PH2150 Scientific Computing Skills

Andrew Casey
a.casey@rhul.ac.uk

May 2016

The aim of this course is to teach you how to use a computer to solve physics and maths based problems. This will be done by introducing you to the programming language *Python* over the first four weeks, and then applying *Python* to solving and visualising physics problems relevant to the first two years of the RHUL undergraduate course. Some books about software list the code that you should type, and sometimes explain what the code is doing. This then looks easy but does not teach you how to write your own code, the approach taken in this course is to ask the question, think about how to solve it using the resources available (including the demonstrators) and then develop the code that successfully solves the problem.

By the end of the course you should all:

- Be comfortable with the use and syntax for *Python 2.7*
- Be able to solve numerical problems using *SciPy* and *NumPy*
- Fit functions to datasets
- Generate 2D and 3D visualisations using *Matplotlib* and *Mayavi*
- Generate scientific reports in Latex
- Many final year projects involve computation and visualisation of data

Contents

1 Course Structure:

The course consists of one term of labs in the teaching Lab T231, typically the first section of each week will consist of a lecture introducing new concepts and that weeks problem sheet. The first four weeks will focus on a introduction to *Python* as a programming language highlighting the syntax and techniques relevant to scientific computing problems, this document forms the part of the notes for the first four weeks but students should follow the further reading suggestions in order to progress more rapidly additional information will be contained within the exercise sheets. This will be followed by six weeks of physics related problems that require the use of *Python* to solve. Each week will be accompanied by an exercise sheet, these will be submitted via *Turnitin* at the end of the last session and will be marked during the rst session of the following week. The exception to this is the post exam work which will be marked in the two weeks of the Autumn term. The marks for these exercises plus an extended exercise/report at the end will form the assessment for this unit. These problems will start gently, but become more complex as your *Python* abilities improve.

2 Getting Started:

2.1 Why *Python*?

You will be aware of a multitude of programming languages and environments in which you could perform scientific computing tasks, you are already familiar with *Mathmatica* from your first year course. In consultation with industries that recruit from physics departments we asked the question *"What Programming language makes a graduate more attractive to you as a potential employee"*, the answer that we got back was that they were less concerned about a specific language but wanted to see evidence that the graduate had performed scientific computing to answer real problems. Larger companies, such as IBM, have in-house languages like *X10* which they expect to have to train all new recruits in its use. Therefore the question falls back to what language can do the tasks that we require as physicists that allows a beginner to do simple things easily but has the potential to develop into a powerful scientific computing tool?

This left us with a list of languages *C*, *MatLab*, *Java* or *Python* all these languages have advantages and disadvantages and some may be better suited to specific applications. For purely mathematical operations *C* (which is a class of languages call compiled) is faster than *Python* (an interpreted language, more of this latter) but this speed saving is only important if you are planning to write code that will be running for long periods of time or repeated on many occasions. Most programs written by physicists are required to solve a particular problem, run once, then discarded. In this scenario the time that is important is the sum of runtime and development time, and it is here where *Python* out performs *C*.

Hopefully through this course you will see that *Python* has a fairly logical format that is readable in an approximation of english combined with mathematical terms. *Python* also has

the advantage that it is free open-source software that is platform independent (Windows, Mac, Linux) so you will all be able to install *Python* on your home PC's or laptops. These benefits are obvious to many people so it is no surprise that *Python* has a growing users base amongst academic researchers (including various groups at RHUL). A quick survey of industrial users of *Python* contains companies such as YouTube; Google (search engine); Industrial, Light and Magic (used for some of the later Star Wars films) and NASA. So once we have decided upon *Python* we will try to employ it in its most generic sense so that the methodologies that you learn could equally be used with any other language you may be required to use in the future.

2.2 The *Python* Environment

In this course we will be using a commercial package of *Python* that is distributed free to Universities and students called *Canopy* (*Enthought Python Distribution*) that is based upon *Python* 2.7.3, this package contains all of the files and libraries that we will need for this course. Additional *modules* that contain specific task orientated solutions can be easily downloaded and incorporated into your *Python* code using the Canopy package manager tool. When looking at code examples it is important to note that there are two distinct variants of *Python*, namely 2.x (such as 2.7.3 that we are using) and 3.x. *Python* 3.x is the newer version which was created such that it is not backwards compatible with 2.x, as all the libraries that we require are coded in 2.x we will remain with this version. There are some syntax changes in 3.x so if you try to run a program written for 3.x in our distribution you most likely will encounter a syntax error, modules exist for converting the script or code of a 3.x program into one compatible with 2.x but they should not be required for this course. When running a program in *Python* you require two things:

- A development environment, where you edit (develop) the code for your programs
- An Interpreter, this is where *Python* translates the source code(that you wrote) into machine code which the computer can understand and run.

As with everything else there are numerous versions of both of these including specialist *Integrated DeveLopment Environments* such as *IDLE* (which you can find as part of the *enthought* distribution), as you become more experienced with *Python* you will discover an environment that best suits the way that you work within *Python*. At its most basic the code can be edited in any text editor (e.g. notepad / word) and the code can be executed from the Command Prompt in windows.

For this course the current teaching lab build is based upon Canopy, which is an integrated development environment, i.e it contains both an editor window and a interpreter window. It can be launched from the start menu via the physics specific applications, within the Enthought folder.

