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   Green sticky if complete

---

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**:
  - This function takes `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**

```
# Functions

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

Green sticky if complete

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**

```
def kelvin_to_celsius(temp):
  return temp - 273.15
```

- **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**:

```
temp_f = 212.0
temp_c = kelvin_to_celsius(fahr_to_kelvin(temp_f))
print(temp_c)
```

SLIDE NEW FUNCTIONS FROM OLD

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

```
def fahr_to_celsius(temp_f):
    return kelvin_to_celsius(fahr_to_kelvin(temp_f))
```

- We can call this just like any other function

```
print('freezing point of water in Celsius:', fahr_to_celsius(32.0))
```

- **THIS IS HOW PROGRAMS ARE BUILT: COMBINING SMALL CHUNKS OF CODE INTO LARGER BITS UNTIL WE GET THE RESULT WE WANT**

SLIDE EXERCISE 01

- **SHOW THE SLIDES FOR THE EXERCISE**

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

Red sticky for a question or issue

Green sticky if complete

- **RETURN TO THE NOTEBOOK**

SLIDE SCOPE

- **Make a Markdown note**

```
## Scope

Variables defined within a function (including parameters) are not available outside
```

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

```python
a = "Hello"

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**

```python
a = "Hello"

def my_fn():
    a = "Goodbye"
    return a

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

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

```python
a = "Hello"

def my_fn(a):
  a = "Goodbye"

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

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)

Red sticky for a question or issue

Green sticky if complete

- **PUT THE NOTEBOOK BACK ON SCREEN WHEN DONE**

SLIDE ANALYSIS

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):
    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()
```

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**

```python
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!')
```

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

```python
for file in files:
    print(file)
    data = numpy.loadtxt(fname=file, delimiter=',')
    detect_problems(data)
    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**

```
# Testing and Documentation

When writing a function, we should

* test output for correctness
* document the expected function
```

- **ADD IMPORTS**

```
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.

```
def centre(data, desired):
    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**

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

- **Let's recentre the data at the value 2**

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

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

SLIDE REAL DATA

- **LOAD THE DATA**

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

- **Let's recentre the data to zero**

```
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.**

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

```
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**

```
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:

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

- **ADD TEXT TO THE NOTEBOOK**

```
## Documentation

We can document what our code is meant to do in several ways

* writing comments in the code
* writing docstrings
* writing documentation documents
```

- 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**

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

- **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

```python
help(print)
```

- module function

```python
help(numpy.mean)
```

- and **if you write it** your own functions

```python
help(centre)
```

- **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**

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

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

- **DEMONSTRATE THE CHANGE**

SLIDE EXERCISE 03

- **MOVE SLIDES TO THE SCREEN**

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

Red sticky for a question or issue

Green sticky if complete

SLIDE ERRORS AND EXCEPTIONS

- **MOVE NOTEBOOK TO THE SCREEN**

SLIDE CREATE A NEW NOTEBOOK

- **Call the notebook** `errors`
- **ADD AN INTRO**

```
# Errors and Exceptions

`Python` provides useful error reports of what has gone wrong, which can help with de
```

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**

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

- **NEW CELL**

```
favourite_ice_cream()
```

SLIDE PARTS OF A TRACEBACK

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-4-8f18c934933f> in <module>()
----> 1 favourite_ice_cream()

<ipython-input-3-3f8910a0f7ad> in favourite_ice_cream()
      3                     "vanilla",
      4                     "strawberry"]
----> 5     print(ice_creams[3])

IndexError: list index out of range
```

- **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!**

```
def some_function()
    msg = "hello, world!"
    print(msg)
     return msg
```

SLIDE SYNTAX TRACEBACK

```
  File "<ipython-input-6-bef8c18baffa>", line 1
    def some_function()
                       ^
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**

```
def some_function():
    msg = "hello, world!"
    print(msg)
     return msg
```

SLIDE NOT QUITE

```
  File "<ipython-input-7-b32ba7f38b6b>", line 4
    return msg
    ^
IndentationError: unexpected indent
```

- `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**

```
---------------------------------------------------------------------------
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**
- **ENTER CODE IN A NEW CELL**

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

- This still gives us a name error

```
 ---------------------------------------------------------------------------
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`
- **ENTER CODE IN NEW CELL**

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

- **This gives us an `IndexError`**

```
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

- **PUT SLIDES ON SCREEN**

```python
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)
```

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**

```
# Defensive Programming

*Defensive programming* is the practice of expecting your code to have mistakes, and
```

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

Assertions are a pythonic way to see if a program's state is correct.

