

---

# **Introduction to Geospatial Python Documentation**

*Release 0*

**Samuel Bowers**

**Sep 19, 2018**



---

## Contents

---

<b>1</b>	<b>Aims</b>	<b>3</b>
<b>2</b>	<b>Instructions</b>	<b>5</b>
<b>3</b>	<b>Contents</b>	<b>7</b>
3.1	Setup instructions . . . . .	7
3.2	Python 101 . . . . .	9
3.3	Introduction to NumPy . . . . .	22
3.4	Introduction to Matplotlib . . . . .	30
3.5	To be continued... . . . . .	41
<b>4</b>	<b>Search</b>	<b>43</b>



Welcome to this very short introduction to using Python for geospatial applications.

We will start with the very basics of Python, and then look at how Python can be used to process, analyse and interpret geospatial data.

Our aim is to introduce you to some of the fundamental concepts of Python. We do not expect you to be able to program fluently in Python by the time you are done (that will take a lot more practice), but we hope to give you an overview of how Python works, and how it might be able to help you in future.

We will aim to build up this documentation over time to give you a record of the methods we've covered. Take note of the URL, these notes may be something that you'll refer back to in future.



# CHAPTER 1

---

## Aims

---

By the end of this tutorial, we hope that you will be able to:

- Write simple Python scripts that use different data types (`strings`, `integers`, `floats`, `boolean`) and how to group them (`lists`), and flow control statements (`for`, `if`, `while`).
- Perform basic data manipulation in Python
- Understand how arrays are used to process image data in NumPy
- Use Matplotlib to make scatter plots and display images



## CHAPTER 2

---

### Instructions

---

- Work through each line of code, and make it run on your own PC
- Do not copy-paste. You'll learn a lot by entering each command manually
- When you encounter an error, in the first instance try and work out the issue yourself. If you're still stuck, do ask for some pointers
- Don't worry about making mistakes, you can't break Python
- Using Google to help with exercises is encouraged
- Go at your own pace; we don't expect you to reach the end of this tutorial today



## 3.1 Setup instructions

### 3.1.1 Installing Python

Before we do anything, we'll need to set up Python to work on your PC. The instructions that follow refer to installation using a Linux PC.

We'll be working with Python 2.7, which is currently the most widely used version of Python. There is a newer version of Python (3.x), which has some small improvements to its syntax, but is not yet widely supported by the scientific packages (or 'modules') we'll be using.

**Anaconda** is one of a number of Python 'distributions', which include the Python interpreter and a range of Python packages. We will be using Anaconda as it is one of the most straightforward to install, and it comes with most of the packages you'll be likely to use.

There are many ways to run Python, including in an Integrated Development Environment (IDE) (e.g. **'Spyder'**, **'PyCharm'**). These are pieces of software that include features for advanced editing, interactive testing, and debugging. We won't be using an IDE as part of this tutorial, but be aware that there exist many IDEs, and you may find in time that you find an IDE that suits you. Or, if you're like me, you may find that you prefer to write scripts in a simple text editor and prefer not to use an IDE at all.

To download and install Anaconda, `cd` to a location you want to save Anaconda, and execute the following:

```
wget https://repo.anaconda.com/archive/Anaconda2-5.1.0-Linux-x86_64.sh
chmod +x Anaconda2-5.1.0-Linux-x86_64.sh
./Anaconda2-5.1.0-Linux-x86_64.sh
```

Accept all the default options in the install.

One important Python module that Anaconda does not come bundled with is `GDAL`. We use `GDAL` to process georeferenced imagery (i.e. satellite data) using Python. To install `gdal`, execute the command:

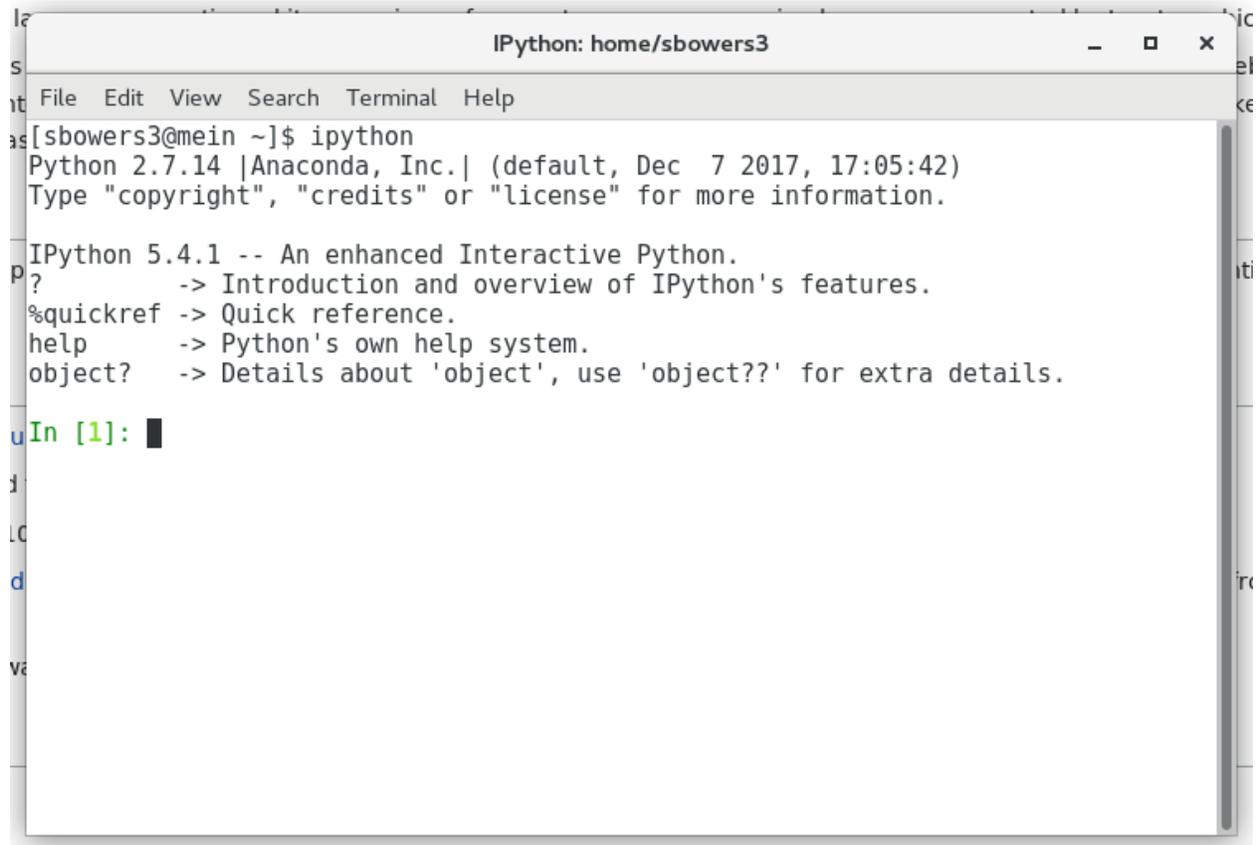
```
conda install -c anaconda gdal
```

### 3.1.2 Using Python

There are two ways to execute Python code:

- By writing code directly into the interpreter window using Python’s ‘interactive mode’ (great for learning)
- By writing scripts in a text editor and then running in an interpreter window (better for data processing)

To open Python’s interactive mode, open a terminal window and type `ipython`. You could also run `python`, but we recommend using IPython (Interactive Python) for it’s user friendliness. You should see something that looks like:



To run a script, you can type `run script_name.py` into the IPython window. Alternatively, from the Linux command line you can execute:

```
python script_name.py
```

We’ll be using Python both interactively and to run scripts in this tutorial.

#### Using Python in interactive mode

Where we want you to enter code directly into the interpreter window in interactive mode, code snippets will be preceded by a `>>>`.

Try typing `import this` into the ‘interpreter window’ followed by the `enter` key. You should see the following:

```
>>> import this
The Zen of Python, by Tim Peters
```

(continues on next page)

(continued from previous page)

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

## Using Python to run scripts

Where we want you to write a script in a text editor and run it, the `>>>` symbol will be absent. There are a number of text editors included with most Linux distributions (e.g. `vim`, `gedit`, `kate`). For this tutorial we'll use `gedit`.

Open a new terminal, `cd` to your working directory and run `gedit`. Type the following into the 'script window', save it as `helloworld.py`:

```
print 'Hello World!'
```

Then, in the interpreter window type `run helloworld.py`. You should see the following:

```
>>> run helloworld.py
Hello World!
```

Don't move on any further until you've been able to run the `import this` command in the interpreter window and managed to run the 'Hello World' script. Ask if you need assistance.

## 3.2 Python 101

Welcome to Python!

Python is a simple programming language with straightforward syntax. It's open source, freely available, and very widely-used. It's also relatively easy to learn, so it's a great place to start to learn programming.

To begin, we'll look at the main types of data used in Python.

### 3.2.1 Strings

Strings are sequences of characters such as letters and numbers. An example of the `string` data type is text.

Try typing the commands following the `>>>` symbols into the interpreter window. Be careful to type it exactly, or you might get an error.

```
>>> print 'Hello world!'
Hello world!

>>> print "I can get Python to print anything I type"
I can get Python to print anything I type

>>> print 'Even numbers like 6 and symbols like #'
Even numbers like 6 and symbols like #
```

With these simple commands, we've instructed Python to print the text we've given it to the screen. By putting those characters inside quotation marks ('', or ""), we've told Python that this data is a the `string` type.

We can also run the `print` command as a script using a text editor. Here Python will execute commands in order from top to bottom. In your text editor, type:

```
print 'Hello world!'
print 'Goodbye world!'
```

... and save the file as `my_first_script.py`. To execute this script, in the interpreter window type `run my_first_script.py`.

```
>>> run my_first_script.py
'Hello world!'
'Goodbye world!'
```

If you manage to replicate this output, congratulations! You've just written your first program in Python.