Fig. 1 shows a screenshot of what you should see if you have launched the correct version of Canopy. Commands can be typed directly into this shell, this is known as interactive mode and each line you type will be run upon pressing enter. This mode is useful for testing

small fragments of code, or using *Python* like a calculator and getting to know how things work. Features of *Python* that you will find useful are that the up/down arrow keys allow you to traceback through all the commands that you have typed into *Python*. The TAB key performs an Auto Complete of the command being input into the shell, if multiple possibilities are available a list of all commands that are compatible are printed above the command line.

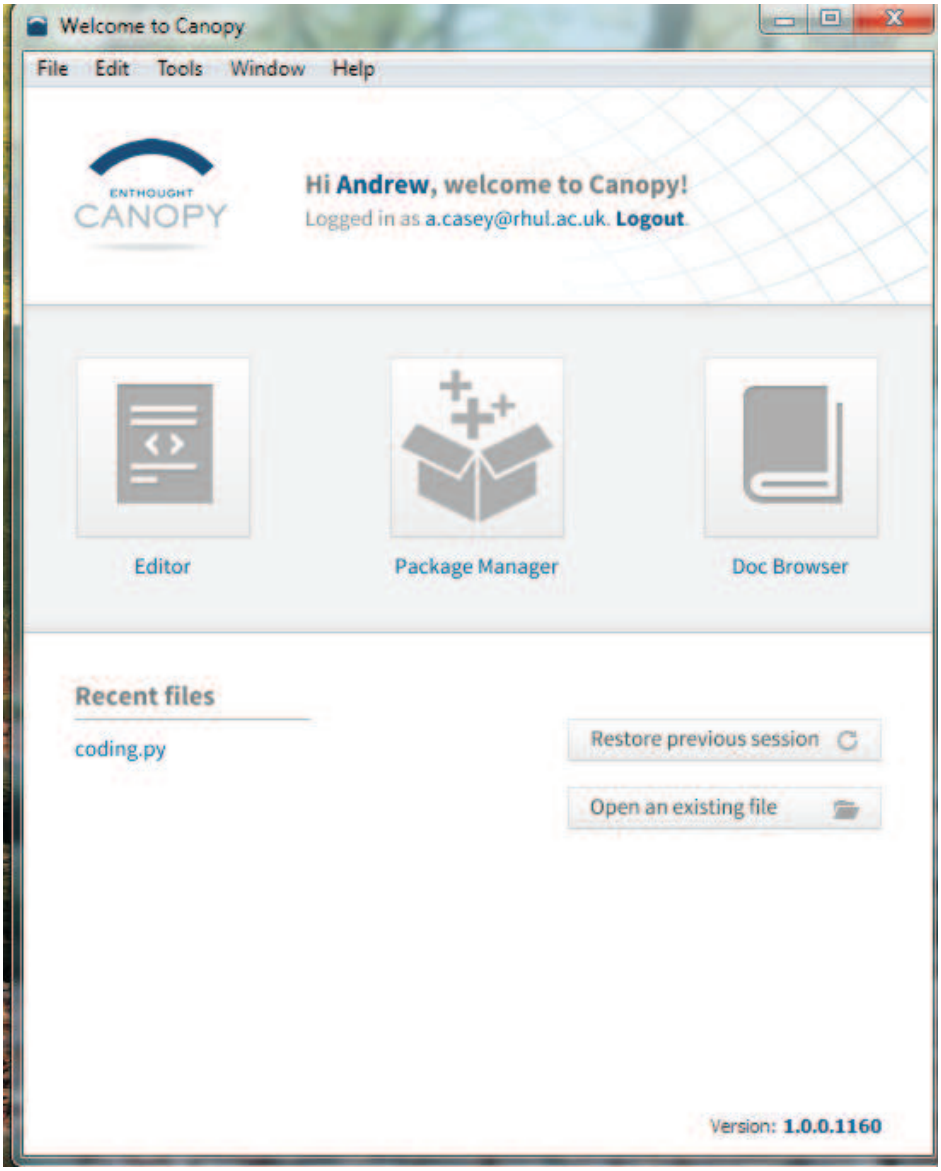

Fig. 1: Click on *Editor* to open the integrated development environment, *Package Manager* allows you to update and install new packages, *Doc Browser* provides links to online help

Although you can use any text editor as a development environment the formatting of *Python* code (in particular indents) is important in the way the code is interpreted. Therefore

it is unwise to use programs such as *Microsoft Word* to edit code as these programs perform hidden editing of the page-layout which is not always easily configured by the user. Instead it is preferable to use a text editor that has been designed for developing code, an alternative editor, *SciTe*, that is part of the distribution is shown in Fig. 2. This text editor is suitable for many languages, and by selecting *Python* from the drop down list your code will be automatically colour coded for greater readability and ease of error checking, for instance comments are shown in green. Code written in the editor can now be saved into your working directory and the script can be called from within the *interpreter*, this allows you to repeat a series of commands without having to type them directly into the *interpreter*.

