# Beginners Python Course

Elliot Lynch

June 13, 2016

## Contents

**1 Introduction** **2**
- 1.0.1 Strengths: . . . 2
- 1.0.2 Weaknesses: . . . 2
- 1.1 General Things to Note: . . . 3
- 1.2 The Golden Rule! . . . 3

**2 Syntax and Language** **4**
- 2.1 General Syntax . . . 4
- 2.2 Boolean Logic and Comparison Operators . . . 5
- 2.3 Arithmatic . . . 5
  - 2.3.1 Integer Arithmatic . . . 5
  - 2.3.2 Floating Point Arithmatic . . . 5
  - 2.3.3 Integer Arithmatic . . . 5
  - 2.3.4 Sequence Arithmatic . . . 6

**3 Third Party Libraries** **6**
- 3.0.5 Inbuilt libraries . . . 6
- 3.0.6 Important third party libraries . . . 6

**4 Sequences (Array Structures) in Python** **7**
- 4.1 Other Data Structures . . . 9
  - 4.1.1 Set Arithmatic . . . 9

**5 Numpy** **10**

**6 Basic Plotting with Pyplot** **12**
- 6.0.2 Example Linestyles . . . 13

**7 Scipy** **13**

**8 Classes** **13**

**9 Try,Except and Exceptions** **13**
9.1 Raising Exceptions . . . . . 14
9.2 Using Try,Except . . . . . 14

**A Basic Language Referance** **15**
A.0.1 Comments . . . . . 15
A.1 Print Statement . . . . . 15
A.1.1 If statement . . . . . 16
A.1.2 Loops . . . . . 16
A.1.3 Functions . . . . . 16
A.1.4 Import statement . . . . . 17
A.1.5 Basic Data Types . . . . . 17
A.1.6 File I/O . . . . . 18

# 1 Introduction

Python is a general purpose interpretive language. Unlike C, python is not compiled but is instead executed at runtime. This gives python a great deal of flexibility at the expense of speed. Python is widely used in scientific computing, particularly as a much more user friendly interface to numerical solvers typically written in C/Fortran.

I appologies in advance for any anti-windows bias in this document. However if you do decide to base your operating system on something other than unix you deserve everything you get.

### 1.0.1 Strengths:

- Interpreted, can run individual commands in consol - very easy to trial code segments.
- Alot of very good third party libaries, e.g. numpy, scipy,matplotlib. Many of these are written in C so are fast.
- Readability
- Comparitively recent language, so objects, lists and other features of a "Modern" Programming language haven't been tact on haphazardly.
- Well documented, far less inaccurate information than other languges (*C pointers*)

### 1.0.2 Weaknesses:

- Interpreted, signficantly slower than compiled languages
- Is very reliant on third party libaries to be truely useful

- Use of indentation for code blocks, aids readability but inconsitency can lead to portability issues.
- alot of changes between python 2 and 3 which are fairly incompatible.

## 1.1 General Things to Note:

Python is an interpreted language and does not need to be compiled before use. This means that python can be run in a terminal or other programming enviroment (e.g. ipython), and commands can be executed line by line in a mannor similar to Matlab. Like Matlab python scripts are not compiled, rather python reads each line in turn and executes it. This has the great advantage that new code can be prototyped, written and run in much less time than for a compiled language. Also python makes a great scripting language and can be used to quickly open and modify large files without having to write and execute a program. However an interpreted code comes at a cost, and is much slower than compiled codes.

The fact that python executes each line in turn is a significant aid to debugging as when your code inevitably crashes it typically exits on the line which caused the crash. Python also deals with all the memory allocation in the code so memory leaks don't (read shouldn't, you can probably do it if you're sufficiently determined) happen in python.

Python uses indentation to denote code blocks in a mannor similar to how C makes use of curly braces { }. This is a great aid to the readability of the code as it forces code to be properly indented and avoids the clutter of have {, } or words such as `do` and `done` littered throughout the code. However it does have one major issue in that it is very inconsistent, with any number of whitespace caracters (tab space etc) usable for indentation. I has unfortunately become not uncommon for programmers on backwards operating systems (i.e. windows) to use tabs for indentation. This is very bad practice and completely ruins the portability of the code, as tab is not consitently defined between operating systems and will normally get converted to an incosistent number of whitespace caracters if the code is run on linux/mac.