But what if we want to include a ' or " symbol in our `string`? If we aren't careful, Python will think that we want to end the `string`. The first option is to enclose single quotation marks within double quotation marks (or double quotation marks within single quotation marks):

```
>>> print "I can even get Python to print a ' symbol."
I can even get Python to print a ' symbol.

>>> print '...and a " symbol'
...and a " symbol.
```

Alternatively, we can use the escape character (`\`) as follows:

```
>>> print 'I can also get Python to print a \' symbol using the escape character.'
I can get Python to print a ' symbol using the escape character

>>> print 'And even a \\ symbol.'
And even a \ symbol.
```

### Exercise: Write a story

Now it's your turn to write a Python script.

- Write a Python script in the script window that tells a short story. Save it as `story.py`, and run it in the interpreter window.

Here's an example that I wrote in a text editor:

```
print 'A little girl goes into a pet shop and asks for a wabbit. The shop keeper
↳looks down at her, smiles and says:'
print '"Would you like a lovely fluffy little white rabbit, or a cutesy wootesly
↳little brown rabbit?'"
print '"Actually", says the little girl, "I don\'t think my python would notice."'
```

... and ran using the interpreter:

```
>>> run story.py
A little girl goes into a pet shop and asks for a wabbit. The shop keeper looks down
↳at her, smiles and says:
"Would you like a lovely fluffy little white rabbit, or a cutesy wootesly little
↳brown rabbit?"
"Actually", says the little girl, "I don't think my python would notice."
```

### An aside: comments

Comments are important aspects of programs. Comments are used to describe what your program does in readable language, or to temporarily remove lines from a program.

```
# This is a comment. When executing this program, Python will ignore this line.
# Anything that comes after the # character is ignored.
print 'This text will be printed' # But not this text
# Comments are often used to disable lines in a program
print 'I want this line to be executed'
# print 'I do not want this line to be executed right now'
```

```
>>> run code.py
This text will be printed
I want this line to be executed
```

I will now use comments to describe the code I give you. Adding comments to your code is good practice as it makes your code easier for other people to use, and helps you to remember how your own code works. Get used to adding comments to the scripts you write.

## 3.2.2 Numerical data

### Integers

An integer is a whole number. Examples of integers are 1, 3, and 10.

We can use Python like a calculator to perform arithmetic on integers. Try the following:

```
>>> 1 + 2 # Addition
3
>>> 1 - 2 # Subtraction
-1
>>> 3 * 4 # Multiplication
12
>>> 5/2 # Division
2
```

The last of these examples is interesting. Why do you think the answer to 5 divided by 2 isn't 2.5?

### Floating point numbers

Unlike integers, floating point numbers (`floats`) have a decimal point. A float can be specified by including a decimal point (e.g. `1.0` or `1.`).

We can apply all the same operators to floats as integers, but they will sometimes behave differently. Try these examples:

```
>>> 3.5 + 1.2
4.7

>>> 8.2/2.3
3.5652173913043477

>>> float(1) # We can convert from integers and floats
1.0

>>> int(1.0) # ...and from floats to integers
1

>>> 3/4.5 # An integer interacting with a float results in a float
0.6666666666666666

>>> 'Hello ' + 'World!' # We can also add strings together
'Hello World!'

>>> '1' + 1.0 # ... but trying to add a string to a float results in an error. Why_
↪is this?
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-1-be0d736a4237> in <module>()
----> 1 '1' + 1.0

TypeError: cannot concatenate 'str' and 'float' objects
```

### An important detour: Variables

In programming, variables are names that we give for something. They ensure that code is readable, and offer a means of 'saving' a value for later use. We assign a value to a variable using the `=` character, with the variable name on the left and its value on the right. Try these examples of how variables work:

```
>>> x = 1 # Define a variable named x

>>> x
1

>>> y = 1 + 2 # Define a variable named y

>>> y
3

>>> x + y # Operators and logical tests work identically with variables
4
```

(continues on next page)

(continued from previous page)

```
>>> var1, var2, var3 = 1, 'two', 3.0 # Multiple variables can be defined at once
>>> print var1, var2, var3
1 two 3.0
```

Now you know the basics of how variables work, you can start to write simple scripts that perform calculations for you. Write out the following program, and save it as `miles_to_km.py`.

```
# Prompt the user to input a distance in miles and save it to a variable named
↳ distance_miles
distance_miles = input('Input a distance in miles: ')

# Convert distance_miles to a float
distance_miles = float(distance_miles)

# Convert the distance to kilometers, and save it to distance_km
distance_km = distance_miles * 1.61

# Convert distance_km to a string
distance_km = str(distance_km)

# Finally, print the distance in kilometers
print 'That is a distance of ' + distance_km + ' kilometers'
```

... and run it in the interpreter window:

```
>>> run miles_to_km.py
Input a distance in miles: 5
That is a distance of 8.05 kilometers
```

Can you make the code run? Note: There are three functions in this script (`input()`, `float()`, and `str()`), which convert between data types. Make sure you understand what each line does.

## Exercises

### 1. Use the `+`, `-`, `*`, and `/` operators to compute the following:

- 123456 plus 654321
- 123456 minus 654321
- 123456 multiplied by 654321
- 123456 divided by 654321

### 2. Can you work out what the following operators do? HINT: You can access help documentation in python with a command

- `%`
- `<`
- `>`
- `**`

### 3. The basal area of a tree (BA) can be calculated using the equation:

- $BA = \pi(DBH/2)^2$

... where DBH is the tree diameter at breast height and  $\pi = 3.14$ . Write a Python script to calculate the basal area of a tree from its diameter at breast height.

### 3.2.3 Boolean data

Boolean data can take two values; True or False (also denoted as 1 or 0). The boolean data type is most commonly associated with logical expressions. The main logical expressions we use in programming are as follows:

```
>>> # An equivalence statement tests whether two arguments have the same value.

>>> True == True
True

>>> True == False
False

>>> False == False
True

>>> # A negation statement tests is the opposite of an equivalence statement.

>>> True != True
False

>>> True != False
True

>>> False != False
False

>>> # The 'and' statement (conjunction) returns True where both arguments are True.

>>> True and True
True

>>> True and False
False

>>> False and False
False

>>> # The 'or' statement (disjunction) returns True where either of the arguments is
↳ True.

>>> True or True
True

>>> True or False
True

>>> False or False
False
```

Boolean logic may be quite different from anything you've learned before, so make sure you understand these statements before going any further.

Logical statements can involve other data types, and also be combined:

```
>>> 1 + 1 == 2
True

>>> 'one' == 'two'
False

>>> x = True

>>> y = False

>>> x and y
False

>>> (False or True) and (False and True) # Can you work out what's happening here?
False
```

### Exercise: logical expressions

Try and predict the outcome of the following boolean expressions. Input them into Python to see how you did.

1. True and True
2. False and True
3. True or False
4. False or True
5. 'this\_string' == 'this\_string'
6. 'this\_string' == 'that\_string'
7. 'this\_string' != 'that\_string'
8. 1 == 1
9. 1 == 1 and 1 == 0
10. 1 == 1 or 1 == 0
11. 1 == 1 and 1 != 0
12. 10 > 5 and 5 < 0
13. True and 3 == 4
14. 1 + 2 != 3
15. (True and False) == (1 and 0)
16. (True or False) and ('this\_string' == 'this\_' + 'string')

### 3.2.4 Groups of things

Frequently we will want to group multiple values together in a single variable. The main mechanisms for doing this in Python are tuples, lists and dictionaries. As our time is limited, we'll only look at lists (and later arrays), but you should be aware that other ways of grouping data exist.

### Creating Lists

Lists are a group of ordered elements. They are created as follows:

```
>>> my_list = [2, 4, 6, 8] # Use square brackets to define a list, separating
↳elements using commas

>>> my_list
[2, 4, 6, 8]

>>> my_list.append(10) # Add a new element to the end of a list

>>> my_list
[2, 4, 6, 8, 10]

>>> my_list.append('dog') # Lists can contain multiple different types of data

>>> my_list
[2, 4, 6, 8, 10, 'dog']

>>> my_list.append([2.0, 3.0, 'cat']) # They can even contain other lists (a 'nested'
↳list)

>>> my_list
[2, 4, 6, 8, 10, 'dog', [2.0, 3.0, 'cat']]
```

### Exercises

1. Try to create your own list containing the letters 'A' to 'F'
2. Append the letter 'G' to the end of your list
3. What happens when you add (+) two lists together?

### Indexing Lists

We can access individual elements of a list using indexing. We do this using square brackets and specifying the numeric location of the elements we want to access.

---

**Note:** In Python (and most other programming languages), we start counting from 0. So the first element of a list is 0, the second element is 1, the third element is 2 etc.

