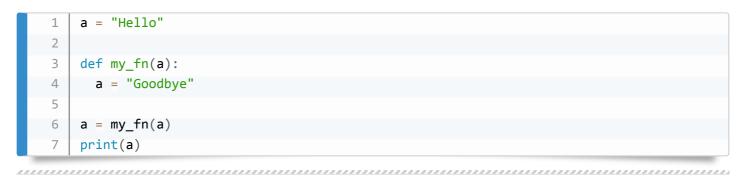
- This is called *function scope*
- DEMO THE CODE BELOW

1	a = "Hello"
2	
3	print(a)

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

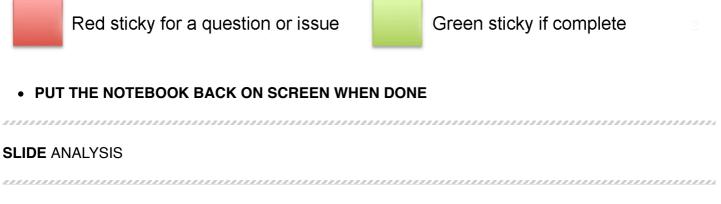


- To move values to and from functions, you should generally return them from the function
- COMPLETE THE CODE EXAMPLE IN THE CELL



SLIDE EXERCISE 02

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



SLIDE TIDYING UP

- Now we can write functions, let's make the inflammation analysis easier to reuse
- ONE FUNCTION PER OPERATION
- **OPEN UP THE FILES.IPYNB NOTEBOOK FROM YESTERDAY
- RESTART AND RUN ALL CELLS
- GUIDE THE STUDENTS THROUGH THE CODE LOGIC: TWO SECTIONS ANALYSE AND DETECT PROBLEMS

SLIDE ANALYSE()

- We'll write a function that plots the data
- WRITE THE FUNCTION BELOW IN THE SAME CELL, WITH COPY AND PASTE
- SPLIT CELLS SO THAT THE FUNCTION AND LOOP ARE SEPARATE

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

RUN THE CELL AND SHOW THAT THE OUTPUT IS THE SAME

SLIDE DETECT_PROBLEMS()

- We'll have a function that checks the data for problems
- * 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>

RUN THE CELL AND SHOW THAT THE OUTPUT IS THE SAME

SLIDE CODE REUSE

- The logic of the code is now easier to understand
- We identify the input files, then apply one function per action in a loop:

.....

- Print the filename
- Load the data with np.loadtxt()
- detect_problems() in the data
- analyse() the data

```
1 for file in files:
2 print(file)
3 data = numpy.loadtxt(fname=file, delimiter=',')
4 detect_problems(data)
5 analyse(data)
```

• THIS HAS ADVANTAGES

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

Red sticky for a question or issue

Green sticky if complete

SLIDE GOOD CODE PAYS OFF

- PUT SLIDES ON SCREEN
- YOU MAY BE ASKING YOURSELF WHY YOU WANT TO BOTHER WITH THIS
- · After 6 months, the referee report arrives and you need to rerun experiments
- Another student is continuing the project
- Some random person reads your article and asks for the code

- Helps spot errors quickly
- · Clarifies structure in your mind as well as in the code
- Saves you time in the long run! ("Future You" will back this up)

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

SLIDE CREATE A NEW NOTEBOOK

- New notebook called testing
- ADD AN INTRO IN MARKDOWN

1	<pre># Testing and Documentation</pre>
2	
3	When writing a function, we should
4	
5	* test output for correctness
6	* document the expected function

ADD IMPORTS

1

import numpy

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

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

SLIDE TEST DATASETS

• ASK THE LEARNERS HOW WE CAN CHECK THAT THE FUNCTION WORKS IN THE WAY WE INTEND

- We could try centre() on our real data, but we don't know what the answer should be!*
- We'll use numpy 's zeros() function to generate an input set where we know the answer

```
• SHOW THE TEST DATA
```

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

• Let's recentre the data at the value 2

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

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

SLIDE REAL DATA

• LOAD THE DATA

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

• Let's recentre the data to zero

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

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

.....