## 1.2 The Golden Rule!

The Golden Rule of Python is to do as little computational work in python as possible by making use of third party libaries.

Particularly For loops are a bad idea! Alot of code that you can write with for loops can also be done using third party libaries. In many cases all the third party libary will do is execute exactly the same loop code (baring possible optimisation) as you would have done. However the major third party libaries are written in C so the code will execute much faster.

# 2 Syntax and Language

The following is a referance to pythons language and Syntax. Python is actually well documented and has a central documentation here `https://docs.python.org/3/`. And of course stack overflow is your eternal saviour. It's important to note that python has changed significantly between version 2 and 3 and in a way that basically breaks everything between the versions. Python 3 is anoyingly more restrictive than 2 but it is more consitent. This section is not meant to be exhastive, that's what the main documentation is for, but to contain enough to get started.

I have also decided to keep more advanced commands such as try/except and classes seperate from this section

## 2.1 General Syntax

Python does not have a line terminating caracter, instead using newline to terminate each line of code (more accurately the line terminating caracter is \n the newline caracter). To indicate code blocks upon which loops,if statements etc act python uses whitespace caracters (stick to useing a consistent number of spaces! Never use tabs.) When you come to the end of the code in the loop,function definition etc this is indicated by returning to the original indentation:

```
for i in [1,2,3]: #note colon to end for loop line
  # this is in the for loop
  print i*2

#this isn't
print 2
```

The line preceding indented code blocks are terminated by a colon e.g.,

```
for i in [1,2,3]:

if (a==b):

def foo():
```

In python you do not need to delcare the type of a variable. This is determind at runtime when you assign to the variable. Python also has no restrictions on type when assigning to an existing variable. So it is possible to assign a string to a variable which is currently a float, this behaves exactly as if the variable was being declared for the first time.

## 2.2 Boolean Logic and Comparison Operators

Python Boolean Logic and Comparisons are the same as C.

| | |
|---|---|
| a == b | a equals b |
| a ! = b | a not equal to b |
| a < b | a less than b |
| a > b | a greater than b |
| a <= b | a less than or equal to b |
| a >= b | a greater than or equal to b |
| not a | boolean not operator |
| a and b | boolean and operator |
| a or b | boolean or operator |
| a is b | checks if a and b are the same object |
| a is not b | checks if a and b are not the same object |
| a in b | checks if a is a member of b, e.g. if it is an element in a list |
| a not in b | checks if a is not a member of b |

## 2.3 Arithmatic

Native python has both integer and floating point arithmatic built in. It also has sequence arithmatic which is used for concatinating strings and certain array like structures such as lists. Python has an operator precedence which is the normal brackets,power,multiplication,addition ordering.

### 2.3.1 Integer Arithmatic

| | |
|---|---|
| a + b | adds two integers |
| a − b | subtracts two integers |
| a / b | integer division, a divided by b and neglects the remainder |
| a % b | modulus, divides a by b and returns the remainder |
| a ∗ b | multiplies two integers |
| a ∗∗ b | raises a to the power of b |

### 2.3.2 Floating Point Arithmatic

The operators for floating point arithmatic are the same as for integer arithmatic except that they operate on floats. The exceptions are:

### 2.3.3 Integer Arithmatic

| | |
|---|---|
| a / b | floating point division, a divided by b including remainder |
| a // b | floor division, this acts like integer division and divided a by b neglecting the remainder |

Python also has an increment operator similar to C's `++`. Preforming `a = a + b` can be done using the `+=` assignment operator by: `a += b`. All

the above operators have similar assignment opertors. So `*=`, `**=`, `/=` etc can be used to multiple/raise to the power/divide the existing value of the variable by a new value.

