Introduction to Python

Mark Slater

UNIVERSITY OF
BIRMINGHAM

Useful Links

- Main python documentation:

  https://docs.python.org/2/

- Good place to ask questions:

  http://stackoverflow.com/

- Google is very good as well!

- My contact details:

  Mark Slater <mslater(AT)cern.ch>, Physics West 317

Course Info and Aims

- The aim of this part of the course is to give some basic experience with Python so you can start writing scripts to help with various types of analysis

- It will cover Python programming from the very basics to using functions, file I/O and external modules

- It will demonstrate these techniques using the example of a cipher program

- At the end there will also be an additional, more advanced exercise about plotting and fitting if we have time!

Preliminary Setup

- There is a skeleton structure for the course on github here:

  https://github.com/drmarkwslater/CCBPythonCourseSep2017

- With your newly acquired git skills, can you:

  1. Fork this repo on github

  2. 'git clone' your copy of the repo to your VM (or wherever you're working)

  3. 'cd' into the 'intro' folder and you're ready to go!

Overview

What is Python?

- Python is a general purpose, extensible object oriented programming language widely used across in the software industry and beyond

- It is not compiled to machine code (like C/C++) allowing it to be a highly dynamic language at runtime and therefore providing you with a lot of control over how code is executed

- This does mean there is a loss in performance but it is unlikely you will hit these issues in your work
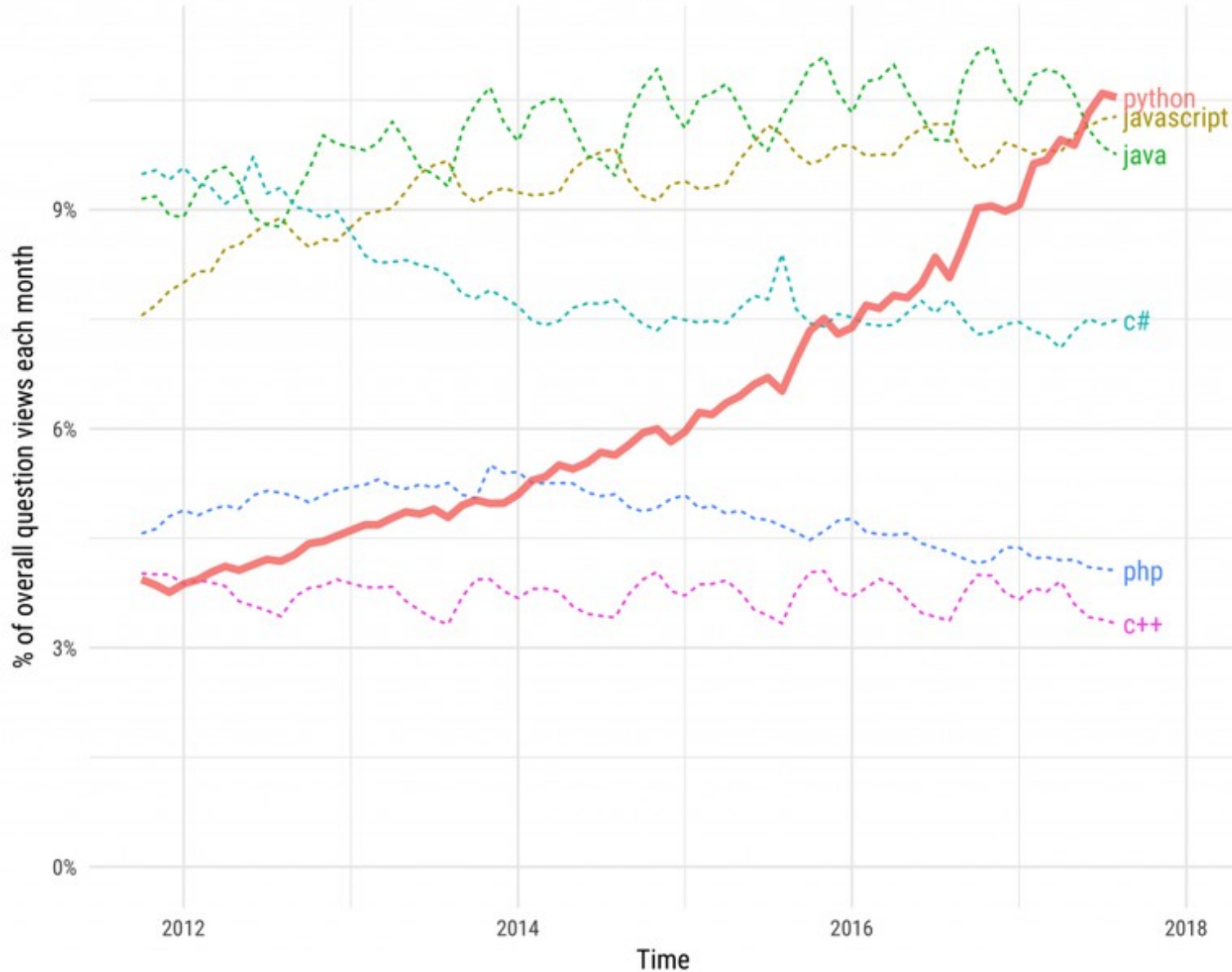
Why Python?