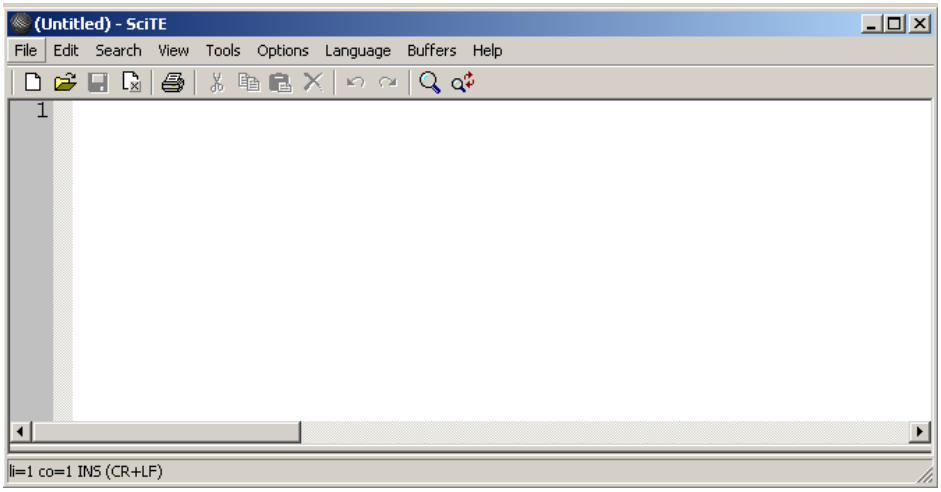

Fig. 2: An alternative development environment *SciTe*, is a generic text editor that has a program language specific colour coding of commands and comments etc that makes it easier to edit and debug code.

2.3 Installing *Python* at home

The distribution that we are using in the teaching lab *Canopy* is available for free to anyone with a *yourname@rhul.ac.uk* email address, enter your details at:

https://www.enthought.com/products/canopy/

The installer is available for Windows /Mac/Linux and SunOS in both 32 and 64 -bit versions, however the course has been designed to operate on the Windows based lab build (There are some issues with some packages on Mac versions).

In order to be able to access your y: drive from home within *Python* you will need to map the network drive into your computers configuration, instructions for doing this can be found on our website here: https://www.royalholloway.ac.uk/it/faq/itfaqs/networkconnectivity/mapdrive.aspx

3 Basic Programming

The following text should be read in conjugation with the exercises provided in the course, it is recommended that you visit the online tutorial at https://docs.python.org/2.7/tutorial/ for a comprehensive introduction to the basics.

3.1 Saving your code

It is very useful to stick to some rules regarding the file names of *Python* programs (and modules within *Python*), otherwise some things might go wrong unexpectedly. Always save the program with the extension .py. Do not put another dot somewhere else in the file name. Only use standard characters for file names: letters, numbers, dash (-) and underscore (_). White space (" ") should not be used at all (use underscores instead). Do not use anything other than a letter (particularly no numbers) at the beginning of a file name. Do not use"non-english" characters (such as ä,ø,è) in your file names, or (even better) do not use them at all when programming.

3.2 Comments

The *Python* interpreter ignores the contents of a line after the "#" symbol, this allows the programmer to add useful comments to remind themselves or explain to others what particular lines of code are supposed to be doing. It is good practice to include comments at the beginning of your programs to explain what the program does, attribute the author and record details of date and version of the program. This may seem unnecessary for the 3-line codes that you will be writing at the beginning of the course but it is a good habit to form. The comment can appear at any point in the line, after encountering the # *Python* moves onto interpreting the next line.

3.3 Operators

In the expression 1 + 2 is equal to 3, 1 and 2 are called operands and + is the operator. *Python* language supports the following types of operators:

- Arithmetic Operators
- Comparison Operators
- Assignment Operators
- Logical Operators
- Conditional Operators

Below the most common operators are defined, the arithmetic operators are represented by:

Operator	Description
+	Addition - Adds values on either side of the operator.
-	Subtraction - Subtracts right hand operand from left hand operand
*	Multiplication - Multiplies values on either side of the operator
/	Division - Divides left hand operand by right hand operand
%	Modulus - Divides left hand operand by right hand operand and returns remainder
**	Exponent - Performs exponential (power) calculation on operators
//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed.

Using these basic operators the *Python* shell can be used as a calculator (with traceback and auto-complete functionality), however this would be a fairly limited calculator. Typing dir() in *PyLab* lists all the named modules, functions variables etc, these built-in functions can be expanded by importing modules, the following example introduces how you can access the *math module* (more on *module*s in the next lecture):

```
# List the functions named within the module: math
import math
print dir(math)
# after the math module has been imported into the Python session you can use
    its functions
x= math.sqrt(5.0)
print x
help(math.sqrt) # returns help on the function sqrt in module math
#explore some of the othe functions contained in the math module
```

import_math_test.py

Using this as a starting point *Python* can be used as a powerful scientific calculator.

The comparison operators:

Operator	Description
==	Checks if the value of two operands are equal or not, if yes then condition becomes true.
!=	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.
<>	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.

The assignment operators:

Operator	Description
=	Simple assignment operator, Assigns values from right side operands to left side operand
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand.
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand
**=	Exponent AND assignment operator, Performs exponential (power) calculation on operators and assign value to the left operand.
//=	Floor Division and assigns a value, Performs floor division on operators and assign value to the left operand

The logical operators:

Operator	Description
and	Called Logical AND operator. If both the operands are true then then condition becomes true.
or	Called Logical OR Operator. If any of the two operands are non zero then then condition becomes true.
not	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.