### 2.3.4 Sequence Arithmatic

Python has arithmatic opertors which act on sequences. These are not the same as array operations! Native python does not have an array structure on which elementwise arithmatic can be preformed like Fortran or Matlab. Instead it has sequences such as lists `[1,2,3]`and strings `"abc"`.Note that python treats strings as lists of caracters so `"abc"`is treated similarly to `["a","b","c"]`(Although a string is not regarded as a list type).

| | |
|---|---|
| a + b | concaternates a and b, with b following on from a. So [1,2] + [3] becomes [1,2,3] |
| a*n | Where a is a sequence and n an integer. Concaternates n copies of a into a single sequence. so [1,2]*2 becomes [1,2,1,2] |

# 3 Third Party Libraries

The key to good python code is the use of libraries. Most things that you want to do in python have already been implimented in a library and are normally much faster than the equivilent code in python.

### 3.0.5 Inbuilt libraries

- math : basic maths library containing definitions of pi, sin etc. It's rare for this to actually be useful as all of it is defined in numpy
- os : extensive library for interacting with the operating system, e.g. open/writing to files, looping over files in a directory etc.
- sys : for interacting with the system enviroment and for passing your code command line arguments. This is actually part of the os library but, unless you are dealing with file paths, is probably all you need.

### 3.0.6 Important third party libraries

- numpy : the main numerical libary in python. Contains the definitions for arrays, matrices complex numbers and many basic mathematical functions/operations. Also includes good support for reading from/writing to deliminated files
- scipy : an extensive library for scientific computations. Includes functions for statistics, ode solvers, least squares inversion/curve fitting and many special functions.

- matplotlib : the main plotting library for python. heavily influenced Matlab plots.
- pylab : a cut down version of matplotlib for when you want to plot the data but don't really need that much control over the presentation. (I use it alot as I am lazy).
- Shapely : 2D geometry and topology library, good for interacting with GIS data/maps. Defines lines/polygons etc and their relationships
- PyQt and PyGTK : for when you have to write a GUI
- Cython : definately not in a beginners course, for integrating C and python.

# 4 Sequences (Array Structures) in Python

Python has a variety of array like structures, some of which have been mentioned above. These are collectively refered to as sequences. It should be noted that there is no array type in native python. Sometimes people refer to sequences (particularly lists) as arrays, however sequences do not behave like arrays and this also confuses them with the actual implementation of arrays in numpy.

As mentioned above sequences have there own arithmatic, which differes from array arithmatic. Sequences are numbered from 0 to n-1 in a similar manor to C arrays. Elements in a sequence can be accessed by:

```
sequence_variable = [1,2,3]

element= sequence_variable[1]
# element now has the value 2.
```

Sequences accept negative indexes which count backwards, so for a sequence of lenght m `sequence_variable[-n]` will give the `m-n` th element. Sequences also admit slicing, where you specify a range of indices and obtain a new sequence with just these elements.

```
sequence_variable = [1,2,3,4,5]

new_sequence= sequence_variable[1:3]
# new_sequence is now [2,3,4]
```

There are three sequences in python, combined with several unordered data structures. I will not cover strings here, except to say that they are a sequence (in a similar way to how they are arrays in C) and many of the operations that can be done on sequences can be done on strings.

The primary sequence in python is the list. Unlike arrays, list need not contain objects of the same type making `["string",11]` a perfectly viable list. Also unlike arrays lists do not have a fixed lenght, and items can be freely appended to them and they can be concaternated. Items in list also have a notion of ordering, in that a list element knows about the elements either side of it.

Lists are thus much more complicated and flexible structures than arrays, however they pay for this by being much slower and less memory efficient.

Lists are never multidimensional, however as a list can contain any python type this can be acheived using nested lists. So a n by m array can be constructed from a list of n lists of length m. There is nothing requireing these lists to be the same length however. It should be noted that nested lists are inherently a bad idea and should be avoided if possible as they are very slow, normally there is a better way of coding something involving a nested list.