SLIDE CHECK PROPERTIES

- ASK LEARNERS HOW THEY COULD VERIFY THE FUNCTION WORKED AS INTENDED
- · We can check properties of the original and centred data
 - mean , min , max , std

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

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

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

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

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

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

SLIDE 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
- Set defaults by assigning a value in the function declaration, as follows:

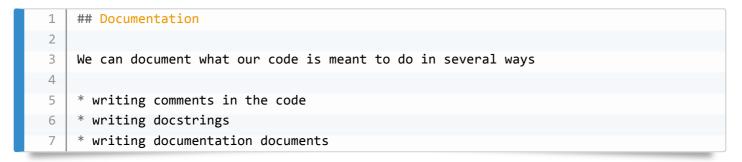
```
1 def centre(data, desired=0.0):
2 """Returns the array in data, recentered around the desired value."""
3 return (data - np.mean(data)) + desired
```

- 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 DOCUMENTING FUNCTIONS

• ADD TEXT TO THE NOTEBOOK



- We can document what our function does by writing comments in the code, and this is a good thing.
- But Python allows us to document what a function does directly in the function using a docstring.
- This is a string that is put in a **specific place in the function definition, and it has special properties that are useful**.
- To add a docstring to our centre() function, we add a string immediately after the function declaration
- ADD DOCSTRING TO EXISTING FUNCTION AND RUN CELL

```
def centre(data, desired):
  1
           """Returns the array in data, recentered around the desired value."""
  2
           return (data - numpy.mean(data)) + desired
  3
 RESTART KERNEL AND RUN ALL

    This documents the function directly in the source code, and it also hooks that documentation into

   Python 's help system.
• We can ask for help on any function using the help() function:

    built-in function

      help(print)
  1
 module function
      help(numpy.mean)
  1

    and if you write it your own functions

      help(centre)
  1

    SHOW LEARNERS HOW DETAILED THE BUILTIN AND NUMPY HELP IS

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

    ADD EXTRA DOCUMENTATION

      def centre(data, desired):
  1
  2
           """Returns the array in data, recentred around the desired value.
  3
  4
           Example
           _ _ _ _ _ _ _ _
  5
           >>> centre([1, 2, 3], 0)
  6
  7
           [-1, 0, 1]
```

• DEMONSTRATE THE CHANGE

.....

return (data - numpy.mean(data)) + desired

SLIDE EXERCISE 03

8 9

• MOVE SLIDES TO THE SCREEN

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

Red sticky for a question or issue

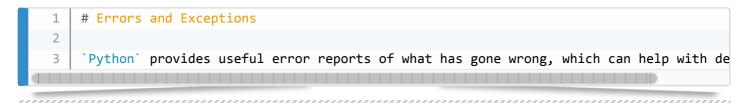


SLIDE ERRORS AND EXCEPTIONS

• MOVE NOTEBOOK TO THE SCREEN

SLIDE CREATE A NEW NOTEBOOK

- Call the notebook errors
- ADD AN INTRO



SLIDE ERRORS

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

......

SLIDE TRACEBACK

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

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

NEW CELL

favourite_ice_cream()

SLIDE PARTS OF A TRACEBACK

```
1
2
    IndexError
                                                 Traceback (most recent call last)
    <ipython-input-4-8f18c934933f> in <module>()
3
    ---> 1 favourite_ice_cream()
4
5
6
    <ipython-input-3-3f8910a0f7ad> in favourite_ice_cream()
7
           3
                                "vanilla",
8
          Δ
                                "strawberry"]
9
      ---> 5
                 print(ice_creams[3])
10
    IndexError: list index out of range
11
```

• TALK THROUGH THE TRACEBACK IN THE NOTEBOOK

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

SLIDE SYNTAX ERRORS

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

1	<pre>def some_function()</pre>
2	<pre>msg = "hello, world!"</pre>
3	<pre>print(msg)</pre>
4	return msg

SLIDE SYNTAX TRACEBACK

```
1 File "<ipython-input-6-bef8c18baffa>", line 1
2 def some_function()
3 ^
4 SyntaxError: invalid syntax
```

• 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 IN PLACE

SLIDE FIXED?