- Though comprised of fairly basic syntax and conventions, it is very powerful. However, it is also easy to learn and produce results with quickly

- The main benefits of Python are readability and ease of use combined with a very large (and growing) set of external modules that will cover a lot of what you need to do (e.g. plotting, statistical analysis, etc.)

- As of writing, it's one of the most used (and fastest growing) languages:

Why Python?

Growth of major programming languages

Based on Stack Overflow question views in World Bank high-income countries

% of overall question views each month

- python
- javascript
- java
- c#
- php
- c++

2012   2014   2016   2018

Time

1. Writing and Running Python Code

Writing and Running Code

There are two ways to run python code: Through the interpreter or as a script

```
>>> f = open('my_data.vcf')
>>> for ln in f.readlines():
...     if ln.find('0/1') > -1:
...         print 'Found 0/1'

>>> f.close()
```

`slatermw$ python`

```
f = open('my_data.vcf')
for ln in f.readlines():
    if ln.find('TAG') > -1:
        print 'Found TAG'

f.close()
```

.py File

`slatermw$ python my_script.py`

The Hello World Program (Ex. 1)

To make a start with python, we will create the ubiquitous 'Hello World' program:

```python
def myprint( ):
    # Function to print hello
    print 'Hello World'

myprint()
```

1. After cd'ing to the 'intro' dir...

2. Run the command 'python'

3. Type in the code to the right (note the spaces/tabs!)

4. Use 'Ctrl+D' to exit

5. Edit the file 'hello.py' – I would suggest using 'gedit'!

6. Type the code again in to the file

7. Back in your terminal, type 'python hello.py'

8. NOTE: You have to be careful about the indentation – either 'Tab' or 3-4 spaces. Be consistent as well!

Basic Syntax of a Python Program

Before we start looking in more detail at Python coding, we will just cover the basic syntax of the program you've just written

A new 'block' of code is usually denoted by a colon (':'). These can be after function definitions, conditionals or loops

A function definition - see later

```python
def myprint( ):
    # Function to print hello
    print 'Hello World'

myprint()
```

Indentations indicate 'blocks' of code – in this case, the body of a function. Python is very strict on the indentation and you MUST be consistent!

It is good practise to add comments to your code – these are ignored by the interpreter but help you explain what you're trying to do, both to other people and yourself a few months on!

2. Types, Objects, Values and Variables

What are Types, Objects, Values and Variables?

Python (and many other languages) have a defined way of holding and organising data within a computer's memory

The appropriate terms are:

- → Types – How to interpret data in a memory location ('object') and what operations can be performed by it
- → Object – Defined area of memory that holds the data ('values') associated with a type
- → Value – Actual data/bits in memory interpreted by the 'type'
- → Variable – A flag or name of an area of memory ('object')

These may seem a bit abstract but it is important to know the difference as we progress to avoid confusion later!
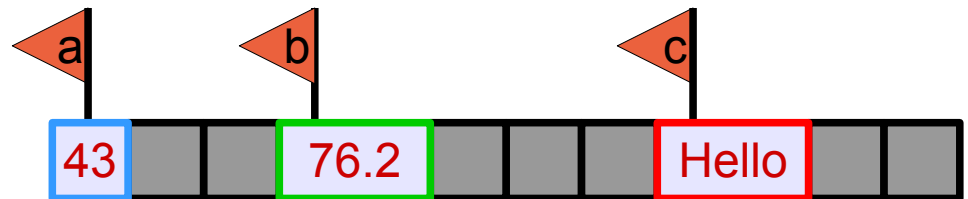
What are Types, Objects, Values and Variables?

In terms of a laptop:

➔ Type – Laptop
➔ Object – The actual physical laptop
➔ Value – The bits and pieces that make up the laptop ('MacBook')
➔ Variable – MARK'S Laptop

```
a = 5
b = 76.2
c = 'Hello'
```


The given code initialises 3 variables – an int, float and string

The variable names are labels that are attached to the objects in memory. The values are the contents of the memory and the type is how python interprets the memory

Basic Types and Initialising Variables

We will start by introducing the basic types that are available in Python and showing how to create them

The built-in basic types are (excluding Lists and Dictionaries – see later!):

> ➜ A boolean – bool (True/False)
>
> ➜ Integer number – int (1, 2, 3, etc.)
>
> ➜ Floating point number – float/double (1.2, 5.7, etc)
>
> ➜ String – an array or list of characters ('hello', 'world', etc.)