The other type of sequence is the tuple. This is an imutable sequence, you declare it and cannot modify it afterwards. It is declared using round bracket `()` in a similar manor to lists,

```
new_tuple = (1,2,"string for variety")

# note for a tuple of one element:
new_tuple = (1,)

# not!

new_tuple = (1)

# as this is just paranthasis around an integer.
```

You can access the elements of a tuple, add and multiply and slice them, but modifying the size of the tuple or the elements it contians is not possible. Tuples are much faster than lists, however in most situations (the exception being when you want to make use of nested tuples) it is better to use a numpy array when speed is your concern.

While lists/tuples can be initialised by typing out each entry individually a more praciticle method is using for loops in lists. This enables lists to be initialised base on another list. for instance:

```
# range(n) is a function that returns a
# list with entries 1,..,n-1

lst1 = range(n)
```

```
# initialise an array where the elements are
# given by some function f(x) of elements
# in lst1

lst2 = [f(el1) for el1 in lst1]
```

## 4.1 Other Data Structures

Python has other non-sequence data structures, which have more specialised uses. These are mostly store unordered data.

Python has a set object built in, this is an unordered collection of unique elements. Set comparison operations are defined between sets, so the use of `set1 < set2` returns `True` if set1 is a proper subset of set2. The following is a referance to pythons set arithmatic:

### 4.1.1 Set Arithmatic

| | |
|---|---|
| a == b | set equallity between a and b |
| a ! = b | set a and b are not equal |
| a < b | a is a proper subset of b |
| a > b | a is a proper superset of b |
| a <= b | a is a subset of b |
| a >= b | a is a superset of b |
| a \| b | union of a with b |
| a − b | set difference between a and b |
| a & b | intersection between a and b |
| a ^ b | symmetric diference between a and b |

It is also possible to preform =— and similar, along with adding and removing specific elements from the set.

Another important data structure is the dictionary, which is related to the data structure often called a hash-map in other languages. This is an unordered collection of key,value pairs where the values stored in the dictionary are referanced by the keys. This is very useful for when you have a collection of properties associated with a label (e.g. a named person/particle/star etc), and enables you to retreive the data from the dictionary associated with this label without having to loop over and checking the elments as you would in a list. As the key acts as a pointer to the associated value (stored data), seaching dictionaries for the data associated with a specific key is very fast.

```
# implimentation of a dictionary
dict_var = {}
```

```
# add some key,value pairs
dict_var["Betelgeuse"] =(90000.0,3300.0)
dict_var["Agol_A"]=(182.0,13000.0)
dict_var["CygnusX1"] =(100000.0,1.0e8,"compact object")

# obtain data for Algol_A
data = dict_var["Agol_A"]

# same implimentation with list:
list_var = [["Betelgeuse",(90000.0,3300.0)],["Agol_A",(182.0,13000.0)],["CygnusX1",

# would need to loop over with an if statement to extract data for a specific entry
```

The keys in for the dictionary need not be strings, but can be any immutable object. In general this means strings,tuples and integers and specifically forbiding floats and lists as these risk being modified - thus no longer providing a one to one map to the stored data.

# 5 Numpy

I have devoted an entire section on numpy due to it's utility. Correct use of numpy is key to any numerical work in python.

The main type in the Numpy library is the numpy array. This is the main implementation of arrays in python (there are others). These are true arrays in the sense that they are a fixed lenght sequence of variables of the same type. Numpy arrays can be multidimenstional and can be obtained from the equivilent multidimentional list by,

```
# we've imported numpy as np

list_variable = [[1,2],[3,4]]

array_variable=np.array(list_variable)
```

Numpy arrays are indexed from zero. They accept slicing and each dimension can be sliced seperately so:

```
# second column of a numpy array:
column=array_variable[:,1]
```

Like there equivilent in Matlab and Fortran numpy array accept elementwise operations. All arithmatic operations preformed on a numpy are preformed element wise. Hence to multiply to sequences of numbers:

```
A = [1,2,3]
B = [3,4,5]

# with lists
C=[]
for i in range(len(A)):
  C += [A[i]*B[i]]

# with arrays
C = np.array(A)*np.array(B)
```

Appart from being slightly shorter the main advantage of doing this is speed. Given that ultimately the computer must loop over each element in turn regardless of how you code it, why should this be? The answer is that the loop in numpy is written in C and thus executes much faster than the equivilent loop written in python. Also numpy arrays are written more efficiently in memory than lists as they do not support nesting.

