

# **Introduction to programming for engineers using Python**

*by*

**Logan G. Page  
Daniel N. Wilke  
Schalk Kok**

Updated:  
December 2012



**UNIVERSITEIT VAN PRETORIA  
UNIVERSITY OF PRETORIA  
YUNIBESITHI YA PRETORIA**

# Table of contents

<b>Table of contents</b>	<b>ii</b>
<b>Chapter 1: Introduction</b>	<b>1</b>
1.1 Computer components . . . . .	1
1.2 What is a computer program? . . . . .	2
1.2.1 Examples of computer programs . . . . .	2
1.3 Developing a program . . . . .	3
1.3.1 Flowcharts . . . . .	3
1.4 Teaching philosophy . . . . .	4
1.5 Formats used in these notes . . . . .	5
<b>Chapter 2: Python introduction</b>	<b>7</b>
2.1 Installing Python . . . . .	8
2.2 Using Python(x,y) . . . . .	10
2.3 Interactive Python Console . . . . .	11
2.3.1 Importing Modules, Functions and Constants . . . . .	14
2.3.1.1 Importing Modules . . . . .	15
2.3.1.2 Importing Functions . . . . .	16
2.3.1.3 Importing Modules (The Wrong Way) . . . . .	17
2.3.2 Math Module Computations . . . . .	18
2.3.3 Help Function . . . . .	19
2.3.3.1 Help for Modules and Functions - “ <code>help(function)</code> ” . . . . .	20
2.3.3.2 Interactive Help - “ <code>help()</code> ” . . . . .	22
2.4 Programming Mode (Spyder) . . . . .	24
2.4.1 Navigating the directory structure from Python . . . . .	26
2.4.1.1 Summary . . . . .	31
2.4.2 Python Script Names . . . . .	31
2.5 Documentation . . . . .	33
2.6 Supplementary Sources . . . . .	35
2.7 Exercises . . . . .	36
<b>Chapter 3: Names and Objects</b>	<b>39</b>
3.1 Numerical Objects . . . . .	39
3.1.1 Memory Model . . . . .	41
3.1.2 Example: Swapping objects . . . . .	43

3.1.3 Assignment Operators . . . . .	45
3.1.4 Integers vs. Floats . . . . .	46
3.1.4.1 Why should you care about this integer division behaviour? . . . . .	48
3.1.4.2 How to avoid integer division . . . . .	48
3.2 String Objects . . . . .	49
3.3 Boolean Objects . . . . .	51
3.4 Acceptable Names . . . . .	52
3.5 Summary . . . . .	53
3.6 Exercises . . . . .	54
<b>Chapter 4: Unconditional loop</b>	<b>57</b>
4.1 The <code>range</code> function . . . . .	59
4.2 The <code>for</code> loop statement . . . . .	60
4.2.1 Example 1 . . . . .	61
4.2.2 Example 2 . . . . .	62
4.2.3 Example 3 . . . . .	63
4.2.4 Example 4 . . . . .	63
4.2.5 Indentation . . . . .	64
4.3 Summing using a <code>for</code> loop . . . . .	66
4.4 Factorial using a <code>for</code> loop . . . . .	67
4.5 Fibonacci series using the <code>for</code> statement . . . . .	69
4.6 Taylor series using the <code>for</code> loop statement . . . . .	72
4.7 Exercises . . . . .	76
<b>Chapter 5: Conditional loop</b>	<b>79</b>
5.1 The <code>while</code> loop statement . . . . .	79
5.2 Conditions . . . . .	80
5.2.1 Conditions as booleans . . . . .	81
5.2.2 Conditions as questions . . . . .	82
5.2.3 Examples of conditions . . . . .	84
5.3 Simulating a <code>for</code> loop statement using a <code>while</code> loop statement . . . . .	85
5.4 Taylor series using the <code>while</code> loop statement . . . . .	86
5.5 Exercises . . . . .	89
<b>Chapter 6: Branching</b>	<b>93</b>
6.1 The <code>if</code> statement . . . . .	93
6.2 Simple example . . . . .	97
6.3 Leg before wicket example . . . . .	98
6.4 Number guessing game . . . . .	101
6.5 The <code>if</code> statement with combined conditions . . . . .	106
6.6 Exercises . . . . .	107
<b>Chapter 7: Additional examples and statements</b>	<b>111</b>
7.1 Additional statements . . . . .	111
7.1.1 <code>lambda</code> function . . . . .	111

---

7.1.2	raw_input statement . . . . .	113
7.1.3	String manipulation . . . . .	116
7.1.3.1	The string plus (+) operator . . . . .	116
7.1.3.2	The string multiply (*) operator . . . . .	117
7.1.3.3	The string formatting operator (%) . . . . .	118
7.1.3.4	String functions . . . . .	121
7.2	Additional examples . . . . .	122
7.2.1	Limit of the natural logarithm . . . . .	122
7.2.2	Numerical differentiation . . . . .	124
7.2.3	Solving a non-linear equation: Newton's method . . . . .	129
7.2.3.1	Example 1 . . . . .	134
7.2.3.2	Example 2 . . . . .	135
7.2.3.3	Example 3 . . . . .	135
7.2.3.4	Example 4 . . . . .	136
7.2.3.5	Example 5 . . . . .	136
7.2.4	Newton's method using numerical gradients . . . . .	138
7.2.5	The bisection method . . . . .	141
<b>Chapter 8:</b>	<b>Numerical integration</b>	<b>147</b>
8.1	Numerical integration using a while loop statement . . . . .	151
8.2	Exercises . . . . .	157
<b>Chapter 9:</b>	<b>Data containers</b>	<b>159</b>
9.1	Available Data Containers in Python . . . . .	159
9.1.1	Lists . . . . .	159
9.1.2	Memory model . . . . .	161
9.1.2.1	Joining lists – the plus operator (+) . . . . .	162
9.1.2.2	Indexing a list . . . . .	163
9.1.2.3	The append and insert functions . . . . .	164
9.1.2.4	Looping through a list . . . . .	167
9.1.2.5	Disadvantage of using lists . . . . .	167
9.1.2.6	Working with lists – examples . . . . .	169
9.1.3	Tuples . . . . .	171
9.1.3.1	Disadvantages of using a tuple . . . . .	172
9.1.3.2	Working with tuples – examples . . . . .	173
9.1.4	Dictionaries . . . . .	174
9.1.4.1	Adding to a dictionary . . . . .	176
9.1.4.2	Looping through a dictionary . . . . .	177
9.1.4.3	Disadvantage of using dictionaries . . . . .	177
9.1.4.4	Working with dictionaries – examples . . . . .	178
9.1.5	Summary . . . . .	179
9.2	Revisiting previous programs . . . . .	180
9.2.1	Taylor series using lists . . . . .	180
9.2.2	Numerical integration using lists . . . . .	182

---

9.3	Sorting algorithms . . . . .	186
9.3.1	Bubble sort algorithm . . . . .	188
9.3.2	Containers inside containers . . . . .	192
9.4	Exercises . . . . .	195
<b>Chapter 10: Vectors and matrices</b>		<b>197</b>
10.1	Simple Computations with vectors and matrices . . . . .	201
10.2	Displaying the components of a vector . . . . .	204
10.3	Displaying the components of a matrix . . . . .	205
10.4	Defining a matrix using the <code>raw_input</code> statement . . . . .	208
10.5	Example: Norm of a vector . . . . .	209
10.6	Example: Dot product . . . . .	211
10.7	Example: Matrix-vector multiplication . . . . .	213
10.8	Example: General vector-vector multiplication . . . . .	216
10.9	Gauss elimination . . . . .	218
10.9.1	Gauss elimination algorithm . . . . .	220
10.9.1.1	Forward reduction step . . . . .	221
10.9.1.2	Back-substitution step . . . . .	223
10.10	Solving linear systems using Python . . . . .	226
10.11	Exercises . . . . .	230
<b>Chapter 11: Functions</b>		<b>235</b>
11.1	Creating new functions . . . . .	236
11.1.1	Example 1 . . . . .	238
11.1.2	Example 2 . . . . .	239
11.2	Functions (Continued) . . . . .	240
11.3	Function with Optional Inputs . . . . .	245
11.4	Gauss elimination using functions . . . . .	247
11.5	Numerical integration using functions . . . . .	251
11.6	Comment statements and code documentation . . . . .	254
11.7	Exercises . . . . .	255
<b>Chapter 12: File handling</b>		<b>259</b>
12.1	The <code>numpy.save</code> and <code>numpy.load</code> commands . . . . .	259
12.2	The <code>numpy.savetxt</code> and <code>numpy.loadtxt</code> commands . . . . .	261
12.3	The <code>csv</code> module . . . . .	264
12.4	Additional methods to read and write data . . . . .	265
12.5	Exercises . . . . .	267
<b>Chapter 13: Graphs</b>		<b>269</b>
13.1	2D Graphs . . . . .	269
13.1.1	Graph annotation . . . . .	271
13.1.2	Multiple plots and plot options . . . . .	274
13.1.3	Superscripts, subscripts and Greek letters . . . . .	279
13.1.4	Other 2D graphs . . . . .	282

---

13.2 3D Graphs . . . . .	282
13.2.1 The <code>plot3D</code> command . . . . .	283
13.2.2 The <code>plot_wireframe</code> and <code>plot_surface</code> commands . . . . .	284
13.3 Subplots . . . . .	289
13.4 Exercises . . . . .	292
<b>Chapter 14: Exception Handling</b>	<b>299</b>
14.1 Text menus and user input . . . . .	299
14.2 Error handling and user input . . . . .	301
<b>Chapter 15: More ways to loop</b>	<b>305</b>
15.1 Recursion: using functions to loop . . . . .	305
15.2 Exercises . . . . .	308
<b>Chapter 16: numpy and scipy capabilities</b>	<b>311</b>
16.1 Solving systems of equations . . . . .	311
16.1.1 Solving linear systems of equations . . . . .	311
16.1.2 Solving overdetermined linear systems of equations . . . . .	312
16.1.3 Solving underdetermined linear systems of equations . . . . .	314
16.1.4 Solving Non-Linear systems of equation . . . . .	316
16.1.5 Equilibrium of non-linear springs in series . . . . .	321
16.2 Polynomials in Python . . . . .	326
16.3 Numerical integration . . . . .	334
16.4 Solving a system of linear differential equations . . . . .	338
16.5 Optimization . . . . .	341
16.6 Exercises . . . . .	347
<b>Chapter 17: What's Next</b>	<b>351</b>
17.1 Python(x,y) Packages . . . . .	352
17.2 Installing New Packages . . . . .	352

# Chapter 1

## Introduction

Welcome to the “Introduction to Programming” course. In this course, you will develop the skills required to develop computer programs in Python. Python is an interpreter (similar to translating german to english) that translates a program or script (readable by humans) into instructions that can be understood and executed by the computer. Python is freely available and free of charge (Open Source) as opposed to other programming languages like Matlab, which is commercially available.

The course will focus on the types of programs engineers require i.e. programs that compute, analyse and simulate a particular system. I use mathematics to teach programming because mathematics require the same logic as programming. I’m of the opinion that if you are good at (applied) mathematics, you will be a good programmer. Furthermore, if you are capable of turning mathematical statements into a program, you will be able to translate any instructions into a program.

### 1.1 Computer components

Before we start to develop new programs, let’s quickly review what equipment we’ll need. The most obvious is the computer itself. A computer consists of

1. A central processing unit (CPU). This is the brain of the computer. It performs all the computations required to perform a certain task. The Intel Pentium iX CPU is most common today.
2. Memory. Personal computer (PC) memory is referred to as RAM (Random Access Memory). The contents of the memory is wiped out if the computer is switched off.

3. Storage device. Usually a permanently mounted magnetic hard drive. The data stored on this device is stored even after the computer is switched off. You save all your files to such a device.

An operating system manages the above components to perform required tasks. Windows XP, 7 and Ubuntu are examples of modern operating systems.

## 1.2 What is a computer program?

A computer program is a list of instructions for the computer to execute / run. Computers are perfect slaves: they perform our tasks without complaint and exactly according to our instructions. Above all, they perform these tasks extremely fast and accurate (they never forget their instructions or their multiplication tables). This implies of course that you need to know exactly which instructions to give in order to obtain the required result. If you don't exactly understand what you want to do and how to do it, your list of instructions to the computer (i.e. your program) will contain logical errors. The computer will execute these flawed instructions exactly as stated, and will obtain the wrong result. This is the reason for one of the famous computer acronyms GIGO (Garbage In, Garbage Out).

### 1.2.1 Examples of computer programs

You should already be familiar with many computer programs. The suite of Microsoft Office programs are programs used to create documents, spreadsheets and presentations. Games are programs for entertainment. Those of you familiar with the Windows operating system, a long list of programs appear if you click on the **Start** menu, followed by **Programs**.

The components of a computer program are

- Input.
- Processing.
- Output.

There can be multiple inputs, processing and outputs. Before any processing can take place, the necessary inputs have to be obtained. In this course, the processing

part is what we will focus most of our attention on. Given certain inputs, we have to give explicit instructions on how to manipulate these inputs to provide a required result. Once the processing is complete, we can provide outputs. The loop can then repeat itself, if necessary.

Even a computer game contains these components. Input is generally obtained from the keyboard, mouse or game controller (joystick), some processing is then performed and output is generated to the screen (in the form of detailed graphics). This input, processing, output loop runs continuously, or at least until you quit the game.

## 1.3 Developing a program

Just because you can use programs that other people have developed, does not mean that you can develop programs yourself. It is a completely different scenario to be the user of a program versus the developer. As a student that has to master the skill of developing new programs, you will have to follow a few basic steps.

1. Get a clear idea of what your program must do. If the program is complicated, break it into smaller chunks. Make sure you know what task each of these chunks must perform.
2. Decide what inputs are required for each of the tasks that your program must perform. Acquire these inputs.
3. Process the inputs as required. The processing could be as simple as sorting a list of inputs, or could be very complicated e.g. the numerical simulation of a large system.
4. Produce the required output. Your program is useless if it processes the inputs correctly, but produces improper or incomplete outputs. The ultimate test of your program is if it produces the correct output.

### 1.3.1 Flowcharts

It is common practise to use flowcharts during the initial program development stages. Flowcharts are schematic representations of a computer program that illustrate the flow of information through this program. A large number of flowchart symbols can be used, but for the sake of simplicity we will only use the following three symbols:

- Oval: Indicates the start or the end of the program.



- Diamond: Indicates a decision or branch in the program.



- Rectangle: All other commands and/or statements.



If you have difficulty developing a program directly (i.e. you generate code as you figure out your program), try using flowcharts. It is sometimes very useful to organise your thoughts. It also provides a schematic outline of your program, which can help to keep the big picture in mind. Flowcharts are not really necessary to develop small programs that produce few outputs, but as the complexity of the program increases flowcharts become indispensable.

An example of a flowchart is depicted in Figure 1.1. The program reads a percentage and checks if it is greater than 50%. If so, it assigns some symbol. If not, it assigns the F symbol. The program then displays the chosen symbol. The details of program is not important here: I'm only trying to give you an example of a flowchart.

It should be clear from the flowchart above that this example has been broken up into two chunks (assign the F symbol; and compute other symbol) and only the first chunk of the program has been outlined with the flowchart. The second chunk `compute symbol` could also be outlined with a flowchart, which would have the inputs ('yes', 'percentage') and give the output ('symbol').

## 1.4 Teaching philosophy

The main goal for this module is to acquire basic programming skills. These skills can only be acquired through practise. It is no use to only use existing programs or to study the examples of other programmers. You have to repeat all the examples in the lectures and notes in Python yourself, and not just simply read or copy them. Try to understand what concept is being taught and then try write a particular Python program from scratch, and only if you get completely stuck, refer to the class example. Just study the example long enough to get going, then close your notes and try to continue.

