

A practical introduction to Python

Q-Step Workshop – 02/10/2019

Housekeeping

- This is a whistle-stop tour.
- Focus on the more “common” aspects of Python programming.
- Today is intended to be the first of a two-part workshop:
 - Today: Absolute basics of Python.
 - Workshop on 06/11/2019: Data Analysis and visualisation with Python.
- Learning a language is all about practice.
- Google is your best friend.

An introduction to coding with Python



An introductory coding course with Python...

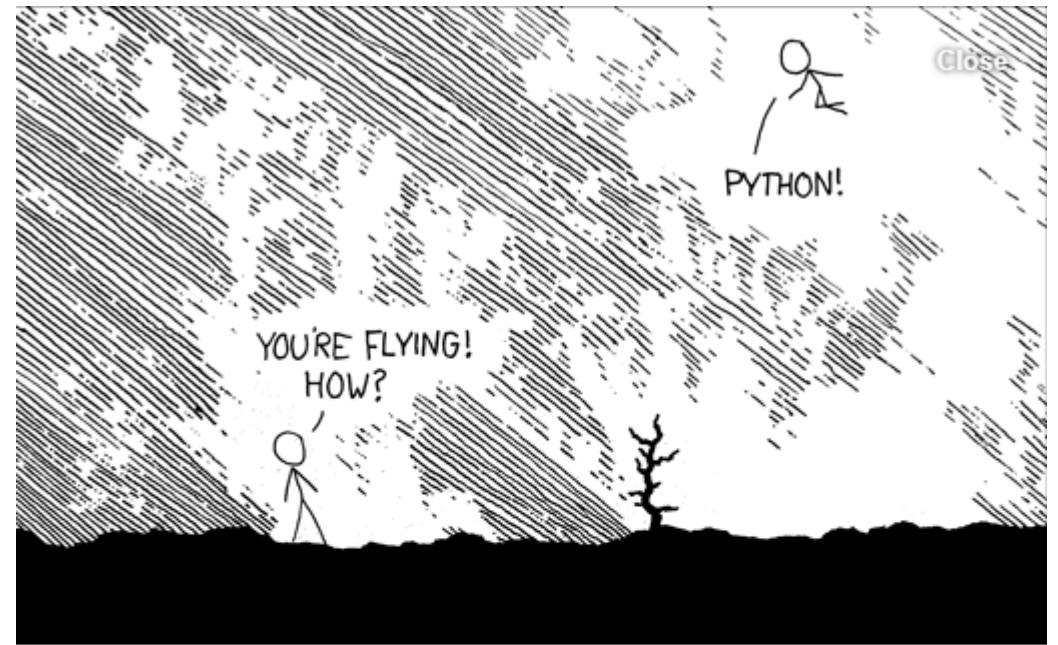
Interpretive vs compiled languages

- Python is an interpretive language.
- This means that your code is not directly run by the hardware. It is instead passed to a *virtual machine*, which is just another programme that reads and interprets your code. If your code used the '+' operation, this would be recognised by the interpreter at run time, which would then call its own internal function 'add(a,b)', which would then execute the machine code 'ADD'.
- This is in contrast to compiled languages, where your code is translated into native machine instructions, which are then directly executed by the hardware. Here, the '+' in your code would be translated directly in the 'ADD' machine code.

Advantages of Python?

Because Python is an interpretive language, it has a number of advantages:

- Automatic memory management.
- Expressivity and syntax that is 'English'.
- Ease of programming.
- Minimises development time.
- Python also has a focus on *importing* modules, a feature that makes it useful for scientific computing.

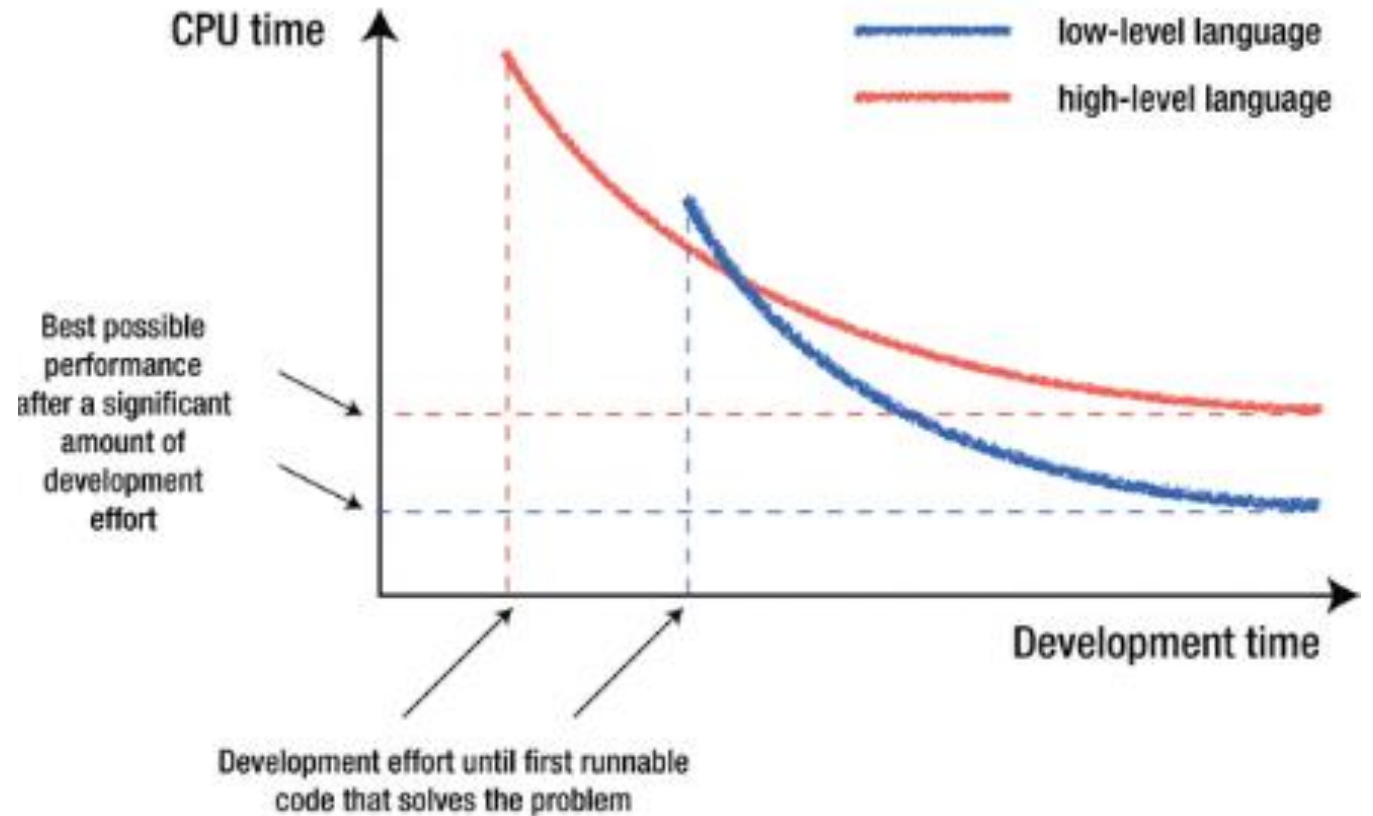


Disadvantages

- Interpreted languages are slower than compiled languages.
- The modules that you import are developed in a decentralised manner; this can cause issues based upon individual assumptions.
- Multi-threading is hard in Python

Which language is the best

- No one language is better than all others.
- The 'best' language depends on the task you are using it for and your personal preference.

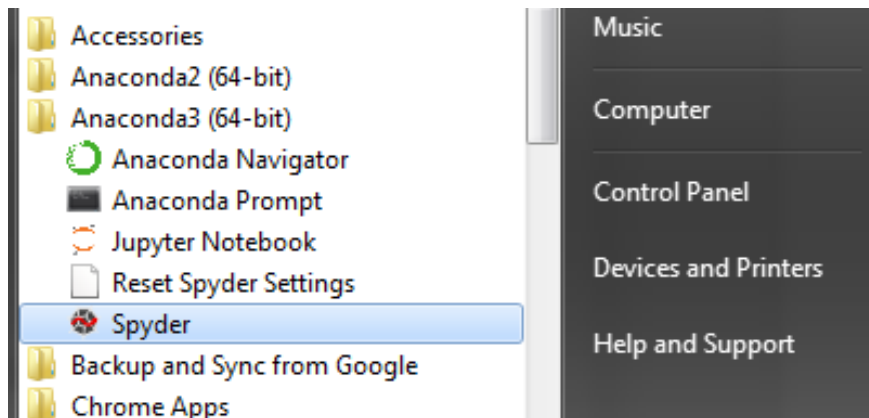


Versions of Python

- There are currently two versions of Python in use; Python 2 and Python 3.
- Python 3 is not backward compatible with Python 2.
- A lot of the imported modules were only available in Python 2 for quite some time, leading to a slow adoption of Python 3. However, this not really an issue anymore.
- Support for Python 2 will end in 2020.

The Anaconda IDE...

- The Anaconda distribution is the most popular Python distribution out there.
- Most importable packages are pre-installed.
- Offers a nice GUI in the form of Spyder.
- Before we go any further, let's open Spyder:



Spyder (Python 3.6)

File Edit Search Source Run Debug Consoles Projects Tools View Help

C:\Users\lb690\.spyder-py3

Editor - C:\Users\lb690\.spyder-py3\temp.py

```
1 # -*- coding: utf-8 -*-
2 """
3 Spyder Editor
4
5 This is a temporary script file.
6 """
7
8 print("Best. Python. Tutorial. Ever!")
```

Usage

Here you can get help of any object by pressing Ctrl+I in front of it, either on the Editor or the Console.

Help can also be shown automatically after writing a left parenthesis next to an object. You can activate this behavior in *Preferences > Help*.

New to Spyder? Read our [tutorial](#)

Help Variable explorer File explorer

IPython console

Console 1/A

Python 3.6.5 [Anaconda, Inc.] (default, Mar 29 2018, 13:32:41) [MSC v.1900 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 6.4.0 -- An enhanced Interactive Python.

In [1]: runfile('C:/Users/lb690/.spyder-py3/temp.py', wdir='C:/Users/lb690/.spyder-py3')
Best. Python. Tutorial. Ever!