3.4 Variables

A variable in computer programming is a storage location associated with a symbolic name that contains a known or unknown value. In *Python* the variable is assigned a value by using the "=" symbol.

>>> $x = 4$ # this is an assignment statement, where x is assigned the integer value 4

The variable name must start with a letter or an underscore. Within *Python* there are several types of variables that we will encounter, these are:

- Numbers
- Strings
- List
- Tuple
- Dictionary
- Boolean
- Object
- None (A special variable type, that means non existent, not known or empty)
- Array (more detail in section 9)

Where the numbers are subdivided into a further four types described next.

3.5 Numbers in *Python*

As a physicist one of our primary interests in *Python* is a tool for manipulating and visualising numbers, as such it is worth spending some time to understand how *Python* (and computers in general) treat numbers. This understanding should help you avoid some of the common pitfalls and be aware of the limitations in accuracy of a numerical program. *Python* has four basic numerical types. There are two types of integer *int* and *long* for integers greater than 32-bit, floating-point numbers *float* and complex floating-point numbers *complex*.

3.5.1 Integer Numbers

In *python* (interactive mode) try the following:

>>> $x = 42$

$>>> type(x)$ # returns the type of variable x

$< type\ 'int' >$

Typing that variable $x = 42$ is interpreted by *Python* as an integer, you do not have to explicitly declare the type by for example:

$>>> x = int(42)$

Strings and floats can be converted to integers by using the *int* constructor, try the following:

$>>> x = int('21')$

$>>> y = int(19.2)$

$>>> print\ \ x, y, x + y$

21 19 41

Here the first line converts the string '21' to an integer 21 (strings responded very differently to maths operations as we will see later) the second line converts the floating point number to an integer by chopping off everything after the decimal point (19.9 also returns 19). The print command returns the values of the variables. One of the common programming errors that novices make results from integer division, in PyLab try the following:
$>>> x = 17/3$

Results in an output of: 5

Which is correct for integer division, but might not have been what you intended a program to do, we will see that for a floating point number: $x = 17.0/3$

5.66666666667

3.5.2 Long Integer Numbers

Python uses at least 32 bits to represent integers, which means that you can store numbers at least as large as $2^{32} - 1$. If you need to store a larger number, *Python* provides the *long* type, which represents arbitrarily large integers. The long type uses more memory and therefore takes longer to perform calculations.

3.5.3 Floating Point Numbers

In *Python*, a floating-point number is represented by a $float$ object, in PyLab type:

$>>> x = 10.1$

$>>> type(x)$

$< float >$

Again you do not need to declare the type, if you want to have a floating point number with a value of 1 you need to type 1. or 1.0, if not *Python* will interpret the number as an integer. Sums involving integers and floats will result in floats. Very large and very small floating-point numbers are represented with exponential notation, which separates out the power of ten, these appear as e+100, in PyLab type:

$>>> x = 10.1 * *100$ # 10.1 to the power 100

$>>> x$ # Typing the variable name will return the value stored

$2.7048138294215165e + 100$

A floating-point number is only an approximation to a real number and is stored in the form: $sign * mantissa * base^{exponent}$

where $-1.0 < (sign * mantissa) < +1$ and $base^{an\ integer\ exponent}$. Therefore the decimal number could be represented as: $1234.56 = +0.123456 * 10^4$, typically computers would use a binary representation $base = 2$, modern computers will use 64-bits to store a float (53 bits for the mantissa and 11 for the exponent) yielding an accuracy of 16 decimal places. The nature of this representation can lead to some surprising consequences when using floats.

In *PyLab*:

$>>> x = 0.001$

$>>> y = (x - 1.0) + 1.0$

$>>> print x,\ y,\ x == y$# x==y is equality testing returning True/False

$< 0.001\ 0.001\ False >$

This outputs seems to contradict itself, but however highlights that what *Python* displays following a print command is not the full stored float value, but has been rounded to a more human friendly number. We can instruct *Python* to display more precision as follows:

$>>> print$"$\%.18f\ \%.18f$"$\%(x, y)$ # %.18f tells Pylab that you want 18 decimal places

Are the values still identical now?

This simple demonstration is meant to reinforce that a $float$ is an approximation that in most instances will give the right answer but the errors can be amplified by poor code, and can give unexpected answers in Boolean (True/False) comparisons. If you are dealing with a discrete number of real objects that can not be sub-divided the integer representation may be more appropriate.

3.5.4 Complex Numbers

A complex number is the sum of a real number and an imaginary number. In *Python* the imaginary component is represented as a number followed by a j.

3.6 Strings

Strings can be specified using single quotes or double quotes, the *string* data type stores a sequence of characters (for example words) individual characters can be addressed within the string as shown in the following example. Subsets of strings can be taken using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the string and working their way from -1 at the end. The plus (+) sign is the string concatenation operator, and the asterisk (*) is the repetition operator. Run the following examples in PyLab to understand the effect of operators on strings. The function str(x) converts the object x into a string data type. More on strings in the next set of notes.

```
str = 'Hello World!'

print str          # Prints complete string
print str[0]       # Prints first character of the string
print str[2:5]     # Prints characters starting from 3rd to 5th
print str[2:]      # Prints string starting from 3rd character
```

```
print str * 2      # Prints string two times
print str + "TEST" # Prints concatenated string
```