``python
assert <condition>, "Some text describing the problem"
``
````

- **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**

```python
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)
```

- **EXECUTE CELL**

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

```python
def normalise_rectangle(rect):
    """Normalises a rectangle to the origin, longest axis 1.0 units."""
    x0, y0, x1, y1 = rect

    dx = x1 - x0
    dy = y1 - y0

    if dx > dy:
        scaled = float(dy) / dx
        upper_x, upper_y = 1.0, scaled
    else:
        scaled = float(dx) / dy
        upper_x, upper_y = scaled, 1.0

    return (0, 0, upper_x, upper_y)
```

- **Test with some values - in the same cell**

```python
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?**

```python
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

- **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**

```python
def normalise_rectangle(rect):
    """Normalises a rectangle to the origin, longest axis 1.0 units."""
    assert len(rect) == 4, "Rectangle must have four co-ordinates"
    x0, y0, x1, y1 = rect

    dx = x1 - x0
    dy = y1 - y0

    if dx > dy:
        scaled = float(dy) / dx
        upper_x, upper_y = 1.0, scaled
    else:
        scaled = float(dx) / dy
        upper_x, upper_y = scaled, 1.0

    return (0, 0, upper_x, upper_y)
```

- **TEST FAILING INPUT AND SHOW ASSERTIONERROR**

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

- **SHOW ANOTHER PROBLEM**

```python
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**

```python
def normalise_rectangle(rect):
    """Normalises a rectangle to the origin, longest axis 1.0 units."""
    assert len(rect) == 4, "Rectangle must have four co-ordinates"
    x0, y0, x1, y1 = rect

    dx = x1 - x0
    dy = y1 - y0

    if dx > dy:
        scaled = float(dy) / dx
        upper_x, upper_y = 1.0, scaled
    else:
        scaled = float(dx) / dy
        upper_x, upper_y = scaled, 1.0

    assert 0 < upper_x <= 1.0, "Calculated upper x-coordinate invalid"
    assert 0 < upper_y <= 1.0, "Calculated upper y-coordinate invalid"

    return (0, 0, upper_x, upper_y)
```

- **TEST FAILING INPUT TO SHOW ASSERTIONERROR**

```python
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*

```python
def normalise_rectangle(rect):
    """Normalises a rectangle to the origin, longest axis 1.0 units."""
    assert len(rect) == 4, "Rectangle must have four co-ordinates"
    x0, y0, x1, y1 = rect
    assert x0 < x1, "Invalid x-coordinates"
    assert y0 < y1, "Invalid y-coordinates"

    dx = x1 - x0
    dy = y1 - y0

    if dx > dy:
        scaled = float(dy) / dx
        upper_x, upper_y = 1.0, scaled
    else:
        scaled = float(dx) / dy
        upper_x, upper_y = scaled, 1.0

    assert 0 < upper_x <= 1.0, "Calculated upper x-coordinate invalid"
    assert 0 < upper_y <= 1.0, "Calculated upper y-coordinate invalid"

    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**

```
## Test-Driven Development

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

```
assert range_overlap([(0.0, 1.0)]) == (0.0, 1.0)
assert range_overlap([(2.0, 3.0), (2.0, 4.0)]) == (2.0, 3.0)
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**

```
assert range_overlap([(0.0, 1.0), (5.0, 6.0)]) == None
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**

```
def test_range_overlap():
    assert range_overlap([(0.0, 1.0)]) == (0.0, 1.0)
    assert range_overlap([(2.0, 3.0), (2.0, 4.0)]) == (2.0, 3.0)
    assert range_overlap([(0.0, 1.0), (0.0, 2.0), (-1.0, 1.0)]) == (0.0, 1.0)
    assert range_overlap([(0.0, 1.0), (5.0, 6.0)]) == None
    assert range_overlap([(0.0, 1.0), (1.0, 2.0)]) == None
```

SLIDE WRITE `RANGE_OVERLAP()`

- **WRITE THE FUNCTION IN THE SAME CELL**

```
def range_overlap(ranges):
    """Return common overlap among a set of (low, high) ranges."""
    lowest = 0.0
    highest = 1.0
    for (low, high) in ranges:
        lowest = max(lowest, low)
        highest = min(highest, high)
    return (lowest, highest)
```

- **RUN THE CELL**
- **TEST IN THE CELL BELOW**

```
test_range_overlap()
```

- This fails:

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-25-cf9215c96457> in <module>()
----> 1 test_range_overlap()

<ipython-input-24-2c4b718b7bc2> in test_range_overlap()
     10 def test_range_overlap():
     11     assert range_overlap([(0.0, 1.0)]) == (0.0, 1.0)
---> 12     assert range_overlap([(2.0, 3.0), (2.0, 4.0)]) == (2.0, 3.0)
     13     assert range_overlap([(0.0, 1.0), (0.0, 2.0), (-1.0, 1.0)]) == (0.0, 1.0)
     14     assert range_overlap([(0.0, 1.0), (5.0, 6.0)]) == None

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**

```python
assert range_overlap([]) == None
```

- Solution:

```python
def range_overlap(ranges):
    """Return common overlap among a set of (low, high) ranges."""
    if not ranges:
        return None
    lowest, highest = ranges[0]
    for (low, high) in ranges[1:]:
        lowest = max(lowest, low)
        highest = min(highest, high)
    if lowest >= highest:  # no overlap
        return None
    else:
        return (lowest, highest)
```