In [2]:

IPython console History log

Variables

- Variables in python can contain alphanumerical characters and some special characters.
- By convention, it is common to have variable names that start with lower case letters and have class names beginning with a capital letter; but you can do whatever you want.
- Some keywords are reserved and cannot be used as variable names due to them serving an in-built Python function; i.e. `and`, `continue`, `break`. Your IDE will let you know if you try to use one of these.
- Python is dynamically typed; the type of the variable is derived from the value it is assigned.

Variable types

- Integer (`int`)
- Float (`float`)
- String (`str`)
- Boolean (`bool`)
- Complex (`complex`)
- [...]
- User defined (`classes`)

- A variable is assigned using the `=` operator; i.e:

```
In: intVar = 5
    floatVar = 3.2
    stringVar = "Food"

    print(intVar)
    print(floatVar)
    print(stringVar)

Out: In [4]: runfile
        1b690/.spyder
        5
        3.2
        Food
```

- The `print()` function is used to print something to the screen.
- Create an integer, float, and string variable.
- Print these to the screen.
- Play around using different variable names, etc.

- You can always check the type of a variable using the `type()` function.

```
In: variable = 100  
    print(type(variable))
```

```
Out: <class 'int'>
```

- Check the type of one of your variables.

- Variables can be *cast* to a different type.

```
In: share_of_rent = 295.50/2.0
print("1:", share_of_rent)
print(type(share_of_rent))
rounded_share = int(share_of_rent)
print("2:", rounded_share)
print(type(rounded_share))
```

```
Out: 1: 147.75
      <class 'float'>
      2: 147
      <class 'int'>
```

Arithmetic operators

The arithmetic operators:

- Addition: +
- Subtract: -
- Multiplication: *
- Division: /
- Power: **

- Write a couple of operations using the arithmetic operators, and print the results to the screen.

```
In: print(5+5)
```

```
x = 2
```

```
y = 10
```

```
print(x/y)
```

```
Out: In [11]:
```

```
1b690/.s
```

```
10
```

```
0.2
```

A quick note on the increment operator shorthand

- Python has a common idiom that is not necessary, but which is used frequently and is therefore worth noting:

```
x += 1
```

Is the same as:

```
x = x + 1
```

- This also works for other operators:

<pre>x += y</pre>	<pre># adds y to the value of x</pre>
<pre>x *= y</pre>	<pre># multiplies x by the value y</pre>
<pre>x -= y</pre>	<pre># subtracts y from x</pre>
<pre>x /= y</pre>	<pre># divides x by y</pre>

Boolean operators

- Boolean operators are useful when making conditional statements, we will cover these in-depth later.
- `and`
- `or`
- `not`

Comparison operators

- Greater than: >
- Lesser than: <
- Greater than or equal to: >=
- Lesser than or equal to: <=
- Is equal to: ==

- Write a couple of operations using comparison operators; i.e.

```
In: intVar = 5
    floatVar = 3.2
    stringVar = "Food"

    if intVar > floatVar:
        print("Yes")

    if intVar == 5:
        print("A match!")
```

```
Out: In [9]: run
      1b690/.spyd
      Yes
      A match!
```

Working with strings

```
In: greeting = 'Hello, Lew!'
print('1:', greeting)
print('2:', len(greeting))
print('3:', greeting[0])
print('4:', greeting[-1])
greeting = greeting.replace("Lew", "class")
print('5:', greeting)
string1 = "Hello"
string2 = "world"
print("1:", string1, string2)
cost = float(35.28)
print("Bar tab = £%f" %cost)
```

```
Out: 1: Hello, Lew!
      2: 11
      3: H
      4: !
      5: Hello, class!
      1: Hello world
      Bar tab = £35.280000
```

- Create a string variable.
- Work out the length of the string.

Dictionaries

- Dictionaries are lists of key-valued pairs.

```
In: prices = {"Eggs": 2.30,  
             "Steak": 13.50,  
             "Bacon": 2.30,  
             "Beer": 14.95}  
print("1:", prices)  
print("2:", type(prices))  
print("The price of bacon is:", prices["Bacon"])
```

```
Out: 1: {'Eggs': 2.3, 'Steak': 13.5, 'Bacon': 2.3, 'Beer':  
14.95}  
2: <class 'dict'>  
The price of bacon is: 2.3
```

Indexing

- Indexing in Python is 0-based, meaning that the first element in a string, list, array, etc, has an index of 0. The second element then has an index of 1, and so on.

```
In: test_string = "Dogs are better than cats"  
    print('First element:', test_string[0])  
    print('Second element:', test_string[1])
```

```
Out: First element: D  
     Second element: o
```

- You can cycle backwards through a list, string, array, etc, by placing a minus symbol in front of the index location.

```
In: test_string = "Dogs are better than cats"  
    print('Last element:', test_string[-1])  
    print('Second to last element:', test_string[-2])
```

```
Out: Last element: s  
     Second to last element: t
```

```
In: test_string = "Dogs are better than cats"
    print('Last element:', test_string[4:])
    print('Second to last element:', test_string[:4])
```

```
Out: Last element: are better than cats
     Second to last element: Dogs
```

```
In: test_string = "Dogs are better than cats"
    print(test_string[5:10])
```

```
Out: are b
```

- Create a string that is 10 characters in length.
- Print the second character to the screen.
- Print the third to last character to the screen.
- Print all characters after the fourth character.
- Print characters 2-8.

Tuples

- Tuples are containers that are immutable; i.e. their contents cannot be altered once created.

```
In: tuple1 = (5, 10)
    print('1:', tuple1)
    print("2:", type(tuple1))
```

Out: 1: (5, 10)
2: <class 'tuple'>

```
In: tuple1[1] = 6
```

Out: TypeError: 'tuple' object does not support item assignment

Lists

- Lists are essentially containers of arbitrary type.
- They are probably the container that you will use most frequently.
- The elements of a list can be of different types.
- The difference between tuples and lists is in performance; it is much faster to 'grab' an element stored in a tuple, but lists are much more versatile.
- Note that lists are denoted by `[]` and not the `()` used by tuples.

```
In: numbers = [1, 2, 3]
print("List 1:", numbers)
print("Type of list 1:", type(numbers))
arbitrary_list = [1, numbers, "Hello"]
print("Arbitrary list:", arbitrary_list)
print("Type of arbitrary list:", type(arbitrary_list))
```

```
Out: List 1: [1, 2, 3]
Type of list 1: <class 'list'>
Arbitrary list: [1, [1, 2, 3], 'Hello']
Type of arbitrary list: <class 'list'>
```

- Create a list and populate it with some elements.

Adding elements to a list

- Lists are mutable; i.e. their contents can be changed. This can be done in a number of ways.
- With the use of an index to replace a current element with a new one.

```
In: numbers = [1, 2, 3]           Out: List 1: [1, 2, 3]
    print("List 1:", numbers)     Amended list 1: [1, 5, 3]
    numbers[1] = 5
    print("Amended list 1:", numbers)
```

- Replace the second element in your string with the integer 2.

- You can use the `insert()` function in order to add an element to a list at a specific indexed location, without overwriting any of the original elements.

```
In: numbers = [1, 2, 3]
    print("List 1:", numbers)
    numbers.insert(2, 'Surprise!')
    print("Amended list 1:", numbers)
```

```
Out: List 1: [1, 2, 3]
      Amended list 1: [1, 2, 'Surprise!', 3]
```

- Use `insert()` to put the integer 3 after the 2 that you just added to your string.

- You can add an element to the end of a list using the `append()` function.

```
In: numbers = [1, 2, 3]
    print("List 1:", numbers)
    numbers.append(4)
    print("Amended list 1:", numbers)

Out: List 1: [1, 2, 3]
      Amended list 1: [1, 2, 3, 4]
```

- Use `append()` to add the string "end" as the last element in your list.

Removing elements from a list

- You can remove an element from a list based upon the element value.
- Remember: If there is more than one element with this value, only the first occurrence will be removed.

```
In: numbers = [1, 2, 3, 3]
    print("List 1:", numbers)
    numbers.remove(3)
    print("Amended list 1:", numbers)
```

```
Out: List 1: [1, 2, 3, 3]
      Amended list 1: [1, 2, 3]
```

- It is better practice to remove elements by their index using the `del` function.

```
In: numbers = [1, 2, 3, 4]
    print("List 1:", numbers)
    del numbers[1]
    print("Amended list 1:", numbers)
    del numbers[-1]
    print("Amended list 2:", numbers)
```

```
Out: List 1: [1, 2, 3, 4]
      Amended list 1: [1, 3, 4]
      Amended list 2: [1, 3]
```

- Use `del` to remove the 3 that you added to the list earlier.

For loops

- The for loop is used to iterate over elements in a sequence, and is often used when you have a piece of code that you want to repeat a number of times.
- For loops essentially say:

“For all elements in a sequence, do something”

An example

- We have a list of species:

```
species = ['dog', 'cat', 'shark', 'falcon', 'deer', 'tyrannosaurus rex']  
for i in species:  
    print(i)
```

- The command underneath the list then cycles through each entry in the species list and prints the animal's name to the screen. Note: The *i* is quite arbitrary. You could just as easily replace it with 'animal', 't', or anything else.

```
In [1]: runfile('//is  
dog  
cat  
shark  
falcon  
deer  
tyrannosaurus rex
```

Another example

- We can also use for loops for operations other than printing to a screen. For example:

```
numbers = [1, 20, 18, 5, 15, 160]
total = 0
for value in numbers:
    total = total + value
print(total)
```

```
In [4]: runfile('')
219
```

- Using the list you made a moment ago, use a for loop to print each element of the list to the screen in turn.