---

Here are some examples of how we index lists:

```
>>> zoo = ['Tiger', 'Parrot', 'Bear', 'Sloth', 'Giraffe'] # A list of animals in a zoo

>>> zoo[0] # Get the first element of the list
'Tiger'

>>> zoo[3] # Get the fourth element of the list
'Sloth'
```

Using negative numbers, we can access elements from the end of a list:

```
>>> zoo[-1] # Get the last element of the list
'Giraffe'

>>> zoo[-2] # Get the second to last element of the list
'Sloth'
```

## Slicing Lists

We can also select multiple elements of a list ('slicing'), as follows:

```
>>> zoo = ['Tiger', 'Parrot', 'Bear', 'Sloth', 'Giraffe']

>>> zoo[1:4] # Get the second to fourth elements from the list
['Parrot', 'Bear', 'Sloth']

>>> zoo[:2] # Get the first two elements of the list
['Tiger', 'Parrot']

>>> zoo[3:] # Get all elements from the fourth onwards
['Sloth', 'Giraffe']

>>> zoo[::2] # Get every second element
['Tiger', 'Bear', 'Giraffe']

>>> zoo[::-1] # Reverse the list
['Giraffe', 'Sloth', 'Bear', 'Parrot', 'Tiger']

>>> # We can apply these same methods to strings

>>> animal = 'Lion'

>>> animal[0]
'L'

>>> animal[1:]
'ion'
```

## List functions

We can determine various properties of list contents using the following functions:

```
>>> my_list = [2, 4, 6, 8, 10]

>>> min(my_list) # Returns the minimum value of a list
2

>>> max(my_list) # Returns the maximum value of a list
10

>>> len(my_list) # Returns the number of elements (length) of the list
5
```

### Exercises:

1. Write a program that takes a `list` as input, reverses it, and `prints` the output.
2. Write a script that prints the highest and lowest values in a list

### 3.2.5 Flow control

Now you know the basics of data types, we can begin to make some complex programs. To do this we need to learn how to control the flow of a program.

#### The `if` statement

An `if` statement looks at whether a boolean statement is `True`, then runs the code under it. If the boolean statement is `False`, the code is skipped.

Have a look at these two `if` statements:

```
>>> cats = 10 # Declare a variable called cats

>>> if cats > 8: #A boolean statement
...:     print 'We have too many cats!' #New block executed where the boolean_
↪statement == True
...:
We have too many cats!

>>> if cats < 3:
...:     print 'We need more cats!'
...:

>>>
```

In the first, the boolean statement was `True`, so Python executed the indented block of code. In the second, the boolean statement was `False`, so Python did not execute the indented block of code.

We can combine multiple boolean tests in a single `if` statement by adding `elif` (“else if”) and `else` statements. To see how this works, write out this program in the script editor, and save it as `cats.py`:

```
cats = 5

if cats > 8:
    print 'We have too many cats!' # This is a new block. It's indented by 4 spaces.
elif cats < 3:
    print 'We need more cats!'
else:
    print 'We have the right number of cats :)'
```

... and run it in the interpreter window:

```
>>> run cats.py
'We have the right number of cats :)'
```

Try setting `cats` to equal 2 or 10. Do you understand what is happening here?

## Exercise

- Can you write an `if` statement that tests whether a `string` begins with the letter 'm'?

## The `for` loop

The `for .. in` statement is a looping statement which iterates over a sequence (e.g. a list). This allows you to perform an operation on each item in the list in turn.

Try these out for yourself to get a feel for how lists work:

```
>>> rainbow = ['red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet']

>>> for colour in rainbow: # Note the colon
...:     print colour
...:
red
orange
yellow
green
blue
indigo
violet

>>> for i in range(6): # For each integer from 0 to 5...
...:     print i
...:
0
1
2
3
4
5
```

Everything that is indented will be executed for each item in the list. `for` loops are very useful for bulk-processing data. For example, by providing a list of filenames we could process a stack of satellite images one at a time.

## Exercises

1. Look at how we used `range` in the last example. What does `range` do?
2. use a `for` loop to print the integers 1 - 10
3. Use a `for` loop to print 'Hello World!' 10 times
4. Write a script to add the integers 1 - 100, and display the total.
5. Write a `for` loop that iterates from 0 to 20. For each iteration, check `if` the current number is even or odd, and print that to the screen. HINT: Remember the `%` operator?

## The `while` statement

A `while` loop is similar to a `for` loop, but will keep executing code so long as a boolean expression is `True`. Unlike a `for` loop which will stop when it gets to the end of a list, a `while` loop can continue indefinitely.

```
>>> count = 0 # Set a variable named 'count' to 0

>>> while count < 5: # As long as count is below 5...
...:     count = count + 1 # ...add 1 to count...
...:     print count # ...and print the value of count
...:
1`
2
3
4
5
```

### Exercises

1. Predict what the following script will do. Try and run it, if you dare...

```
i = 0

while True:
    i = i + 1
    print i
```

1. Can you repair the script to stop iterating when i reaches 10000?

### Combining conditional statements

Consider the following program, which is a number guessing game:

```
number = 50

your_guess = input('Enter an integer : ') # Input an integer

if your_guess == number:
    # This is a new block. It is indented by 4 spaces.
    print 'Well done, you correctly guessed the number!'

elif your_guess < number:
    # This block will be executed where the guess is lower than the correct number
    print 'Wrong! The correct number is higher than that.'

else:
    # This block will be executed where neither of the two conditions above are_
    ↪reached.
    print 'Wrong! The correct number is lower than that.'

print 'Done' # This final statement is always executed
```

Save it as `if_game.py`, and try and play it:

```
>>> run if_game.py
Enter an integer : 20
Wrong! The correct number is higher than that.
Done

>> run if_game.py
```

(continues on next page)

(continued from previous page)

```

Enter an integer : 60
Wrong! The correct number is lower than that.
Done

>> run if_game.py
Enter an integer : 50
Well done, you correctly guessed the number!
Done

```

It works, but we can include a while statement to improve this game. Be very careful with indentation of the blocks, as the if statement is nested inside the while statement so some blocks are indented by 8 spaces.

```

number = 50
run_code = True

while run_code:

    # This is a new block. It is indented by 4 spaces.
    your_guess = input('Enter an integer : ') # Input an integer

    if your_guess == number:
        # This is another new block. It is indented by 8 spaces.
        print 'Well done, you correctly guessed the number!'
        run_code = False # This will exit the while statement

    elif your_guess < number:
        # This block will be executed where the guess is lower than the correct number
        print 'Wrong! The correct number is higher than that'

    else:
        # This block will be executed where neither of the two conditions above are
        ↪reached.
        print 'Wrong! The correct number is lower than that'

print 'Done' # This final statement is always executed

```

Save it as while\_game.py, and try and play it:

```

>>> run while_game.py
Enter an integer : 20
Wrong! The correct number is higher than that
Enter an integer : 60
Wrong! The correct number is lower than that
Enter an integer : 50
Well done, you correctly guessed the number!
Done

```

## Exercise

1. Can you modify the number guessing game to limit the number of guesses allowed to 10?
2. Search the internet to work out how to generate a random integer between 1 and 100. Can you modify the game to randomly generate a new integer each time it is run?

### 3.2.6 Functions

You've already used several functions in python. Examples are `len`, `max`, and `input`. These in-built functions are useful, but sometimes you want a function that can perform a more specific operation. With Python, you can create your own functions, which you can re-use multiple times in a script.

Here's an example of a very simple function, which we will name `hello`:

```
>>> def hello():
...:     print 'Hello!'
...:
>>> hello()
'Hello!'
>>> hello()
'Hello!'
```

This simple function had no inputs or output variables. Here's a more complex function that converts distance from miles to kilometers. This function will have one input (`distance_miles`) and one output (`distance_km`).

```
>>> def miles_to_km(distance_miles):
...:     distance_km = distance_miles * 1.61 # Convert miles to kilometers
...:     return distance_km # And return the distance in kilometers
>>> distance_km = miles_to_km(5)
>>> distance_km
8.05
```

You will recall we wrote a complete script to do this earlier. Now that it is contained within a function, we can easily repeat the calculation multiple times:

```
>>> miles_to_km(1)
1.61
>>> miles_to_km(200)
322.0
```

We won't look any further into functions as part of this tutorial, but you should be aware that they exist, and that the more complex your programs get the more essential they become to keeping your scripts tidy.

#### Exercise:

1. Write a function to convert temperature from fahrenheit to celcius. The equation to convert from °F to °C is:
  - $[^{\circ}\text{C}] = ([^{\circ}\text{F}] - 32) * 5/9$

## 3.3 Introduction to NumPy

An array is a way of storing several items (e.g. integers) in a single variable. Arrays are a systematic arrangement of objects, usually arranged in rows and columns. NumPy is a package for scientific computing in Python, which adds support for large multi-dimensional arrays. Arrays are similar to lists, but:

- They can be very large and multi-dimensional

- They are memory efficient, and provide fast numerical operations
- They can contain only a single data type in one array

A good example of a form of data that numpy is indispensable for is an image, such as data from an Earth Observation satellite.

The numpy module can be imported into Python with the code:

```
>>> import numpy
```

... however, because numpy is used so frequently, it is usually shortened to np by importing as:

```
>>> import numpy as np
```

### 3.3.1 Creating numpy arrays

The easiest way to create a numpy array is using a list and the `np.array` function:

```
>>> import numpy as np
>>> a = np.array([1,2,3])
>>> a
array([1, 2, 3])
```

Arrays can be multi-dimensional (2D, 3D, 4D...):

```
>>> b = np.array([[4,1,3],[7,2,5],[1,8,7]]) # A 2D array
>>> b
array([[4, 1, 3],
       [7, 2, 5],
       [1, 8, 7]])
```

Arrays have a number of attributes:

```
>>> a.ndim # Number of dimensions
1
>>> a.shape # Shape of array
(3,)
>>> b.ndim
2
>>> b.shape
(3, 3)
```

There are other ways to create numpy arrays:

```
>>> np.arange(10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.arange(0, 50, 5) #Numbers from 0 to 50 in steps of 5
array([ 0,  5, 10, 15, 20, 25, 30, 35, 40, 45])
```

(continues on next page)

(continued from previous page)

```

>>> np.linspace(10, 20, 21) #Numbers from 10 to 20, linearly split into an array of
↳21 elements
array([ 10. ,  10.5,  11. ,  11.5,  12. ,  12.5,  13. ,  13.5,  14. ,
       14.5,  15. ,  15.5,  16. ,  16.5,  17. ,  17.5,  18. ,  18.5,
       19. ,  19.5,  20. ])

>>> np.zeros(10)
array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])

>>> np.ones(10)
array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.])

>>> np.ones((3,3))
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])