To declare a variable (a named object of this type), all you need to do is provide a variable name and initialise it to whatever starting value you want:

```python
a = 5
b = 76.2
c = 'Hello'
```

This will create an object of the requested type (i.e. assign the appropriate memory) and initialise it with the value

It also assigns this object with the given variable name so you can refer to it later

Getting Experience with Variables (Ex. 2)

- Now either edit the code in your 'Hello World' script or directly use the python interpreter to do the following:

    → Create, modify (e.g. add 1 to it) and output an integer variable

    → Create a decimal ('float') variable and output this

    → Create a string variable and print it

- Note that you can use the same variable name for each of the above variables but it's good practise not to reuse variable names for different objects in the same area of code

- As you've probably guessed, to output things to the screen, use the 'print' statement:

```
print 'Hello World'
```

3. Operators

Operators (1)

- Through the introduction to variable creation, you have also now met the idea of operators

- Operators are symbols that perform a specific operation on one or more objects. A subset of these are the arithmetic operations you're familiar with:

    - Multiplication: a * b

    - Addition: a + b

    - Subtraction: a - b

    - Division: a / b

- For example, 'a + b' is the addition operator being applied to the objects 'a' and 'b'

- Also note that operators in Python are nothing 'special' – they are essentially shorthand calling other bits of code

Operators (2)

As well as these arithmetic operators, there are several more language specific ones:

- ➔ Assignment: a = b
- ➔ Addition/Subtraction and assignment: a += b, a -= b
- ➔ Exponent: a ** b (e.g. 10**3 = 1000)
- ➔ Modulus: % (e.g. 9 % 4 = 1)
- ➔ Floor Division: a // b (e.g. 9 // 4 = 2)
- ➔ Subscript/Array: []

What each operator does is entirely a property of the type(s) they are operating on though generally it is obvious what effect they will have
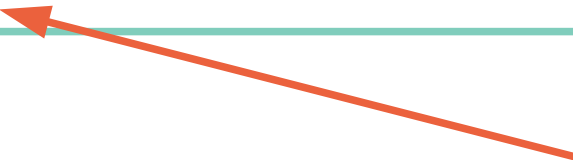
Note that as in Maths, operators have precedence and you should use brackets appropriately!

Single Character Access

As another example of simple operator use, we will look at the 'string' type

As mentioned, this type can be considered an array or list of characters (chars) and so to access a single character from that array, you use the '[]' operator

```python
# Assign a variable for the string
a = 'Hello World'

# Print the whole string
print a

# output 'o'
print a[4]
```

The array operator references the zero-indexed list of letters in string and returns the appropriate element

Getting Experience with Variables (Ex. 3)

- Again, either edit the code in your 'Hello World' script or directly use the python interpreter to do the following:
    - ➜ Create float and integer variables
    - ➜ Output the product of these
    - ➜ See what happens when you divide a float and an int and then two ints
    - ➜ Create a string variable
    - ➜ Output various letters from your string variable

4. Program Flow

Program Flow

- It would be difficult to do much with a language if a program was just executed from top to bottom and you couldn't control what parts of the code were executed

- There are a number of ways provided to gain this control over the program:

  - → Conditionals
  - → Loops
  - → Functions

- We'll go over each one in turn and give you a chance to use them as we start to develop the cipher code

Program Flow – Scope and Indentation

- Before we go into the main ways of controlling program flow, it's important to understand the idea of scope. This refers to 'blocks of code' that, in Python, are indicated by levels of indentation

- Generally, all program flow statements are terminated by a colon and followed by a code block that is indented relative to where it is defined

- This indentation can be tabs or a certain number of spaces (3-4 usually) but it MUST be consistent!

- An error you may see quite often is an 'Indentation Error' – this is when Python encountered more/less indentation without an appropriate Program Flow Control statement

Program Flow – Scope and Indentation

```python
def my_func():
    # Start of a function is indented
    # from where it was declared
    a = 10
    for i in range(0, a):
        # loop needs to be indented
        print i

    print 'all done!'

# 'Global' namespace - no indentation!
my_func()
```

All Program Flow statements (a function in this case) requires the associated code to be indented from where it is declared

As the 'for' loop is already indented in the function, it's associated code must be indented a second time

If code is in the 'global' namespace, it doesn't have any indentation

5. Conditionals

Program Flow - Conditionals

- Conditionals allows different code blocks to be executed based on the outcome of a simple test. It uses a number of additional operators, including:

  - ➜ Comparison: ==

  - ➜ Greater/Less than: > / <

  - ➜ Greater/Less than or equal to: >= / <=

  - ➜ Not equal to: !=

  - ➜ 'not' to negate a boolean value

  - ➜ 'in' to test if an object is present in a list/string – see later!

- Examples of this are shown on the next slide, but the main points are:

  - ➜ The 'if', 'elif' and 'else' keywords are used

  - ➜ The conditions must evaluate to True or False

  - ➜ The code must be indented

Program Flow - Conditionals

```python
a = 1
b = 2
mybool = True

if a == b:
    # a equals b so do this
    print 'a equals b'
elif mybool:
    # This code will actually run as mybool is True
    # equivalent to 'if mybool == True'
    print 'mybool was True'
elif not mybool:
    # 'not' negates a boolean
    # i.e. this will be called if mybool is False
    print 'mybool was False'
else:
    # if none of these apply, do this
    print 'none of the conditions were met'
```

6. Loops

Program Flow – Loops

- Loops are very useful for re-using code – a very important practise in all code development, not just Python. Loops allow you to repeat a block of code a set number of times, over a list of objects or until a condition is met

- The first type of loop we will look at is the 'while' loop. This will continue to run iterations of the loop code until the given condition evaluates to False

```python
i = 1
while i < 5:
    print i
    i += 1
```

Some useful points to note are:

→ Like conditionals, the code to be executed must be indented

→ You must declare any loop variables before the 'while' statement

→ The evaluation of the condition is done at the beginning of each loop

→ The loop will continue until the condition is False, so be careful of infinite loops!