• SHOW AND RUN FIXED CODE

1	<pre>def some_function():</pre>
2	<pre>msg = "hello, world!"</pre>
3	<pre>print(msg)</pre>
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
- ENTER CODE IN A NEW CELL

print(a)

• We have a NAME ERROR

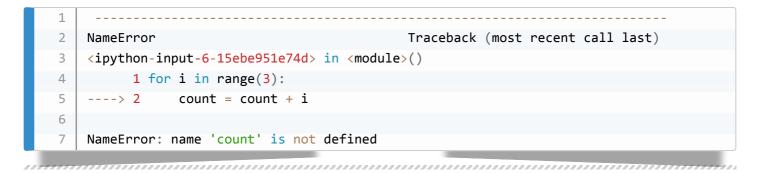
```
1 -----
2 NameError Traceback (most recent call last)
3 <ipython-input-5-c5a4f3535135> in <module>()
4 ----> 1 print(a)
5
6 NameError: name 'a' is not defined
```

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

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

1

• This still gives us a name error



SLIDE INDEX ERRORS

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

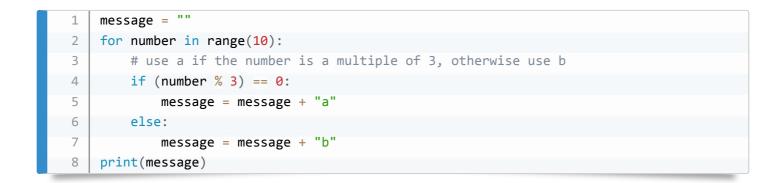
```
1 letters = ['a', 'b', 'c']
2 for letter in range(4):
3 print("Letter", letter, "is", letters[letter])
```

• This gives us an IndexError

```
1
    Letter #1 is a
2
    Letter #2 is b
    Letter #3 is c
3
    4
5
    IndexError
                                            Traceback (most recent call last)
    <ipython-input-7-656a22fa6ec5> in <module>()
6
          3 print("Letter #2 is", letters[1])
7
     4 print("Letter #3 is", letters[2])
8
    ----> 5 print("Letter #4 is", letters[3])
9
10
11
    IndexError: list index out of range
```

SLIDE EXERCISE 04

• PUT SLIDES ON SCREEN



Red sticky for a question or issue

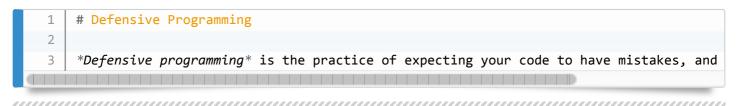
Green sticky if complete

SLIDE DEFENSIVE PROGRAMMING

• PUT NOTEBOOK BACK ON SCREEN

SLIDE CREATE A NEW NOTEBOOK

- Call it defensive
- ADD INTRO IN MARKDOWN

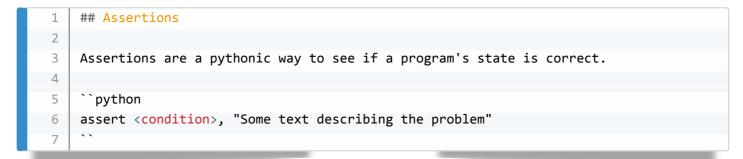


SLIDE DEFENSIVE PROGRAMMING

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

SLIDE ASSERTIONS

• ADD INTRODUCTORY TEXT



- Assertions are a Pythonic way to see if code runs correctly
 - 10-20% of the Firefox source code is assertions/checks on the rest of the code!
- We assert that a *condition* 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.

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)
```

EXECUTE CELL

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

• The traceback tells us there is an AssertionErroe and highlights which assertion failed.

- SLIDE WHEN TO USE ASSERTIONS
- Assertions are useful in three circumstances:
- preconditions must be true at the start of an operation
- · postcondition something guaranteed to be true when an operation completes
- invariant something always true at a particular point in code
- PUT EXAMPLE CODE IN NEW CELL

```
1
    def normalise rectangle(rect):
2
         """Normalises a rectangle to the origin, longest axis 1.0 units."""
 3
        x0, y0, x1, y1 = rect
4
        dx = x1 - x0
5
        dy = y1 - y0
6
7
8
        if dx > dy:
9
             scaled = float(dy) / dx
10
             upper_x, upper_y = 1.0, scaled
11
         else:
             scaled = float(dx) / dy
12
             upper_x, upper_y = scaled, 1.0
13
14
15
         return (0, 0, upper_x, upper_y)
```

• Test with some values - in the same cell

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

• DO ALL INPUTS MAKE SENSE?

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

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

......

SLIDE PRECONDITIONS

1

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

• MAKE CHANGE IN CELL