```

You may have noticed that in some instances array elements are shown with a trailing dot (e.g. 1. vs 1). This indicates the type of numerical data held in the array (i.e. integer or float). Be aware that a numpy array can only hold one data type at once.

```

>>> a
array([1, 2, 3])

>>> a.dtype
dtype('int64')

>>> c = np.array([1., 2., 3.])

>>> c.dtype
dtype('float64')

```

### Exercise: Array creation

1. Create an array containing the numbers 1 to 100, in steps of 0.5
2. Create an array containing all odd numbers from 1 to 100
3. Make an array containing the integer 10, repeated 20 times

### 3.3.2 Array operations

Numpy arrays can be modified using the standard python operators:

```

>>> a + 1 # Adds 1 to all elements
array([ 2,  3,  4])

>>> a - 1 # Subtracts 1 from all elements
array([ 0,  1,  2])

>>> a * 5 # Multiplies all elements by 5
array([ 5, 10, 15])

>>> a / 10. # Divides all elements by 10. Why do we divide by a float here?
array([ 0.1,  0.2,  0.3])

```

We can also perform operations on two arrays:

```
>>> a * a # What's happening here?
array([1, 4, 9])

>>> b + b # What's happening here?
array([[ 8,  2,  6],
       [14,  4, 10],
       [ 2, 16, 14]])

>>> a * b # What's happening here?
array([[ 4,  2,  9],
       [ 7,  4, 15],
       [ 1, 16, 21]])
```

### 3.3.3 Array indexing

Much like lists, we can access the elements of an array independently. Make sure you understand what is happening in each of these examples:

```
>>> a[0]
1

>>> a[1]
2

>>> a[-1]
3

>>> b[1]
array([7, 2, 5])

>>> b[1,1]
2
```

In a similar manner to lists, arrays can also be sliced into parts:

```
>>> x = np.arange(100)

>>> x[10:20] # Get the elements from 10 to 20
array([10, 11, 12, 13, 14, 15, 16, 17, 18, 19])

>>> x[::10] # Get every 10th element
array([ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90])

>>> y = np.random.rand(6,6) # Create a 6 by 6 array of random numbers

>>> y[:3,:3] # Select the first 3 rows and columns
array([[ 0.1545669 ,  0.35908932,  0.30732055],
       [ 0.21162311,  0.09059191,  0.87824174],
       [ 0.72614825,  0.0266589 ,  0.41946432]])
```

### 3.3.4 Assignment

Using similar methods, we can assign values to elements of an array

```

>>> y[:3,:3] = 0 # Set the first 3 rows and columns to equal 0

>>> y
array([[ 0.          ,  0.          ,  0.          ,  0.984333   ,  0.79586604,  0.99353897],
       [ 0.          ,  0.          ,  0.          ,  0.3648008  ,  0.81364244,  0.83354187],
       [ 0.          ,  0.          ,  0.          ,  0.42934255,  0.52064217,  0.25940776],
       [ 0.95443339,  0.83289332,  0.91049323,  0.71452678,  0.13792483,  0.79273019],
       [ 0.4731708  ,  0.01571735,  0.98596698,  0.95775551,  0.11409062,  0.72255358],
       [ 0.87815504,  0.60418293,  0.17141781,  0.44434767,  0.56713818,  0.
↪53995463]])

>>> y[-1,-1] = 1 # Set the lower right array element to equal 1
array([[ 0.          ,  0.          ,  0.          ,  0.984333   ,  0.79586604,  0.99353897],
       [ 0.          ,  0.          ,  0.          ,  0.3648008  ,  0.81364244,  0.83354187],
       [ 0.          ,  0.          ,  0.          ,  0.42934255,  0.52064217,  0.25940776],
       [ 0.95443339,  0.83289332,  0.91049323,  0.71452678,  0.13792483,  0.79273019],
       [ 0.4731708  ,  0.01571735,  0.98596698,  0.95775551,  0.11409062,  0.72255358],
       [ 0.87815504,  0.60418293,  0.17141781,  0.44434767,  0.56713818,  1.]])

```

### Exercise: Array manipulation

1. Create the following arrays using the methods we have covered:

```

array([3, 2, 1])

array([ 0, 20, 40, 60, 80, 100, 120, 140, 160, 180, 200, 220, 240,
        260, 280, 300, 320, 340, 360, 380, 400, 420, 440, 460, 480, 500,
        520, 540, 560, 580, 600, 620, 640, 660, 680, 700, 720, 740, 760,
        780, 800, 820, 840, 860, 880, 900, 920, 940, 960, 980])

array([[ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  4.,  1.],
       [ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.],
       [ 2.,  2.,  2.,  2.,  2.]])

```

1. Can you make a three dimensional array of random numbers?
2. Create the following array. Use the indexing methods to extract the four outlined areas:

5	2	5	7	9
1	2	3	2	1
3	6	8	10	0
8	2	9	3	7
4	0	3	1	7

### 3.3.5 Functions

Numpy includes a large number of [useful functions](#).

Here are some examples of commonly used functions.

```

>>> a = np.arange(1,11)

>>> np.sum(a) # The sum total of an array
55

>>> np.min(a) # Minumum value of an array
1

>>> np.max(a) # Maxiumum value of an array
10

>>> np.mean(a) # Mean value of an array
5.5

>>> b = a.reshape(5,2) # Change the dimensions of a shape

>>> b
array([[ 1,  2],
       [ 3,  4],
       [ 5,  6],
       [ 7,  8],
       [ 9, 10]])

```

(continues on next page)

(continued from previous page)

```
>>> np.sum(b, axis=0) # Sum along the vertical axis of the array
array([25, 30])

>>> np.sum(b, axis=1) # Sum along the horizontal axis of the array
array([ 3,  7, 11, 15, 19])
```

**Exercise: More array manipulation**

1. Create the following array (in one line of code!)

$$a = \begin{pmatrix} 1 & 2 & \dots & 10 \\ 11 & 12 & \dots & 20 \\ \vdots & \ddots & \ddots & \vdots \\ 91 & 92 & \dots & 100 \end{pmatrix}$$

1. Using this array:
  - Calculate the sum total of the array
  - Calculate the mean of the array
  - Calculate the mean of each row of the array
  - Extract the 3rd row of the array (numbers 21 to 30)

**3.3.6 Masking**

For some kinds of data, it can be useful to mask (or hide) certain values. For example, cloud-covered pixels in an optical satellite image. Masks are boolean arrays (values of `True` and `False`), where values of `True` refer to locations that should be masked.

```
>>> x = np.random.rand(6,6)

>>> x > 0.5
array([[ True,  True, False, False, False, False],
       [ True, False, False,  True, False, False],
       [ True, False,  True, False,  True, False],
       [False,  True, False, False, False,  True],
       [False,  True, False, False, False, False],
       [ True, False,  True, False,  True,  True]], dtype=bool)

>>> np.logical_and(x > 0.5, x < 0.75) # Multiple conditions can be combined
array([[ True,  True, False, False, False, False],
       [False, False, False,  True, False, False],
       [ True, False, False, False,  True, False],
       [False,  True, False, False, False, False],
       [False, False, False, False, False, False],
       [False, False,  True, False, False, False]], dtype=bool)
```

(continues on next page)

(continued from previous page)

```
>>> x[x > 0.9] # Extract all elements that meet given criteria
array([ 0.9354168 ,  0.91462   ,  0.98655895,  0.91459135,  0.90945349])
```

Since the masks are also numpy arrays, these can be stored and used to mask other arrays.

```
>>> mask = x > 0.9

>>> mask # Mask shows True where x > 0.9
array([[False, False, False, False, False, False],
       [ True, False, False, False, False, False],
       [False, False, False, False, False, False],
       [False, False, False, False, False,  True],
       [False,  True, False, False, False, False],
       [False, False, False, False,  True,  True]], dtype=bool)

>>> y = np.random.rand(6,6)

>>> y[mask] # Extract the values of y where x > 0.9
array([ 0.28332556,  0.20372563,  0.15115744,  0.08839921,  0.83573746])
```

We can apply multiple conditions these boolean masks using the `and` and `or` statements we looked at in the previous section. However, the syntax is a little different:

```
>>> mask = np.logical_or(x < 0.25, x > 0.75) # Masks everything below 0.25 and over 0.75

>>> mask = np.logical_and(x > 0.25, x < 0.75) # Masks everything over 0.25 but below 0.75
```

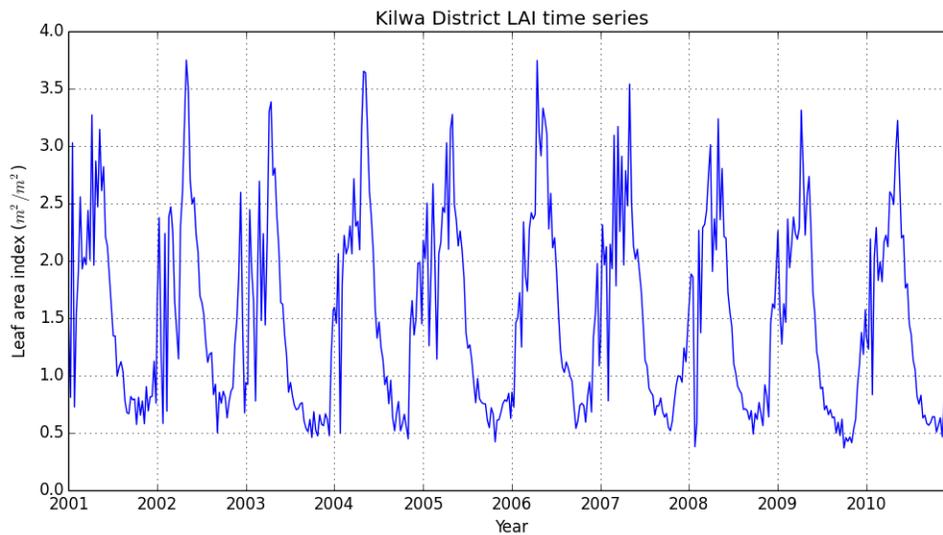
There is also a special class of numpy array known as a masked array, which we'll look at in the next section. You can use this to store both data and a mask in one object. You can create a masked array as follows:

```
>>> masked_array = np.ma.array(x, mask = mask) # Creates a masked array.
```

### Exercise: Looking at some real data

Here we'll take a look at some real data from [MODIS](#). I have extracted a time series of leaf area index (LAI) estimates every 8 days over the years 2001 - 2010 over Kilwa District in southeastern Tanzania. The data can be downloaded here: [kilwaLAI.npz](#).