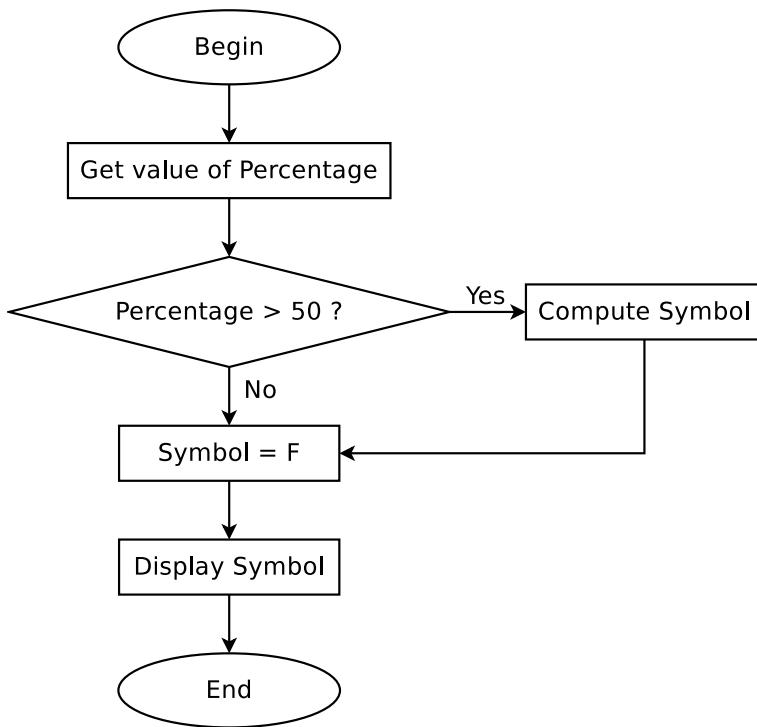


Figure 1.1: Example of flowchart

Furthermore, you will not be allowed to make use of Python's extensive list of modules and functions. Rather, we will develop our own versions of existing Python functions and compare our results to that of Python. You will not acquire the necessary programming skills if you only use existing programs. During tests and exams, the instructions will be clear enough to distinguish whether or not specific Python modules and / or functions may be used.

## 1.5 Formats used in these notes

In these notes I will make use of different text formatting as well as differently formatted blocks of text or code. Below, I have listed the different formats used in these note as well as an explanation of there purpose:

*Application* - This in-text formatting will be used to refer to an application or software package.

`code snippet` - This in-text formatting will be used to refer to a small piece of code, variable name or command.

**More Info:**

This formatted text block will be used to provide you with additional information regarding a certain topic being discussed.

**Take Note:**

This formatted text block will be used for important information that you must be aware of. It is recommended that you take a moment to fully understand the information given in this type of text block and where possible practise the concepts being discussed.

The following formatted block will be used for program outputs as well as programs typed in the *IPython Console*. This block will have line numbers on the left side of the frame and a light grey font colour inside the frame.

```
1 In [1]: from math import sin  
2  
3 In [2]:
```

The following formatted block will be used for programs typed up in the *Spyder*. This block will have line numbers on the left side of the frame and colour highlighted code inside the frame.

```
1 for i in range(10):  
2     print i  
3 print "finished"
```

# Chapter 2

## Python introduction

Python was developed by Guido van Rossum in the late eighties and early nineties at the National Research Institute for Mathematics and Computer Science in the Netherlands. Python is currently one of the most popular dynamic programming languages, along with Perl, Tcl, PHP, and Ruby. Although it is often viewed as a "scripting" language, it is really a general purpose programming language along the lines of Lisp or Smalltalk (as are the others, by the way). Today, Python is used for everything from throw-away scripts to large scalable web servers that provide uninterrupted service 24x7. It is used for GUI and database programming, client- and server-side web programming, and application testing. It is used by scientists writing applications for the world's fastest supercomputers and by children first learning to program.

Python is a high-level, interpreted, interactive and object oriented-scripting language. Python was designed to be highly readable which uses English keywords frequently where other languages use punctuation and it has fewer syntactical constructions than other languages.

1. Python is Interpreted: This means that it is processed at runtime by the interpreter and you do not need to compile your program before executing it. This is similar to PERL and PHP.
2. Python is Interactive: This means that you can actually sit at a Python prompt and interact with the interpreter directly to write your programs.
3. Python is Object-Oriented: This means that Python supports Object-Oriented style or technique of programming that encapsulates code within objects.
4. Python is Beginner's Language: Python is a great language for the beginner programmers and supports the development of a wide range of applications, from simple text processing to WWW browsers to games.

Some of Python's feature highlights include:

1. Easy-to-learn: Python has relatively few keywords, simple structure, and a clearly defined syntax. This allows the student to pick up the language in a relatively short period of time.
2. Easy-to-read: Python code is much more clearly defined and visible to the eyes.
3. Easy-to-maintain: Python's success is that its source code is fairly easy-to-maintain.
4. A broad standard library: One of Python's greatest strengths is the bulk of the library is very portable and cross-platform compatible on UNIX, Windows, and Macintosh.
5. Interactive Mode: Support for an interactive mode in which you can enter results from a terminal right to the language, allowing interactive testing and debugging of snippets of code.
6. Portable: Python can run on a wide variety of hardware platforms and has the same interface on all platforms.
7. Extendable: You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient.
8. Databases: Python provides interfaces to all major commercial databases.

## 2.1 Installing Python

Python is available for download from the new Click-UP system. It is recommended that you download the installation file from campus to save yourself internet usage from home.

- Insert your flash disk into a USB port on a computer in the computer labs.
- Open *Internet Explorer* (or your favourite internet browser)
- Log in at the University Portal and go to the new Click-UP system. Then go to the Downloads section of the MPR213 module page.
- Click on the **Python(x,y)-2.7.3.1.exe** file and save it to your flash disk. The file size is **510 MB** so make sure you have enough space on your flash disk.

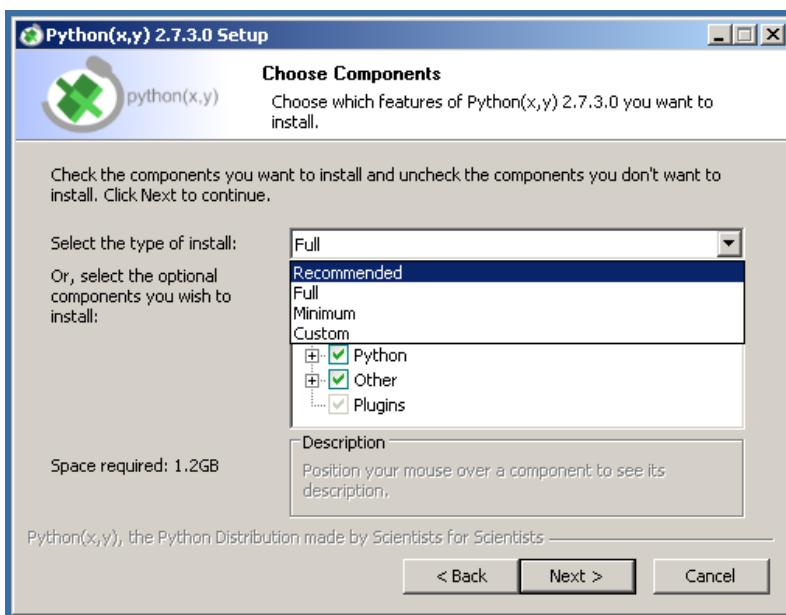


More Info:

Python is also available for download from the official site:  
<http://code.google.com/p/pythonxy/wiki/Downloads?tm=2>

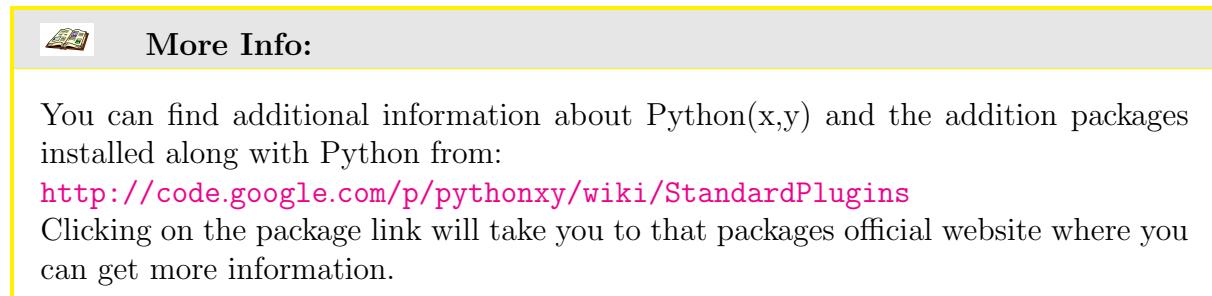
Now that you have downloaded the Python files required for installation, let us install Python onto your personal computer. You do not need to install it onto the lab computers as they already have it installed.

- Insert your flash disk into a USB port on your personal computer.
- Open *My Computer* and navigate to where you saved the *Python(x,y)-2.7.3.1.exe* file on your flash disk.
- Double click the *Python(x,y)-2.7.3.0.exe* file to install it.
- The *Python(x,y) 2.7.3.1 Setup Wizard* box will be displayed and in it the *License Agreement* box, read through the licence agreement and then click **I Agree** to proceed with the installation.
- The *Choose Components* box will be displayed. Choose either **Full** or **Recommended** for the “*Select the type of install:*” drop down box. Then click **Next** to proceed with the installation.



- The *Choose Install Location* box will be displayed. Click **Next** to proceed with the installation.
- The *Choose Start Menu Folder* box will be displayed. You are now ready to install Python, click **Install** to install Python.

- It will take several minutes to install Python, but when the installation has finished, the *Installation Complete* box will be displayed. Click **Next / Finish** to exit the *Setup Wizard*. You are now ready to run Python.



## 2.2 Using Python(x,y)

You start the Python(x,y) Home Menu (shown in Fig. 2.1) by double clicking on the Python(x,y) icon on the desktop or by going to the **Start** menu, then **Programs**, then **Python(x,y)**.

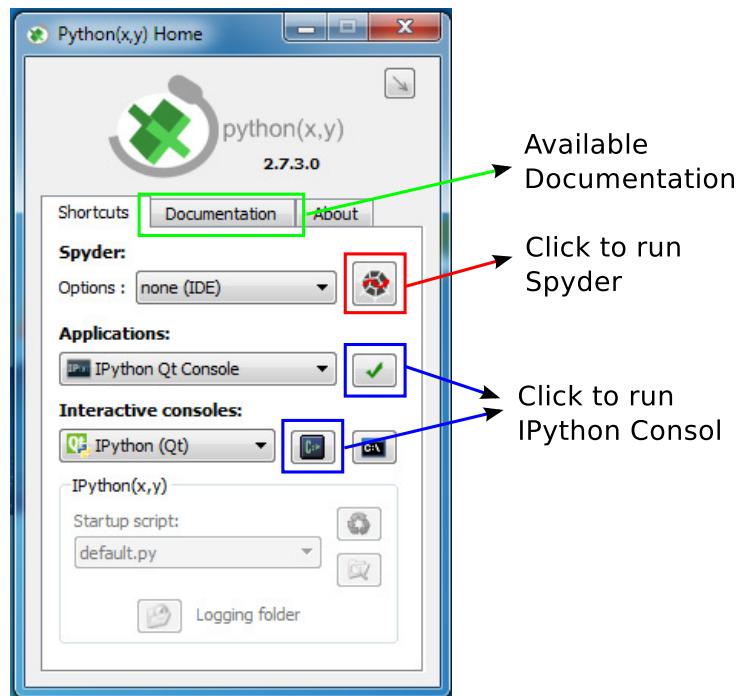


Figure 2.1: Python(x,y) Home Menu

## 2.3 Interactive Python Console

To start an interactive *IPython Console* click on the console button as shown in Fig. 2.1. The optional interactive consoles to choose from in the drop down box are: IPython (sh), IPython (Qt) and Python. For the sake of this course it is not important to know the difference between the different consoles and you are free to use any one of them. All of these interactive consoles run Python !

In these notes all examples are given either using the IPython (Qt) interactive console or using the Spyder editor as explained in the next section (Section 2.4).

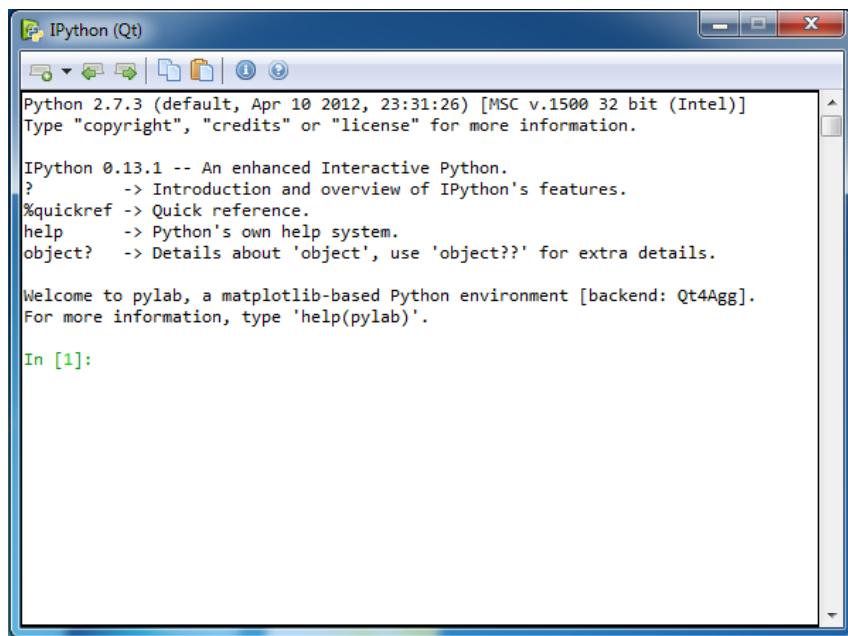


Figure 2.2: IPython Console

Once you start up the *IPython Console* you will see the special prompt (`In [1]:`) appears in the *IPython Console*. This means that Python is awaiting instructions. We can type in a command and press the *Enter* key. The command will be executed, output will be displayed (if the command requires it) and the special prompt will appear again, awaiting the next instruction. Using Python in this way is called the interactive mode. We execute commands one at a time and immediately see the outputs as they are produced.

If we enter

```
In [1]: 2 + 3
```

and press the *Enter* key, the following output is produced

```
1 In [1]: 2 + 3
2 Out [1]: 5
3
4 In [2]:
```

The special prompt (`In [1]:`) changes to (`Out [1]:`) and the output of 5 is displayed to indicate that our instruction was computed successfully. The special prompt (`In [1]:`) is then incremented from 1 to 2 (`In [2]:`) indicating that Python is waiting for the second input command. Notice that when you type you will see that there is a space directly after the special prompt. It is only there for readability to separate your instruction from the special prompt.

**Take Note:**

- 1) I number the program lines in all the examples I include in these notes. This is only so that I can refer to specific program line/s without you having to count lines from top to bottom.
- 2) Do not enter line numbers whenever you write a program in Python

**Take Note:**

- 1) The special prompt is the instruction number Python is waiting for, this is not a line number for the code.
- 2) To avoid confusion I will replace the instruction number from the special prompt with the sharp symbol (#) in the notes (e.g. `In [#]:`) unless required for the purposes of explanation.

If we however type

```
1 In [#]: a
```

and press the *Enter* key, the following output is produced

```
2 -----
3 NameError          Traceback (most recent call last)
4 /home/logan/<ipython-input-5-60b725f10c9c> in <module>()
5     ----> 1 a
```

```

6
7     NameError: name 'a' is not defined
8
9 In [#]:
```

We see what is called a **Traceback**. This indicates that Python was unable to compute our instruction. The **Traceback** traces the error to where it was occurred and Python thus tells you why and where it could not compute your instruction. In this case, you can see that above the special prompt there is a line that starts with “**NameError**”. Python tells us here what type of error was encountered and then the message for this error type: “**name 'a' is not defined**”. This information tells us that Python does not know what **a** is. Where an error is encountered will be discussed in the Debugging section of the notes (Section TODO??).