Numpy arrays can be initialised using lists, but has a varaiety of functions for generating arrays:

```
# initialise an array with n zeros
ar1 = np.zeros(n)
ar2 = np.zeros_like(ar1) # array of zeros with same shape as ar1

# initialise an array with n zeros
ar1 = np.ones(n)
ar2 = np.ones_like(ar1)

# initialise array with n linearly increasing values
# between strt and end
ar1 = np.linspace(strt,end,n)
ar2 = np.linspace(strt,end) # defaults to n = 50
```

A very useful function in numpy is `np.where`. This sets values in a numpy array accourding to a conditional, and is esentially equivilent to looping over the array and setting the value with an if/else statement. This should almost always be used instead of for loops with if statements.

```
# use of the where statement
# for ar1,ar2 a numpy array

# the syntax is np.where(conditional,value if true,value if false)
# can set with arrays
ar3 = np.where(ar1<0,ar1,ar2)

# or numbers
ar3 = np.where(ar1<0,1,0)

# can also compare arrays for the conditional
ar3 = np.where(ar1<ar2,1,0)
```

In addition to arrays, numpy has an implimentation of matrices - with all the normal definitions of matrix multiplication, inverse, determinant.

Numpy also contains a module numpy.ma which contains an implementation of masked arrays. These are used for dealing with invalid data, but are also often of use for selecting subsets of arrays, particularly with the use of `np.where` . These are used as follows:

```
# numpy.ma must be imported seperately
import numpy.ma as ma

# masked arrays have the mask optional
# argument which accepts a list containing
# 1 or 0 of the same shape as the array and
# masks elments which are marked with a 1
# these are still in the array but are no longer
# used for computation
# for array ar1:

masked_ar = ma.masked_array(ar1,mask = [0,0,1,0])

# this has masked the 3rd element
```

# 6 Basic Plotting with Pyplot

The main plotting library for python is Matplotlib, which is esentially a copy of MatLabs plotting library with most of the syntax unchanged. The simplest way of plotting things in python is to use pylab, which is a module of matplotlib. When using Matplotlib alot needs to be specified before you can plot anything, which gives a great deal of flexibility at the expense of

lazieness. Pylab is quite different to plot two arrays x and y the syntax is beutifully simple:

```
# import pylab as pb

pb.plot(x,y)
pb.show()
```

For a scatter plot:

```
pb.plot(x,y,"x")
pb.show()
```

The `"x"` here is the linestyle, this is an optional argument that can be passed to pb.plot, the default is straight lines. This defines if/what lines to use, what to plot on the datapoints and the colours (On which note Matplotlib/pylab is American and thus mispells it color instead). Some example linestyles:

### 6.0.2 Example Linestyles

| | |
|---|---|
| `"-"` | solid line |
| `"r-"` | red solid line |
| `"k--"` | black dashed line |
| `"x"` | crosses |
| `"bo"` | blue circles |
| `"g:>"` | green dotted line with triangles |

Plotting multiple lines/datasets can be done by calling pb.plot multiple times before the pb.show - you plot all the data and then call show and there is no need to specifiy `hold` as in MatLab.

# 7 Scipy

# 8 Classes

# 9 Try,Except and Exceptions

Sometimes you have to interact with real users, and inevitably they will do stupid things. How do you check to see if the user has given you sensible inputs when you ask for it. This is done with the try,except statements in conjuction with Exceptions.

## 9.1 Raising Exceptions

Python will raise it's own exceptions if it recieves incorrect inputs, however often you may wish to deliberately raise an exception. This is done using the raise command:

```
# raise the base exception
raise Exception

# raise a specific error (TypeError)
raise TypeError

# raise specific error with additional information
# which will be printed when the error is raised
raise TypeError("A type error was raised")
```

Python has a large number of Errors predefined and most circumstances are covered by one of them. However there are times, particularly when developing classes that you may want to define your own. This is done by writing an error class, which makes use of inheritance:

```
# a custom error which inherits from
# exception
class CustomError(Exception):
  pass
```

This will now behave like any other Exception.