The range() function

- The `range()` function generates a list of numbers, which is generally used to iterate over within for loops.
- The `range()` function has two sets of parameters to follow:

`range(stop)`

stop: Number of integers (whole numbers) to generate, starting from zero. i.e:

```
for i in range(5):  
    print(i)
```

```
In [6]: runfile('/  
0  
1  
2  
3  
4
```

`range([start], stop[, step])`

start: Starting number of the sequence.

stop: Generate numbers up to, but not including this number.

step: Difference between each number in the sequence

i.e.:

```
for i in range(3,6):  
    print(i)
```

```
In [7]: runfile('/  
3  
4  
5
```

```
for i in range(4, 10, 2):  
    print(i)
```

```
In [8]: runfile  
4  
6  
8
```

Note:

- All parameters must be integers.
- Parameters can be positive or negative.
- The `range()` function (and Python in general) is 0-index based, meaning list indexes start at 0, not 1. eg. The syntax to access the first element of a list is `mylist[0]`. Therefore the last integer generated by `range()` is up to, but not including, stop.

- Create an empty list.

```
new_list = []
```

- Use the range() and append() functions to add the integers 1-20 to the empty list.

```
for i in range(1, 21):  
    new_list.append(i)
```

- Print the list to the screen, what do you have?

```
print(new_list)
```

Output: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]

The `break()` function

- To terminate a loop, you can use the `break()` function.
- The `break()` statement breaks out of the innermost enclosing `for` or `while` loop.

```
for i in range(1, 10):  
    if i == 3:  
        break  
    print(i)
```

```
In [9]: runfile('//isa  
1  
2
```

The `continue()` function

- The `continue()` statement is used to tell Python to skip the rest of the statements in the current loop block, and then to continue to the next iteration of the loop.

```
for i in range(1, 10):  
    if i == 3:  
        continue  
    print(i)
```

```
In [10]: runfile('//isa
```

```
1  
2  
4  
5  
6  
7  
8  
9
```

While loops

- The while loop tells the computer to do something as long as a specific condition is met.
- It essentially says:

“while this is true, do this.”

- When working with while loops, its important to remember the nature of various operators.
- While loops use the `break()` and `continue()` functions in the same way as a for loop does.

An example

```
species = ['dog', 'cat', 'shark', 'falcon', 'deer', 'tyrannosaurus rex']
i = 0
while i < 3:
    print(species[i])
    i = i + 1
```

```
In [11]: runfile('///i:
dog
cat
shark
```

A bad example

```
counter = 0
while counter <= 100:
    print(counter)
    counter + 99
```

- Create a variable and set it to zero.
- Write a while loop that states that, while the variable is less than 250, add 1 to the variable and print the variable to the screen.

```
In: counter = 0
```

```
while counter < 250:  
    counter += 1  
    print(counter)
```

```
Out: 246  
247  
248  
249  
250
```

- Replace the < with <=, what happens?

```
249  
250  
251
```


For loop vs. while loop

- You will use for loops more often than while loops.
- The for loop is the natural choice for cycling through a list, characters in a string, etc; basically, anything of *determinate* size.
- The while loop is the natural choice if you are cycling through something, such as a sequence of numbers, an *indeterminate* number of times until some condition is met.

Nested loops

- In some situations, you may want a loop within a loop; this is known as a nested loop.
- What will the code on the right produce?
- Recreate this code and run it, what do you get?

```
In: for x in range(1, 11):  
    for y in range(1, 11):  
        print('%d * %d = %d' %(x, y, x*y))
```

```
Out: In [16]: runfile('//i:  
1 * 1 = 1  
1 * 2 = 2  
1 * 3 = 3  
1 * 4 = 4  
1 * 5 = 5  
1 * 6 = 6  
1 * 7 = 7
```

Conditionals

- There are three main conditional statements in Python; `if`, `else`, `elif`.
- We have already used `if` when looking at `while` loops.

```
In: school_night = True
    if school_night == True:
        print("No beer")
    else:
        print("You may have beer")
```

Out: No beer

```
In: school_night = False
    if school_night == True:
        print("No beer")
    else:
        print("You may have beer")
```

Out: You may have beer

An example of elif

```
In: Lew_is_tired = False
    Lew_is_hungry = True
    if Lew_is_tired is True:
        print("Lew has to teach")
    elif Lew_is_hungry is True:
        print("No food for Lew")
    else:
        print("Go on, have a biscuit")
```

```
Out: No food for Lew
```

Functions

- A function is a block of code which only runs when it is called.
- They are really useful if you have operations that need to be done repeatedly; i.e. calculations.
- The function must be defined before it is called. In other words, the block of code that makes up the function must come before the block of code that makes use of the function.

```
In: def practice_function(a, b):  
      answer = a * b  
      return answer  
  
x = 5  
y = 4  
calculated = practice_function(x, y)  
print(calculated)
```

```
Out: In [10]: runfi  
1b690/.spyder-  
20
```

- Create a function that takes two inputs, multiplies them, and then returns the result. It should look some like:

```
def function_name(a, b):  
    do something  
    return something
```

```
def multiply_function(a, b):  
    result = a * b  
    return result
```

- Create two different lists of integers.
- Using your function, write a nested for loop that cycles through each entries in the first list and multiples it by each of the entries in the second list, and prints the result to the screen.

```
In: def multiply_function(a, b):  
    result = a * b  
    return result  
  
numbers_list = [1, 2, 3]  
multiplier_list = [2, 4]  
for n in numbers_list:  
    print("_____")  
    for m in multiplier_list:  
        current_answer = multiply_function(n, m)  
        print("The answer to %d * %d is: " %(n, m), current_answer)
```

```
Out: _____  
The answer to 1 * 2 is: 2  
The answer to 1 * 4 is: 4  
  
_____  
The answer to 2 * 2 is: 4  
The answer to 2 * 4 is: 8  
  
_____  
The answer to 3 * 2 is: 6  
The answer to 3 * 4 is: 12
```

Multiple returns

- You can have a function return multiple outputs.

```
In: def multiply_function(a, b):  
    result = a * b  
    result2 = result * result  
    return result, result2  
  
numbers_list = [1, 2, 3]  
multiplier_list = [2, 4]  
for n in numbers_list:  
    print("_____")  
    for m in multiplier_list:  
        current_answer, current_answer2 = multiply_function(n, m)  
        print("The answer to %d * %d is: " %(n, m), current_answer)  
        print("The result of this squared is: ", current_answer2)
```

```
Out: _____  
The answer to 1 * 2 is: 2  
The result of this squared is: 4  
The answer to 1 * 4 is: 4  
The result of this squared is: 16  
  
_____  
The answer to 2 * 2 is: 4  
The result of this squared is: 16  
The answer to 2 * 4 is: 8  
The result of this squared is: 64  
  
_____  
The answer to 3 * 2 is: 6  
The result of this squared is: 36  
The answer to 3 * 4 is: 12  
The result of this squared is: 144
```

Reading and writing to files in Python: The file object

- File handling in Python can easily be done with the built-in object `file`.
- The `file` object provides all of the basic functions necessary in order to manipulate files.
- Open up notepad or notepad++. Write some text and save the file to a location and with a name you'll remember.

The `open()` function

- Before you can work with a file, you first have to open it using Python's in-built `open()` function.
- The `open()` function takes two arguments; the name of the file that you wish to use and the mode for which we would like to open the file

```
practiceFile = open('Practice_file_for_IOC.txt', 'r')
```

- By default, the `open()` function opens a file in 'read mode'; this is what the 'r' above signifies.
- There are a number of different file opening modes. The most common are: 'r'=read, 'w'=write, 'r+'=both reading and writing, 'a'=appending.
- Use the `open()` function to read the file in.

The `close()` function

- Likewise, once you're done working with a file, you can close it with the `close()` function.
- Using this function will free up any system resources that are being used up by having the file open.

```
practiceFile.close()
```

Reading in a file and printing to screen example

Using what you have now learned about for loops, it is possible to open a file for reading and then print each line in the file to the screen using a for loop.

- Use a for loop and the variable name that you assigned the open file to in order to print each of the lines in your file to the screen.

```
In: practiceFile = open('practice_file.txt', 'r')
     for line in practiceFile:
         print(line)
```

```
Out: In [42]: runfile('//isad.isadroot.
             User/Desktop')
             The first line of text

             The second line of text

             The third line of text

             The fourth line of text

             I'm bored now, you get the idea
```

The read() function

- However, you don't need to use any loops to access file contents. Python has three in-built file reading commands:

1. `<file>.read()` = Returns the entire contents of the file as a single string:

```
practiceFile = open('practice_file.txt', 'r')
print(practiceFile.read())
```

```
In [44]: runfile('///isad.isadroot.e
User/Desktop')
The first line of text
The second line of text
The third line of text
The fourth line of text
I'm bored now, you get the idea
```

2. `<file>.readline()` = Returns one line at a time:

```
practiceFile = open('practice_file.txt', 'r')
print(practiceFile.readline())
```

```
In [51]: runfile('///isad.i
User/Desktop')
The first line of text
```

3. `<file>.readlines()` = Returns a list of lines:

```
practiceFile = open('practice_file.txt', 'r')
print(practiceFile.readlines())
```

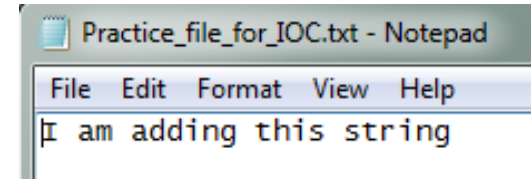
```
In [52]: runfile('///isad.isadroot.ex.ac.uk/UOE/User/Desktop/IOC_test.py',
wdir='///isad.isadroot.ex.ac.uk/UOE/User/Desktop')
['The first line of text\n', 'The second line of text\n', 'The third line of
text\n', 'The fourth line of text\n', "I'm bored now, you get the idea\n"]
```

The write() function

- Likewise, there are two similar in-built functions for getting Python to write to a file:

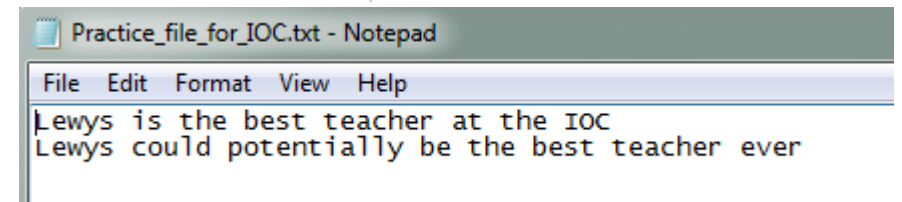
1. `<file>.write()` = Writes a specified sequence of characters to a file:

```
practiceFile = open('Practice_file_for_IOC.txt', 'w')
practiceFile.write('I am adding this string')
```



2. `<file>.writelines()` = Writes a list of strings to a file:

```
testList = ['Lewys is the best teacher at the IOC\n', 'Lewys could potentially be the best teacher ever\n']
practiceFile = open('Practice_file_for_IOC.txt', 'w')
practiceFile.writelines(testList)
```



- Important: Using the `write()` or `writelines()` function will overwrite anything contained within a file, if a file of the same name already exists in the working directory.