→ Use 'break' to get out of the loop and 'continue' to skip the iteration

Aside: Calling Functions

- Though we will be covering these in more detail later, I will briefly go over calling functions as these will be needed for the next exercise

- When you call a function, you are sending the program off to run another section of code before returning back to you either having done something to object(s) or returned some information

- Functions can either be called with the object as an argument or ON the object itself with the '.' operator:

```python
# Create a string
mystr = 'hello world'

# change this to lower case
mystr = mystr.lower()

# Find out how many letters
print len(mystr)

# find the first instance of a letter
pos = mystr.find('l')

# split the string according to white space
tokens = mystr.split()
```

Transliterate User Input (Ex 4)

- We have now covered enough material to start writing our Caesar Cipher program

- As the Caesar cipher is a classical cipher, we have to impose the following restrictions:

  → Letters must be only one case (we have chosen upper-case)

  → Numbers must be changed to words

  → Any other non-alphanumeric characters need to be removed

- The next exercise is to take some input from the user, change it based on the above restrictions and then print it out

Transliterate User Input (Ex 4)

- To help you with writing the transliteration code, we'll break it down into steps and you should tackle each one in turn – the file to use is in the 'caesar_cipher' directory

- Check each individual step is working and outputting what you expect before moving on to the next

```
# Take each letter from user input and in each case:

#    - Convert to upper case

#    - Change numbers to words

#    - Ignore any other (non-alpha) characters

#    - In each case, add result to a string variable

# print out the new string
```

Use the following code to loop over user input

```
in_string = raw_input('Input: ')
while i < len(in_string):
    # use in_string[i] to access each element
    i = i + 1
```

Use a set of if..elif..else blocks for now

Use Google to find functions that check if a char is alphabetic and shift it to uppercase

7. Lists, Dictionaries, Tuples

Collections

- For the next part of the cipher code we will need to start using arrays or collections of objects

- There are additional basic types in Python designed to deal with these collections (or objects that 'contain' a set of other objects). There are many provided but we'll focus on the most frequently used:

  - ➔ List – a simple list of objects
  - ➔ Dictionary – an associative list of objects (e.g. 1 → 'ONE')
  - ➔ Tuple – An immutable (cannot be changed) list

- The most useful of these are the first two – I include the tuple as you may come across things that look like lists but are created using parentheses ('()') instead of square brackets ('[]') – these are tuples!

Lists

- A list can hold any array of types and can be altered by adding, removing or modifying elements

- A string is very similar to a list and can be manipulated in a very similar way

- You've already met how to access single elements in a string using square brackets '[]' and the same works for a list

- You can also access a subset of elements in a list (or string) using the 'slicing' method

- This is where you specify a range within the brackets using a colon ':' rather than a single number, e.g. mylist[3:7] will return a list containing elements 3, 4, 5 and 6

- If you miss out the first or last index, it will return all the elements from the start or to the end of the list respectively

- You can also specify elements from the end of a list using a negative index, e.g. mylist[-1] returns the last element in the list

Lists

```python
# Create a list of strings
mylist = ['Hello', 'World', 'and', 'everyone', 'in', 'it']

# Take a few slices - Note: zero indexed!
a = mylist[3]
b = mylist[0:2]
c = mylist[-1]

# this will give: everyone ['Hello', 'World'] it
print a, b, c

# Can add lists (and strings) together
d = b + mylist

# How many elements are in the list?
print len(mylist), len(d)

# Add and remove some elements
mylist.append('!')
print mylist

mylist.remove('World')
del mylist[2]
print mylist
```

Create a list of objects using square brackets and a comma separated list

Also use the square brackets to access elements of the list

You can use the 'append' and 'remove' methods to change the contents of the list. There are many other methods to manipulate a list – here's a good start:
https://docs.python.org/2/tutorial/datastructures.html

Dictionaries

- A dictionary is similar to a list but instead of storing objects in order, it associates them with a 'key' which can be referenced to access the object

- As with lists, anything can be stored as value or key

```python
# Create a dictionary
# (and that's TOTALLY my age :)
dict = { 'First Name' : 'Mark', 'Surname': 'Slater', 'Age': 25 }

# access some elements
print dict['First Name']
print dict['Age']

# Extend/Delete elements
dict['School'] = 'EPS'
del dict['Age']

# Some useful functions
print dict.keys()
print dict.values()
```

Use square brackets and a key value to access a value

Create a dictionary using curly brackets, key, value pairs separated by a colon and elements separated by a comma

Add Key/Value pairs to a dictionary by just specifying the new key. Use 'del' to delete a pair.