We can use (or abuse) Python as a calculator, i.e. always use it in the interactive mode to compute numerical values. The symbols used for simple computations are summarised in Table 2.1.

Computation	Math Symbol	IPython Console
Addition	+	In [#]: 5 + 2
Subtraction	-	In [#]: 5 - 2
Multiplication	*	In [#]: 5 * 2
Division	/	In [#]: 5 / 2
Power	$x^y$	In [#]: 5 ** 2

Table 2.1: Simple Computations in Python

Play around with the operations above and make sure you know how to use them.



### Take Note:

Note the unexpected output when dividing an **odd integer** by 2, i.e. `In[#]: 5 / 2` returns 2 and not 2.5. At first glance this seems like Python is making a mistake, but this behavior is indeed correct and will be explained in Section 3.1.4.



### More Info:

Both the IPython and Python consoles run Python. The IPython console has been modified to be more user friendly and easier to use and debug, thus we will be using the *IPython Console* in this course.

### 2.3.1 Importing Modules, Functions and Constants

Python on its own is a very lightweight programming language and has been designed in such a fashion that not all usable functions in Python are available as soon as it starts up.

For example if we enter the following into the IPython Console:

```
1 In [#]: cos(1)
```

and press the *Enter* key, the following output is produced

```
2 -----
3 NameError          Traceback (most recent call last)
4 /home/logan/<ipython-input-6-edaadd132e03> in <module>()
5     1 cos(1)
6
7 NameError: name 'cos' is not defined
8
9 In [#]:
```

Which tells us that Python doesn't know what `cos` is. In order to tell Python what any extra functions (like `cos`) are we first need to bring them into the IPython Console, only then will Python know what these functions are and what they do. Extra functions and objects (like  $\pi$ ) are stored in what is called a module.

You can consider these modules as folders on your computer. Folders are used to store and structure many different files on your computer in order to make it easier for the user to find and work with them, similarly modules store and structure different functions and objects that can be used in Python.

For example the exponential ( $e^x$ ), logarithms (`ln`, `log10`, etc.) and trigonometric (`cos`, `sin`, `acos`, etc.) functions are stored in the `math` module. The objects  $\pi$  and  $e$  are also stored in the `math` module.

So how do we bring these extra functions into the IPython Console? The answer is with the (`import`) statement. There are several ways to `import` these extra function: you can either `import` the full module (Section 2.3.1.1) or you can `import` the functions and objects (Section 2.3.1.2) contained inside a module.

### 2.3.1.1 Importing Modules

Here I will show you how to `import` the `math` module. As you have guessed by now the `math` module contains mathematical functions and objects.

We `import` the `math` module by entering the following in the IPython Console:

```
1 In [#]: import math
```

We have now told Python what `math` is and thus we can now use the `math` module in Python. If we type `In [#]: math` in the IPython Console we get the following output:

```
2 In [#]: math
3 Out [#]: <module 'math' (built-in)>
4
5 In [#]:
```

So we can see that Python knows that `math` is a module. To be able to use the functions and objects inside the `math` module we use a dot (.) after a `math` statement as follows:

```
5 In [#]: math.cos(1)
6 Out [#]: 0.5403023058681398
7
8 In [#]: math.sin(1)
9 Out [#]: 0.8414709848078965
10
11 In [#]: math.pi
12 Out [#]: 3.141592653589793
```

Looking at line 5 you can see that the dot (.) tells Python that the function `cos` resides in the module `math`.



**More Info:**

Type `help(math)` into the *IPython Console* (after importing the `math` module) to see a list of all the functions stored in this module



#### More Info:

You can also `import` a module and give it a new name to use in your program:

```
1 In [#]: import math as some_other_name
2
3 In [#]: some_other_name.cos(1)
4 Out [#]: 0.5403023058681398
5
6 In [#]:
```

#### 2.3.1.2 Importing Functions

If we want to only `import` (use) a few functions and objects from the `math` module we can do that straight away by using the `from` statement as follows:

```
1 In [#]: from math import pi
2
3 In [#]: from math import cos
4
5 In [#]: from math import sin
6
7 In [#]: from math import tan
```

Here we are almost literally telling Python: “from the module named `math` import the object `pi`” and “from the module named `math` import the function `cos`” etc.

In the example above you can see that there is a lot of repetitive typing: each time we `import` a function or object we repeat “`from math import`”. Because we are importing functions and objects from the **same** module we can consolidate the above example into one line as follows:

```
1 In [#]: from math import pi, cos, sin, tan
```

Now we tell Python: “from the module named math import the object pi and the function cos and the function sin etc.”

So now that Python knows what these extra functions are we can easily use them as follows:

```
1 In [#]: pi
2 Out [#]: 3.141592653589793
3
4 In [#]: cos(1)
5 Out [#]: 0.5403023058681398
6
7 In [#]: sin(1)
8 Out [#]: 0.8414709848078965
```



#### More Info:

Again you can also **import** a function from a module and give it a new name to use in your program:

```
1 In [#]: from math import cos as some_other_name
2
3 In [#]: some_other_name(1)
4 Out [#]: 0.5403023058681398
5
6 In [#]:
```

#### 2.3.1.3 Importing Modules (The Wrong Way)

The following example will **import** everything contained inside the **math** module. This is the **wrong way** of importing functions and objects from a module and should only be used in rare / special cases.

```
1 In [#]: from math import *
```

This tells Python: “from the module named math import everthing”  
These extra functions can now be used as shown previously:

```
1 In [#]: cos(1)
2 Out [#]: 0.5403023058681398
3
4 In [#]: sin(1)
5 Out [#]: 0.8414709848078965
6
7 In [#]: pi
8 Out [#]: 3.141592653589793
```



#### Take Note:

Often you will find online documentation or online worked examples that use this method for importing functions and constants and the only reason for this example in the notes is to let you know that this is the **wrong way** of import functions and constants. You should always use the `import` methods described in Sections 2.3.1.1 and 2.3.1.2 above, and only in very very rare cases should you use this method.

### 2.3.2 Math Module Computations

Now that we know how to bring these extra functions into the IPython Console lets look at some of the functions in the `math` module. A list of a few of the `math` module functions is shown below.

- Exponential function: `exp`  
e.g.  $e^1$  is computed by  
`In [#]: exp(1)`
- Base 10 logarithms are computed using `log10`  
`In [#]: log10(100)`
- Natural logarithms (`ln`) are computed using `log`  
`In [#]: log(2.718)`
- Trigonometric functions (radians): `sin`, `cos` and `tan`  
`In [#]: sin(1)`
- Inverse trigonometric functions (radians): `asin`, `acos`, `atan`  
`In [#]: atan(1)`
- Factorial  $n!$  is computed using the `factorial` command  
e.g.  $5!$  is computed as  
`In [#]: factorial(5)`

Play around with the operations above and make sure you know how to use them.



### Take Note:

Trigonometric functions work in radians, i.e. if you want to compute  $\sin(90^\circ)$ , you must first convert  $90^\circ$  to radian using the `math.radians()` function, e.g:

```
1 In [#]: from math import radians, sin  
2  
3 In [#]: sin(radians(90))  
4 Out [#]: 1.0  
5  
6 In [#]:
```

### 2.3.3 Help Function

The interactive mode is also used whenever you need help on some Python command. I have more than 3 years Python programming experience and I still sometimes forget the correct syntax of an Python command. The help function is probably the Python command I use most frequently, so I think it is good advice to get acquainted with the help function as soon as possible.

If you simply type “help” in the IPython Console you will see the following output:

```
1 In [#]: help  
2 Out [#]: Type help() for interactive help, or  
           help(object) for help about object.  
3  
4 In [#]:
```

As you can see from the above output there are two ways of using the help function in Python.



### More Info:

Syntax refers to the correct usage of a programming language command. It is similar to making a grammar mistake in a real language. If you use a command incorrectly Python will produce an error message, indicating that you made some mistake.

**More Info:**

- 1) The `help(object)` statement is used for imported modules, functions and objects.
- 2) The `help()` statement starts an interactive help system in the *IPython Console* and is used for core Python statements like 'and', '+', 'for', 'if', etc.

### 2.3.3.1 Help for Modules and Functions - “`help(function)`”

For help on a module or function you simply type `help(module)` or `help(function)` in the *IPython Console* and press the *Enter* key. If you type in a valid module or function name into the `help()` function, text will appear in the *IPython Console* that gives more details about the module or function you typed in. As an example, type `help(math)` and press the *Enter* key. The following text will appears in the *IPython Console*:

```
1 In [#]: import math
2
3 In [#]: help(math)
4
5 Help on built-in module math:
6
7 NAME
8     math
9
10 FILE
11     (built-in)
12
13 DESCRIPTION
14     This module is always available. It provides access to
15     the mathematical functions defined by the C standard.
16
17 FUNCTIONS
18     acos(...)
19         acos(x)
20
21             Return the arc cosine (measured in radians) of x.
22
23     acosh(...)
24         acosh(x)
25
26             Return the hyperbolic arc cosine (measured in radians)
27             of x.
```

```
28      asin(...)  
29          asin(x)  
30  
31          Return the arc sine (measured in radians) of x.  
32  
33      asinh(...)  
34          asinh(x)  
35  
36          Return the hyperbolic arc sine (measured in radians)  
37          of x.  
38  
39      atan(...)  
40          atan(x)  
41  
42          Return the arc tangent (measured in radians) of x.  
43  
44      atan2(...)  
45          atan2(y, x)  
46  
47          Return the arc tangent (measured in radians) of y/x.  
48          Unlike atan(y/x), the signs of both x and y are  
49          considered.  
50  
51      atanh(...)  
52          atanh(x)  
53  
54      .  
55      .  
56      .  
57  
58      (END)
```

The output above has been modified for easier display and printing of these notes, try this at home to see the actual output. At the end of the help information you will see an (END) line. Type **q** (short for quit) to exit the help information. You can also get the help information on functions contained inside the **math** module, for example try: `help(math.cos)`.



**Take Note:**

Remember to first `import` the module (in this case the `math` module) or function into the *IPython Console* before using the `help(object)` statement !!



### More Info:

In the *IPython Console* you can also use a single or double question (?) or ??? after the imported module, function or constant to get help information:

```
1 In [#]: from math import cos
2
3 In [#]: cos?
4 Type:      builtin_function_or_method
5 Base Class: <type 'builtin_function_or_method'>
6 String Form:<built-in function cos>
7 Namespace: Interactive
8 Docstring:
9 cos(x)
10
11 Return the cosine of x (measured in radians).
12
13 In [#]:
```

#### 2.3.3.2 Interactive Help - “`help()`”

For interactive help you simply type `help()` in the *IPython Console* and press the *Enter* key:

```
1 In [#]: help()
2
3 Welcome to Python 2.7!  This is the online help utility.
4
5 If this is your first time using Python, you should
6 definitely check out the tutorial on the Internet at
7 http://docs.python.org/tutorial/.
8
9 Enter the name of any module, keyword, or topic to get
10 help on writing Python programs and using Python modules.
11 To quit this help utility and return to the interpreter,
12 just type "quit".
13
```

```
14 | To get a list of available modules, keywords, or topics,  
15 | type "modules", "keywords", or "topics". Each module  
16 | also comes with a one-line summary of what it does; to  
17 | list the modules whose summaries contain a given word  
18 | such as "spam", type "modules spam".  
19 |  
20 help>
```

You will see that the Python `help()` command will display a welcome message and then the special prompt will change from `In [#]:` to `help>`. The interactive help has a wealth of information. Unfortunately typing “`help> modules`” into the help system will most likely cause the *IPython* Console to crash (close), this is a bug that is currently being fixed by the Python developers. Typing “`help> keywords`” or “`help> topics`” works fine and will display a list of Python keywords and help topics respectively.

If you now type in a valid Python command, keyword or help topic name after this special prompt, text will appear in the *IPython* Console that gives more detail about the command you typed in. As an example, type “`help> +`” and press the *Enter* key. The following text will appears in the *IPython* Console:

```
1 help> +  
2 Summary  
3 *****  
4  
5 The following table summarizes the operator precedences in  
6 Python, from lowest precedence (least binding) to highest  
7 precedence (most binding). Operators in the same box have  
8 the same precedence. Unless the syntax is explicitly given,  
9 operators are binary. Operators in the same box group left  
10 to right (except for comparisons, including tests, which all  
11 have the same precedence and chain from left to right  
12 --- see section *Comparisons* --- and exponentiation, which  
13 groups from right to left).  
14  
15 +-----+-----+  
16 | Operator | Description |  
17 +=====+=====+  
18 | ‘‘+‘‘, ‘‘-‘‘ | Addition and subtraction |  
19 +-----+-----+  
20 | ‘‘*‘‘, ‘‘/‘‘, ‘‘%‘‘ | Multiplication, division, remainder |  
21 | ‘‘//‘‘, ‘‘%‘‘ | [8] |  
22 +-----+-----+
```

```
23 | ``+x``, ``-x``, | Positive, negative, bitwise NOT |
24 | ``~x``           |
25 +-----+
26 | ``**``          | Exponentiation [9]           |
27 +-----+
28
29 - [ Footnotes ]-
30
31 [8] The ``%`` operator is also used for string formatting;
32     the same precedence applies.
33 (END)
```

The output above has been modified for easier display and printing of these notes, try this at home to see the actual output. Again at the end of the help information you will see an (END) line. Type **q** (short for quit) to exit the help information. Spend some time using the `help` functions to become familiar with the Python documentation environment. This will enable you to help yourself in a way that would enable you to focus, find and correct your problem. In addition it will allow you to explore functions that are not taught in this course but that Python has to offer.



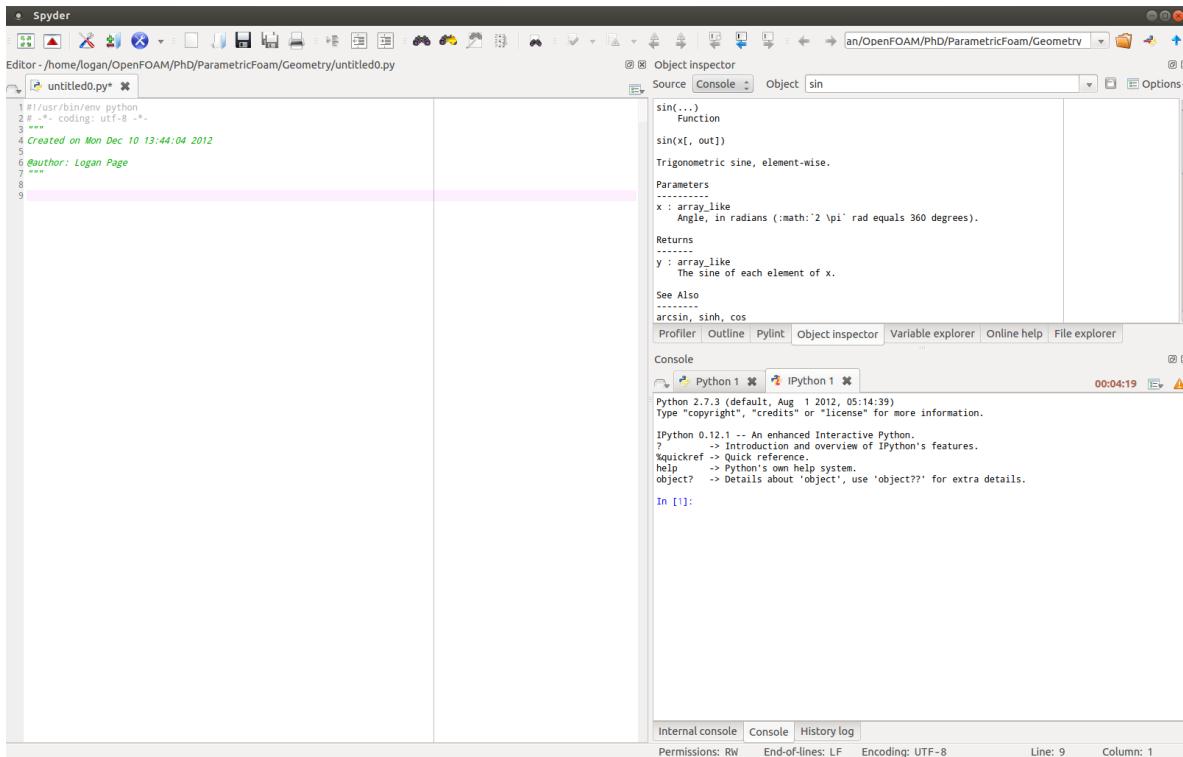
#### More Info:

If you don't want to go through the interactive help system you can enclose the keyword in quotation marks, e.g: `help('+')`, `help('and')`, `help('keywords')`, etc.