Practice – writing to a file in Python

Part 1:

- Open the file you created in the last practice and ready it for being written to.
- Write a string to that file. Note: this will overwrite the old contents.
- Remember to close the file once you are done.

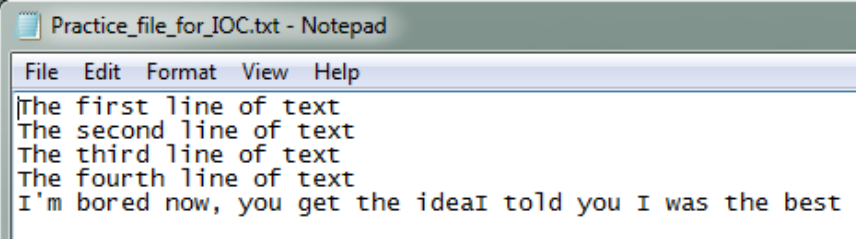
Part 2:

- Create a list of strings.
- Use the `open()` function to create a new `.txt` file and write your list of strings to this file.
- Remember to close the file once you are done.

The append() function

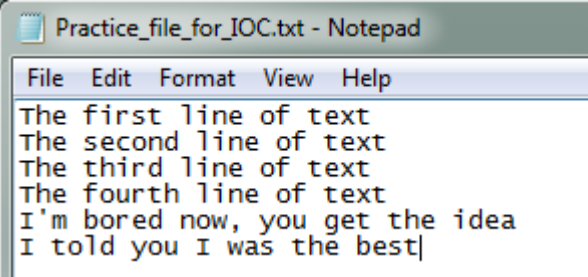
- If you do not want to overwrite a file's contents, you can use the `append()` function.
- To append to an existing file, simply put `'a'` instead of `'r'` or `'w'` in the `open()` when opening a file.

```
practiceFile = open('Practice_file_for_IOC.txt', 'a')
testLine = '\nI told you I was the best'
practiceFile.write(testLine)
```



Practice_file_for_IOC.txt - Notepad

```
File Edit Format View Help
The first line of text
The second line of text
The third line of text
The fourth line of text
I'm bored now, you get the ideaI told you I was the best
```



Practice_file_for_IOC.txt - Notepad

```
File Edit Format View Help
The first line of text
The second line of text
The third line of text
The fourth line of text
I'm bored now, you get the idea
I told you I was the best|
```

Practice – appending to a file in Python

- Open the text file you created in part two of the writing to a file practice, and ready it for appending.
- Define a string object.
- Appending this new string object to the file.
- Remember to close the file once you are done.

A word on import

- To use a package in your code, you must first make it accessible.
- This is one of the features of Python that make it so popular.

```
In: import datetime
    current_time = datetime.datetime.now()
    print(current_time)
```

- There are pre-built Python packages for pretty much everything.

```
In: import antigravity
```

Plotting in Python

- Before creating an plots, it is worth spending sometime familiarising ourselves with the [matplotlib](#) module. It will save a lot of time later on.

Some history....

- `Matplotlib` was originally developed by a neurobiologist in order to emulate aspects of the MATLAB software.
- The pythonic concept of importing is not utilised by MATLAB, and this is why something called `Pylab` exists.
- `Pylab` is a module within the Matplotlib library that was built to mimic the MATLAB style. It only exists in order to bring aspects of `NumPy` and `Matplotlib` into the namespace, thus making for an easier transition for ex-MATLAB users, because they only had to do one import in order to access the necessary functions:

```
from pylab import *
```

- However, using the above command is now considered bad practice, and `Matplotlib` actually advises against using it due to the way in which it creates many opportunities for conflicted name bugs.

Getting started

- Without `Pylab`, we can normally get away with just one canonical import; the top line from the example below.
- We are also going to import `NumPy`, which we are going to use to generate random data for our examples.

```
import matplotlib.pyplot as plt
import numpy as np
```

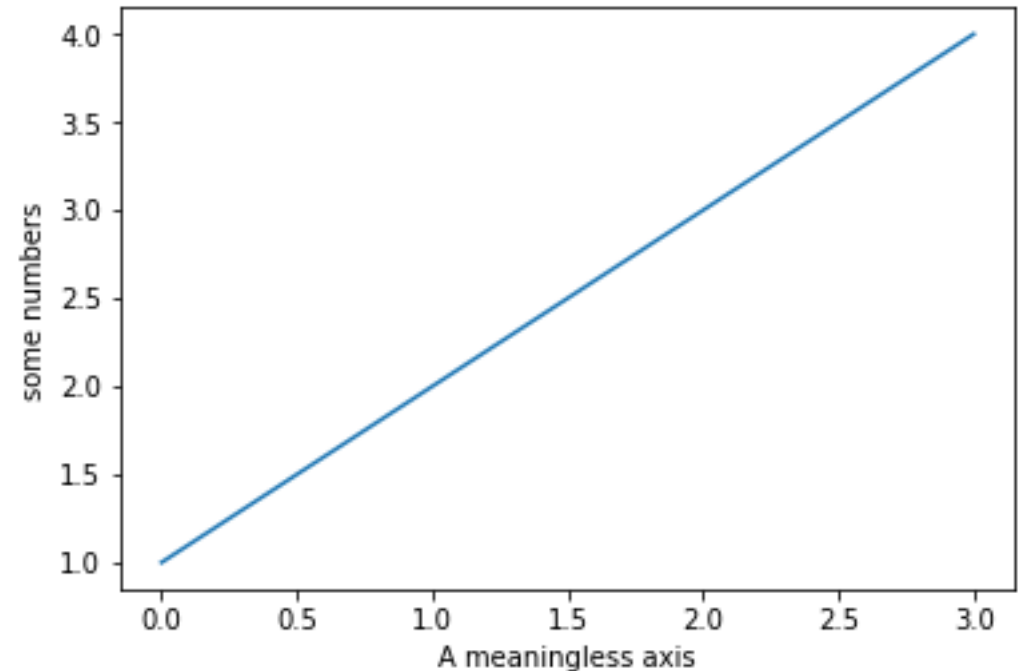
Different graph types

- A simple line graph can be plotted with `plot()`.
- A histogram can be created with `hist()`.
- A bar chart can be created with `bar()`.
- A pie chart can be created with `pie()`.
- A scatter plot can be created with `scatter()`.
- The `table()` function adds a text table to an axes.
- Plus many more....

Our first plot

```
import matplotlib.pyplot as plt
import numpy as np
plt.plot([1,2,3,4])
plt.ylabel('some numbers')
plt.xlabel('A meaningless axis')
plt.show()
```

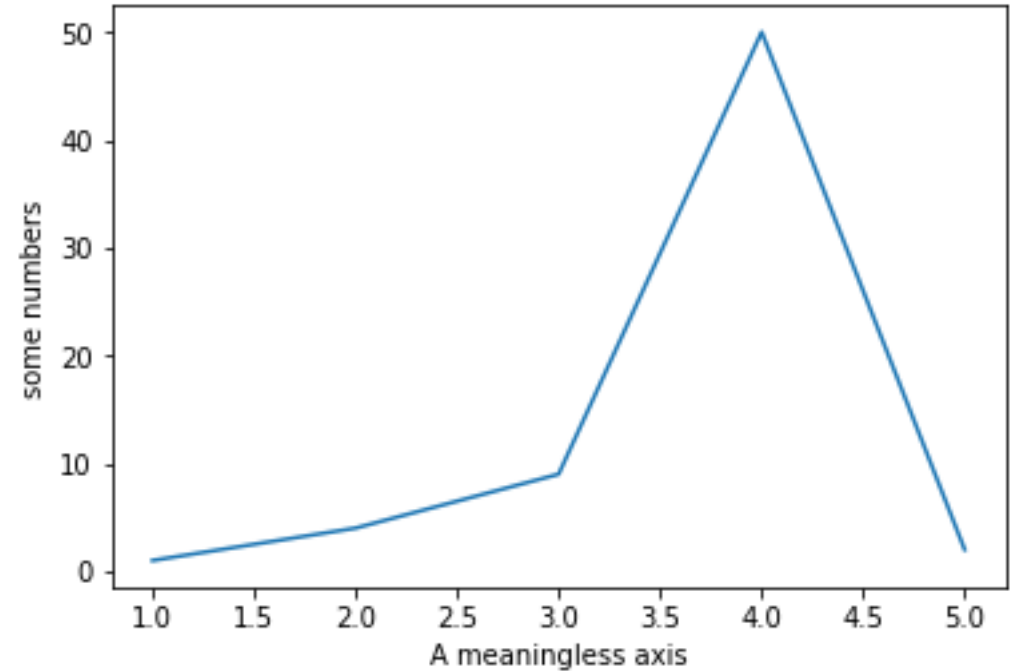
- You may be wondering why the x-axis ranges from 0-3 and the y-axis from 1-4.
- If you provide a single list or array to the `plot()` command, Matplotlib assumes it is a sequence of y values, and automatically generates the x values for you.
- Since python ranges start with 0, the default x vector has the same length as y but starts with 0.
- Hence the x data are [0,1,2,3].