You can access a list of the keys and values using these functions. More useful functions can be found here: https://docs.python.org/2/tutorial/datastructures.html

Revisiting Program Flow – The 'for' Loop

- Now we've got to grips with collections (lists and dictionaries primarily), we can now introduce the other major type of loop you'll use – the 'for' loop.

- As with the 'while' loop, this loops over a code block but in this case it perform the iteration for each object in the given list, assigning each element to a given variable

```python
for i in [1, 2, 3, 4]:
    print i

for i in range(1, 5):
    print i
```

- Some useful points to note are:

  - → A cycle of the loop will happen with the given variable set to each object in the list

  - → You can use 'break' to terminate a loop and 'continue' to skip to the next iteration

  - → The loop variables are just like normal variables and can be accessed both in the loop and after it

  - → The 'range' function can be useful as this creates a list of numbers from the first value to the last minus one

Improving the Transliterate Code (Ex 5)

- Now you know about lists and dictionaries, you can make the following improvements to your cipher code

    1. Instead of looping with a 'while' loop and a counter, change it to a 'for' loop going over each element in the string

    2. Instead of using lots of if..elif...else statements to convert numbers to words, set a dictionary to map from one to the other, i.e. mydict['1'] = 'ONE' then just check if the char is in the dictionary keys

8. Command Line Arguments

Command Line Arguments

- After you've got your code working as you want it, it would be rather annoying if you had to keep editing the file just to change one or two settings

- This is where command line arguments can be very useful as you can provide additional information to your script that it can then act on, e.g. which file to load, how to interpret a file, etc.

- To gain access to the command line arguments supplied to your program, you will need to access the external 'sys' module

- Note that I'll be just showing a very basic method of handling command line arguments. For a much more powerful method, see:

https://docs.python.org/2.7/library/argparse.html

Aside: Using External Code

- As we've mentioned before, reusing code is a very good idea as less code generally means fewer bugs!

- A key part of this, as well as meaning you don't have to write code for everything yourself is using external modules that provide code and data you can use

- You can ask Python to load these modules using the 'import' statement, after which you can access the functions and data provided by the module

```python
# import the 'os' module
import os

# Access things from the module using the '.' operator
print os.environ
print os.uname()
```

Command Line Arguments

Import the sys module into your program

```python
# Get access to the sys module
import sys

# print out the command line arguments
for arg in sys.argv:
    print 'Argument:   ' + arg

    # Look for '--help'
    if arg == '--help':
        print 'Help requested!'
```

sys.argv is a list containing all the supplied command line options

Loop over the command line arguments and print them out. If –help is found, print something

```
EPSC02PN49MFVH8:~ slatermw$ python cmdline.py test1 --help test2
Argument:   cmdline.py
Argument:   test1
Argument:   --help
Help requested!
Argument:   test2
EPSC02PN49MFVH8:~ slatermw$
```

Adding Command Line Options (Ex 6)

- To allow our cipher program to read input from a given file and output to a different file, we'll now add command line argument handling

- You need to make your program accept 3 options:

  - --help: Give some info about the program and how to use it
  - -i <file>: Specify an input file to pull the text from - store in a variable
  - -o <file>: Specify an output file to save the text to - store in a variable

- Note: To do the input and output files, in addition to checking for the flag, you need to save the following arguments into variables:

```python
# print out the command line arguments (first is script name!)
i = 1
while i < len(sys.argv):
    print 'Argument:   ' + sys.argv[i]
    # Look for '-i' and store following argument
    if sys.argv[i] == '-i':
        # sys.argv[i+1] is the next arg
        # don't forget to skip it with i += 1!
        pass
    i += 1
```

9. Functions

Functions (1)

- As mentioned several times now, its good coding practise to reuse as much code as possible. The main way of doing this in Python is through the use of functions that can then be called in other code blocks. Just like a variable, a function must be declared before it can be used:

def <function_name> (<arguments>):

- After declaration, the function is called by just giving the function name and the required arguments in brackets (as we've already seen).

- At this point, the program flow jumps to this function until it hits a 'return' statement or the end of it's scope (i.e. the end of the indented code block)

- Some points to note:

  → Variables declared in a function aren't visible outside of the function

  → You can specify default values for arguments in a function

  → When calling a function, you can specify arguments by name

  → Unless they are immutable types (Numbers, bools, strings) the object itself is passed to function (see next slide)

Program Flow – Functions (2)

Define a function that prints a list and changes the input arguments

```python
def print_list( inlist, line_msg = '' ):
    "print the values in a list"
    for val in inlist:
        print line_msg + ":  " + val
    inlist.append('All done')
    line_msg = 'All done'
    return 'Returning...'

# Call the function a couple of times
print_list( ['hello', 'world'], 'MSG1' )

mylist = ['test1', 'test2', 'test3']
my_msg = 'MSG2'
out = print_list( line_msg = my_msg, inlist = mylist)

# This will show the addition to the list but
# the message won't be changed
print mylist, my_msg

# We also have the output from the function stored
print out

# This would cause an error
#print val
```

Note that, the list will be changed but the string will not!