## 2.4 Programming Mode (Spyder)

The interactive modes discussed above are only useful for a few scenarios. The usual situation when we are developing a program is that we want to execute a large collection of commands. It is very limiting to enter and execute these commands one at a time in interactive mode.

Rather, we will type out our collection of commands, one below the other, in a file, save this file somewhere on our computer and then run the file (execute the collection of commands). We will be using the *Spyder* IDE (Integrated Development Environment) for creating Python scripts that contain our collection of commands. To start *Spyder* first run the Python(x,y) Home Menu (as shown in Section 2.2) and click on the button to run *Spyder*.



The *Spyder* application is split into the following windows:

- Editor (where we will write our programs)
- Variable Explorer
- Object Inspector
- File Explorer
- and many others...
- See <http://packages.python.org/spyder/>  
For documentation on each of these windows.

Once you have typed all the statements in your Python script, Click on the **File** menu, then Click **Save as**. Enter the name of your program, which can be anything as long as you obey the rules for file names (More on this in the Section 2.4.2). and add the .py file extension. All Python scripts require the .py extension and are referred to as a .py-files in these notes. Before clicking the **Save** button make sure that you are saving the file in the directory you intended. If you are unsure, look at the top of the **Save File** window. You will see a **Save in** window. Click on the downward pointing arrow head, which will make a directory list appear. You can now navigate to the directory where you want to save your work and then click the **Save** button.

Once the file is saved, you can execute the list of commands in the Python script by typing `run` and the name of the .py-file in the *IPython Console*, followed by pressing the *Enter* key:

```
1 In [#]: run program1.py
```

Alternatively you can run the Python script directly from the *Spyder* IDE by clicking on the `Run` menu and then on `Run` (Short-cut key: F5).



#### Take Note:

When running a Python script from the *IPython Console*, you have to make sure that the *IPython Console* is pointing to the directory where you saved the .py-file. See Section 2.4.1 for navigating the directory structure.



#### More Info:

The official *Spyder* website will give you more information on how to use *Spyder* as well as how to use the individual windows inside the *Spyder* application:  
<http://packages.python.org/spyder/>

### 2.4.1 Navigating the directory structure from Python

The computer has a hard drive with many files on them e.g. pictures, music, system files and executable programs. In order to keep track of where the files are they are grouped together using a directory structure (almost like filing documents in a filing cabinet). The Python package is grouped in its own directory. To become familiar with the directory structure click on **Start**, then double click on **My Computer**. You will see icons and folders depending on your computer. Double click on **HDD (C:)** this will let you go to the C: hard drive and display the content of this hard drive. Now double click on **Program Files (x86)**. You will now see many directories. Search for the one called **pythonxy** and double Click on it. You will now see some folders and files inside the **pythonxy** directory. The window you are currently viewing the files and directories in is called *Windows Explorer*.

When you start the *IPython Console* (depending on how you start it) it will point to one of the following directories:

(A) C:\Users\<user name>    Or

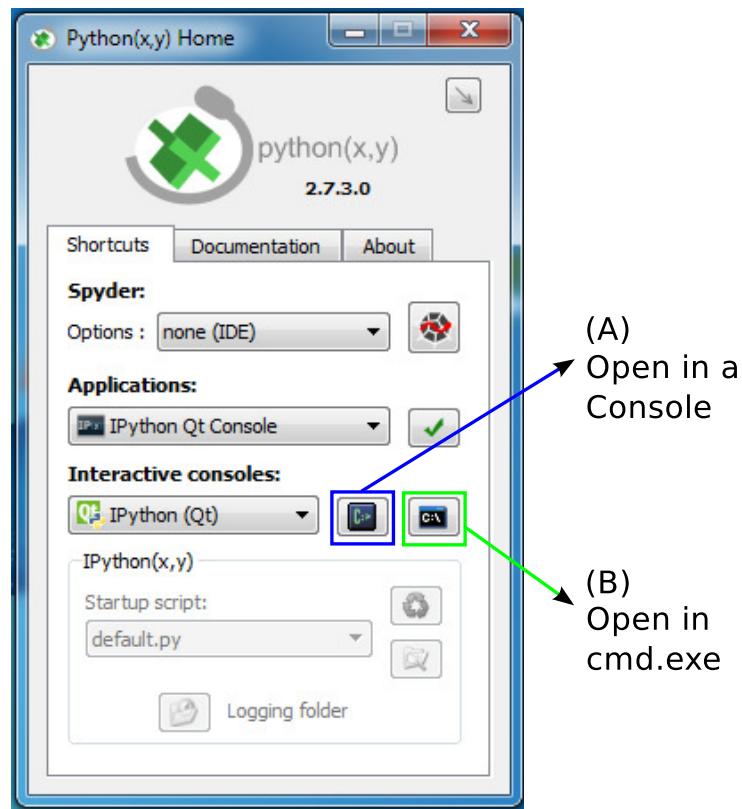


Figure 2.3: Python(x,y) Home Menu

(B) C:\Users\<user name>\.xy\startups

When the *IPython Console* starts up it will point to (A), above, if you start the *IPython Console* in a console; and it will point to (B) if you start the *IPython Console* in cmd.exe. This is shown in Fig. 2.3.

You can use the `pwd` command to see where the *IPython Console* is pointing (ignore the `u` below, this explanation is outside the scope of this course, and only consider the directory location shown between the quotation marks):

```

1 In [#]: pwd
2 Out[#]: u'C:\\Users\\MPR213'
3
4 In [#]:

```



**Take Note:**

<user name> will be replaced with your user name on your computer, in my case <user name> is equal to MPR213.

**Take Note:**

Windows systems use the backslash (\) to separate directories, files and drives. And this notation will be used in the notes.

Python (on a Windows system) uses either two backslashes (\\) or a forward slash (/) to separate directories, files and drives.

For now lets assume that the *IPython Console* is pointing the "C:\Users\<user name>" directory. When I type `ls` the following appears on the screen

```
1 In [#]: ls
2 Volume in drive C has no label.
3 Volume Serial Number is 24A9-A80B
4
5 Directory of C:\Users\MPR213
6
7 2012/12/10  11:11 PM    <DIR>          .
8 2012/12/10  11:11 PM    <DIR>          ..
9 2012/12/10  10:31 PM    <DIR>          .ipython
10 2012/12/13  11:01 AM   <DIR>          .matplotlib
11 2012/12/10  10:02 PM   <DIR>          .xy
12 2012/12/10  08:46 PM   <DIR>          Contacts
13 2012/12/10  10:35 PM   <DIR>          Desktop
14 2012/12/10  08:46 PM   <DIR>          Documents
15 2012/12/10  08:46 PM   <DIR>          Downloads
16 2012/12/10  08:46 PM   <DIR>          Favorites
17 2012/12/10  08:46 PM   <DIR>          Links
18 2012/12/10  08:46 PM   <DIR>          Music
19 2012/12/10  08:46 PM   <DIR>          Pictures
20 2012/12/10  08:46 PM   <DIR>          Saved Games
21 2012/12/10  08:46 PM   <DIR>          Searches
22 2012/12/10  11:11 PM          107 Untitled0.ipynb
23 2012/12/10  08:46 PM   <DIR>          Videos
24                      1 File(s)        107 bytes
25                      16 Dir(s)     33 892 466 688 bytes free
26
27 In [#]:
```

The output of `ls` shows the content of the current directory and it is the same as

when you browse the directories using **My Computer** (*Windows Explorer*). When using *Windows Explorer* it is easy to distinguish between files and directories as the icons are there to assist you.

When using **ls** you can distinguish files from directories in the following way. Files usually have an extension i.e. a filename followed by a dot (.) and then followed by an extension. For example, Untitled0.ipynb has a filename Untitled0 and an extension ipynb. The extensions tell Windows about the content of the file so that when you double click on it Windows knows what application or program to use to open the file. The directories only have a name without an extension. This is because directories are not opened by an application or program, they only group files. Therefore Windows only needs to know the name of the directory you want to go to from your current directory to go there. Directories also have a DIR keyword placed in the third column of information when you use the **ls** command.

You can change from your current directory or current hard drive to another directory or hard drive by using the **cd** command. If you type **cd Documents** and press the *Enter* key you have changed from your current directory (`C:\Users\<user name>`) to the "`C:\Users\<user name>\Documents`" directory. If you now type **ls** the following is displayed on your screen:

```
1 In [#]: pwd
2 Out [#]: u'C:\\Users\\MPR213'
3
4 In [#]: cd Documents
5
6 In [#]: pwd
7 Out [#]: u'C:\\Users\\MPR213\\Documents'
8
9 In [#]: ls
10 Volume in drive C has no label.
11 Volume Serial Number is 24A9-A80B
12
13 Directory of C:\\Users\\MPR213\\Documents
14
15 2012/12/13 11:11 PM <DIR> .
16 2012/12/13 11:11 PM <DIR> ..
17 2012/12/13 11:11 PM 0 New Text Document.txt
18 1 File(s) 0 bytes
19 2 Dir(s) 33 175 799 744 bytes free
20
21 In [#]:
```

which shows you that there are 2 directories and 1 files in your current (C:\Users\<user name>\Documents) directory. You can type `cd ..` to go back to the directory you just came from. By typing `cd ..` you move one directory up (to the parent directory). E.g. the `pythonxy` directory is in the **Program Files (x86)** directory and therefore the **Program Files (x86)** directory is one directory up from (or the parent directory of) the `pythonxy` directory.

To change to a different hard drive, first open **My Computer**. **My computer** will show you which hard drives are available on your computer and what their drive letters are (e.g. C, E, F). If I type `cd F:/` I change from my C: hard drive to my F: drive. By typing `cd C:/` I am back on my C: drive.



#### Take Note:

To change drives you have to use the forward slash (/) after the drive letter and double colon (:).

E.g. `cd C:` - wont work !

E.g. `cd C:\` - wont work !

Only `cd C:/` will work.

This is important when you work in the labs because you need to go to your H: drive and save your data there, otherwise your data will be lost when you log out. This is also important for tests and exams as you will be in the labs when you write them. If for some reason your computer hangs or the power fails then at least the data you saved on your H: drive will still be there.

By typing `cd C:/` I have moved to the C: drive. I am now in the root directory of my C:/ drive. It is called the root directory as there are no other directories up from this directory. When you type `ls` you will see that there are no `[.]` or `[..]` in the list of directories.

Let us go back to the pythonxy default directory by typing `cd C:/Program Files (x86)/pythonxy` and pressing the *Enter* key. Note that I have used the forward slash here to separate the drive and directories.

To show you what happens when you use `cd` incorrectly let us try to change directory to a file. When you type `cd License-en.rtf` the following appears on the screen:

```
1 In [#]: cd C:/Program Files (x86)/pythonxy
2
3 In [#]: cd License-en.rtf
4 [Error 267] The directory name is invalid: u'License-en.rtf'
```

```
5 C:/Program Files (x86)/pythonxy  
6  
7 In [#]:
```

This is because `License-en.rtf` is a file (with the `.rtf` extension) and not a directory.

#### 2.4.1.1 Summary

- `pwd` - Prints the current directory, to where the *IPython Console* is pointing, to the screen.
- `ls` - Print the contents of the current directory to the screen.
- `cd directory_name` - Changes the current directory to the directory specified (in this case `directory_name`).
- Examples:
  - `cd C:/` - Changes to the `C:` root directory.
  - `cd ..` - Changes to the parent directory (one directory up).
  - `cd ~/Documents` - Changes to your “My Documents” Directory.
  - `cd C:/Program Files (x86)/pythonxy` - Changes to the Python(x,y) default installation directory.
  - `cd C:/Program Files (x86)/pythonxy/doc` - Changes to the Python(x,y) default documentation directory.



#### More Info:

`cd ~/Documents` is the same as  
`cd C:/Documents and Settings/<user_name>/Documents` (where `<user_name>` is your user name on your computer).  
Both of these commands take you to your “My Documents” folder.

#### 2.4.2 Python Script Names

It is extremely important to only use allowable Python script names. Python script names can contain any alphabetical characters (a-z, A-Z) as well as all the numerals (0-9), but the first character has to be an alphabetical character. Python script names may not contain any spaces or special symbols (! @ # \$ % ^ & \ + - ) except the underscore

(-). Python is also case sensitive, i.e. it distinguishes between upper case and lower case letters.

It is therefore important that you enter the name of the file as you saved it. Therefore if your Python script name with extension is `program1.py` then you have to run it by typing `run program1.py` in Python. If you type `run PROGRAM1.py` then Python will complain with a similar error message to the one below:

```
1 In [#]: run PROGRAM1.py
2 ERROR: File 'u'PROGRAM1.py',' not found.
3 In [#]:
```

Use the `ls` statement to view the files in your current directory. How the files are displayed (i.e. upper and lower case letters) using this command is also the way you should type them in the *IPython Console*.



#### Take Note:

You should avoid giving a program the same name as any of the Python keywords: type `help('keywords')` to see the list:

```
1 and      elif      if       print
2 as       else      import   raise
3 assert   except   in       return
4 break    exec     is       try
5 class    finally  lambda  while
6 continue for     not     with
7 def     from     or      yield
8 del      global   pass
```



#### More Info:

Because Python is so widely used in all areas of industry and research, there is a programming standard (coding conventions) for how you write up your programs. This standard can be found at:

<http://www.python.org/dev/peps/pep-0008/>

Although as engineers we are not programmers, I do believe in doing things right from the start and as such I will be referencing sections of the PEP8 standard where needed.

PEP8 standard for program names:

- Python script should have short, all-lowercase names (e.g. `my_program.py` and not `My_Program.py`). Underscores can be used in the Python script name if it improves readability.

## 2.5 Documentation

Python has a wealth of online documentation and worked out examples. If you ever get stuck, or the information from the `help()` function is insufficient, *Google* or any other search engine will be your best friend.

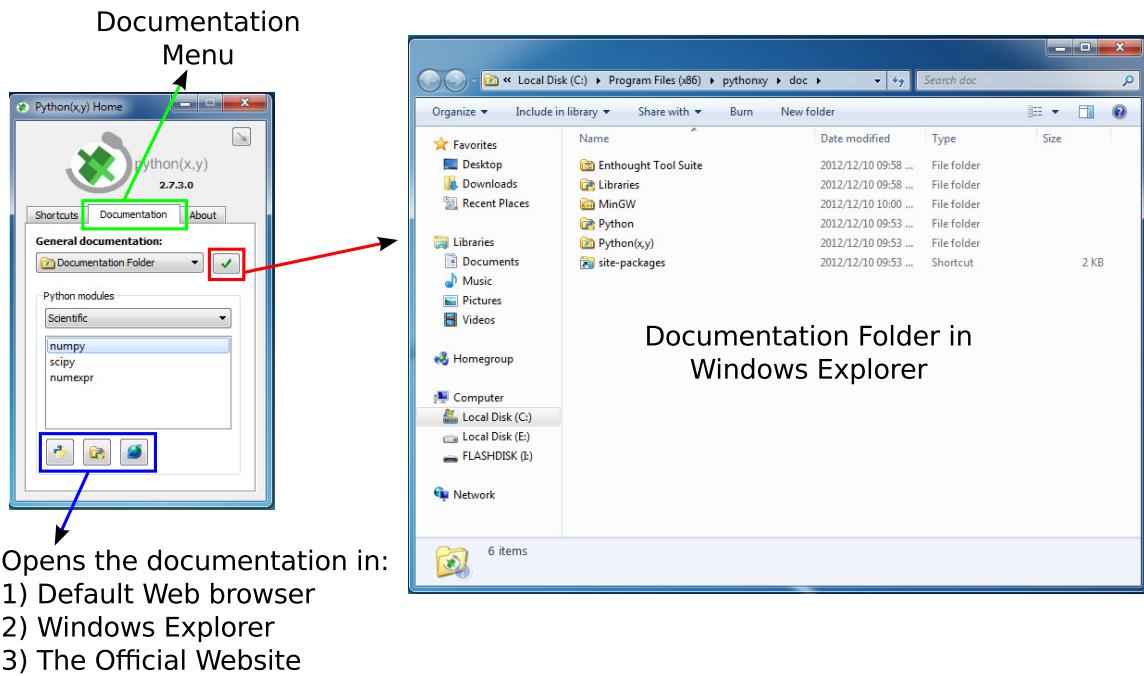


Figure 2.4: Python(x,y) Documentation

Over and above what you can find online there is also a lot of documentation installed with the Python(x,y) Home Menu (shown in Fig. 2.4). Clicking on the little “tick” symbol will open up *Windows Explorer* to where the documentation is saved on your Computer.