The plot() function

- The `plot()` argument is quite versatile, and will take any arbitrary collection of numbers. For example, if we add an extra entry to the x axis, and replace the last entry in the Y axis and add another entry:

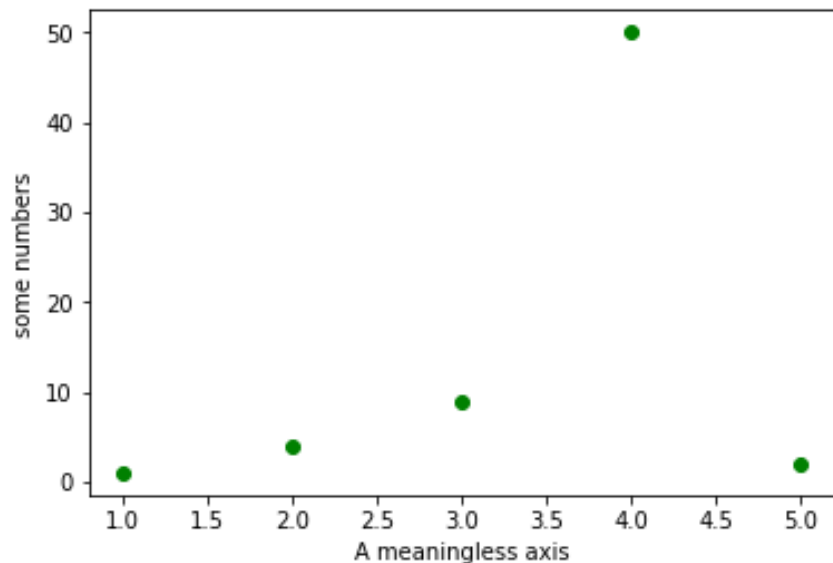
```
import matplotlib.pyplot as plt
import numpy as np
plt.plot([1, 2, 3, 4, 5], [1, 4, 9, 50, 2])
plt.ylabel('some numbers')
plt.xlabel('A meaningless axis')
plt.show()
```



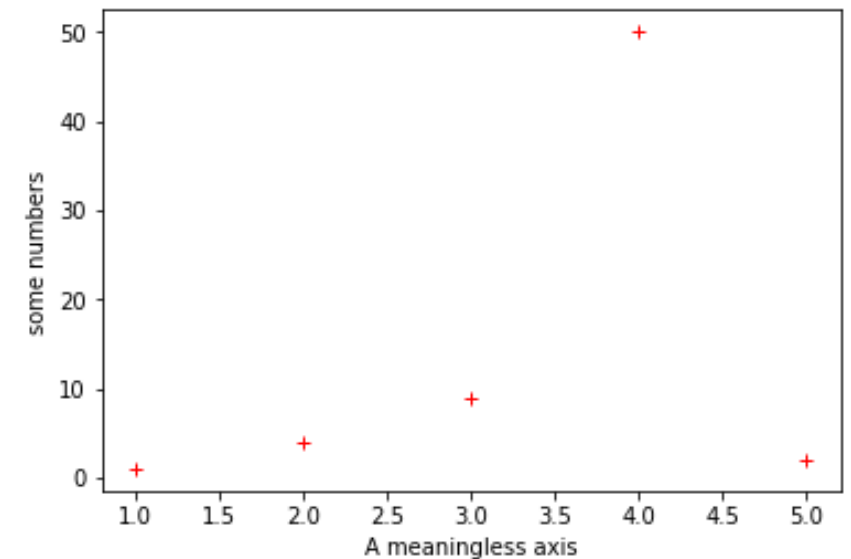
The plot() function

- The `plot()` function has an optional third argument that specifies the appearance of the data points.
- The default is `b-`, which is the blue solid line seen in the last two examples. The full list of styles can be found in the documentation for the `plot()` on the Matplotlib page

```
plt.plot([1, 2, 3, 4, 5], [1, 4, 9, 50, 2], 'go')
```



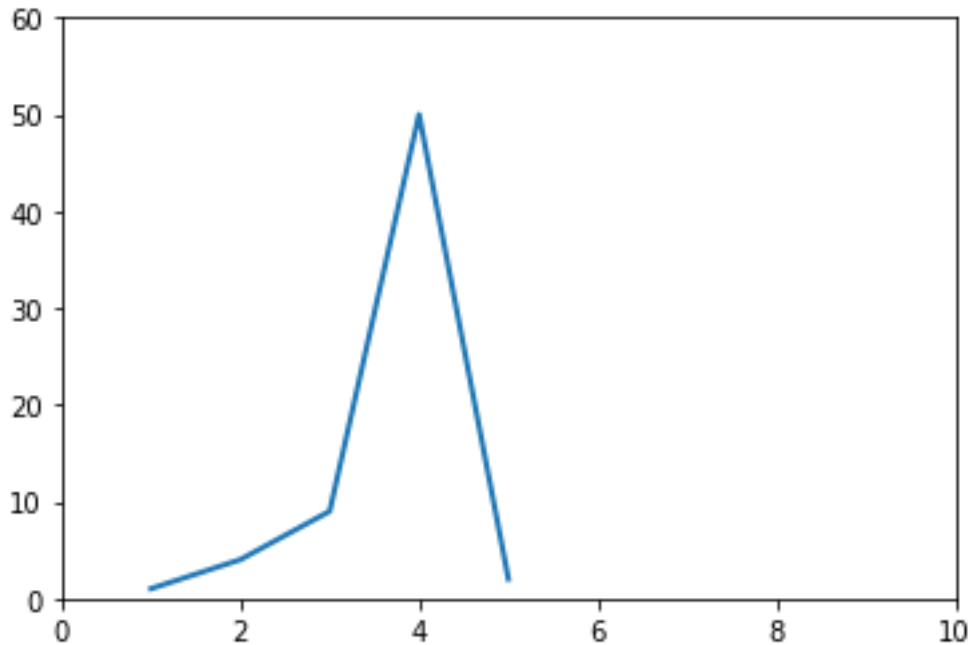
```
plt.plot([1, 2, 3, 4, 5], [1, 4, 9, 50, 2], 'r+')
```



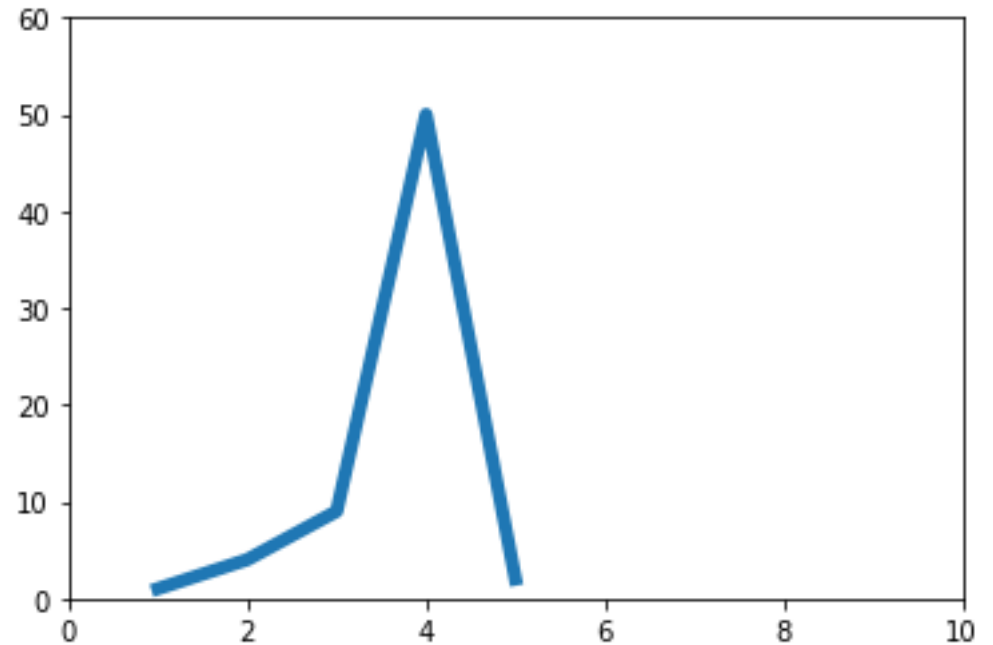
The plot() function

- You can quite easily alter the properties of the line with the `plot()` function.

```
import matplotlib.pyplot as plt
import numpy as np
plt.plot([1, 2, 3, 4, 5], [1, 4, 9, 50, 2], '-', linewidth=2.0)
plt.axis([0, 10, 0, 60])
plt.show()
```



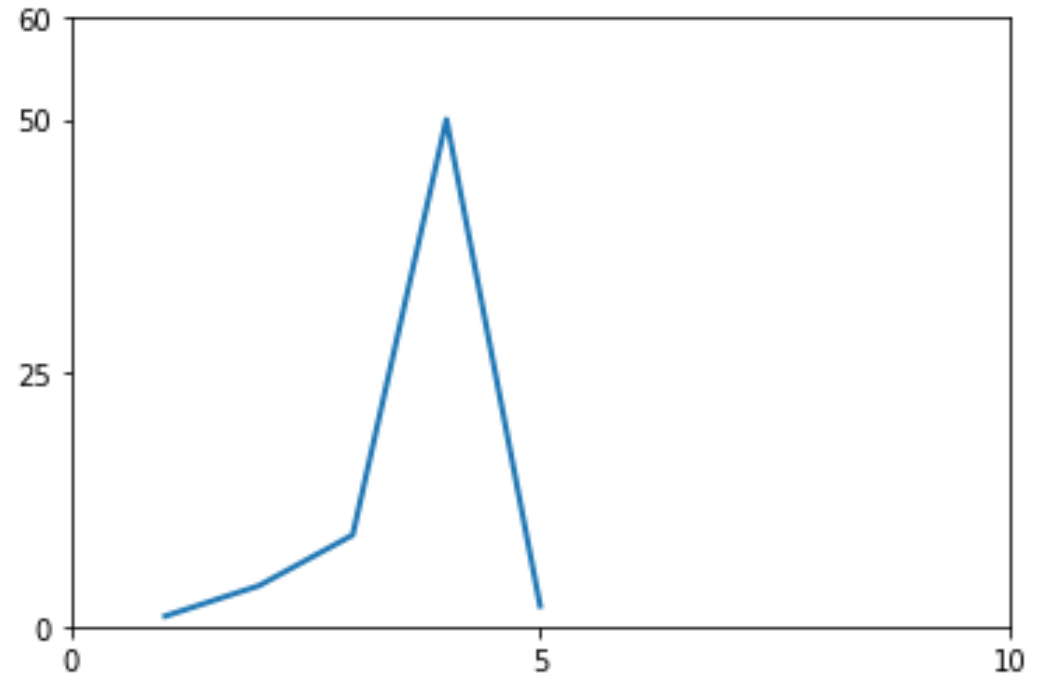
```
import matplotlib.pyplot as plt
import numpy as np
plt.plot([1, 2, 3, 4, 5], [1, 4, 9, 50, 2], '-', linewidth=5.0)
plt.axis([0, 10, 0, 60])
plt.show()
```