We can use the output from a function as given by the 'return' statement that it exited with

'val' was declared in the function and so is not available outside

Start Using Functions (Ex. 7)

- Now you have learnt what functions are and why they are used, we can now start using them in your cipher code

- For the next exercise, move your transliteration code within the loop into a function called 'transformChar' that takes a string and returns a string after applying the transliteration:

```python
def transformChar( in_char ):
    'Take the input char, apply transliteration and return'
```

- Don't forget to put the function above your main code so it is defined before being referenced

- You will also need to alter the loop in your main code appropriately to call this

Returning More Than One Value

- Returning one value from a function is useful for a single answer, but quite often you will need to return several values

- In python, you can return multiple values with the return statement and 'unpack' them automatically from the calling code:

```python
def myfunc():
    return 1, 'hello'

var1, var2 = myfunc()
```

- However, this can quickly become a problem if the number of values returned changes or you forget which order they come in

- A better method is to return a dictionary:

```python
def myfunc():
    return {'var1':1, 'var2':'hello' }

var_dict = myfunc()
```

Continue Using Functions (Ex. 8)

- To continue 'cleaning up' your main code, you should now move the code for dealing with the command line arguments to a function called 'parseCommandLine'

- This should take no arguments but return a dictionary with the following key:value pairs:

  - → help_req: boolean, True if –help was found
  - → in_file: String, Input file if specified
  - → out_file: String, Output file if specified

- This should leave you with the main (global namespace) code looking something like this:

```python
# <transformChar function>

# <parseCommandLine function>

# ----------------------------------
cmd_args = parseCommandLine()
if cmd_args['help_req']:
    # print help and quit
    sys.exit(0)

in_string = raw_input('Input: ')
out_string = ''
for in_char in in_string:
    out_string += transformChar( in_char )
```

10. File Operations

File Operations

- One of the main things you will want to do is load files and process their contents as well as write out some processed data

- I will just be dealing with text files in this course as they will be the most useful to you, however extending to any binary data is fairly easy

- The main way to access files is with the 'open' function

- This returns what's called a 'file object' – basically a reference to the open file which has a number of useful functions to interact with it

- There are many options for dealing with files, see the following link for more info:

https://docs.python.org/2/tutorial/inputoutput.html

File Operations

Open a file called 'hello.py' for reading (the default)

Close the file – this will happen automatically when the variable goes out of scope

```python
# Open a file for reading
myfile = open('hello.py')

# print the contents
out_str = ''
for ln in myfile.readlines():
    print ln
    out_str += ln.split()[0]

# close this file
myfile.close()

# Open another for writing
myfile2 = open('myout.txt', 'w')
myfile2.write(out_str)
myfile2.close()
```

The 'readlines' function returns a list of all the lines (including a new line '\n' character). This makes it easy to loop over each line in turn

After opening another file for writing, tell python to just write the contents of the string to the file

Start using Input/output Files (Ex. 9)

- Now we have the knowledge necessary to implement reading the input for the cipher from a file rather than the keyboard as well as writing the output to a file as well

- You already have the command line options that provide you with the input and output files (if they were provided) so now you should do the following:

  1. If an input file is given, load the string to be processed from that instead of asking for input (use the 'read' function rather than 'readlines')

  2. If an output file is given, write the output to that instead of the screen

11. Caesar Cipher

Caesar Cipher

- We can now finish off the last part of our program – the actual Cipher code!