The data look like this:



We'll learn to make a plot like this in the next section.

First, download the data and load into Python using the following commands.

```
>>> data = np.load('kilwaLAI.npz') # make sure that you put the path to your copy of
↳ kilwaLAI.npz

>>> LAI = data['LAI'] # Leaf area index, units of m2/m2
>>> year = data['year'] # 2001 to 2010
>>> month = data['month'] # 1 to 12
```

Explore the data contained in these four arrays. See if you can use the functions of `numpy` to answer the following questions.

1. What is the average LAI in Kilwa District over the whole monitoring period?
2. What is the minimum and maximum LAI observed in Kilwa District?
3. What proportion of the observations is LAI above 2.0?
4. What was the average LAI in February 2005?
5. In what months is LAI in Kilwa District at its maximum and minimum? (Hint: This is difficult. Consider using a `for` loop.)

## 3.4 Introduction to Matplotlib

Matplotlib is a module that is used for plotting data. Take a look at the [Matplotlib gallery](#) to see all the plots you can make with matplotlib given enough time.

Here we'll cover two of the most commonly used functions of matplotlib; making scatterplots and displaying images.

We import matplotlib as follows:

```
>>> import matplotlib.pyplot as plt
```

### 3.4.1 Plotting data

As an example dataset, we'll look at the relationship between tree diameter (DBH) and tree height from some measurements in miombo woodland in Tanzania. Allometric relationships similar to this are central to estimating the biomass in forest plots.

The data we'll use is named `treeData.csv`.

First, we'll load the data into Python:

```
>>> import numpy as np # Make sure numpy is loaded

>>> tree_data = np.loadtxt('treeData.csv',skiprows=1,delimiter=',') # Load in data.

>>> tree_data
array([[ 37.6 ,  11.5 ],
       [ 20.1 ,   7.5 ],
       [ 32.5 ,  10.5 ],
       [ .... ,  .... ],
       [ 54.5 ,  12.75],
       [ 11.9 ,   6.25],
       [ 11.5 ,   5.75]])
```

The first column of this dataset contains estimates of tree DBH (units = cm), and the second column contains estimates of tree height (units = m). At the moment the data have been loaded as a single 2 dimensional array. We can separate it into two arrays as follows:

```
>>> DBH = tree_data[:,0] # Index every row and the first column

>>> DBH
array([ 37.6,  20.1,  32.5,  47.4,  37.9,  38.8,  41.1,  22.9,  24.1,
        40.2,  36.9,  32.7,  38.9,  27.6,  39.6,  30.1,  77.3,  35.7,
        36.8,  71. ,  62. ,  50.4,  65.7,  16.8,  61.4,  17.9,  19.8,
        20.1,  17.1,  13.3,  25.8,  34.4,  29.4,  25.8,  13.8,  13.4,
        34.5,  57.5,  23.2,  21. ,  71.1,  49.5,  21.4,  25.5,  47. ,
        10.7,  21.9,  19.7,  18.8,  42.9,  65.3,  18.4,  19. ,  50.6,
        51.5,  43.3,  28. ,  19. ,  15.9,  39.3,  41.1,  38.1,  13.9,
        10.3,  10.2,   9.7,  16.6,  28.8,  11.7,  16.6,  15.1,  39.8,
        25.9,  46.3,  27.1,  52.4,  49.7,  36.2,  26.1,  25. ,  21.4,
        20.1,  29.9,  23.1,  74. ,  62.1,  46.2,  16.8,  23.4,  20.4,
        64.4,  33.6,  59.2,  32. ,  18.5,  15.8,  13.8,  15.8,  42.7,
        24.1,  16.5,  10.8,  11.3,  57.2,  47.7,  27.7,  21.2,  17.5,
        13.3,  11.9,  31.2,  28.2,  34. ,  43.4,  31.7,  24. ,  18.3,
        27. ,  23. ,  33.8,  46.4,  19. ,  18. ,  40. ,  27.9,  13.8,
        19. ,  19.2,  17.8,  12.9,  35.9,  18.2,  56. ,  22. ,  30.9,
        25.4,  41.8,  14.2,  53.7,  28.8,  34.9,  26.7,  45.1,  12.1,
        26. ,  10.8,  16.2,  13.1,  27.2,  25.9,  48.1,  25.5,  48.2,
        52.8,  32.3,  40.2,  32. ,  24.1,  13.5,  37.5,  20.8,  42. ,
        20.6,  16.6,  20. ,  20.3,  21.1,  18.8,  17.1,  16.9,  20.3,
        15.4,  48.8,  39.5,  49.7,  54.3,  59.4,  12.5,  31.9,  37.1,
        16.5,  68. ,  14.4,  43.9,  26.6,  27.9,  72.5,  33.8,  53.2,
        27.2,  30.7,  49.4,  10.1,  38.1,  29.2,  10.7,  20.2,  23.9,
        29.1,  25.3,  26.2,  51.5,  51.4,  14. ,  12.9,  14.7,  33.3,
        67.5,  26.7,  20.3,  27.1,  17.4,  15.9,  15. ,  17.5,  37. ,
        13.7,   8.4,  10.5,  11.5,  13.2,  13.7,   8.9,  11.7,  10.9,
        67.1,  71.8,  35.1,  35. ,  17. ,  21.1,  29.7,  18.1,  14. ,
        33.2,  56.7,  13.4,  13. ,  14.2,  51. ,  29.6,  35.2,  22.1,
        50.5,  41.9,  39.6,  31. ,  46.1,  42.9,  45.9,  28.9,  40.3,
```

(continues on next page)

(continued from previous page)

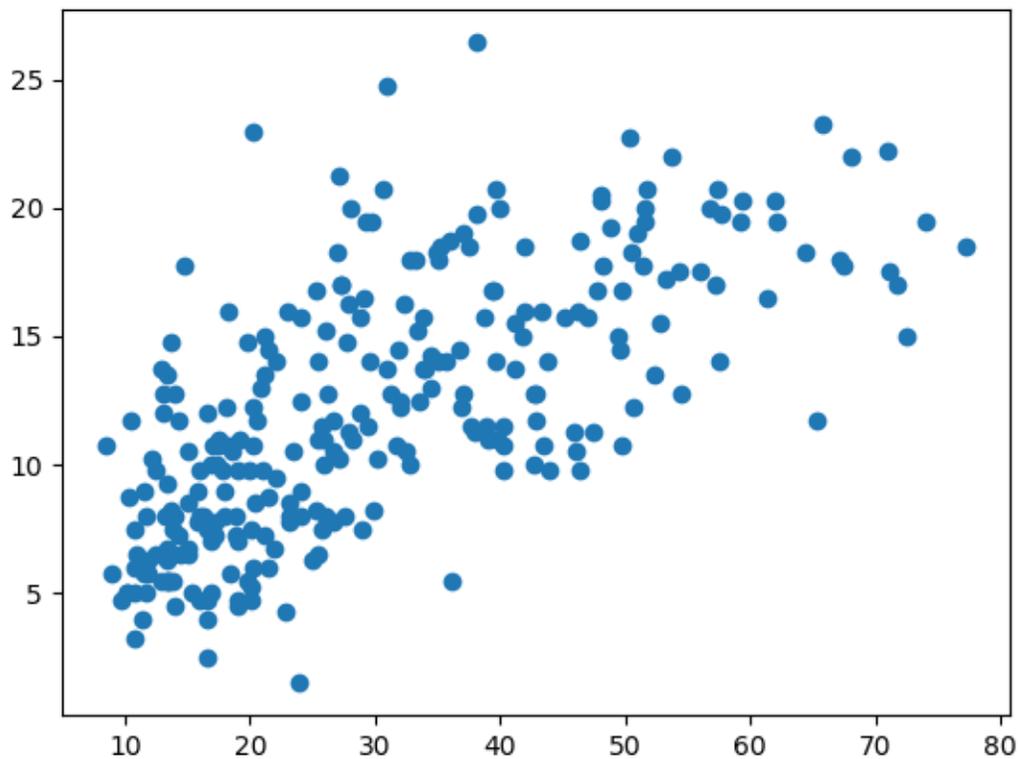
```
39.1, 48. , 15.1, 13.3, 13.4, 16. , 12.4, 42.7, 57.7,
21.4, 25.2, 51.7, 57.3, 23.2, 32.7, 15. , 43.7, 54.5,
11.9, 11.5])
```

```
>>> height = tree_data[:,1] # Index every row and the second column
```

Next, we'll build a scatter plot to show the relationship between DBH and tree height using matplotlib:

```
>>> plt.scatter(DBH, height) # Build scatter plot
<matplotlib.collections.PathCollection at 0x30fdd50>

>>> plt.show()
```