Altering tick labels

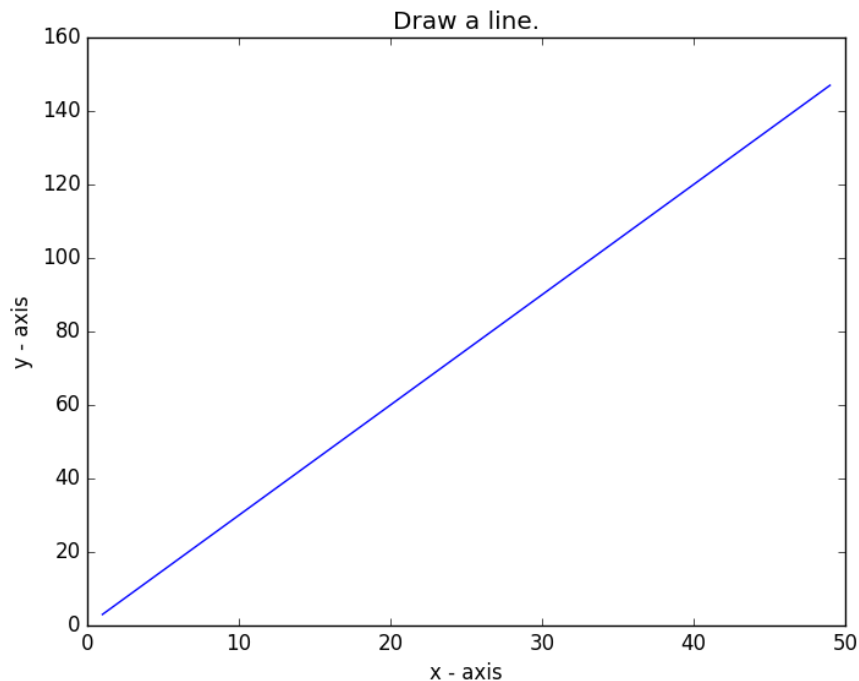
- The `plt.xticks()` and `plt.yticks()` allows you to manually alter the ticks on the x-axis and y-axis respectively.
- Note that the tick values have to be contained within a list object.

```
import matplotlib.pyplot as plt
import numpy as np
plt.plot([1, 2, 3, 4, 5], [1, 4, 9, 50, 2], '--', linewidth=2.0)
plt.axis([0, 10, 0, 60])
plt.xticks([0, 5, 10])
plt.yticks([0, 25, 50, 60])
plt.show()
```



Practice - Basic line graph

Let's write a Python program to draw a line graph with suitable labels for the x-axis and y-axis. Include a title.

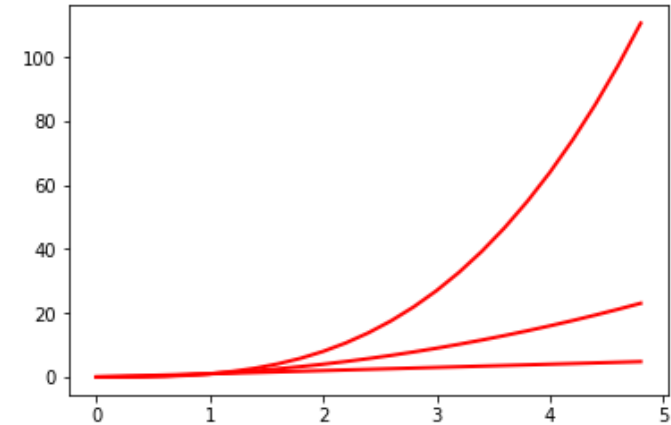


```
import matplotlib.pyplot as plt
X = range(1, 50)
Y = [value * 3 for value in X]
print("Values of X:")
print(range(1,50))
print("Values of Y (thrice of X):")
print(Y)
# Plot lines and/or markers to the Axes.
plt.plot(X, Y)
# Set the x axis label of the current axis.
plt.xlabel('x - axis')
# Set the y axis label of the current axis.
plt.ylabel('y - axis')
# Set a title
plt.title('Draw a line.')
# Display the figure.
plt.show()
```

The setp() function

- The `setp()` allows you to set multiple properties for a list of lines, if you want all the lines to be matching.

```
import numpy as np
import matplotlib.pyplot as plt
# evenly sampled time at 200ms intervals
t = np.arange(0., 5., 0.2)
# red dashes, blue squares and green triangles
lines = plt.plot(t, t, 'b-', t, t**2, 'r-', t, t**3, 'g-', linewidth=2.0)
plt.setp(lines, color='r', linewidth=2.0)
# or MATLAB style string value pairs
plt.setp(lines, 'color', 'r', 'linewidth', 2.0)
plt.show()
```



- You can use the `setp()` function along with either the `line` or `lines` function in order to get a list of settable line properties.

```
lines = plt.plot(t, t, 'b-', t, t**2, 'r-', t, t**3, 'g-', linewidth=2.0)
plt.setp(lines)
```

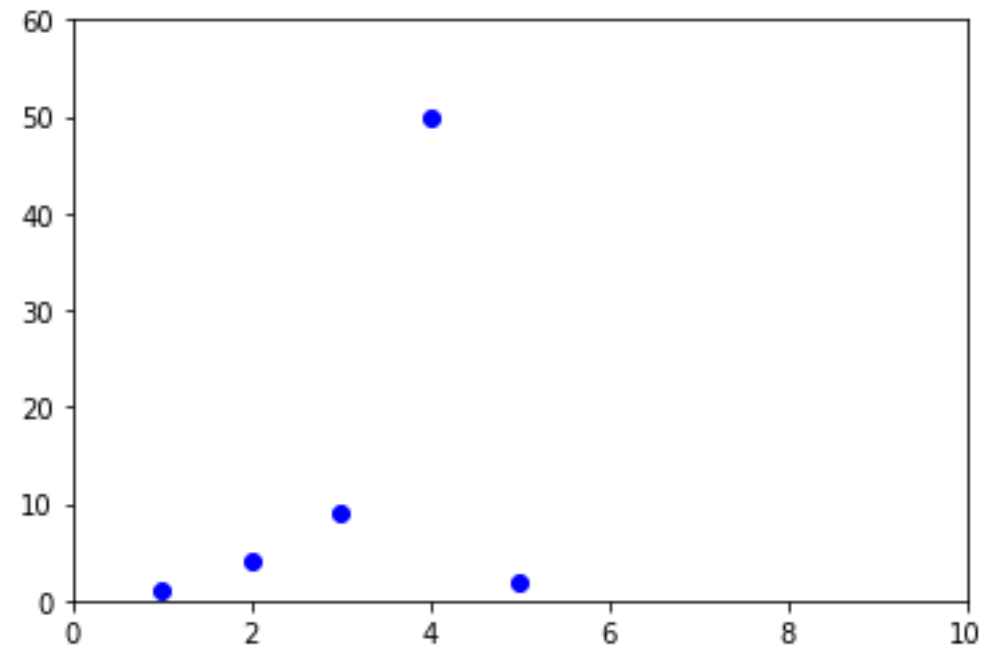
```
In [72]: runfile('///isad.isadroot.ex.ac.uk/UOE/User/Desktop/IoC_plottin
Desktop')
agg_filter: a filter function, which takes a (m, n, 3) float array an
alpha: float (0.0 transparent through 1.0 opaque)
animated: bool
antialiased or aa: bool
clip_box: a `.Bbox` instance
clip_on: bool
clip_path: [(~matplotlib.path.Path`, `.Transform`) | `.Patch` | None
color or c: any matplotlib color
contains: a callable function
```

The axis() function

- The `axis()` function allows us to specify the range of the axis.
- It requires a list that contains the following:

[The min x-axis value, the max x-axis value, the min y-axis, the max y-axis value]

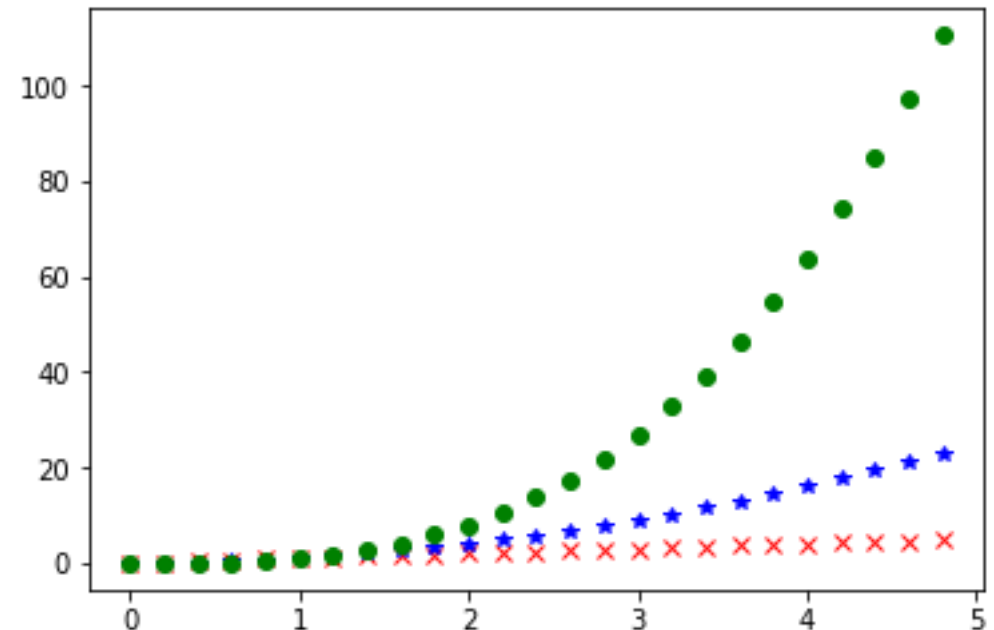
```
import matplotlib.pyplot as plt
import numpy as np
plt.plot([1, 2, 3, 4, 5], [1, 4, 9, 50, 2], 'bo')
plt.axis([0, 10, 0, 60])
plt.show()
```



Matplotlib and NumPy arrays

- Normally when working with numerical data, you'll be using NumPy arrays.
- This is still straight forward to do in Matplotlib; in fact all sequences are converted into NumPy arrays internally anyway.