- This is a basic substitution cipher that swaps an input letter with one given by a key:
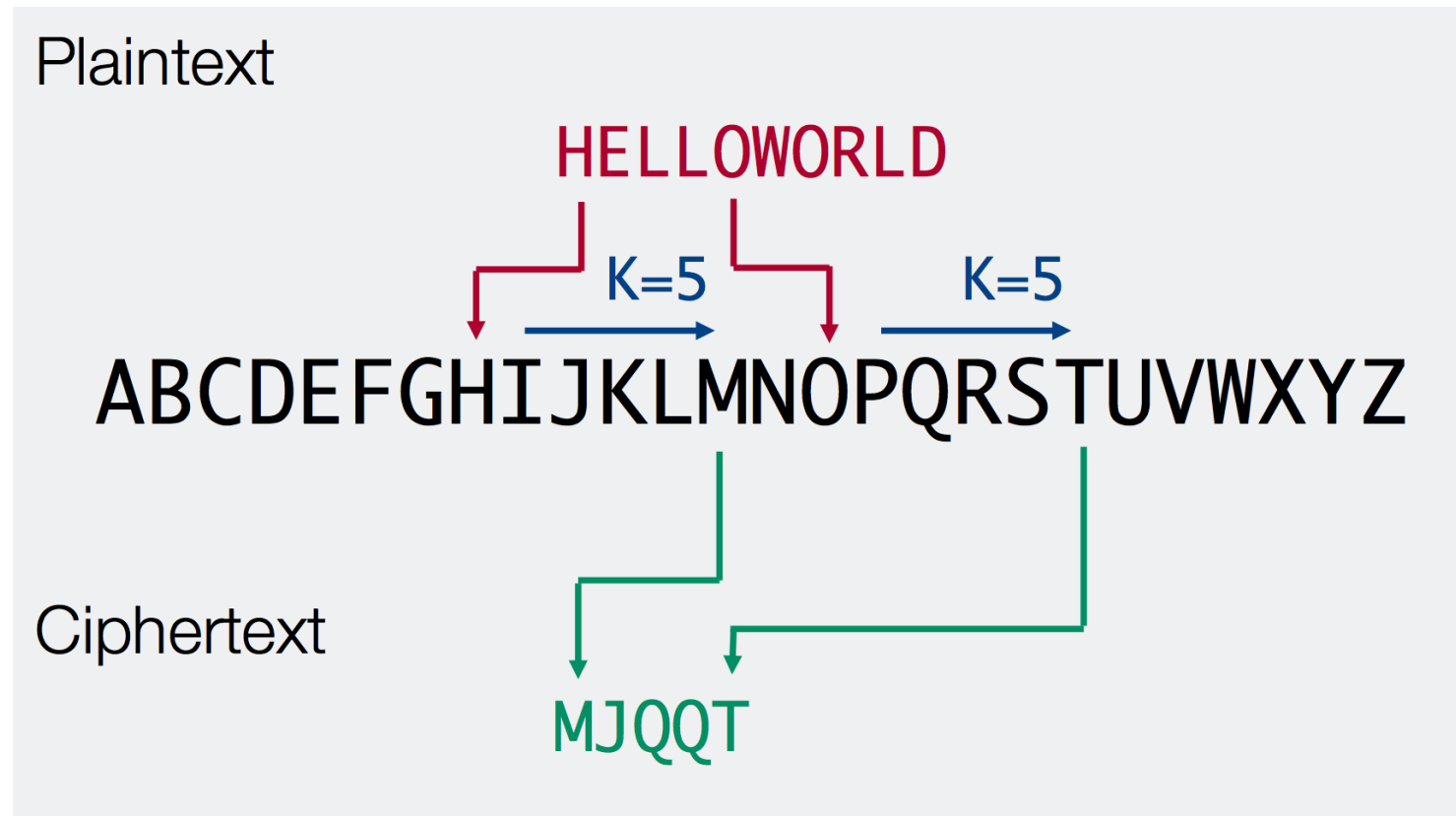
Caesar Cipher (Ex 10)

- Our code already 'sanitises' the input text so at this point we only need to worry about applying the actual cipher

- Before this however, you should extend the parseCommandLine function to allow the user to provide a key – use 'int()' to convert from a string to an integer

- As with the transliterate code, we'll tackle the caesar ciper by breaking up the problem into sections and doing each one in turn

- You should create a new function (as given in the next slide) that takes an input string and key and returns the cipher text

- Then, write code that does what each comment asks for

- Make sure each bit of code does what is required by checking the variables before and after the additional code

- When you're happy, move on to the next bit – you should only need ~1-2 lines of code per comment

- Run your code over 'dickens.txt' with a key of 5 and compare your output to the 'dickens_cipher.txt' file using the 'diff' shell command as a check!

Caesar Cipher (Ex 10)

```python
def runCaesarCipher( plain_text, key ):
    'Apply the caesar cipher method on the given text'

    # Create the alphabet container and output string

    # Loop over the input text

    # For each character find the corresponding position in the alphabet
    # (use find or index depending on how you set up the alphabet)

    # Apply the shift to the position, handling correctly potential wrap-around

    # Determine the new character and add it to the output string

    # Finally (after the loop), return the output string
```

12. Basic Data Analysis

Performing a Basic Data Analysis

- Now we have covered the basics of Python, we'll now use that to cover a more practical example that will be more relevant to your research

- The next few slides will take you through extracting data from a text file, plotting the results and then fitting a curve to the data

- It will use a number of external modules: matplotlib, numpy and scipy

- This is a fairly advanced example so do please ask for help if you need it!

Extracting Data from a Text File

- The first step of this data analysis is to retrieve the appropriate data from a text file

- I have already pre-generated some data that you can find in the 'data_analysis' folder in the git repo – datasets.txt

- If you look at the file, you'll see 4 columns:

  1. The 'time' (x value) of the data point

  2. Value of an exponential (e^x) function

  3. Value of a 4th order polynomial

  4. Value of a gaussian/normal distribution

- Note that the time values are not regular and there is some noise applied to each of the curves

- Also, comments are denoted by a '#' at the start of the line

Extracting Data from a Textfile (Ex. 11)

- You should create a new program file for this section

- Then, code each of the following steps so you open the file and retrieve the data:

  1. Initialise 4 lists to take the 't' values and each of the 3 datasets

  2. Open the file and loop over each line (use 'readlines' here!)

  3. Ignore any line starting with '#' - don't forget how to access single characters!

  4. Split the line up into it's tokens

  5. Add each to the appropriate lists given it's position (use the 'float' function to convert the string to a number)

- As before, you only need a few lines of code for each stage and you should test after each point to make sure it's working

- You may find the following useful:

```python
# Set up a string of words with white space
mystr = '   my    test string'

# Remove whitespace/newlines from front and back
print mystr.strip()

# split up string into a list of the tokens
toks = mystr.split():
```

Plotting in Python

- Now we have the data in lists we want to plot it

- This is actually made quite easy thanks to the 'matplotlib' library

- It has many ways of plotting data including contour plots, histograms, error bars, etc.