```
def normalise_rectangle(rect):
1
2
         """Normalises a rectangle to the origin, longest axis 1.0 units."""
         assert len(rect) == 4, "Rectangle must have four co-ordinates"
3
        x0, y0, x1, y1 = rect
4
5
6
        dx = x1 - x0
7
        dy = y1 - y0
8
9
         if dx > dy:
10
             scaled = float(dy) / dx
11
             upper_x, upper_y = 1.0, scaled
12
        else:
13
             scaled = float(dx) / dy
             upper_x, upper_y = scaled, 1.0
14
15
16
         return (0, 0, upper_x, upper_y)
```

• TEST FAILING INPUT AND SHOW ASSERTIONERROR

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

• SHOW ANOTHER PROBLEM

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

SLIDE POSTCONDITIONS

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

```
def normalise_rectangle(rect):
 1
2
         """Normalises a rectangle to the origin, longest axis 1.0 units."""
         assert len(rect) == 4, "Rectangle must have four co-ordinates"
 3
         x0, y0, x1, y1 = rect
4
 5
6
         dx = x1 - x0
7
         dy = y1 - y0
8
9
         if dx > dy:
10
             scaled = float(dy) / dx
             upper_x, upper_y = 1.0, scaled
11
12
         else:
             scaled = float(dx) / dy
13
14
             upper_x, upper_y = scaled, 1.0
15
         assert 0 < upper_x <= 1.0, "Calculated upper x-coordinate invalid"</pre>
16
17
         assert 0 < upper_y <= 1.0, "Calculated upper y-coordinate invalid"</pre>
18
19
         return (0, 0, upper_x, upper_y)
```

• TEST FAILING INPUT TO SHOW ASSERTIONERROR

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

• This isn't our code's fault!

- The problem is that the input values have the upper-right corner below the lower left corner
- We need to add another precondition

```
def normalise_rectangle(rect):
1
2
         """Normalises a rectangle to the origin, longest axis 1.0 units."""
3
         assert len(rect) == 4, "Rectangle must have four co-ordinates"
4
         x0, y0, x1, y1 = rect
5
         assert x0 < x1, "Invalid x-coordinates"</pre>
         assert y0 < y1, "Invalid y-coordinates"</pre>
6
7
8
         dx = x1 - x0
9
         dy = y1 - y0
10
         if dx > dy:
11
             scaled = float(dy) / dx
12
13
             upper_x, upper_y = 1.0, scaled
14
         else:
15
             scaled = float(dx) / dy
             upper_x, upper_y = scaled, 1.0
16
17
         assert 0 < upper x <= 1.0, "Calculated upper x-coordinate invalid"
18
         assert 0 < upper_y <= 1.0, "Calculated upper y-coordinate invalid"</pre>
19
20
21
         return (0, 0, upper_x, upper_y)
```

• DEMONSTRATE THE ERROR THAT'S RAISED

SLIDE NOTES ON ASSERTIONS

.......................

• PUT SLIDES ON SCREEN

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

SLIDE TEST-DRIVEN DEVELOPMENT

SLIDE A PROBLEM

- We want to write a function that identifies when two or more ranges (eg. time-series overlap).
- The range of each input is given as a pair of numbers: (start, end)
- · We want the largest range that all the inputs include

• ASK LEARNERS HOW THEY WOULD GO ABOUT THE PROCESS

SLIDE A NOVICE'S APPROACH

- 1. Write a function: range_overlap()
- 2. Call the function interactively on two or three test inputs
- 3. If the answer is wrong, fix the function
- This works thousands of scientists are doing it right now!

.....

SLIDE A PROGRAMMER'S APPROACH

- 1. Write a short function for each test
- 2. Write a range_overlap() function that should pass those tests
- 3. If any answers are wrong, fix it and re-run the test functions
- WHy DO IT THIS WAY?
- We have to say what the function does in detail before we write it clarity of thought, aids design
- Avoids confirmation bias we have to think about what could go wrong before we write the function, not write a function and confirm that it works on sample data

SLIDE TEST FUNCTIONS

- PUT THE NOTEBOOK ON SCREEN
- Add an intro