There's a lot we can do to improve this diagram, such as using different colours and markers, and adding labels.

```
>>> plt.scatter(DBH, height, c='darkgreen', marker='x', s=10) # Try other colours [c],
↪ sizes [s], and marker [marker] types!
<matplotlib.collections.PathCollection at 0x3123450>

>>> plt.xlabel('DBH (cm)') # Set x-axis label
<matplotlib.text.Text at 0x3112890>

>>> plt.ylabel('Tree height (m)') # Set y-axis label
<matplotlib.text.Text at 0x30997d0>
```

(continues on next page)

(continued from previous page)

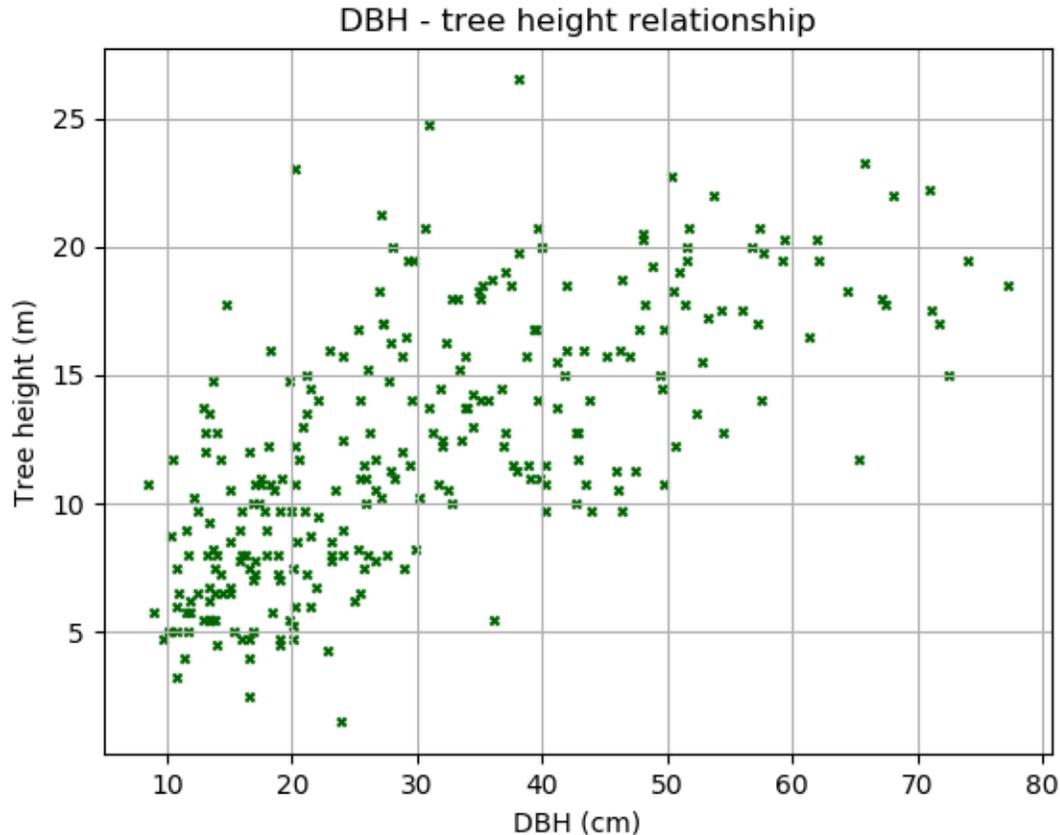
```

>>> plt.title('DBH - tree height relationship') # Set title
<matplotlib.text.Text at 0x312fd90>

>>> plt.grid() # Add a grid

>>> plt.show()

```



We can also add the linear line of best fit. First determine the slope and intercept of the best fit line:

```

>>> par = np.polyfit(DBH, height, 1, full=True)

>>> slope=par[0][0]

>>> intercept=par[0][1]

```

Then plot:

```

>>> plt.scatter(DBH, height, c='darkgreen', marker='x', s=10)
<matplotlib.collections.PathCollection at 0x5c21ed0>

>>> plt.xlabel('DBH (cm)') # Set x-axis label
<matplotlib.text.Text at 0x5c4eed0>

```

(continues on next page)

(continued from previous page)

```
>>> plt.ylabel('Tree height (m)') # Set y-axis label
<matplotlib.text.Text at 0x5c1b8d0>

>>> plt.title('DBH - tree height relationship')
<matplotlib.text.Text at 0x5c14b50>

>>> plt.grid()

>>> x = np.arange(0,81,1)

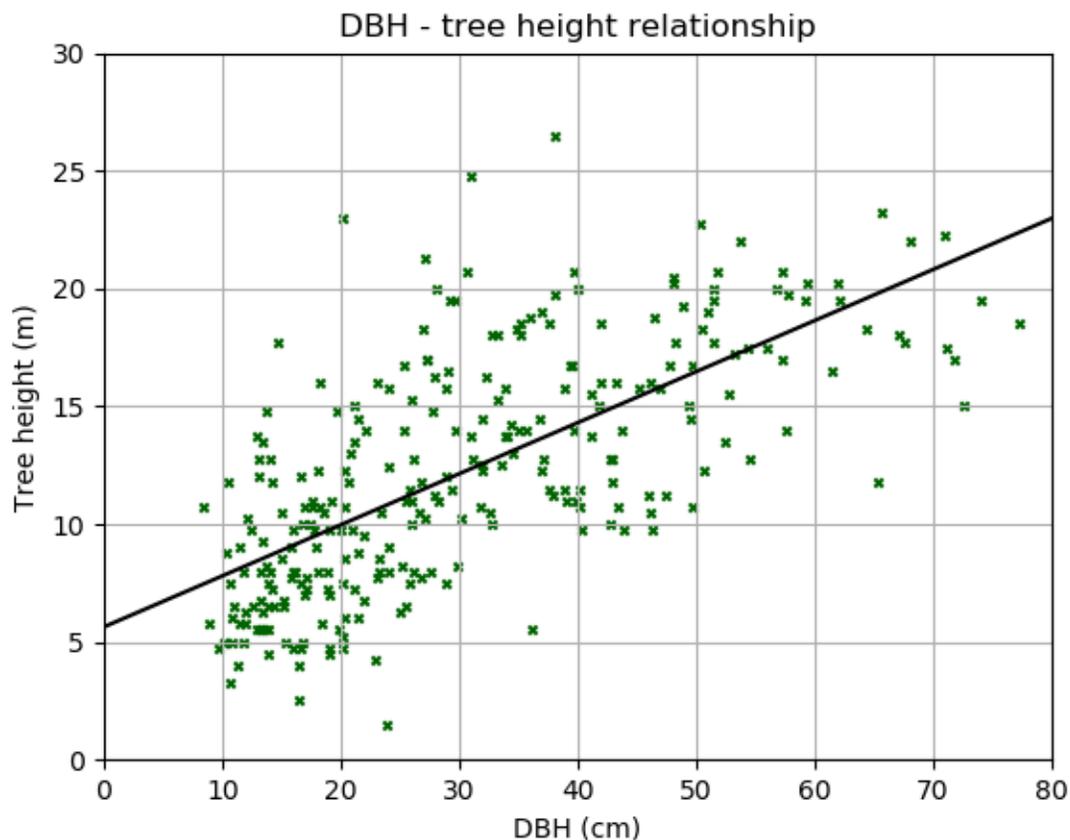
>>> y = x * slope + intercept # y = mx + c

>>> plt.plot(x,y,'k') # 'k' for black
[<matplotlib.lines.Line2D at 0x5c28610>]

>>> plt.xlim(0,80) # Set x-axis limit
(0, 80)

>>> plt.ylim(0,30) # Set y-axis limit
(0, 30)

>>> plt.show()
```



How useful do you think this relationship is? Should we really have used a straight line?

### Exercise: Building a biomass-backscatter relationship

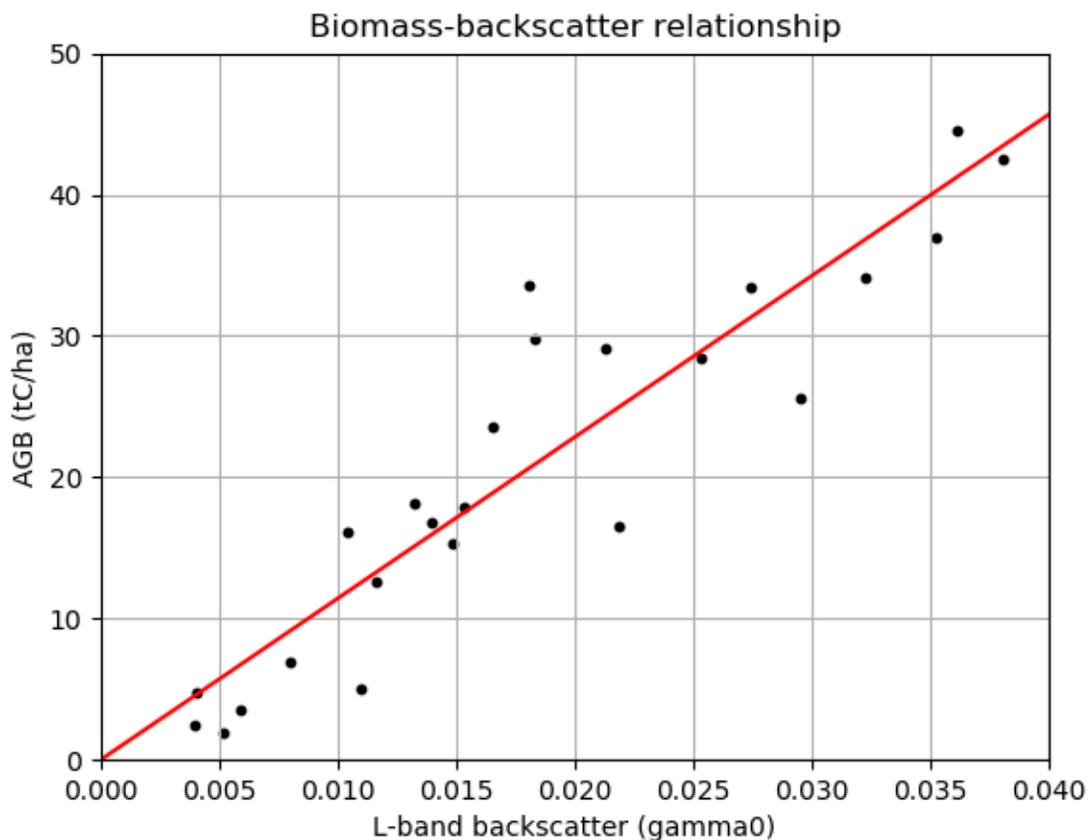
L-band radar backscatter shows a strong response to variation in aboveground biomass (AGB), a property which can be used to map forest biomass. Here we're going to generate a linear biomass-backscatter relationship using data from Tanzania.