string_test.py

3.7 Lists

Lists are the most versatile of *Python*'s compound data types. A list contains items separated by commas and enclosed within square brackets ([ ]). Items belonging to a list can be of different data type. Like strings the values stored in a list can be accessed using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the list and working their way to end-1. The plus (+) sign is the list concatenation operator, and the asterisk (*) is the repetition operator. Run the following example in PyLab to understand the effect of operators on lists.

```
list = [ 'abcd', 100 , 3.14, 'Lev', 4+2j ]
tinylist = [123, 'Brian']

print list          # Prints complete list
print list[0]       # Prints first element of the list
print list[1:3]     # Prints elements starting from 2nd till 3rd
print list[2:]      # Prints elements starting from 3rd element
print tinylist * 2  # Prints list two times
print list + tinylist # Prints concatenated lists
```

list_test.py

3.8 Tuples

We have seen that lists and strings have many common properties, such as indexing and slicing operations. They are two examples of sequence data type a third type is the *tuple*. A *tuple* consists of a number of values separated by commas, for instance:

```
# tuple_test.py

first = (1, 2, 3)
second = (4, 5, 6)

print "len(first) : ", len(first)
print "max(first) : ", max(first)
print "min(first) : ", min(first)
print "first + second :", first + second
print "first * 3 : ", first * 3
print "1 in first : ", 1 in first
print "5 not in second : ", 5 not in second
first(2)=5 # this will create an error as you cannot change a value in a tuple
```

tuple_test.py

Running this test demonstrates one of the properties of tuples, that the values are immutable, which is useful if you want to store data (such as coordinates) that you want to remain fixed during the course of a program.

3.9 Dictionaries

A dictionary is basically an efficient table that maps *keys* to *values*. Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by *keys*, which can be any immutable type; strings and numbers can always be keys. Tuples can be used as keys if they contain only strings, numbers, or tuples. You can not use lists as keys. It is best to think of a dictionary as an unordered set of key: value pairs, with the requirement that the keys are unique (within one dictionary). A pair of braces creates an empty dictionary, {}. Placing a comma-separated list of key:value pairs within the braces adds initial key:value pairs to the dictionary; this is also the way dictionaries are written on output.

The main operations on a dictionary are storing a value with some key and extracting the value given the key. It is also possible to delete a key:value pair with del. If you store using a key that is already in use, the old value associated with that key is forgotten. The *keys()* method of a dictionary object returns a list of all the keys used in the dictionary, in arbitrary order (if you want it sorted, just apply the *sorted()* function to it). To check whether a single key is in the dictionary, use the *in* keyword.

Here is a small example using a dictionary:

```
# example of telephone numbers stored in a dictionary
tel = {'Andrew': 4351, 'Gill': 3506} #creates the dictionary
tel['Ian'] = 3483 # adds a key:value pair to the dictionary
print tel
# should return {'Gill': 3506, 'Ian': 3483, 'Andrew': 4351}
print tel['Andrew']
#should return 4351
del tel['Gill'] # deletes key:value pair
tel['Tim'] = 3448
print tel
# {'Tim': 3448, 'Ian': 3483, 'Andrew': 4351}
print tel.keys()
#['Tim', 'Ian', 'Andrew']
print sorted(tel.keys())
#['Andrew', 'Ian', 'Tim']
print 'Andrew' in tel
#True
```

dictionary_example.py

4 Input/Outputs

The simplest way to produce output (which we have encountered already) is using the *print* statement where you can pass zero or more expressions, separated by commas. This function

converts the expressions you pass it to a string and writes the result to standard output as follows:

```
print "Python should help me understand complex physics concepts,", "I hope!";
```

print_output_test.py

Returns "Python should help me to understand complex physics concepts, I hope!"
To receive input from a user of your program there exist two built in *Python* functions:

- *raw_input*, allows the user to input strings
- *input*, allows the user to input numbers and Booleans

The *raw_input*([prompt]) function reads one line from standard input and returns it as a string

```
str = raw_input("Enter your input: ");
print "Received input is : ", str
```

raw_input_test.py

This would prompt you to enter any string and it would display same string on the screen. If I type "I am confused", it's output would be "I am confused". The *input* function is equivalent to the *raw_input* except that it will evaluate the *Python* expression, i.e it will perform operations on the input value before returning an answer, for example:

```
age = float(raw_input("Enter your age: ")); # input statement is not defined
    on Canopy
print "You are old enough to buy alcohol: ", age>18.0
```

input_test.py

Returns "You are old enough to buy alcohol: True", if the user entered an age greater than 18.

4.1 Opening and Closing Files:

Until now, you have been reading and writing to the standard input and output. Now we will see how to deal with actual data files. *Python* provides basic functions and methods necessary to manipulate files by default. Most of the file manipulation can be done using a **file** object.

4.2 The *open* Function:

Before you can read or write a file, you have to open it using *Python*'s built-in *open*() function. This function creates a file object which would be utilized to call other support methods associated with it.

```
file object = open(file_name [, access_mode][, buffering])
#The file_name: argument is a string value that contains the name of the file
    that you want to access.
#access_mode: The access_mode determines the mode in which the file has to be
    opened ie. read, write append etc. A complete list of possible values is
    given below in the table. This is optional parameter and the default file
    access mode is read (r)
#buffering: If the buffering value is set to 0, no buffering will take place.
    If the buffering value is 1, line buffering will be performed while
    accessing a file.
```

file_syntax_test.py

Here is a table of some of the more common used modes for opening a file:

Mode	Description
r	Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
w	Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing
a	Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.