The Python(x,y) Documentation Tab is also sorted into different types of categories and modules to make it easier to browse through all the available documentation (As shown in Fig. 2.5).

The *Spyder Object inspector* window will also fill with documentation as you type your program, see for example Fig 2.6. And lastly the *Spyder Online help* window has even

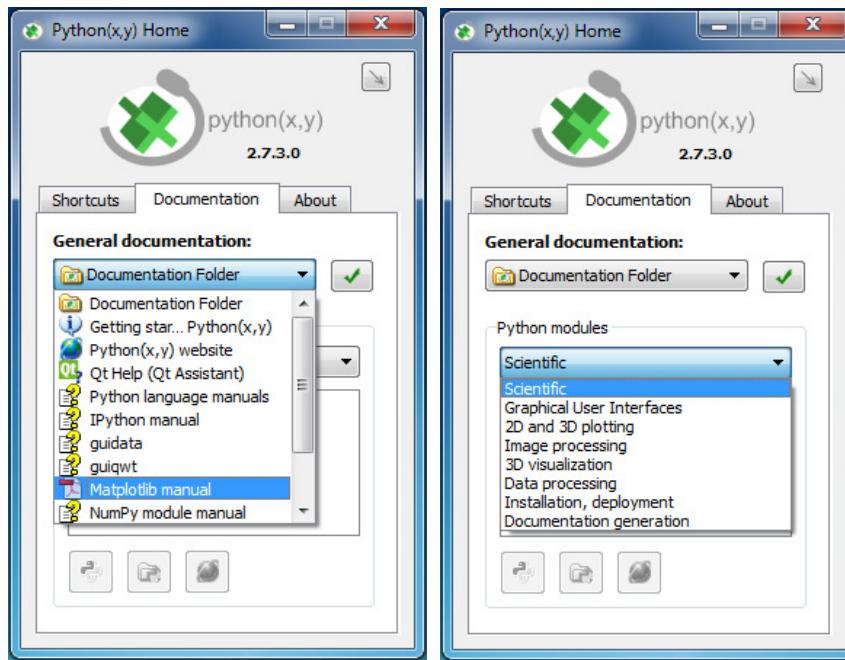


Figure 2.5: Python(x,y) Documentation - Categories

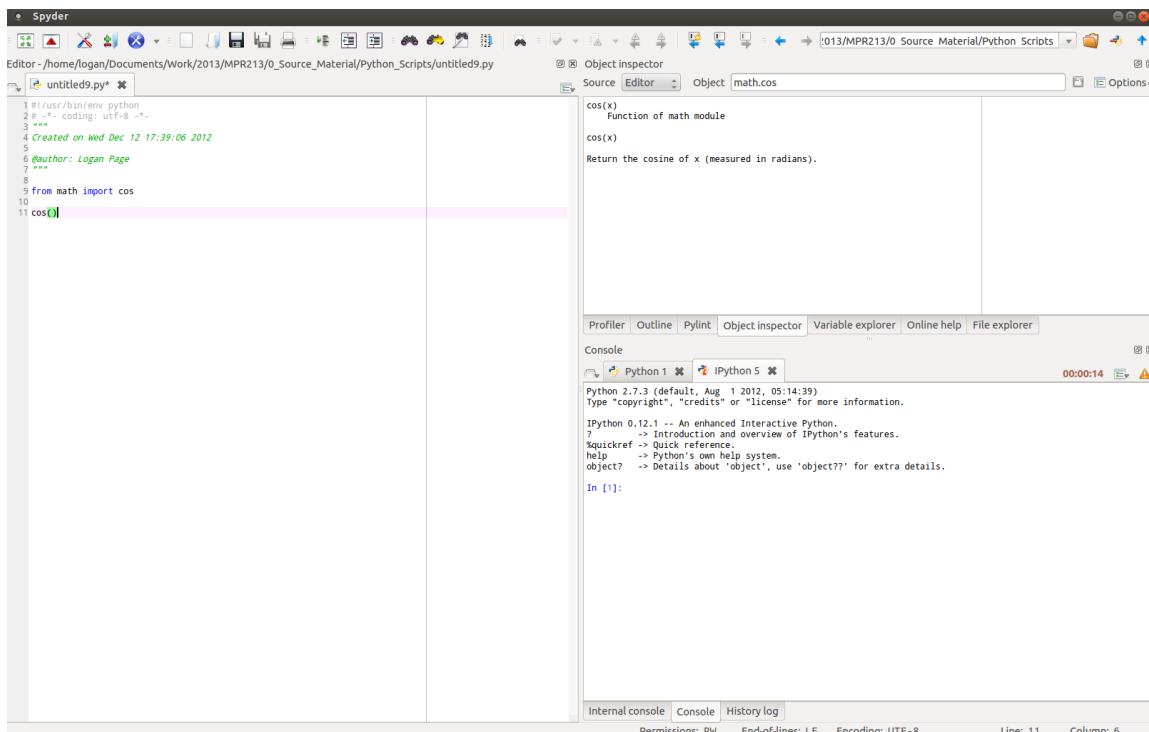


Figure 2.6: Spyder Object Inspector Window

more online documentation available, see for example Fig. 2.7.



Figure 2.7: Spyder Online Help Window

## 2.6 Supplementary Sources

The following sources have great examples and explanations on a lot of topics we will discuss in these notes. So if at any stage you find yourself needing more help or additional examples regarding a specific topic or concept these sources should be your first stop, thereafter “Google”.

1. [http://en.wikibooks.org/wiki/Non-Programmer's\\_Tutorial\\_for\\_Python\\_2.6](http://en.wikibooks.org/wiki/Non-Programmer%27s_Tutorial_for_Python_2.6)
2. <http://docs.python.org/2/tutorial/index.html>
3. <http://pythonbooks.revolunet.com/>

There is also a series of 24 lectures titled *Introduction to Computer Science and Programming* delivered by Eric Grimson and John Guttag from the MIT available at <http://academicearth.org/courses/introduction-to-computer-science-and-programming>. This is aimed at students with little or no programming experience. It aims to provide students with an understanding of the role computation can play in solving problems.

## 2.7 Exercises

1. Run the interactive *IPython Console*.
2. For each term in expressions below (Eq. 2.1 and Eq. 2.2), which arithmetic operation does Python compute first, (addition and subtraction) or (multiplication and division). Note the significance of the brackets ()

$$(300 - 154)/(11/2 + 16 \times 6) \quad (2.1)$$

and

$$300 - 154/11/2 + 16 \times 6 \quad (2.2)$$

[Hint: see the help information on operators - `help('+')`]

3. What is the value of  $154/11/2$ ? Does 154 first get divided by 11 to give 14 which is then divided by 2? Or does 11 get divided by 2 to give 5.5 such that 154 can be divided by 5.5?
4. Identify the order in which the operations will execute by placing round brackets () around the operations that execute together:

$$250 - 350/35/5 + 125 \times 3^2/2 \quad (2.3)$$

[e.g.  $12 + 3 \times 4 = (12 + (3 \times 4))$ ]

[Hint: You know you've made a mistake if the answer changes, as the () will force operations to occur together: e.g.  $12 + 3 \times 4 = 24$  and  $((12 + 3) \times 4) = 60$ ]

5. Can you say with confidence whether the expression below will give the same or a different answer to the expression above:

$$((250 - (350/(35/5))) + (125 \times 3^2)/2) \quad (2.4)$$

[If you were wrong or are still unsure investigate grouping different operations.]



### Take Note:

Use brackets to avoid confusion in your operations.

6. Compute the answer of

$$\frac{154 - 17}{29 + 54 \times 3} \quad (2.5)$$

7. Compute the answer of

$$154 - \frac{17}{29} + 54 \times 3 \quad (2.6)$$

8. Import the math module. Then Type `help(math.cos)` and `help(math.acos)`.
9. Compute the answer of:

$$5^3 - 0.5^4 \times \frac{15 + \pi}{3.5} + 121 \times 4 \quad (2.7)$$

10. What is the difference between the functions `math.log` and `math.log10`?
11. Compute the unknown variable  $b$ :

$$b \times 4 + 2 = \sqrt{19} - \cos(\pi) \quad (2.8)$$

12. Compute the unknown variable  $a$ :

$$a \times \sqrt{\log_{10}(90)} + 4\sin(\pi/8) = \frac{\cos(\pi/4)}{\tan(\sqrt{111})} \quad (2.9)$$

13. This question is only to make you aware of Python's complex number capabilities. In this course we do not consider complex numbers, but I feel it is important to be aware of it. Try the following examples in the *IPython Console*:

```
1 import cmath
2 cmath.sqrt(-1)
3 1+2j
4 (1+3j) + (9+4j)
5 (1+3i) * (9+4i)
```

You can type your own complex numbers in e.g. by typing `1+2j`. You can add complex numbers e.g. `(1+3i)+(9+4i)`. You can multiply complex numbers e.g. `(1+3i)*(9+4i)`.

If you find the answers to be surprising don't worry, the formal theory behind complex numbers will be treated in subjects to come.



# Chapter 3

## Names and Objects

When we used Python interactively as a calculator, each number we typed was actually created as an object in the computers' memory. So typing `1 + 2` into the *IPython Console* will create three objects in memory: the object 1; the object 2; and the object 3 (the result of  $1 + 2$ ). It is important to note that everything in Python is an object in memory.

Names are used in Python to reference these objects in memory. As engineers, we usually work with numbers, so most of the names we will deal with will reference numerical objects. Python is unlike traditional programming languages (e.g. C, Pascal and Fortran) in that we do not have to define names. Defining names basically means that at the start of our program we have a list of all the names and their respective types (e.g. do we store a real number, integer or string in the variable).

### 3.1 Numerical Objects

In Python (and all other programming languages I'm familiar with), a object (value) is assigned to a name by simply typing the name, followed by an equal sign, followed by the numerical value e.g.

```
1 In [#]: a = 2.0
2
3 In [#]:
```

The above statement first creates the object 2.0 in memory and then binds the name `a` to that object. You can enter the above line in the interactive mode or in the programming

mode. You can now type **a** and press the *Enter* key and the following will appear

```
3 In [#]: a
4 Out [#]: 2.0
5
6 In [#]:
```

We do not get the error message we got when we started with Python in Section 2.3. Remember we typed **a** and received an error message

```
-----
NameError          Traceback (most recent call last)
/home/logan/<ipython-input-5-60b725f10c9c> in <module>()
----> 1 a

NameError: name 'a' is not defined

In [#]:
```

Since we have now defined **a** we may use it in expressions just like numbers. We can use the name **a** in a subsequent computation in a program. The value 2.0 will be substituted for every name **a** in the expression, since the name **a** is bound to the object 2.0. As an example, consider the program line

```
6 In [#]: b = a + 3.0
7
8 In [#]: b
9 Out [#]: 5.0
10
11 In [#]:
```

The object 3.0 is first created in memory, then the object 2.0 is substituted into the above expression, in the place of the name **a**. 3.0 is added to 2.0 and the resulting 5.0 object is created in memory. Finally the name **b** is bound to this resulting 5.0 object. All of this is done in line 1 in a fraction of a second. Names are bound to objects until we explicitly change the binding to something else e.g.

```
11 In [#]: a = 4.0
12
13 In [#]:
```

The above statement will first create the object 4.0 in memory and then bind the name `a` to this new object, thus removing the previous binding to the 2.0 object. The name `b` is still bound to the object 4.0 and thus will remain unchanged. If we require an updated evaluation of `b`, we must again type:

```
13 In [#]: b = a + 3.0
14
15 In [#]: b
16 Out [#]: 7.0
17
18 In [#]:
```



#### Take Note:

Please check the line numbers in each of the examples above to see how they follow on from one another.



#### Take Note:

Only a name is allowed at the left of the equal sign, and all the quantities (numbers, other names and objects) that appear to the right of the equal sign must be known. Therefore, make sure that a object (value) has been bound to a name before you use that name to the right of the equal sign to define another name.

### 3.1.1 Memory Model

Hopefully the previous section was not too confusing, I will try to clarify the “name-binding-object” behaviour of Python with a few memory model figures and explanations in this section.

Four columns are shown in Figure 3.1, the first column is the line number for the Python statement, the second the Python statement. This is what we see when creating a Python program or when we use Python in an interactive mode. What happens behind the scenes, between Python and the computers memory, is shown in the last two columns.

Line No.	Python Statement	Names	Objects (in memory)
1	a = 1	a	1 <small>integer object</small>
2	b = 1	b	
3	b = 2	b	2 <small>integer object</small>
4	c = a + b	c	3 <small>integer object</small>

Figure 3.1: Memory model for names bound to objects (example 1)

If we consider Figure 3.1 and go through each of the Python statements:

1. the object 1 is created in memory and the name **a** is bound to that object.
2. the object 1 was already created in memory in line 1, but now we bind another name **b** to this same object. So at this point we have two names (**a** and **b**) bound to the same object 1.
3. the object 2 is created in memory and the name **b** is now bound to the new object 2, thus removing the previous binding to the object 1.
4. the objects 1 and 2 are added together and the resulting object 3 is created in memory and the name **c** is used to bind this resulting object 3.

Line No.	Python Statement	Names	Objects (in memory)
1	a = 2.0	a	2.0 <small>float object</small>
2	b = a + 3.0	b	3.0 <small>float objects</small> 5.0
3	a = 4.0	a	4.0 <small>float object</small>
4	b = a + 3.0	b	7.0 <small>float object</small>

Figure 3.2: Memory model for names bound to objects (example 2)

The previous figure (Figure 3.2) shows the memory model for the examples shown in the previous section. Review the previous section (Section 3.1) and see if you can link the explanations, given in the examples, with the memory model shown above.



#### Take Note:

Objects in memory can have multiply names bound to them.

### 3.1.2 Example: Swapping objects

Let's consider the problem of swapping the objects bound by two names `a` and `b`. A first attempt might be:

```

1  a = 2.0
2  b = 4.0
3  a = b
4  b = a

```

Line No.	Python Statement	Names	Objects (in memory)
1	<code>a = 2.0</code>	<code>a</code>	<code>2.0</code> <small>float object</small>
2	<code>b = 4.0</code>	<code>b</code>	<code>4.0</code> <small>float object</small>
3	<code>a = b</code>	<code>a</code>	
4	<code>b = a</code>	<code>b</code>	

Figure 3.3: Memory model – swapping objects (example 1)

The above program does not succeed in our goal. As program line 3 is executed, the name `a` is bound to the object 4.0. Program line 4 then binds name `b` to the same object 4.0. All that the above program achieves is to bind all names to the same object 4.0 (this is shown in the memory model in figure 3.3). So how do we perform the swapping correctly?