## 9.2 Using Try,Except

`try` and `except` are commands in python used primarilly for error handling. They are often used in conjuction with raw_input and the raise statement to handle data inputed by the user at runtime, which is not gatunteed to be in the correct format. Python runs the code in the `try` block, however if this fails with an error the code in the `except` block is run instead and any results (e.g. changing variables) from the `try` block are discarded.

```
try:
  # execute this block of code
  #
  #
except:
  # if the above code fails
  # execute this instead
```

An example of how to use this is when you require the user to input data - say a float:

```
try:
  # ask the user for input using raw_input
  # which returns what they type as a string
  usr_input=raw_input("Input a Float")
  # convert the input to a float
  x = usr_input
except:
  # if something went wrong - say the users
  # an idiot and writes a sentance instead of
  # a number
  raise TypeError("Thats not actually a number")
```

There is another use for try, except and that is to handle data with missing/invalid values. This is really an abuse of the try, except routine as you are effectively using it as an if statement, with except being used as else. This is dangerous as any bug in the try block will not cause the code to crash/raise an exception. However it's a rather useful abuse and so long as your aware of the dangers of doing so (and preferably if the code doesn't have to be used by anyone else), then it is acceptable.

# A Basic Language Referance

### A.0.1 Comments

```
# this is a comment in python

'''
This is a multiline comment in python
it can also be used to define multiline
string and is quite good for documentation
'''
```

## A.1 Print Statement

```
a_variable = "to be printed"

# python 2 print syntax
print a_variable

#python 3 print syntax
```

```
print(a_variable)

# all types in python can be printed,
# however what python actually prints
# is very type dependent.
```

### A.1.1 If statement

```
# if statement for python, here bool
# is a boolean variable or an integer
# 1 or 0. The brackets are entirely optional

if (bool):
  print "true"

# also can use logical conditionals

if (a == b):
  print "true"
```

### A.1.2 Loops

```
#for loop

For i in [1,2,3]:
  print i

# while loop
while true:
  print 1
```

### A.1.3 Functions

```
# define a function

def foo(a,b):
 # note the indent
 # I also do not need to tell python what
 # type the function takes/returns as this
 # is determined at runtime
  return a + b
```

### A.1.4 Import statement

```
#the import statement for importing libaries/functions

# imports the whole library
# call members of the libary by prefixing with the name
# of the libary e.g.
# numpy.foo()

import numpy

# imports the whole library but changes the prefix to np
# so a member of the libary can be called by:
# np.foo()

import numpy as np

# import a single member of the libary

from numpy import foo

# this can no be called using:
# foo()

# note these are used to import your own libraries
# so the command:

import mylibrary

# will look for mylibrary.py in the local directory and
# run this, thus enabling all the functions/varaibles
# in mylibrary.py to be used
```

### A.1.5 Basic Data Types

```
# The type of a python variable is determined at runtime
# based on what is assigned to it.

# declare a string
var = "a string"

# floating point number
```

```
var = 1.0

#integer
var = 1

# warning it is a common mistake in python to do
# the following:

var = 1/2

# this is not 1/2! As 1 and 2 are integers, integer division is
# preformed, the value of var is 0
# to assign a value of a half floats must be used:

var = 1.0/2.0

# a list, this is similar to an array, except that it is extendable
# i.e. is not of a fixed length

var = [1,2,3]
```

### A.1.6 File I/O

```
# to open a file for reading in the local directory
# called read.txt

f = open("read.txt","r")

# read a line
line = f.readline()

# read entire file
data = f.read()

# open file for writing

g = open("write.txt","w")

# write to file
g.write(data)

# add a new line
```

```
g.write("a new line\n")

# you should close files when they are do
# as python doesn't have a compiler to tell
# it when they are no loner in use
f.close()
g.close()
```