The data can be found in '[biomassBackscatter.csv](#)', the first column is radar backscatter ( $s_0$ ) and the second column is AGB (tC/ha).

1. Plot the relationship between biomass and l-band radar backscatter. Include:

- A point for each data point
- A line showing the linear relationship between the biomass and backscatter
- Axis labels and a title

You should aim to create a plot that looks something like this:



### 3.4.2 Displaying images

As an example dataset, we'll look at a Digital Elevation Model (DEM) of Mozambique from the [Shuttle Radar Topography Mission](#).

This data is available for download as a GeoTiff file named [MozambiqueDEM.tif](#).

First, we'll load the data into Python using the [Geospatial Data Abstraction Library \(GDAL\)](#) module. GDAL provides an extremely useful set of tools to process geospatial imagery. We'll cover GDAL in more detail in the next section..

**Warning:** GDAL is notoriously difficult to set up. If you are getting errors, download [MozambiqueDEM.npy](#) and use the following line to load the DEM instead:

```
DEM = np.load('MozambiqueDEM.npy')
```

```
>>> import gdal # Or from osgeo import gdal

>>> ds = gdal.Open('MozambiqueDEM.tif') # Open the geotiff file

>>> DEM = ds.ReadAsArray() # Read the data into a numpy array

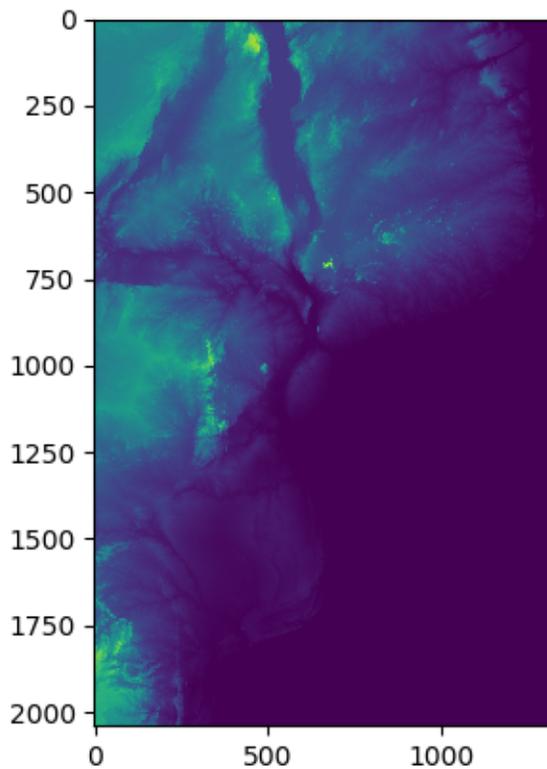
>>> DEM # The units of this data are metres
array([[1476, 1480, 1474, ..., 0, 0, 0],
       [1490, 1491, 1479, ..., 0, 0, 0],
       [1500, 1501, 1494, ..., 0, 0, 0],
       ...,
       [1669, 1701, 1768, ..., 0, 0, 0],
       [1660, 1686, 1732, ..., 0, 0, 0],
       [1661, 1695, 1791, ..., 0, 0, 0]], dtype=int16)
```

We can display the DEM as an image using matplotlib:

```
>>> plt.imshow(DEM) # Plot image

>>> plt.show() # Show image
```

You should see something that looks like this:



In this image you should be able to make out the coastline of Mozambique, as well as locations of major lakes and mountains. However, we can do a lot to improve this image with the options provided by matplotlib.

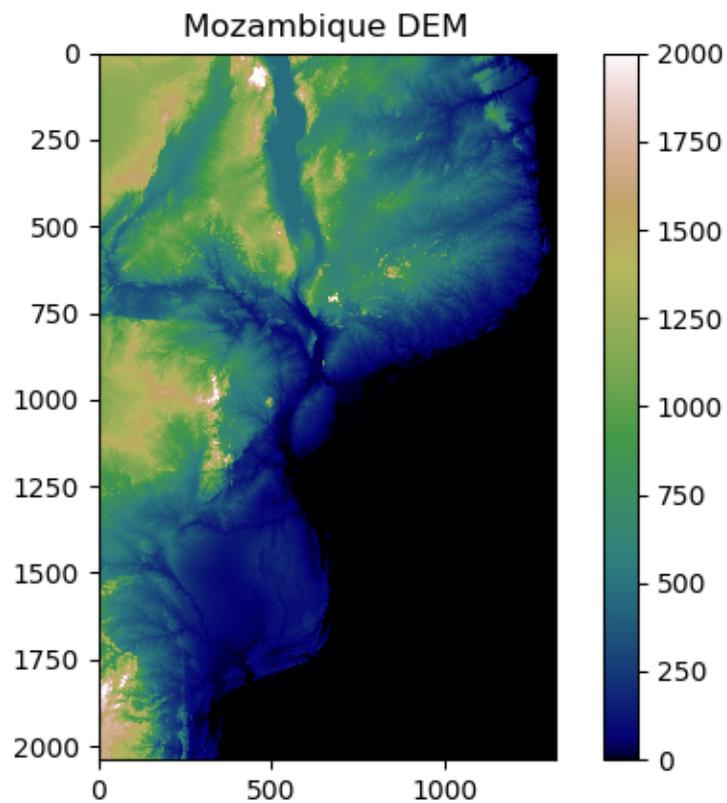
We'll apply a better colour map to the image, specify limits on the color map, show a colour bar to aid interpretation, and add a title. Matplotlib comes with a [range of colourmaps](#) built-in. Pick a colourmap you like, and apply as follows:

```
>>> plt.imshow(DEM, cmap = 'gist_earth', vmin = 0, vmax = 2000) # Choose color map,
↳and limits

>>> plt.colorbar() # Adds a colour bar to the image

>>> plt.title('Mozambique DEM') # Adds a title

>>> plt.show()
```



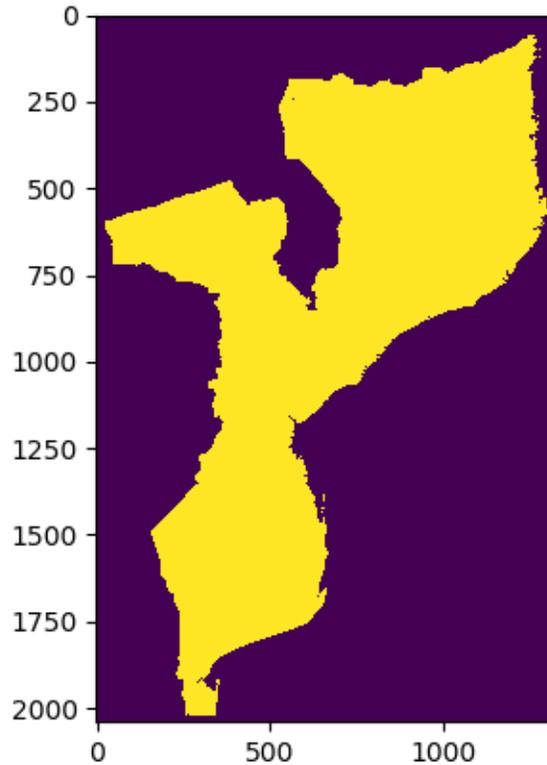
Make sure you understand all the options we specified there. Try out your own colour maps and limits!

### Adding masks

Usually we are interested only in one area of interest. In this case, we might want to mask out anything outside of Mozambique.

First load and display `MozambiqueMask.tif` (or `MozambiqueMask.npy`), which indicates which pixels fall within Mozambique. You can follow the same procedure as we used to load the DEM.