```
1 ## Test-Driven Development
2
3 In test-driven development, we write tests that assert what functions should do befor
```

- Here are three test functions for a hypothetical range_overlap() function
- 1. single range returns itself
- 2. simple overlap of two ranges
- 3. simple overlap of three ranges

```
1 assert range_overlap([(0.0, 1.0)]) == (0.0, 1.0)
2 assert range_overlap([(2.0, 3.0), (2.0, 4.0)]) == (2.0, 3.0)
3 assert range_overlap([(0.0, 1.0), (0.0, 2.0), (-1.0, 1.0)]) == (0.0, 1.0)
```

- ENTER FUNCTIONS IN A CELL AND RUN
- NOTE THAT IN THE ABSENCE OF A FUNCTION, IT FAILS

- NOTE THAT WE HAVE IMPLICITLY DEFINED WHAT OUR INPUT AND OUTPUT LOOK LIKE
- NOTE THAT WE'RE MISSING A CASE WITH NO OVERLAP
- How should we define a result where there is no overlap? DISCUSS WITH LEARNERS Return
 (0, 0); return None?
- Are our ranges (x, y) or [x, y]? do they meet when we have [(0, 1), (1, 2)]

• ASSUME

- Return None when there's no overlap
- · Overlaps must have non-zero width
- ADD TWO MORE TESTS

1 assert range_overlap([(0.0, 1.0), (5.0, 6.0)]) == None
2 assert range_overlap([(0.0, 1.0), (1.0, 2.0)]) == None

SLIDE MAKE A TEST FUNCTION

- Wrap the assertions in a function
- DO THIS IN THE SAME CELL

1	<pre>def test_range_overlap():</pre>
2	<pre>assert range_overlap([(0.0, 1.0)]) == (0.0, 1.0)</pre>
3	<pre>assert range_overlap([(2.0, 3.0), (2.0, 4.0)]) == (2.0, 3.0)</pre>
4	assert range_overlap([(0.0, 1.0), (0.0, 2.0), (-1.0, 1.0)]) == (0.0, 1.0)
5	<pre>assert range_overlap([(0.0, 1.0), (5.0, 6.0)]) == None</pre>
6	<pre>assert range_overlap([(0.0, 1.0), (1.0, 2.0)]) == None</pre>

SLIDE WRITE RANGE_OVERLAP()

• WRITE THE FUNCTION IN THE SAME CELL

```
def range_overlap(ranges):
1
        """Return common overlap among a set of (low, high) ranges."""
2
        lowest = 0.0
3
4
       highest = 1.0
5
        for (low, high) in ranges:
            lowest = max(lowest, low)
6
7
            highest = min(highest, high)
        return (lowest, highest)
8
```

• RUN THE CELL

1

• TEST IN THE CELL BELOW

• This fails:

```
1
2
    AssertionError
                                               Traceback (most recent call last)
3
    <ipython-input-25-cf9215c96457> in <module>()
    ----> 1 test_range_overlap()
4
5
6
    <ipython-input-24-2c4b718b7bc2> in test_range_overlap()
7
         10 def test_range_overlap():
         11 assert range_overlap([(0.0, 1.0)]) == (0.0, 1.0)
8
9
                assert range_overlap([(2.0, 3.0), (2.0, 4.0)]) == (2.0, 3.0)
    ---> 12
               assert range_overlap([(0.0, 1.0), (0.0, 2.0), (-1.0, 1.0)]) == (0.0, 1.0)
10
         13
         14
                assert range_overlap([(0.0, 1.0), (5.0, 6.0)]) == None
11
12
13
    AssertionError:
```

- SECOND TEST FAILS
- We're initialising lowest and highest to arbitrary values we should really do this from the data
- always initialise from data a very sound rule!

SLIDE EXERCISE 05

- PUT SLIDES ON SCREEN
- add a test

assert range_overlap([]) == None

• Solution:

```
def range_overlap(ranges):
1
2
         """Return common overlap among a set of (low, high) ranges."""
        if not ranges:
3
            return None
4
5
         lowest, highest = ranges[0]
        for (low, high) in ranges[1:]:
6
7
             lowest = max(lowest, low)
             highest = min(highest, high)
8
9
         if lowest >= highest: # no overlap
10
            return None
11
         else:
12
           return (lowest, highest)
```