Once a file is opened and you have one *file* object, you can get various information related to that file. It is good practice to close the file using close() after you have finished intergating it, for example

```
# Open a file
fo = open("Physics.txt", "w")
print "Name of the file: ", fo.name

# Close opend file
fo.close()
```

openclose_syntax_test.py

The file object provides a set of access methods to extract or insert information. We will see how to use read() and write() methods to read and write files.

4.3 The write() Method:

The *write*() method writes any string to an open file. It is important to note that *Python* strings can have binary data and not just text.

Syntax: fileObject.write(string); Here passed parameter is the content to be written into the open file.

Example:

```
# Open a file
fo = open("foo.txt", "wb")
fo.write( "Python is a great language.\nYeah its great!!\n");
# Close opend file
fo.close()
```

open_syntax_test.py

The above method would create foo.txt file and would write given content in that file and finally it would close that file. If you would open this file, it would have following content

Python is a great language. Yeah its great!!

4.4 The read()Method:

The read() method reads a string from an open file. It is important to note that *Python* strings can have binary data and not just text.

Syntax: fileObject.read([count]); Here the passed parameter is the number of bytes to be read from the opened file. This method starts reading from the beginning of the file and if count is missing then it tries to read as much as possible, may be until the end of file. Example:

```
# Open a file
fo = open("foo.txt", "r+")
str = fo.read(20);
print "Read String is : ", str
# Close opend file
fo.close()
```

read_syntax_test.py

This produces the following result: "Python is"

5 Formatting

At this stage in the course the programs are going to start to get more complex, so it is worth raising the issue of programming style now. The originators of *Python* came to the conclusion that code is read more often than it is written, and hence the readability of a program is of high importance. The following link is to a document that summarises the community accepted style guide. Writing programs consistent with the style guide makes the program more readable and easier to export to other *Python* users. http://www.python.org/dev/peps/pep-0008/ *Style Guide for Python Code*

You should download and save a copy of the style guide. Some of the ethos has been captured in this text by Tim Peters,

The Zen of *Python*

Beautiful is better than ugly.

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

6 Control Structures

So far the programs that we have considered have had a linear flow through the instructions, the syntax of control structures allows us to break that linear flow, for example by iterating an operation until a condition is met (*while* and *for* loops) or branching into multiple paths (*If* Statements). The following section briefly introduces all three structures, but to gain a more comprehensive understanding you must practice writing examples and look at online sources or text books.

6.1 The *while* loop

A while loop executes repeatedly a set of statements as long as a boolean condition is true:

```
while condition: # :marks the beginning of the indented block of statements
    < statement1 >
    < statement2 >
    < ..... >
<first statement after loop >
```

All statements in the loop must be indented. The loop ends when an unindented statement is encountered.

An example of a while loop that repeats the algorithm until the conditions is met.

```
print '-----------------------'     # table heading
C = -20                           # start value for C
dC = 5                            # increment of C in loop
while C <= 40:                    # loop heading with condition
    F = (9.0/5)*C + 32            # 1st statement inside loop
    print C, F                    # 2nd statement inside loop
    C =+ 1                  # 3rd statement inside loop
print '-----------------------'     # end of table line (after loop)
```

whileloop.py

6.2 The *for* loop

The Nature of For Loops. When data are collected in a list, we often want to perform the same operations on each element in the list. We then need to walk through all list elements. Computer languages have a special construct for doing this conveniently, and this construct is in *Python* and many other languages called a for loop. Let us use a for loop to print out all list elements:

```
Cdegrees = [-20, -15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40]
print '    C    F'
for C in Cdegrees:
    F = (9.0/5)*C + 32
    print '%5d %5.1f' % (C, F) # see how a headline is inserted into the list
```

for_loop_example.py

6.3 The *if* statement

The flow of a computer program is often required to branch, i.e. if a condition is met it will carry out one set of instructions and if not it will perform another set of instructions. In *Python* this is carried out by the *if-else* structure. The general structure is:

if condition: # :marks the beginning of the indented block of statements
    < block of statements, executed if condition is True >
else:
    < block of statements, executed if condition is False >

With the keyword *elif*, short for *else if*, we can create several mutually exclusive *if* tests which allow for multiple branching of the program flow:
if condition:
    < block of statements >

elif condition2:
    < block of statements >
elif condition3:
    < block of statements >
else:
    < block of statements >

The following example shows how a *function* can be defined with multiple expressions in different ranges using the *elif* statement.

```
x=0.5
if x < 0:
    print x
elif 0 <= x < 1:
    print x*2
print 'z'



#         return 0.0
#     elif 0 <= x < 1:
#         return x
 #    elif 1 <= x < 2:
 #        return 2 - x
#    elif x >= 2:
 #       return 0.0
```

elif_example.py

7 Functions, Classes, Modules, Packages

Some definitions to help you keep track:

- **A Variable:** Attaching a name to a value.
- **A Statement:** a block of code can be used for assignment, control flow, exceptions, function definition.
- **A Function:** A series of statements which returns some value to a caller. It can also be passed zero or more arguments which may be used in the execution of the body.
- **An Argument:** A value passed to a function or method, assigned to a named local variable in the function body.
- **A Class:** A template for creating user-defined objects. Class definitions normally contain method definitions which operate on instances of the class