You should end up with something that looks a bit like this:

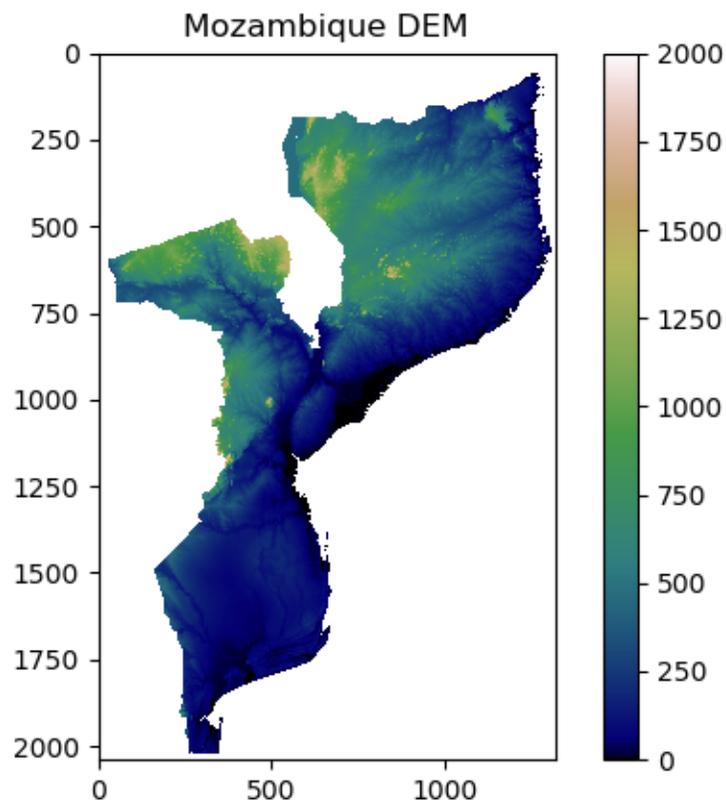


We'll create a masked array from the DEM, that excludes anything outside of the mask:

```
>>> DEM_mask = np.ma.array(DEM, mask = Mozambique_mask)

>>> DEM_mask
masked_array(data =
  [[-- -- -- ..., -- -- --]
  [-- -- -- ..., -- -- --]
  [-- -- -- ..., -- -- --]
  ...,
  [-- -- -- ..., -- -- --]
  [-- -- -- ..., -- -- --]
  [-- -- -- ..., -- -- --]],
  mask =
  [[ True  True  True ...,  True  True  True]
  [ True  True  True ...,  True  True  True]
  [ True  True  True ...,  True  True  True]
  ...,
  [ True  True  True ...,  True  True  True]
  [ True  True  True ...,  True  True  True]
  [ True  True  True ...,  True  True  True]],
  fill_value = 999999)
```

When you plot your new masked array, does it look like this? If not, can you fix it?



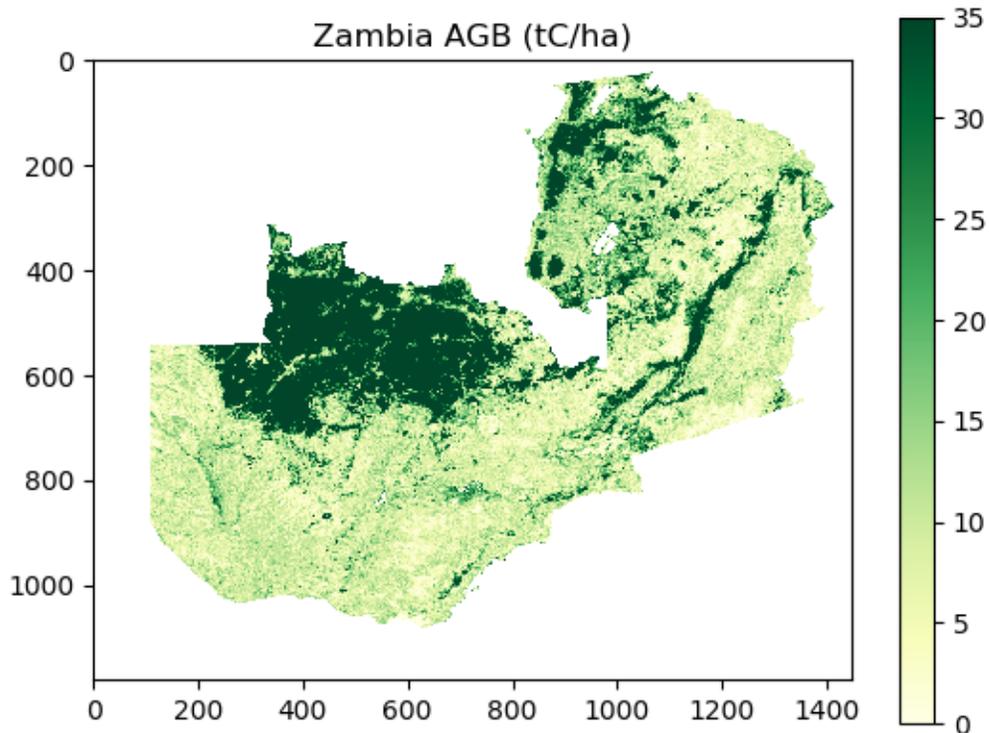
### Exercise: Your turn

I've provided you with some data from the [Avitabile pan-tropical biomass map](#), covering the area of Zambia. The data are in two files, just like the Mozambique DEM example.

- `AGBZambia.tif` (or `AGBZambia.npy`)
  - Data have units of tonnes of carbon per hectate (tC/ha)
  - Data have an equal area projection, with each pixel representing 1 km<sup>2</sup>
  - No data is indicated by a value of `-3.3999999999999996e+38`
- `ZambiaMask.tif` (or `ZambiaMask.npy`)
  - 1 where pixel falls inside Zambia, 0 where pixel falls outside Zambia

With this data, address the following tasks:

1. Load in the data to python, and create a masked array of Zambia's AGB
2. Visualise the data (see below for an example)
3. Calculate Zambia's total AGB stocks according to the Avitabile biomass map. Take care with units and no data values!



## 3.5 To be continued...

### 3.5.1 Further reading

Learning to write useful scripts in Python takes a lot of time and commitment. The best way to learn is by doing.

If you would like to learn more about Python independently, there are loads of free online resources. Here are the sites that I've found particularly useful in learning to use Python:

- [Learn Python the hard way](#): an excellent online Python course
- [Codecademy](#): another well-known Python course
- [Stack Overflow](#): for diagnosing errors
- [Google](#): seriously!

### 3.5.2 Stay in touch

Please do email either Sam ([sam.bowers@ed.ac.uk](mailto:sam.bowers@ed.ac.uk)) or Simone ([simone-vaccari@itsi.co.uk](mailto:simone-vaccari@itsi.co.uk)) if you'd like any assistance or any further tips. We'll be very happy to help.

Happy coding!

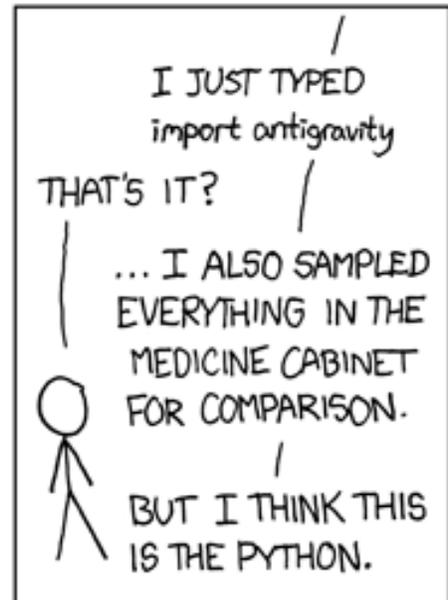
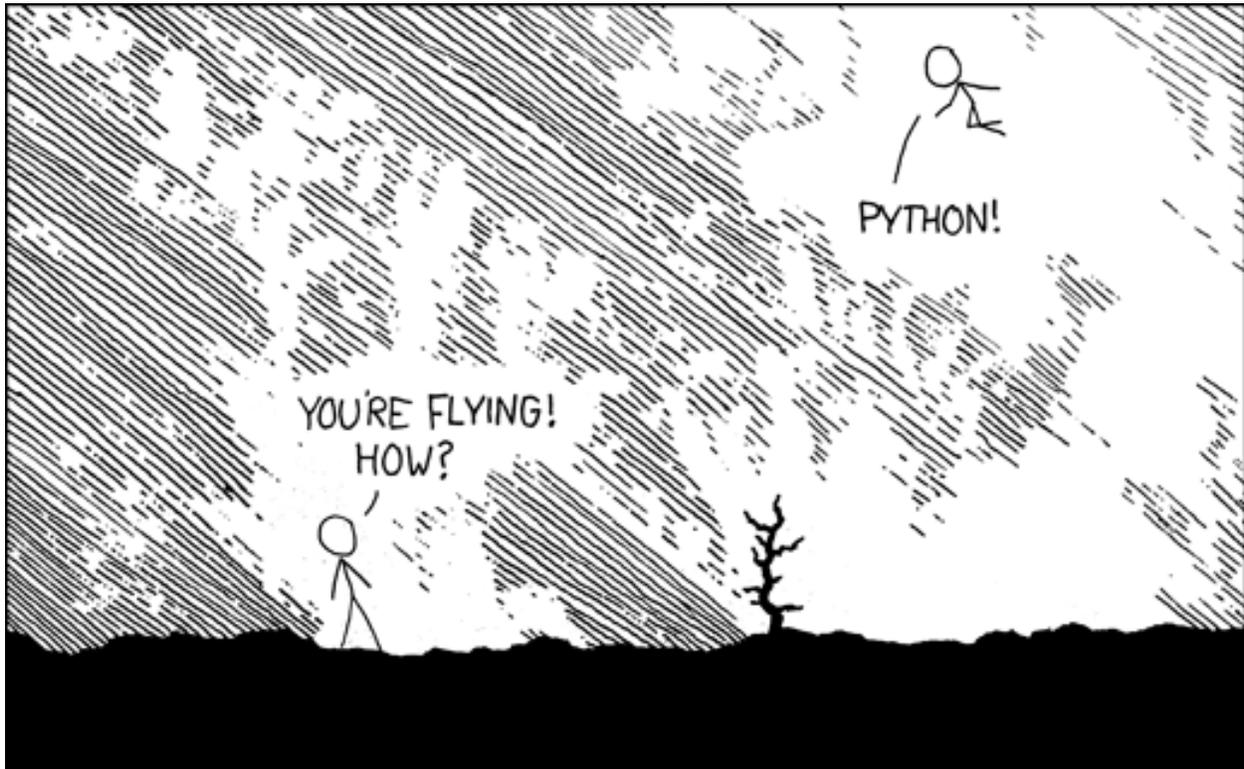


Fig. 1: Souce: XKCD.

---

Search

---

- search



THE UNIVERSITY *of* EDINBURGH