The problem with the above program is that as soon as we bind the name `a` to a new object, its original binding (to object 2.0) is removed. The name `a` can only be bound to one object, the one it was most recently bound to. Names have no memory of their

previously bound objects. The way we work around this problem is by using a new name that remains bound to the first object. Here's the correct way to swap two objects:

```

1 a = 2.0
2 b = 4.0
3 a_copy = a
4 a = b
5 b = a_copy

```

Line No.	Python Statement	Names	Objects (in memory)
1	a = 2.0	a	2.0 float object
2	b = 4.0	b	4.0 float object
3	a_copy = a	a_copy	
4	a = b	a	
5	b = a_copy	b	

Figure 3.4: Memory model – swapping objects (example 2)

Note that I used the name `a_copy` which will remain bound to the object 2.0. Using descriptive names such as `a_copy` helps a lot to make sense of your programs. It also helps other programmers if they have to use or modify your programs (or lecturers that have to grade your programs).

An alternative solution to the object swapping problem is to use a new name that remains bound to the second object, but then the program must read:

```

1 a = 2.0
2 b = 4.0
3 b_copy = b
4 b = a
5 a = b_copy

```

This example illustrates that there is more than one correct method to achieve a certain goal. You as the programmer will have to make sure that whatever logic you use is consistent with the goal and will produce the correct output.

### 3.1.3 Assignment Operators

We can also use the same name to the left and to the right of the equal sign. Remember that when we use the equal sign that everything to the right has to be known or defined whereas there is only one name to the left of the equal sign. What happens in the following code?

```
1 number = 5  
2 number = number + 3
```

Line 1 first creates the object 5 in memory and then binds the name `number` to the 5 object, therefore we can now use `number` on the right hand side of the equal sign. On line 2 the right hand side gets computed until everything is done and then the name (on the left hand side of the equal sign) gets bound to the resulting object (from the computation of the right hand side of the equal sign). Therefore by the time line 2 is reached, the name `number` is bound to the object 5 to which the object 3 gets added to give the resulting object 8. The name `number` is then bound to this resulting 8 object. Using names in this fashion allows you to remember what you have computed so far, and reduces the number of names needed in your programs. Similarly, if we add line 3

```
3 number = number * 2
```

the name `number` is now bound to the object 16. This is a powerful technique which comes in handy in many situations. There is also a “shorthand” method for modifying variables in this fashion. Consider the following code:

```
1 number = 5  
2 number += 3  
3 number *= 2
```

This example does exactly the same as the above example. Line 1 binds the name `number` to the object 5, line 2 adds the object 3 to the 5 object and line 3 multiplies the object 2 with 8 object. This technique is know as *Assignment Operators*. The Table 3.1 shows a summary of the assignment operators (assume `a` holds the value of 10).

Play around with these operators and make sure you are comfortable with both methods of updating / changing the same variable.

Line No.	Python Statement	Names	Objects (in memory)
1	number = 5	number	5 integer object
2	number += 3	number	3 8 integer objects
3	number *= 2	number	2 16 integer objects

Figure 3.5: Memory model – assignment operators

Computation	Example	Operator
Addition	a = a + 10	a += 10
Subtraction	a = a - 10	a -= 10
Multiplication	a = a * 2	a *= 2
Division	a = a / 2	a /= 2
Power	a = a ** 2	a **= 2

Table 3.1: Assignment Operators in Python

### 3.1.4 Integers vs. Floats

For numerical objects there are two distinct kinds of types, namely an integer and a floating point number (real number). We can see the type of variable we are using by typing `type(variable)` into the *IPython Console*:

```

1 In [#]: a = 10
2
3 In [#]: type(a)
4 Out [#]: int
5
6 In [#]: b = 2.0
7
8 In [#]: type(b)
9 Out [#]: float

```

Recall, in Section 2.3, that typing `5 / 2` into the *IPython Console* returned 2 and not 2.5, this is because Python sees that both the 5 and the 2 are integers and neither of them are floats and thus Python returns the result as an integer (throwing away the 0.5).

Consider the following example:

```
1 In [#]: a = 5
2
3 In [#]: a / 2
4 Out [#]: 2
5
6 In [#]: a / 2.0
7 Out [#]: 2.5
8
9 In [#]: b = 5.0
10
11 In [#]: b / 2
12 Out [#]: 2.5
13
14 In [#]:
```

As you can see from the above example:

- If both the numerator and the denominator are integers then Python returns the result as an integer
- If either the numerator or the denominator is a float then Python will return the result as a float



#### Take Note:

- 1) Any Python computation with only integers will return integer results, i.e. `In [#]: 5 / 2` returns 2
- 2) This holds for assignment operators as well:

```
1 In [#]: a = 5
2
3 In [#]: a /= 2
4
5 In [#]: a
6 Out [#]: 2
```

3) Make sure you understand this behaviour and that you practice a few example to see the difference between integers and floats.

#### 3.1.4.1 Why should you care about this integer division behaviour?

Integer division can result in surprising bugs: suppose you are writing code to compute the mean value  $m = (x + y)/2$  of two numbers  $x$  and  $y$ . The first attempt of writing this may read:

```
1 m = (x + y) / 2
```

Suppose this is tested with  $x = 0.5$  and  $y = 0.5$ , then the example above computes the correct answers  $m = 0.5$  (because  $0.5 + 0.5 = 1.0$ , i.e. a  $1.0$  is a floating point number, and thus  $1.0 / 2$  evaluates to  $0.5$ ).

Or we could use  $x = 10$  and  $y = 30$ , and because  $10 + 30 = 40$  and  $40 / 2$  evaluates to  $20$ , we get the correct answer  $m = 20$ . However, if the integers  $x = 0$  and  $y = 1$  would come up, then the code returns  $m = 0$  (because  $0 + 1 = 1$  and  $1 / 2$  evaluates to  $0$ ) whereas  $m = 0.5$  would have been the right answer. We have many possibilities to change the example above to work safely, including these three versions:

```
1 m = (x + y) / 2.0
```

```
1 m = (x + y) * 0.5
```

```
1 m = float(x + y) / 2
```

#### 3.1.4.2 How to avoid integer division

There are two ways to avoid the integer division problem and ensure that the results of the computation is always returned as a floating point number:

1. Make use of Python's future division: it has been decided that from Python version 3.0 onwards the division operator will return a floating point number even if both the numerator and denominator are integers. This feature can be activated in older Python versions (2.x) with the `from __future__ import division` statement:

```
1 In [#]: from __future__ import division
2
3 In [#]: 15/6
4 Out [#]: 2.5
5
6 In [#]: 21/7
7 Out [#]: 3.0
8
9 In [#]:
```

2. Ensure that at least one number (either the numerator or the denominator) is a floating point number (as discussed in section 3.1.4), the division operator will return a floating point number. This can be done by writing 15. or 15.0 instead of 15, or by forcing conversion of the number to a float, i.e. use float(15) instead of 15:

```
1 In [#]: 15 / 6
2 Out [#]: 2
3
4 In [#]: 15. / 6
5 Out [#]: 2.5
6
7 In [#]: 15.0 / 6
8 Out [#]: 2.5
9
10 In [#]: 15 / 6.
11 Out [#]: 2.5
12
13 In [#]: float(15) / 6
14 Out [#]: 2.5
15
16 In [#]: 15 / float(6)
17 Out [#]: 2.5
18
19 In [#]:
```

## 3.2 String Objects

So far, I have only used names that have been bound to numerical objects. I hope you realise that numbers aren't the only type of information available, so you should recognise

the need to also save other types of information. The type of non-numerical object we will use most often is the string object. A string object is created by enclosing characters in either single, double or triple quotes e.g.

```
1 In [#]: name = 'John'
2
3 In [#]: name = "John"
4
5 In [#]: name = '''John'''
6
7 In [#]: name = """John"""
```

The above statements create a string object in memory that has the content John and then binds the name `name` to this string object. We can assign any content to a string object, but remember to enclose the string in either single, double or triple quotes. Valid examples are

```
1 address = "101 My street, My Suburb, My Town, 0001."
2 phone_number = "(012) 420 2431"
3 zodiac_sign = "Gemini"
```

If you want to include a double quote character as part of the string, add a backslash symbol (\) before the quote character (otherwise Python will interpret the quote character as the end of the string) i.e.

```
1 quote = "Lecturer: \"I'm tired of lazy students.\" "
2 reply = "Students: \"We're tired, not lazy!\" "
```

After you entered the above statements, type `quote` and press enter or type `print quote` and press enter. The following will appear in the *Command Window*:

```
1 In [#]: quote = "Lecturer: \"I'm tired of lazy students.\" "
2
3 In [#]: print quote
4 Lecturer: "I'm tired of lazy students."
```

Do you see that the double quote characters are now interpreted as part of the string object? String objects are often used to create appropriate feedback to the users of your programs.



#### More Info:

The `print` command is used to print information to the *IPython Console*:

`print 2.0` - will print the value 2.0

`print "hello"` - will print the string `hello`

`print var1` - will print the value of the variable `var1`

`print var1, var2` - will print the value of the variable `var1` followed by a space followed by the value of the variable `var2`

### 3.3 Boolean Objects

Another type of object I want you to be aware of is a boolean object, which can only have one of two values, namely `True` or `False`. This type of object is commonly used in `while` loop statements (Section 5.1) and `if` statements (Section 6.1), and I will show examples of the use of this object in the relevant sections of these notes.

You create the boolean object, with the `True` or `False` values, and bind a name to it the same way you would any other object:

```
1 In [#]: a = True
2
3 In [#]: a
4 Out [#]: True
5
6 In [#]: b == False
7
8 In [#]:
```



#### Take Note:

`True` and `False` must have the capital letters T and F respectively.

If you use `true` (lower case t) Python will give you an error message.

## 3.4 Acceptable Names

Names have the same restrictions as program names. The case sensitive argument also applies to names and therefore the name `Value1`, `VALUE1` and `VaLuE1` are distinct and each can be bound to the same or to a different object.

When you bind the name of an imported function name in Python to a new object, you can no longer access the original function any more. Consider the example where you bind the name `sin` (after importing `sin` from the `math` module) to a new object:

```
1 In [#]: from math import sin
2
3 In [#]: type(sin)
4 Out[#]: builtin_function_or_method
5
6 In [#]: sin = 4.0
7
8 In [#]: type(sin)
9 Out[#]: float
10
11 In [#]:
```

From lines 3 and 4 you can see that the imported name `sin` is recognised as a function. When you bind the name `sin` to the object `4.0` (line 6) the previous binding to the original function is removed. You can also see in lines 8 and 9 that `sin` is now a float object.

If you tried to call `sin` now as if it were a function e.g. `sin(1)` you would get the following error message:

```
11 In [#]: sin(1)
12 -----
13 TypeError      Traceback (most recent call last)
14 <ipython-input-75-93e746b069d9> in <module>()
15     ----> 1 sin(1)
16
17 TypeError: 'float' object is not callable
18
19 In [#]:
```

**Take Note:**

Names should not be any of the Python keywords: type `help('keywords')` to see the list:

1	and	elif	if	print
2	as	else	import	raise
3	assert	except	in	return
4	break	exec	is	try
5	class	finally	lambda	while
6	continue	for	not	with
7	def	from	or	yield
8	del	global	pass	

**More Info:**

PEP8 standard for names:

- “lowercase with words separated by underscores as necessary to improve readability”, e.g. `my_value` and not `My_Value`.
- names should be at least 3 characters in length and descriptive.

The PEP8 standard is available at:

<http://www.python.org/dev/peps/pep-0008/>

## 3.5 Summary

Summary of the objects encountered thus far:

- Numerical Objects:
  - Integer Objects: `my_int = 5`
  - Float Objects (real numbers): `my_float = 2.3`
- String Objects: `my_name = "Logan"`
- Boolean Objects: `is_correct = True`

**More Info:**

You can also convert between object types using the following statements:

1. `int()` - Convert a string or number to an integer
2. `float()` - Convert a string or number to a floating point number (real number)
3. `str()` - Return a string representation of a number

More information on each of these will be given as and when they are used in the examples in these note.

## 3.6 Exercises

1. Assigning an object to the name `calc = 5*5*5 + 4*4`. Which operators are executed first (addition and subtraction) or (multiplication and division)?  
Redo the computation `a = 5*5*5 + 4*4` using only `**` and `+` operators as opposed to `*` and `+`.
2. Compute the unknown variable  $a$  in the equation:

$$a \times \sqrt{b} = \sin(d) + c \quad (3.1)$$

with  $b = \ln(7)$ ,  $c = 16^2$  and  $d = \pi/2$ .

[Answer : 184.23]

3. Compute the unknown variable  $a$  in the equation:

$$a \times \sqrt{b} + c = \frac{\cos(d)}{\tan(e)} \quad (3.2)$$

with  $b = \log_{10}(90)$ ,  $c = 4\sin(\pi/8)$ ,  $d = \pi/4$  and  $e = \sqrt{111}$ .

4. Compute the following result:

$$\frac{(\pi + 5)^2(\sqrt{5} - e^2)}{(1 - e^\pi)(1 + \pi^e)} \quad (3.3)$$

Attempt at least two methods to compute the above i.e. enter the complete computations as a single programming line, and break the computation into as many lines as required to make the logic a bit easier.

5. Compute the terminal velocity of a body in a non-dense medium by using the following:

$$v_t = \sqrt{\frac{2mg}{\rho AC_d}}$$

with:

$C_d$  – coefficient of drag 0-1(dimensionless)

$m$  – mass(kg)

$\rho$  – density of medium in which body is falling ( $kg/m^3$ )

$A$  – cross sectional area of body perpendicular to direction of flow ( $m^2$ )

6. Compute the velocity at a given time  $t$  given by:

$$v(t) = \sqrt{\frac{2mg}{\rho AC_d}} \tanh \left( t \sqrt{\frac{g\rho C_d A}{2m}} \right)$$



# Chapter 4

## Unconditional loop

Throughout this course, I will use mathematical concepts to illustrate programming principles. I do this because that is why engineers need to program: we will find some mathematical expression or method in some textbook, and we want to use it to solve an engineering problem.

As a first example, consider the sum of the first 100 integers:

$$\sum_{n=1}^{100} n \quad (4.1)$$

which can be written as  $1 + 2 + 3 + 4 + \cdots + 100$ . We would like to add the first hundred integers, the question is how to go about it (if we pretend not to know about Gauss's little formula). We can add the first hundred numbers by adding each consecutive number to the sum already calculated. Clearly this would be a time consuming approach which becomes more time consuming when we increase the first hundred to a thousand or million.

Let us go through a calculator exercise in Python and add the first couple of numbers to a name called `mysum`. Remember we first need to define the name `mysum`. Here is where logic and understanding comes into play as more than one possibility exist. I'll explain two possible approaches that come to mind.

We can define `mysum` to be equal to the first term we want to add i.e. 1 and then add the numbers 2 to 100 to it. Therefore a program that adds the first five numbers appears as follows:

OR

```

1 mysum = 1
2 mysum = mysum + 2
3 mysum = mysum + 3
4 mysum = mysum + 4
5 mysum = mysum + 5
6 print mysum

```

```

1 mysum = 1
2 mysum += 2
3 mysum += 3
4 mysum += 4
5 mysum += 5
6 print mysum

```

Alternatively, because we are doing addition we can define `mysum` to be equal to 0 since  $0 + a = a$ . Therefore setting `mysum` equal to 0 does not affect the calculation. Adding from 0 to 100 is the same as adding from 1 to 100. This approach results in the program:

OR

```

1 mysum = 0
2 mysum = mysum + 1
3 mysum = mysum + 2
4 mysum = mysum + 3
5 mysum = mysum + 4
6 mysum = mysum + 5
7 print mysum

```

```

1 mysum = 0
2 mysum += 1
3 mysum += 2
4 mysum += 3
5 mysum += 4
6 mysum += 5
7 print mysum

```

Both these approaches are tedious, especially when considering that we would like to add the first 100 numbers. It is clear that in both approaches there is a pattern of what we are doing

- We firstly initialize the name `mysum`.
- We then add a number from the sequence to `mysum` and store the result back into `mysum`.
- We continue until we have added all the numbers in the sequence.