- A typical example is:

```python
# import the matplotlib library
import matplotlib.pyplot as plt

# plot an y=x curve with red circles
plt.plot([1,2,3,4], [1,2,3,4], 'ro', label='y=x')

# Add a y=x^2 curve as a blue line
plt.plot([1,2,3,4], [1,4,9,16], 'b-', label='y=x^2')

plt.xlabel('x')
plt.xlabel('y')
Plt.legend()
plt.show()
```
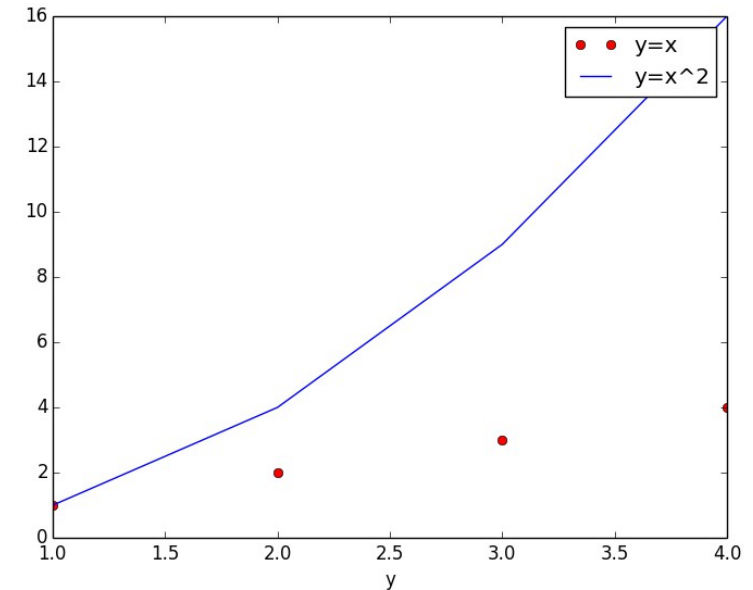

- I would highly recommend having a look at the documentation:

https://matplotlib.org/contents.html

Plotting data from a file (Ex 12)

- Using the code on the previous slide, you should now be able to extend your data analysis code to do the following:

  1. Import matplotlib

  2. Plot each of the three curves on the same axes

  3. They should be plotted with different coloured points

  4. Label the axes and provide a legend

Fitting in Python

- All that's left to do in this basic data analysis is to fit the data to a given a model(s) to see what parameters are best

- Again, this is made fairly simple for us by the external 'scipy' and 'numpy' modules that provide simple functions to do this

- To use this function, you need to provide a model for the data and the data itself, and scipy will return the best fit parameters

- You can then use these parameters to plot the resulting fit

- See the next slide for an example of how to do it!

- As with matplotlib, do go through the documentation to see what else is there:

https://www.scipy.org/docs.html

Fitting in Python

Note that we need to use the 'numpy' version of the exponential function to avoid problems when running the fit

Can specify bounds for the fit if you're having trouble getting it to converge

```python
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
import numpy as np

# assume we have a decaying exponential
def model(x, a, b, c):
    return a * np.exp(-b * x) + c

# assume xdata and ydata are lists of the data and then do the fit
popt, pcov = curve_fit(model, xdata, ydata, bounds=([0, 0, 0], [3., 2., 1.]))

# popt is now an array of the optimal values
# can just pass this to the model using the '*' operator to unpack
ydata_fit = []
for x in xdata:
    ydata_fit.append( model(x, *popt) )

# can now plot the fitted data
plt.plot(xdata, ydata_fit, 'r-', label='fit')
```

Fitting Models to the Data (Ex 13)

- Again, as you already have the data in lists, you can apply the code in the previous slide to do a best fit for the 3 models:

  1. Set up 3 functions as the models for each dataset
     → n0 * exp(lambda * x)
     → a + x*b + x*x*c + x*x*x*d + x*x*x*x+e
     → n0 * exp(-1 * (x – mu) * (x – mu) / (2 * sd * sd)

  2. Perform a fit on each to obtain the best parameters

  3. Plot these as (labelled) curves to your final plot

  4. Print out the best fit parameters for the 3 models

- Note that you may need to specify bounds for the exponential fit to get a decent convergence!

Solutions and Links again!

- To get the 'solutions' for the exercises, have a look at:

  https://github.com/drmarkwslater/CCBPythonCourseSep2017Solutions

- Main python documentation:

  https://docs.python.org/2/

- Good place to ask questions:

  http://stackoverflow.com/

- Google is very good as well!

- My contact details:

  Mark Slater <mslater(AT)cern.ch>, Physics West 317