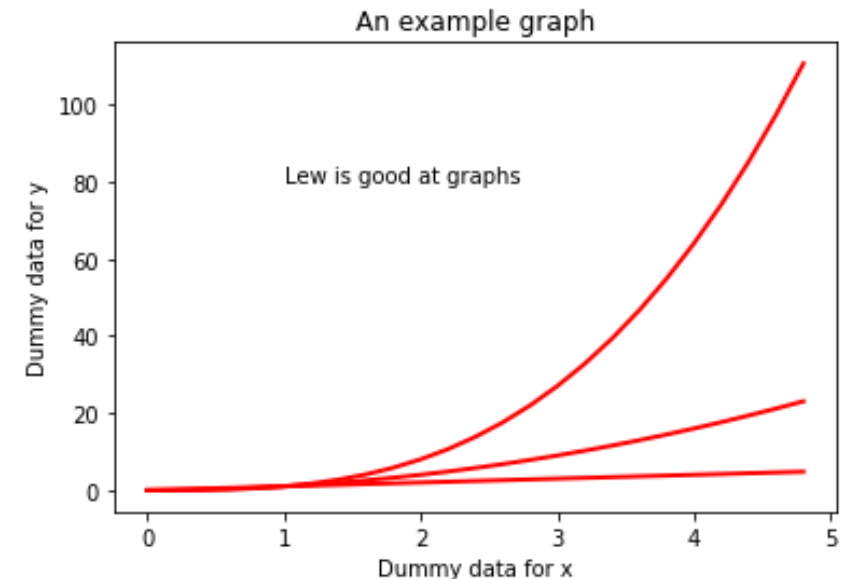
```
import numpy as np
import matplotlib.pyplot as plt
# evenly sampled time at 200ms intervals
t = np.arange(0., 5., 0.2)
# red dashes, blue squares and green triangles
plt.plot(t, t, 'rx', t, t**2, 'b*', t, t**3, 'go')
plt.show()
```



Working with text

- There are a number of different ways in which to add text to your graph:
 - `title()` = Adds a title to your graph, takes a string as an argument
 - `xlabel()` = Add a title to the x-axis, also takes a string as an argument
 - `ylabel()` = Same as `xlabel()`
 - `text()` = Can be used to add text to an arbitrary location on your graph. Requires the following arguments:
`text(x-axis location, y-axis location, the string of text to be added)`
- Matplotlib uses TeX equation expressions. So, as an example, if you wanted to put $\sigma_i = 15$ in one of the text blocks, you would write `plt.title(r'$\sigma_i=15$')`.

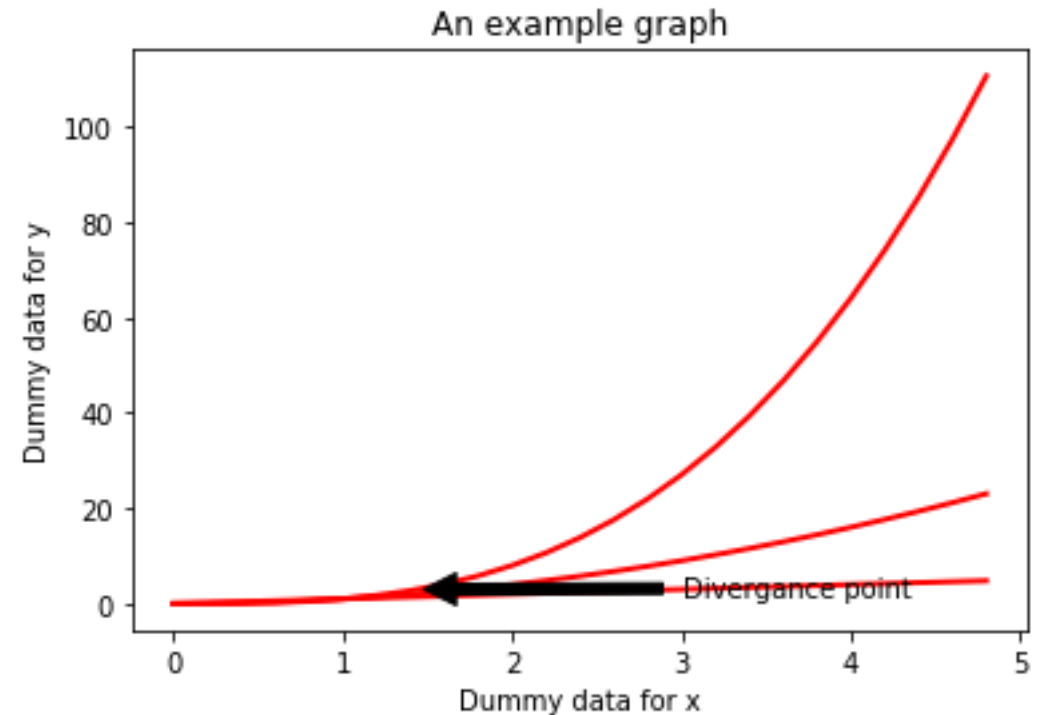
```
import numpy as np
import matplotlib.pyplot as plt
# evenly sampled time at 200ms intervals
t = np.arange(0., 5., 0.2)
# red dashes, blue squares and green triangles
lines = plt.plot(t, t, 'b-', t, t**2, 'r-', t, t**3, 'g-', linewidth=2.0)
plt.setp(lines, color='r', linewidth=2.0)
plt.xlabel('Dummy data for x')
plt.ylabel('Dummy data for y')
plt.title('An example graph')
plt.text(1, 80, 'Lew is good at graphs')
plt.setp(lines, 'color', 'r', 'linewidth', 2.0)
plt.show()
```



Annotating data points

- The `annotate()` function allows you to easily annotate data points or specific area on a graph.

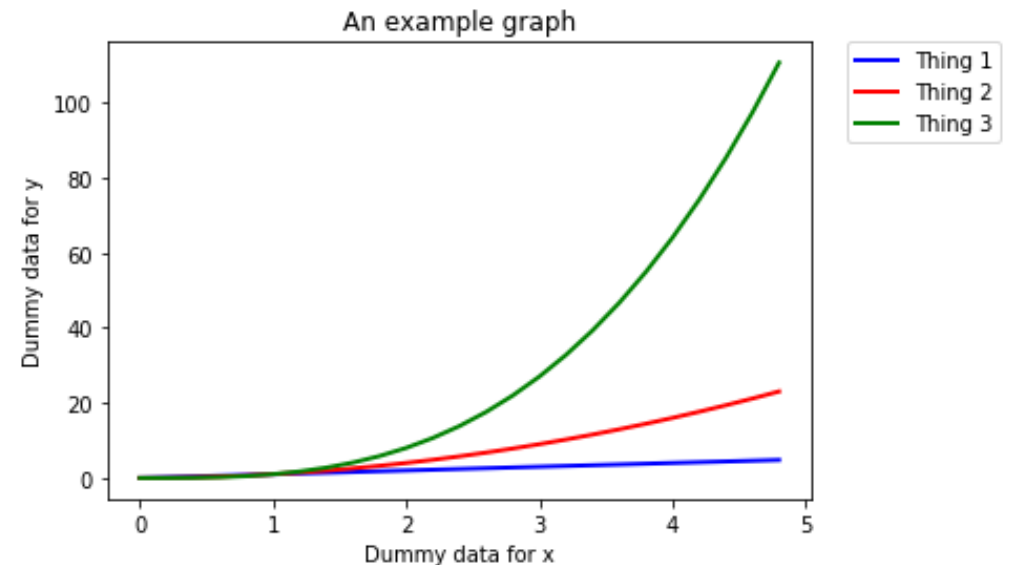
```
t = np.arange(0., 5., 0.2)
# red dashes, blue squares and green triangles
lines = plt.plot(t, t, 'b-', t, t**2, 'r-', t, t**3, 'g-', linewidth=2.0)
plt.setp(lines, color='r', linewidth=2.0)
plt.xlabel('Dummay data for x')
plt.ylabel('Dummy data for y')
plt.title('An example graph')
plt.annotate('Divergance point', xy=(1.4, 3), xytext=(3, 1.5),
            arrowprops=dict(facecolor='black', shrink=0.05),
            )
plt.setp(lines, 'color', 'r', 'linewidth', 2.0)
plt.show()
```



Legends

- The location of a legend is specified by the `loc` command. There are a number of in-built locations that can be altered by replacing the number. The Matplotlib website has a list of all locations in the documentation page for `location()`.
- You can then use the `bbox_to_anchor()` function to manually place the legend, or when used with `loc`, to make slight alterations to the placement.

```
import numpy as np
import matplotlib.pyplot as plt
# evenly sampled time at 200ms intervals
t = np.arange(0., 5., 0.2)
# red dashes, blue squares and green triangles
lines = plt.plot(t, t, 'b-', linewidth=2.0, label='Thing 1')
lines = plt.plot(t, t**2, 'r-', linewidth=2.0, label='Thing 2')
lines = plt.plot(t, t**3, 'g-', linewidth=2.0, label='Thing 3')
plt.xlabel('Dummy data for x')
plt.ylabel('Dummy data for y')
plt.title('An example graph')
plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
plt.show()
```



Saving a figure as a file

- The `plt.savefig()` allows you to save your plot as a file.
- It takes a string as an argument, which will be the name of the file. You must remember to state which file type you want the figure saved as; i.e. png or jpeg.
- Make sure you put the `plt.savefig()` before the `plt.show()` function. Otherwise, the file will be a blank file.