Firstly, we recognise that we know exactly how many numbers we wanted to add i.e. from 1 to 100. Secondly we see a pattern when we add the numbers. We can use a `for` loop statement to exploit these two aspects of the problem we wanted to solve. Before we jump into the `for` loop statement I first need to cover the `range` function as this will be used in the `for` loop statement itself.

## 4.1 The range function

The `range` function is used to create a list of integers. The syntax of the `range` function is:

```
1   range(start, end, increment)
```

Where `start` is the first integer in the list, `end` is the last integer in the list, and `increment` is the step size. So we can use the `range` functions as follows:

```
1   In [#]: range(0, 10, 1)
2   Out[#]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
3
4   In [#]:
```



### Take Note:

The last integer in the list (`end`) is always omitted !!!

So, for example, `range(5, 12, 1)` will give a list of integers from 5 to 11 in incremental steps of 1.

The first integer in the list (`start`) is always included.

From this example we can see that the `range` function has created a list of integers starting from 0 and ending at 9. The `range` function created this list by first starting at 0 (`start = 0`) and then adding the increment 1 (`increment = 1`) to this value to get the next value, it keeps doing this as long as the value is less than the end value 10 (`end = 10`).

Both the `increment` and the `start` values can be omitted (left out). If we don't specify a `start` value, the default value of 0 is used. If we don't specify an `increment` value, the default value of 1 is used. Consider the following examples:

`start = 0, end = 10, and increment = 3` are specified:

```
1   In [#]: range(0, 10, 3)
2   Out[#]: [0, 3, 6, 9]
3
4   In [#]:
```

`start = 2, end = 10, and increment = 5` are specified:

```
1 In [#]: range(2, 10, 5)
2 Out[#]: [2, 7]
3
4 In [#]:
```

`start = 0 and end = 10` are specified. `increment` is equal to the default value of 1:

```
1 In [#]: range(0, 10)
2 Out[#]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
3
4 In [#]:
```

`start = 6 and end = 10` are specified. `increment` is equal to the default value of 1:

```
1 In [#]: range(6, 10)
2 Out[#]: [6, 7, 8, 9]
3
4 In [#]:
```

Only `end = 5` is specified. `increment` is equal to the default value of 1 and `start` is equal to the default value of 0:

```
1 In [#]: range(5)
2 Out[#]: [0, 1, 2, 3, 4]
3
4 In [#]:
```

## 4.2 The for loop statement

A `for` loop statement is a command used to repeat a section of code for a predetermined number of times. The syntax of the `for` loop statement is:

---

```
1   for index in range(first, last, increment):  
2       program statements to be repeated
```



#### Take Note:

Note the double colon (:) at the end of the `for` loop statement in line 1. This tells Python where the `program statements` inside the `for` loop start.

If the double colon (:) is left out you will see an error message.

where `index` is a name that gets bound to each of the integer objects in the list of integers (created by the `range` command) during every repetition of the `for` loop until it reaches the last integer object in this list. This happens automatically. We do not need to define `index` as a variable having an initial value `first` and we do not have to increment `index` with `increment` every time the loop is repeated. In fact, we must not. If we do these things, the program will not function properly. Any number of statements can be contained within the `for` loop statement. A few simple examples of `for` loop statements follow.

#### 4.2.1 Example 1

```
1   for i in range(3):  
2       print "i = ", i
```

which creates the following output:

```
1   i = 0  
2   i = 1  
3   i = 2
```

This example can also be written as follows:

```
1   int_list = range(3)  
2   print "int list = ", int_list  
3   for i in int_list:  
4       print "i = ", i
```

which creates the following output:

```

1 int_list = [0, 1, 2]
2 i = 0
3 i = 1
4 i = 2

```

Line No.	Python Statement	Names	Objects (in memory)			
1	a = range(3)	a →	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>1</td><td>2</td></tr></table>	0	1	2
0	1	2				
2	for i in a:					
3-0	print "i = ", i	i ↗				
3-1	print "i = ", i	i ↗				
3-2	print "i = ", i	i ↗				

Figure 4.1: Memory model – `for` loop statement (example 1)

You can see from this example that the name `i` is bound to each consecutive integer object of `int_list` as the `for` loop progresses. The memory model shown for this example (figure 4.1) is not 100% correct, but for explanation purposes of this section it is fine. This memory model will be reviewed and explained correctly in section 9.1.2.4.

#### 4.2.2 Example 2

```

1 for i in range(0, 6, 2):
2     print "i = ", i

```

which creates the following output:

```

1 i = 0
2 i = 2
3 i = 4

```

### 4.2.3 Example 3

```
1 for i in range(5, 0, -2):
2     print "i = ", i
```

which creates the following output:

```
1 i = 5
2 i = 3
3 i = 1
```

Again this example can also be written as follows:

```
1 int_list = range(5, 0, -2)
2 print "int_list = ", int_list
3 for i in int_list:
4     print "i = ", i
```

which creates the following output:

```
1 int_list = [5, 3, 1]
2 i = 5
3 i = 3
4 i = 1
```

### 4.2.4 Example 4

```
1 cnt = 0
2 print "cnt = ", cnt
3 for k in range(5):
4     cnt = cnt + 1
5     print "cnt = ", cnt
```

which creates the following output:

```
1 cnt = 0
2 cnt = 1
3 cnt = 2
4 cnt = 3
5 cnt = 4
6 cnt = 5
```

This last example is included to demonstrate that the index variable used in the **for** loop statement (**k** in the case above) does not have to be used inside the **for** loop statement. In such a case the **for** loop is simply used to repeat a certain set of commands (or a single command as above) a known number of times. It also illustrates that only the commands contained in the **for** loop statement are repeated. Since *cnt* = 0 is outside the **for** loop statement it is only executed once at the start, when the program is executed.

#### 4.2.5 Indentation

Indentation means the amount of white space placed in front of your code. Python, unlike other programming languages, uses this white space to determine what statements are contained inside the **for** loop statement. Other programming languages will either require brackets (, (), []) or keywords ('end') to tell that programming language which statements are part of the **for** loop statement. Python uses white space, 4 spaces to be exact !!! Consider the following example:

```
1 cnt = 0
2 for i in range(3):
3     cnt += i
4     print "I'm inside the for loop"
5 print "I'm outside the for loop"
```

When you execute this code you will see the following output:

```
1 I'm inside the for loop
2 I'm inside the for loop
3 I'm inside the for loop
4 I'm outside the for loop
```

From this you can see that line 4 is executed 3 times by the `for` loop statement and after the `for` loop has finished only then is line 5 executed.



#### Take Note:

Python requires 4 spaces before `all program statements` inside a `for` loop statement. The example above has 4 spaces in front of the statements on lines 3 and 4 to indicate that these lines are inside the `for` loop statement.

Lets look at what happens if the code indentation is wrong. The above example has been modified and only 3 spaces have been added in front of the statement on line 4:

```
1 a = 0
2 for i in range(3):
3     a += i
4     print "I'm not indented properly"
5 print "I'm outside the for loop"
```

If you run this example now you will see an error message similar to the following:

```
1     print "I'm not indented properly"
2
3 IndentationError: unindent does not match any
4         outer indentation level
```



#### Take Note:

Be VERY careful and aware of this when typing your programs, as this can sometimes lead to unexpected results and logic errors. There is a big difference between the following two examples:

```
1 cnt = 0
2 for i in range(10):
3     print "cnt = ", cnt
4     cnt += i
5 print "cnt = ", cnt
```

```
1 cnt = 0
2 for i in range(10):
3     print "cnt = ", cnt
```

```
4     cnt += i
5     print "cnt = ", cnt
```

Make sure you know the difference in the results of these two examples and why they are different !!!



### More Info:

*Spyder* will automatically add 4 spaces when you push the `tab` key. And `Shift + tab` will automatically remove 4 spaces. This can also be used if you have highlighted several lines of code. Try it !!

If this doesn't work properly, Click on the `Tools` menu in *Spyder*, then Click on `Preferences`, then Click on `Editor` (on the left). Under the `Advanced Settings` tab (on the right) make sure that "Indentation characters" is set to "4 spaces" and that "Tab always indent" is un-ticked.

## 4.3 Summing using a for loop

Let's revisit the program that computes the sum of the first 100 integers. We can easily perform this inside a `for` loop statement. Let's consider the first approach where we initialised `mysum` to 1:

```
1 mysum = 1
2 for nextnumber in range(2, 101):
3     mysum = mysum + nextnumber
4 print mysum
```

If we consider our second approach where we initialised `mysum` to 0 the program would look something like this:

```
1 mysum = 0
2 for nextnumber in range(1, 101):
3     mysum += nextnumber
4 print mysum
```

Can you see how easy the for statement makes the computation. If we want to sum the first 512 numbers we only have to change **end** value used in the **range** function in the **for** loop statement.

Alternatively, we can use the for loop statement to merely execute the loop a specified number of times without using the **index** name inside the for loop. Therefore, to sum the first hundred integers together, we need to generate the correct integer to add to the sum ourselves as shown in the example below:

```
1 mysum = 0
2 nextnumber = 1
3 for i in range(101, 201):
4     mysum += nextnumber
5     nextnumber += 1
6 print mysum
```

The index name **i** starts at 101. It increments by 1 every time we go through the for loop and stops when the index name **i** is equal to 200. Clearly **i** does not contain any of the integers between 1 and 100 and since we want to add the first hundred numbers we need to compute the correct integer to add ourselves. This is achieved by using the **nextnumber** name which is initialised to 1 in line 2. We add **nextnumber** to **mysum** where after we add 1 to **nextnumber** so that **nextnumber** is equal to 2 the next time we go through the loop. Once we added **nextnumber** the second time we go through the loop 1 will be added to **nextnumber** so that **nextnumber** is equal to 3 and so on. Every time, before we complete the loop and after we've added **nextnumber** to **mysum** we add 1 to **nextnumber** so that the next number in the summation sequence is added the following time we go through the loop.

## 4.4 Factorial using a for loop

Consider a program that computes  $10!$  i.e.  $1 \times 2 \times 3 \dots 10$ . Again we know exactly how many times we want to multiply numbers with each other. We also see a pattern in what we want to compute. We can therefore use a **for** loop statement to compute it.

Let us call the name **myfactorial**. Before we can use **myfactorial** on the right hand side we need to initialise it. Would it work if we initialize as **myfactorial = 0**? What would the result be? As we are busy with multiplication the result will always be 0.

In this case we need to initialise **myfactorial** to 1. This can be viewed as a number

that does not affect our calculation i.e.  $a \times 1 = a$  or either as the first number of the sequence. For the former, the program is

```
1 myfactorial = 1
2 for nextnumber in range(1, 11):
3     myfactorial = myfactorial * nextnumber
4 print myfactorial
```

and for the latter, we start multiplying from 2 instead of 1,

```
1 myfactorial = 1
2 for nextnumber in range(2, 11):
3     myfactorial *= nextnumber
4 print myfactorial
```



#### Take Note:

It is important to realise that both programs are correct. More importantly you need to develop your own way of coding your programs as there is no unique solution to a programming problem. Programs are based on the logic and understanding of the programmer.

**Think and understand before you type.**

For instance we can use the for loop to merely execute the loop a specified number without using the `index` name inside the for loop. Therefore, to compute the product of the first 10 integers we need to generate the correct integer to multiply to the product ourselves as shown in the example below:

```
1 myfactorial = 1
2 nextnumber = 2
3 for i in range(22, 31):
4     myfactorial *= nextnumber
5     nextnumber += 1
6 print myfactorial
```

The index name `i` starts at 22. It increments by 1 every time we go through the for loop and stops when the index name `i` is equal to 30. Clearly `i` does not contain any of the integers between 1 and 10. As we want to compute the product the first 10 integers

we need to compute the correct integer that needs to be multiplied with `myfactorial` ourselves. This is achieved by using the `nextnumber` name which is initialised to 2 in line 2. We then multiply `nextnumber` to `myfactorial` where after we add 1 to `nextnumber` so that `nextnumber` is equal to 3 the next time we go through the loop. Once we have multiplied `nextnumber` to `myfactorial` the second time we go through the loop 1 will be added to `nextnumber` so that `nextnumber` is equal to 4 and so on. Everytime before we complete the loop and after we've multiplied `nextnumber` with `myfactorial` we add 1 to `nextnumber` so that the next number in the factorial sequence is multiplied the following time we go through the loop.



#### Take Note:

The `for` loop statement is used for unconditional looping, i.e. we know in advance how many times a section of code has to be repeated. At no point do we have to check if a certain condition is True before we execute the loop another time.



#### More Info:

Remember that you can use the `math.factorial(10)` function to verify the results of these examples in this section.

## 4.5 Fibonacci series using the `for` statement

A very famous series was proposed by the mathematician Fibonacci: The first two terms of the series are given by  $L_0 = 0$  and  $L_1 = 1$ . Hereafter, each new term in the series is given by the sum of the previous two terms in the series i.e.

$$L_k = L_{k-1} + L_{k-2} \quad \text{for } k = 2, 3, 4, \dots \quad (4.2)$$

Using the above formula, the following sequence is generated: 0, 1, 1, 2, 3, 5, 8, 13, ... .

Let us first do this problem step by step using Python as a calculator. Let us compute the fifth term ( $k = 4$ ),

```

1 term_k_minus_2 = 0
2 term_k_minus_1 = 1
3 term_k = term_k_minus_1 + term_k_minus_2
4 term_k_plus_1 = term_k + term_k_minus_1
5 term_k_plus_2 = term_k_plus_1 + term_k
6 print term_k_plus_2

```

Let's pretend we are only interested in the fifth term ( $k = 4$ ). The solution above requires five names with five values. If we require the twentieth term ( $k = 19$ ), we would have twenty names using this approach. Here we have written a program that does not exploit the pattern we see from the mathematics i.e. the Fibonacci series only uses the  $(k - 1)$  and  $(k - 2)$  terms to compute the  $k^{\text{th}}$  term i.e. for  $k = 2$  we require  $k = 1$  and  $k = 0$  terms, for  $k = 3$  we require  $k = 2$  and  $k = 1$  terms etc.

We only have to remember the previous two terms to compute the next term. Let us do that by swapping objects. The computed  $k^{\text{th}}$  term in the  $k^{\text{th}}$  iteration becomes the  $(k - 1)^{\text{th}}$  when we increment  $k$  by 1. Similarly the  $(k - 1)^{\text{th}}$  term becomes the  $(k - 2)^{\text{th}}$  term. Let us rewrite the above code to use only three variables,

```
1 term_k_minus_2 = 0
2 term_k_minus_1 = 1
3 term_k = term_k_minus_1 + term_k_minus_2 #k = 2
4 term_k_minus_2 = term_k_minus_1
5 term_k_minus_1 = term_k
6 term_k = term_k_minus_1 + term_k_minus_2 #k = 3
7 term_k_minus_2 = term_k_minus_1
8 term_k_minus_1 = term_k
9 term_k = term_k_minus_1 + term_k_minus_2 #k = 4
10 print term_k
```



#### Take Note:

The sharp symbol (#) in a Python program is used to add comments to that program giving more information regarding a certain line or several lines of code. Anything written after the sharp symbol (#) will not be executed by Python.

```
1 for i in range(10):
2     # print "I am not part of the program"
3     # this line nor the one above will be executed
4     print "I am part of the program"
```

Make sure you understand what the output of this example will be and why !