- **A Method:** A function which is defined inside a class body.
- **An Attribute:** A variable which is defined inside a class body.
- **A Module:** A file containing *Python* definitions (functions, variables and objects) themed around a specific task. The module can be imported into your program. The convention is that all import statements are made at the beginning of your code.
- **A Package:** Is a set of modules or a directory containing modules linked through a shared theme. Importing a package does not automatically import all the modules contained within it. Some useful packages: *SciPy* (maths, science and engine ring module), *Numpy* (Numerical *Python* for arrays, fourier transforms and more), *matplotlib* (graphical representation), *mayavi* (3D visualisation module in Enthought distribution)

7.1 Functions

So far we have encountered some of *Python*'s built-in functions, these are functions that are always available such as print () or input (). We have also seen how functions can be imported from a module such as *math*,

>>> *from math import* * # imports all of the functions contained in the math module
>>> *dir*(*math*) #lists all of the functions within math

A complete list of available functions and how to use them can be found in the *Python* help files or in the http://docs.python.org/index.html *Python* documentation v2.7.3.

This section outlines how to define you own functions.

```
#Fibonacci_example.py

def fib(n):    # write Fibonacci series up to n
    """Print a Fibonacci series up to n.
    """ # the information entered here is returned by help(fib)
    a, b = 0, 1
    while a < n:
        print a, #, makes print stay on the same line
        a, b = b, a+b


# Now call the function we just defined:
fib(2000)
# returns 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

fibonacci_example.py

- Function = a collection of statements we can execute wherever and whenever we want

- Function can take input objects and produce output objects
- Functions help to organize programs, make them more understandable, shorter, and easier to extend

The keyword *def* introduces a function definition. It must be followed by the function name and the parenthesized list of formal parameters. The statements that form the body of the function start at the next line, and must be indented (4 spaces). The first statement of the function body can optionally be a string literal; this string literal is the function's documentation string, or docstring, and can be accessed through the help(function_name) statement. The function can be modified to return a list of numbers, rather than printing them:

```
#Fibonacci_example2.py

def fib2(n):    # write Fibonacci series up to n
    """Return a list containing the Fibonacci series up to n."""
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)
        a, b = b, a+b
    return result

# Now call the function we just defined:
f100=fib2(100)
f100                # write the result
# [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

fibonacci_example2.py

Here the return statement returns with a value from a function. The statement *result.append(a)* calls a method of the list object result. A *method* is a *function* that 'belongs' to an *object* and is named *obj.methodname*, where *obj* is some object (this may be an expression), and *methodname* is the name of a method that is defined by the object's type. Different types define different methods. Methods of different types may have the same name without causing ambiguity. (It is possible to define your own object types and methods, using *classes*) The method *append()* shown in the example is defined for list objects; it adds a new element at the end of the list. In this example it is equivalent to result = result + [a], but more efficient. Working through the exercises with reference to online material will aid in your understanding of implementing functions in your programs.

8 Object Orientated Programming OOP

Object orientated programming, OOP, is a style of programming that is supported by *Python*, you will come across many references to it while looking at the literature. The goal of OOP is to more efficiently organise code and secondly to encourage the re-use of code.

The concept and its application are the subject of many books, we will return to it later in the course but at this stage I will highlight the some of the syntax of objects (that you have already encountered). Within *Python* nearly everything can be thought of as an object, from the *Python* Language Reference *"Objects are* Python*'s abstraction for data. All data in a* Python *program is represented by objects or by relations between objects. (In a sense, and in conformance to Von Neumann's model of a "stored program computer", code is also represented by objects.) Every object has an identity, a type and a value. An object's identity never changes once it has been created; you may think of it as the object's address in memory."*

So you have already created and operated on objects, the following example demonstrates the object-like properties of lists, write a similar program and run in Canopy.

```
# An illustration of the object syntax related to lists
testlst = [1,2,3,4,5,6] # create a list object
print dir(testlst)
""" All objects have their own namespace containing all variable and function
names that are defined for that object. The dir(object_name) function
returns a list of all names defined for the object."""
#
#
print testlst
testlst.append(7) # here we are using the '.' (dot) operator to append an
    element to the list
print testlst
testlst.reverse() # reverses order of list
print testlst
testlst.index (4) # returns the 4th element of the list
```

object_syntax.py

In this example you can read the '.' *dot* operator as: " ask object *testlst* to *do* something" as: append a element to the list; reverse the elements in the list or return the value at a given index. The dot operator can access the namespace of the object (testlst). Variables and functions defined in object namespaces are called *attributes* and *methods* of the object. In the case of testlst.append(x), .append is a method acting on the list object.

9 Numerical Scientific Computing using *NumPy*

NumPy is the fundamental package for scientific computing with *Python*. It contains among other things:

- a powerful N-dimensional array object
- sophisticated (broadcasting) functions
- tools for integrating C/C++ and Fortran code
- useful linear algebra, Fourier transform, and random number capabilities

These notes will only cover the basic of the usage of *NumPy*, a fuller understanding can be found in this book by one of the developers of *NumPy*:
http://www.tramy.us/numpybook.pdf or via the *SciPy Numpy* website http://numpy.scipy.org/ and associated links.