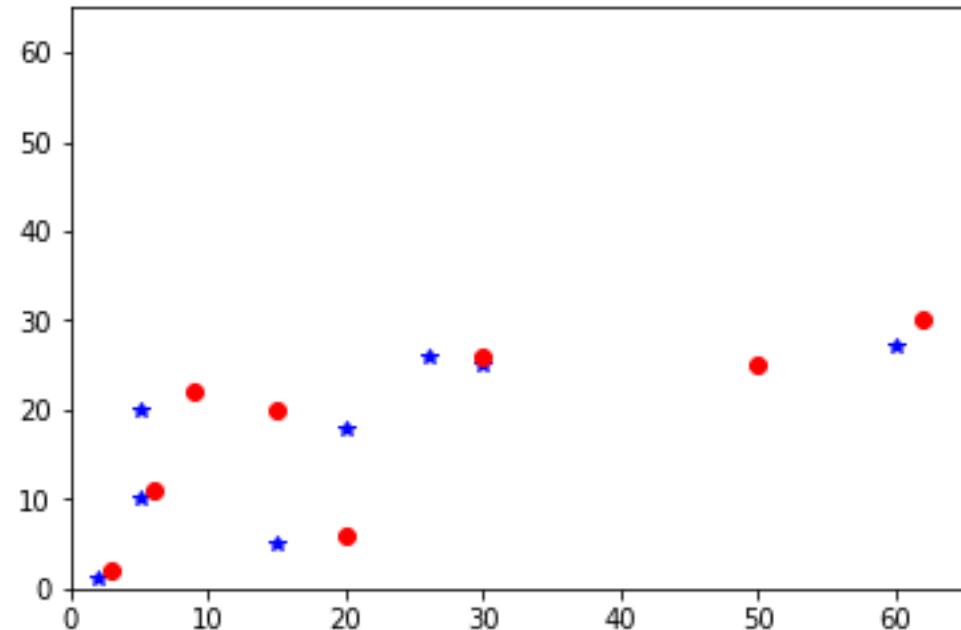
```
t = np.arange(0., 5., 0.2)
# red dashes, blue squares and green triangles
lines = plt.plot(t, t, 'b-', t, t**2, 'r-', t, t**3, 'g-', linewidth=2.0)
plt.setp(lines, color='r', linewidth=2.0)
plt.xlabel('Dummy data for x')
plt.ylabel('Dummy data for y')
plt.title('An example graph')
plt.text(1, 80, 'Lew is good at graphs')
plt.setp(lines, 'color', 'r', 'linewidth', 2.0)
plt.savefig('test.png')
plt.show()
```

Scatter plot exercise

Let's write a Python program to plot quantities which have an x and y position; a scatter graph.

```
import numpy as np
import pylab as pl

# Make an array of x values
x1 = [2, 15, 5, 20, 5, 30, 26, 60]
# Make an array of y values for each x value
y1 = [1, 5, 10, 18, 20, 25, 26, 27]
# Make an array of x values
x2 = [3, 20, 6, 15, 9, 30, 50, 62]
# Make an array of y values for each x value
y2 = [2, 6, 11, 20, 22, 26, 25, 30]
# set new axes limits
pl.axis([0, 65, 0, 65])
# use pylab to plot x and y as red circles
pl.plot(x1, y1, 'b*', x2, y2, 'ro')
# show the plot on the screen
pl.show()
```



Debugging

- Debugging is in fundamental aspect of coding, and you will probably spend more time debugging than actually writing code.
- EVERYONE has to debug, it is nothing to be ashamed of.
- In fact, you should be particularly concerned if you do write a programme that does not display any obvious errors, as it likely means that you are just unaware of them.
- There are a number of debugging programmes available to coders. However, debugging the most common issues that you'll encounter when developing programmes can be done by following a few key principles.
- However, always remember that sometimes fixing a bug can create new bugs.



Print everything

- When debugging, the most important function at your disposal is the `print` command. Every coder uses this as a debugging tool, regardless of their amount of experience.
- You should have some sense as to what every line of code you have written does. If not, print those lines out. You will then be able to see how the values of variables are changing as the programme runs through.
- Even if you think you know what each line does, it is still recommended that you print out certain lines as often this can aid you in realising errors that you may have overlooked.

Print examples

Did this chunk of code run?

```
i = 0
while i < 6:
    string = '*'
    length = i * 3
    for j in range(length):
        string = string + '*'
    i += 1
print("Got here")
```

Yes, it did.

```
In [135]: runfile('//is
Got here
```

I want the value of variable to be 10 upon completion of the for loop. Did the for loop work correctly?

```
variable = 1
for i in range(10):
    variable += 1
print("variable", variable)
```

No.

```
In [133]: runfile('//
variable: 11
```

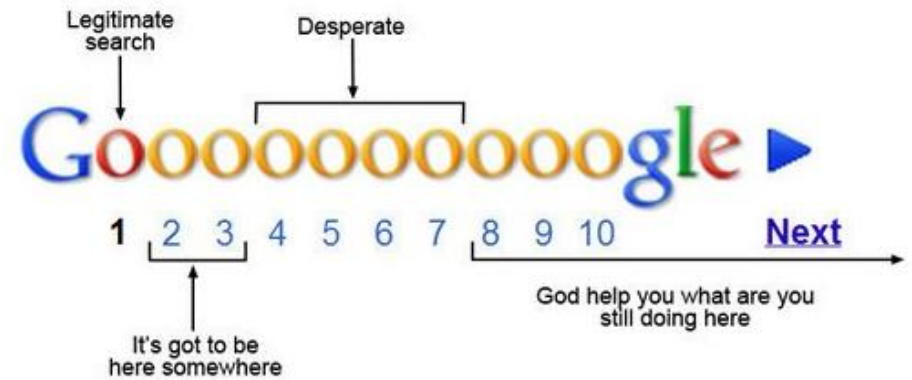
Run your code when you make changes

- Do not sit down and code for a hour or so without running the code you are writing. Chances are, you will never get to the bottom of all of the errors that your programme reports when it runs.
- Instead, you should run your script every few minutes. It is not possible to run your code too many times.
- Remember, the more code you write or edit between test runs, the more places you are going to have to go back an investigate when your code hits an error.

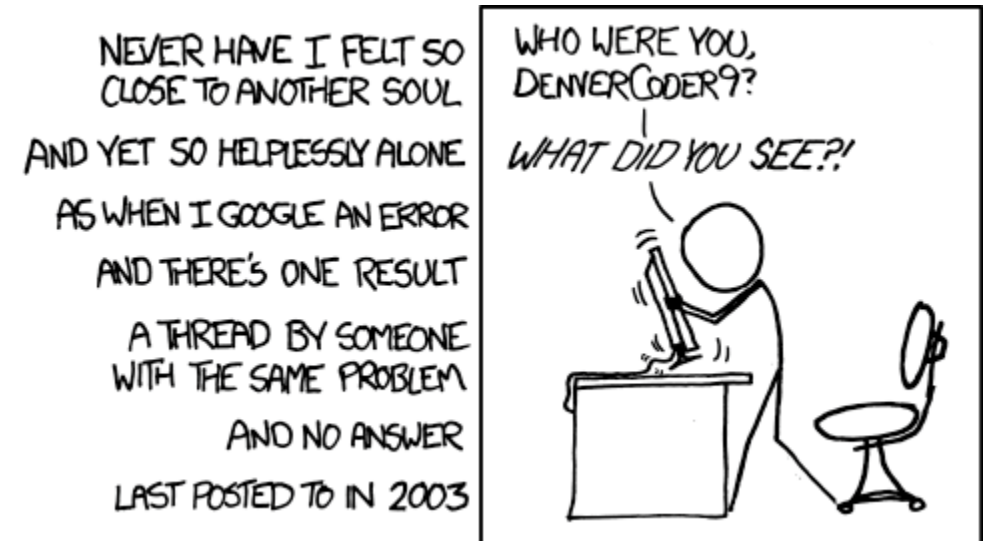
Read your error messages

- Do not be disheartened when you get an error message. More often than not, you'll realise what the error is as soon as you read the message; i.e. the for loop doesn't work on a list because the list is empty.
- This is particularly the case with Python, which provides you with error messages in 'clear English' compared to the cryptic messages given by offered by other languages.
- At the very least, the error message will let you know which lines is experiencing the error. However, this may not be the line causing the error. Still, this offers a good starting point for your bug search.

Google the error message



- If you cannot work out the cause of an error message, google the error code and description.
- This can sometimes be a bit of a hit-or-miss, depending on the nature of the error.
- If your error is fairly specific, then there will nearly always be a webpage where someone has already asked for help with an error that is either identical or very similar to the one you are experiencing; stackoverflow.com is the most common page you'll come across in this scenario.
- Do make sure that you read the description of the problem carefully to ensure that the problem is the same as the one you are dealing with. Then read the first two or three replies to see if page contains a workable solution.



Comment out code

- You can often comment out bits of code that are not related to the chunk of code that contains the error.
- This will obviously make the code run faster and might make it easier to isolate the error.

Binary searches

- This method draws upon a lot of the methods we have already covered.
- Here, you want to break the code into chunks; normally two chunks, hence this method's name.
- You then isolate which chunk of code the error is in.
- After which, you take the chunk of code in question, and divide that up, and work out which of these new chunks contains the error.
- So on until you've isolate the cause of the error.

Walk away

- If you have been trying to fix an error for a prolonged period of time, 30 minutes or so, get up and walk away from the screen and do something else for a while.
- Often the answer to your issue will present itself upon your return to the computer, as if by magic.

Phrase your problem as a question

- Many software developers have been trained to phrase their problem as a question.
- The idea here is that phrasing your issue in this manner often helps you to realise the cause of the problem.
- This often works!



Walter

Ask someone

- If all else fails, do not hesitate to ask a colleague or friend who is a coder and maybe familiar with the language for help.
- They may not even need to be a specialist, sometimes a fresh pair of eyes belonging to someone who is not invested in the project is more efficient at helping you work out your issue than spending hours trying to solve the issue on your own or getting lost the internet trying to find a solution.

Any questions?

Useful resources

- There are two great online resources for learning this language through practical examples. These are the Code Academy (<https://www.codecademy.com/catalog/subject/web-development>) and Data Camp (https://www.datacamp.com/?utm_source=adwords_ppc&utm_campaignid=805200711&utm_adgroupid=39268379982&utm_device=c&utm_keyword=data%20camp&utm_matchtype=e&utm_network=g&utm_adpostion=1t1&utm_creative=230953641482&utm_targetid=kw-298095775602&utm_loc_interest_ms=&utm_loc_physical_ms=1006707&gclid=EAlaIQobChMI3o2iqtbV2wIVTkPTCh2QRA19EAAAYASAAEgLZdPD_BwE).