9.1 Arrays

As we have seen, a list in *Python* is an ordered set of values, such as a set of integers or a set of floats. There is another object in *Python* that is somewhat similar, an array. An array is also an ordered set of values, but there are some important differences between lists and arrays:

- The number of elements in an array is fixed. You cannot add elements to an array once it is created, or remove them.
- The elements of an array must all be of the same type, such as all floats or all integers. You cannot mix elements of different types in the same array and you cannot change the type of the elements once an array is created.

Lists, as we have seen, have neither of these restrictions and, on the face of it, these seem like significant drawbacks of the array. Why would we ever use an array if lists are more flexible? The answer is that arrays have several significant advantages over lists as well:

- Arrays can be two-dimensional, like matrices in algebra. That is, rather than just a one-dimensional row of elements, we can have a grid of them. Indeed, arrays can in principle have n-dimensions. Lists, by contrast, are always just one-dimensional containers.
- Arrays behave roughly like vectors or matrices: you can do arithmetic with them, such as adding them together, and you will get the result you expect. As we have seen this is not true with lists, addition results in concatenation of the list.
- Arrays work faster than lists. Especially if you have a very large array with many elements then calculations may be significantly faster using an array.

An important note is the that *Numpy*'s array class is called *ndarray*. It is also known by the alias array. Note that *numpy.array* is not the same as the Standard *Python* Library class *array.array*, which only handles one-dimensional arrays and offers less functionality. *Numpy* builds on earlier array package *Numeric* and older versions of text books or examples may refer to this. In *Numpy* dimensions are called axes. The number of axes is rank. For example, the coordinates of a point in 3D space [1, 2, 1] is an array of rank 1, because it has one axis. That axis has a length of 3. In example pictured below, the array has rank 2 (it is 2-dimensional). The first dimension (axis) has a length of 2, the second dimension has a length of 3.

```
#Creating Arrays in Numpy

import numpy as np
x=np.array([[1,2,3,4],[5,6,7,8]])
print 'size of array', np.size(x)
print 'shape of array', np.shape(x)
print 'x=', x

y=np.linspace(0,1,20) # from 0 to 1 in 20 evenly spaced steps
print 'y=',y
z=np.arange(1,10,0.1) #from 1 to 10 in steps of 0.1
print 'z=',z
zz=z.reshape(30,3) # reshape the 1,90 array into a 30,3 array
print zz
```

array_ex1.py

Some of the attributes of an array are:

ndarray.ndim the number of axes (dimensions) of the array. In the *Python* world, the number of dimensions is referred to as rank.

ndarray.shape returns the dimensions of the array. This is a tuple of integers indicating the size of the array in each dimension. For a matrix with n rows and m columns, shape will be (n,m). The length of the shape tuple is therefore the rank, or number of dimensions, $ndim$.

ndarray.size the total number of elements of the array. This is equal to the product of the elements of shape.

ndarray.dtype an object describing the type of the elements in the array. One can create or specify dtype's using standard *Python* types. Additionally *NumPy* provides types of its own. numpy.int32, numpy.int16, and numpy.float64 are some examples.

ndarray.itemsizethe size in bytes of each element of the array. For example, an array of elements of type float64 has itemsize 8 (=64/8), while one of type complex32 has itemsize 4 (=32/8). It is equivalent to ndarray.dtype.itemsize.

ndarray.data the buffer containing the actual elements of the array. Normally, we won't need to use this attribute because we will access the elements in an array using indexing facilities.

10 Further Reading

With all further reading please remember that we are using the 2.x variant of *Python*.

10.1 Books

There are many free e-books available for learning how to program in *Python*:

- http://files.swaroopch.com/python/byteofpython_120.pdf*A Byte of Python* by C.H.Swaroop

- http://learnpythonthehardway.org/book/ *Learn Python the Hard Way* by Zed A.Shaw(By which the author means typing lots of examples) has a free *html* version

Some (not-so-free) books, I have highlighted these books as unlike some of the other books available they were not written for computer scientists:

- *A Primer on Scientific Programming with Python*, by Hans Petter Langtangen, Springer 2012
- *Numerical Methods in Engineering with Python* Jaan Kiusalaas 2nd Ed 2010
- *Introduction to Numerical Programming: A Practical Guide for Scientists and Engineers Using Python and C/C++* Titus Beu, CRC Press 2014

10.2 Useful Websites

As with other languages there is a large community of users who post their problems and solutions onto a number of websites and forums, the open source nature of *Python* encouraged its use and hence there is a large resource of material that can be tapped into, as physicists we do not want to spend our time re-inventing the wheel. The following websites can be used as staring points when exploring this information:

http://docs.python.org/index.htmlPython v2.7.3 documentation

http://numpy.scipy.org/*numpy* is a fundamental package for scientific computing with *Python* building on *numpy* the package *scipy* (pronouced "sigh-pie") provides many resources for scientific computing http://www.scipy.org/

http://www.enthought.com/ the commercial distribution of *Python* that we are using in the teaching lab.

http://en.wikibooks.org/wiki/Python_Programming a wikibooks *Python* Guide

http://en.wikibooks.org/wiki/Non-Programmer%27s_Tutorial_for_Python_2.6 another wikibook

http://http://stackoverflow.com/ forum where people post questions and solutions to programming problems, authors do not always agree on the best solution.

https://github.com/python another programming forum.