Let us see how we would go about computing the 25<sup>th</sup> term ( $k = 24$ ) of the Fibonacci series. We need to compute an additional 23 terms since the first two terms are given. Can you see why the `for` loop statement is applicable? We know how many terms we want to compute and we have a pattern of what we would like to repeat 23 times. So, let us compute the twenty fifth term ( $k = 24$ ) using the `for` loop statement:

```
1 term_k_minus_2 = 0
2 term_k_minus_1 = 1
3 for k in range(2, 25):
4     term_k = term_k_minus_1 + term_k_minus_2
5     term_k_minus_2 = term_k_minus_1
6     term_k_minus_1 = term_k
7 print "term_k = ", term_k
```

It is important to note that when we swap the objects we do it in such a way that we don't end up with all names bound to the same object. We therefore start with the `term_k_minus_2` name in line 5, since its current content was used to compute the current  $k^{\text{th}}$  Fibonacci term but it is not used to compute the next Fibonacci term. We therefore bind the name `term_k_minus_2` to the same object bound by `term_k_minus_1`. Thereafter we bind the name `term_k_minus_1` to the same object bound by `term_k`, that we just computed. Now we are ready to increment  $k$  to compute the next term of the Fibonacci series. When  $k$  is incremented, and the loop repeats, the name `term_k` will get bound to a new object resulting from `term_k_minus_1 + term_k_minus_2`.

The program is ended with a single statement after the `for` loop, with the only purpose to provide some output: we want to print the value contained in the name `term_k` to the *IPython Console*. We do so by using the `print` statement. The following output is produced:

```
9 term_k = 46368
```

As stated before, we could write many programs that will do the same thing depending on the logic the programmer follows. The important aspect is to understand what you want to do, work out the logic of how you would do it. You can then program what you require to end up with a program that will solve your problem.

Here is another example of a program that computes the 24<sup>th</sup> term of the Fibonacci series:

```
1 term_k_minus_2 = 0
2 term_k_minus_1 = 1
3 term_k = term_k_minus_1 + term_k_minus_2
4 for k in range(3, 25):
5     term_k_minus_2 = term_k_minus_1
6     term_k_minus_1 = term_k
```

```

7     term_k = term_k_minus_1 + term_k_minus_2
8     print "term_k = ", term_k

```

See if you can work out the logic behind this program and why it works.

## 4.6 Taylor series using the for loop statement

A Taylor series is an approximation to any function using a series of polynomial terms. In the limit of using an infinite number of terms, the approximation is exact. Review Taylor series in your mathematics textbook if you've forgotten the details. Although a Taylor series does not intuitively require the **for** loop statement, it is useful to illustrate the **for** loop statement.

A feature of a Taylor series approximation is that each new term contributes less to the function approximation than the terms before. So, at some point the additional Taylor terms don't contribute much to the overall function value and we might as well stop adding additional terms. Therefore, when we use a computer to compute a function value using a Taylor series approximation, we will always use a finite number of terms. As an example, the value of  $\pi$  can be computed from the following Taylor series:

$$\pi = 4 \sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1} = 4 \left[ 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \dots \right] \quad (4.3)$$

Suppose that we want to approximate the value of  $\pi$  by using the above Taylor series. If we want to use a **for** loop statement to perform this computation, we must decide how many terms we want to add up. Suppose we choose 100 terms. A program that accomplishes this is:

```

1 taylor_pi = 0.0
2 for k in range(100):
3     taylor_pi += (4.0 * (-1)**k) / (2*k + 1)
4 print "pi = ", taylor_pi

```

This programs produces the output

```
pi = 3.13159290356
```

Compared to the true value of  $\pi = 3.14159\dots$ , the 1st 100 terms of the Taylor series approximation is only accurate to 2 digits. Note that since we start counting at  $k = 0$ , we only need to repeat the loop up to  $k = 99$  to add up the first 100 terms. Also note that we must initialise the name `taylor_pi` to zero outside the `for` loop statement. If we do not do this, Python will produce an error message upon executing line 3 for the first time: we're instructing Python to add the value  $4*(-1)^k/(2*k+1)$  to the name `taylor_pi` (whose value is unknown). If we did not assign zero to the name `taylor_pi` outside the `for` loop statement, we cannot expect Python to perform the required computation. The program is ended with a single statement after the `for` loop statement, with the only purpose to provide some output: we want to print the function value approximation contained in the name `taylor_pi` to the *IPython Console*.

You can also perform the task by first computing the next Taylor term in the series and then adding it to the sum of terms in a next line:

```

1 taylor_pi = 0.0
2 for k in range(100):
3     taylor_term = (4.0 * (-1)**k) / (2*k + 1)
4     taylor_pi += taylor_term
5 print "pi = ", taylor_pi

```

If we want to compute  $\pi$  more accurately, we have to sum more terms in the Taylor series. If we compute the sum of the first 1000 terms (by simply changing line 2 to: `for k in range(1000)`), the output is

```
pi = 3.14059265384
```

Even the first 1000 terms in the Taylor series produces an approximate value for  $\pi$  that is only accurate to 3 digits. Before you decide that Taylor series approximations are of no use, consider the approximation to the exponential function:

$$e^x = 1 + x + \frac{1}{2}x^2 + \frac{1}{6}x^3 + \dots = \sum_{k=0}^{\infty} \frac{x^k}{k!} \quad (4.4)$$

Let's use this Taylor series approximation to compute the value of  $e$  (i.e. compute  $e^x$  for  $x = 1$ ). The following program will add the first 20 terms:

```

1 from math import factorial
2

```

```
3 taylor_exp = 0.0
4 x_val = 1.0
5 for k in range(20):
6     taylor_exp += x_val**k / factorial(k)
7 print "exp = ", taylor_exp
```

You should already be quite familiar with the `math.factorial(k)` function to compute  $k!$ . The example above produces the following output:

```
exp = 2.71828182846
```

which is accurate to all 15 digits. Here we have an example where a few terms in this Taylor series produces a very accurate approximation, whereas the 1000-term approximation to  $\pi$  was still inaccurate.

Let us pretend that the `math.factorial` function does not exist. How could we go about computing the factorial of a number  $k$ . We could use the `for` loop statement as we have done in Section 4.4 to compute the factorial of  $k$ , i.e.  $k! = 1 \times 2 \times 3 \cdots \times (k - 2) \times (k - 1) \times k$ . Before we add a term to `taylor_exp` we must make sure we have computed the factorial correctly. The following program computes the first 20 terms of the Taylor series of  $e^x$  by using the `for` loop statement to compute the factorial:

```
1 taylor_exp = 0.0
2 x_val = 1.0
3 for k in range(20):
4
5     factorial = 1
6     for i in range(2, k+1):
7         factorial *= i
8
9     taylor_exp += x_val**k / factorial
10 print "exp = ", taylor_exp
```

In the program above we compute the factorial of  $k$  for every iteration from scratch i.e. during every iteration we initialize `factorial` to 1 (line 5) and then compute the appropriate factorial before we compute the next term we want to add to `taylor_exp`.

**Take Note:**

Notice the indentation for this example.

Lines 4 to 9 require 4 space in front of the program statements to tell Python that they are inside the first (outer) `for` loop.

Lines 7 and 8 require 4 + 4 spaces in front of the program statements to tell Python that they are inside the second (inner) `for` loop.

Make sure you understand the behaviour and output of the following two examples and why they are different !!

```

1 a = 0.0
2 for i in range(10):
3     a += 10 * 1
4     for j in range(5):
5         a /= 2
6 print "a = ", a

```

```

1 a = 0.0
2 for i in range(10):
3     a += 10 * 1
4 for j in range(5):
5     a /= 2
6 print "a = ", a

```

However, if consider that  $6! = 6 \times 5! = 6 \times 5 \times 4!$  and inspect Equation (4.3) we see that we are required to compute  $k!$ ,  $k = 0, 1, 2, \dots$  as we proceed through the iterations of the `for` loop statement. We could therefore use the factorial of the previous iteration and multiply it with the required number to compute the factorial of the current iteration. The following program is an example of such an implementation:

```

1 taylor_exp = 1.0      #changed from 0.0 to 1.0
2 x_val = 1.0
3 factorial = 1
4 for k in range(1, 19):
5     factorial *= k
6     taylor_exp += x_val**k / factorial
7 print "exp = ", taylor_exp

```

We initialise the name `factorial` outside the `for` loop statement to 1. Then for the

current iteration we compute the appropriate factorial for this iteration by multiplying the previously calculated factorial of the previous iteration with `k`. Also note that instead of initialising `taylor_exp` to 0 we now initialise `taylor_exp` to 1. Also in the `for` loop statement we start `k` at 1 instead of 0. By initialising `taylor_exp` to 1 we have already included the term when  $k$  is equal to 0 into our summation. We can therefore start `k` at 1 instead of 0 to avoid multiplying `factorial` with 0 i.e. when `k = 0`:

```
5   factorial *= k
```

then `factorial` is equal to zero and remains so thereafter irrespective of the value of `k`.



#### Take Note:

It is important to note that when you change a part of your code, you have to revisit the rest of the code to ensure that everything is still working as expected.

## 4.7 Exercises

1. Write a program where you multiply two positive integers by using the addition `+` operator. You are only allowed to use the multiplication `*` operator to verify your answer. First assign the two numbers to two names: `number1` and `number2`. Recall that you can write

$$7 \times 4 = 7 + 7 + 7 + 7 = 4 + 4 + 4 + 4 + 4 + 4 \quad (4.5)$$

2. Simulates the throwing of a dice 5 times using a `for` loop statement and `print` the result of each throw to the screen.

[Hint: Use the `random` module to generate a random integer between 1 to 6.]

3. I heard a story that the famous mathematician Gauss was rather a handful in class. He always completed his assignment much faster than the other kids and then created havoc due to boredom. In an attempt to keep Gauss busy for a while, his teacher told Gauss to add the first 1000 integers, as “punishment”. A minute later Gauss completed his task, because he reasoned as follows: If I have to add  $0, 1, 2, \dots, N - 2, N - 1, N$ , then I can create the number pairs  $1 + N, 2 + (N - 1), 3 + (N - 2)$  and so on. The sum of each of these pairs is  $N + 1$ , and there is  $\frac{N}{2}$  of these pairs. So the sum is simply given by

$$\sum_{n=0}^N n = \frac{N}{2}(N + 1) \quad (4.6)$$

You can test the above formula for small values of  $N$ . Then write a *Python* program that checks Gauss's formula for  $N = 1000$ . So I want you to add the first 1000 integers (using a `for` loop statement), and then compare it to the formula above.

4. We can break up the addition of the first 1000 integers by adding the odd numbers and then by adding the even numbers. You can check your calculation by adding the odd and even numbers which must then equal the answer given in the previous question.

Add all the odd numbers between 0 and 1000 i.e.

$$\sum_{n=1}^{500} 2n - 1 \quad (4.7)$$

Now add all the even numbers between 0 and 1000 i.e.

$$\sum_{n=1}^{500} 2n \quad (4.8)$$

and store it in another variable.

It is expected that the sum of the even sequence to be bigger by 500 than the sum of the odd sequence, since every number of the even sequence is 1 bigger than the corresponding number of the odd sequence and 500 numbers are added.

See if you can program the even and odd summations in two different ways:

- By converting the mathematical statements directly into a program i.e. force yourself to use a `for` loop statement that starts at 1 and ends at 500.
  - By using the `for index in range(begin, end, increment)` such that `index` takes on the values of the numbers that needs to be added in the summation. For example when we want compute  $3+6+9$  we can let `index` take on the values of the numbers in the sequence we want to add by using `for index in range(3, 10, 3)`. Here the variable `index` will take on the following values 3, 6 and 9 as it progresses through the loop.
5. An alternating series is a series where the terms alternate signs. See if you can program the addition of the first 100 terms of the following alternating series:

$$\sum_{n=1}^{100} (-1)^{n+1} \frac{1}{n} = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \dots - \frac{1}{100} \quad (4.9)$$

6. Find the pattern of the following series and then compute the 30th term of the series:

$$\frac{1}{1} + \frac{1}{2} - \frac{1}{4} + \frac{1}{8} - \frac{1}{16} + \frac{1}{32} - \dots$$

7. The Basel problem was first posed by Pietro Mengoli in 1644 and was solved by Leonhard Euler in 1735 which brought Leonhard Euler immediate fame at the age of 28. Euler showed that the sequence

$$\frac{1}{1} + \frac{1}{4} + \frac{1}{9} + \frac{1}{16} + \frac{1}{25} + \frac{1}{36} + \dots$$

converges to  $\pi^2/6$ . Find the pattern of this series and then compute the first 100 and then first 1000 terms of this series and compare the accuracy to  $\pi^2/6$ .

8. A very famous series was proposed by the mathematician Fibonacci: The first two terms of the series are given by  $L_0 = 0$  and  $L_1 = 1$ . Hereafter, each new term in the series is given by the sum of the previous two terms in the series i.e.

$$L_k = L_{k-2} + L_{k-1} \quad \text{for } k = 2, 3, \dots \quad (4.10)$$

Using the above formula, the following sequence is generated: 0, 1, 1, 2, 3, 5, 8, 13, .... Write a program that computes the first 20 terms of the Fibonacci sequence and displays it on the screen.

9. The sine function can be approximated by the following infinite series:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} \dots \quad (4.11)$$

Write a program that will compute up to the  $\frac{x^{19}}{19!}$  term using the unconditional **for** loop statement. Use  $x = \frac{\pi}{4}$  in your code.

10. The wind chill factor (WCF) indicates the perceived air temperature to exposed skin and is given by:

$$WCF = 13.12 + 0.6215T_a - 11.37v^{0.16} + 0.3965T_av^{0.16}$$

with  $T_a$  the air temperature in degrees Celcius and  $v$  the air speed in  $km/h$  (measured at 10 metres standard anemometer height).

Write a program that displays a collection of WCF's using nested **for** loop statements. The temperature ( $T_a$ ) must range from -20 to 55 degrees Celcius in steps of 5 and wind speed ( $v$ ) must rang from 0 to 100  $km/h$  in increments of 10.

# Chapter 5

## Conditional loop

In the previous section we had an example where a few terms in the Taylor series for  $e^x$  produced a very accurate approximation, whereas approximation to  $\pi$  of the 1000<sup>th</sup> term was still inaccurate. So how can we tell in advance how many terms will be required to get an accurate function value? We can't. The only benefit of using the `for` loop statement to compute the approximate value to a function is it's simplicity. In general our goal would be to compute an accurate function value rather than just add some predetermined number of terms. So how should we approach the problem of computing an accurate function value using a Taylor series approximation ?

I can think of a couple of ideas.

1. We can keep on adding terms to the approximation and keep track of the function value. If the function value changes become insignificant, stop adding terms.
2. We can keep track of the size of each additional Taylor term. If the size of the new Taylor term is insignificantly small, stop adding terms.

Clearly the `for` loop statement cannot be used for the above ideas. We need a type of loop that will only be repeated if some condition is true. The `while` loop statement performs this function.

### 5.1 The `while` loop statement

The `while` loop statement is used for conditional loops, i.e. a section of code is repeated only while some specified condition remains true. The syntax of the `while` loop statement is:

```
1  while condition:  
2      program statements to be repeated
```

I'll repeat again, just to emphasise the syntax: the section of code contained in `while` loop statement is repeated only while the `condition` remains `True`.



#### Take Note:

Note the double colon (:) at the end of the `while` loop statement in line 1. This is the same as the `for` loop statement and tells Python where the `program statements` inside the `while` loop start.

If the double colon (:) is left out you will see an error message.



#### Take Note:

The rules for indentation for the `while` loop are the same as for the `for` loop, see Section 4.2.5.

Python requires 4 spaces before `all program statements` inside the `while` loop.

## 5.2 Conditions

Let us look at what these conditions more closely. Conditions can be viewed as booleans (or questions) that we are allowed to make (or ask), which Python will tell us if they are `True` (*yes*) or `False` (*no*). Python can only tell us `True` or `False`.



#### Take Note:

It is important to note that every name in a condition has to be defined.

I'll explain both viewpoints as some students may find considering conditions as booleans easier to understand, rather than to consider the conditions as questions or *vice versa*. It is important to go through both and then to choose the one that is aligned with your way of thinking i.e. the one you best understand and the one you are the most comfortable with.

### 5.2.1 Conditions as booleans

When we view conditions as booleans in Python then every time we make a comparison, Python will tell us whether the comparison is `True` or `False`. We can only make certain types of comparisons. The comparisons we are allowed to make are listed:

1. The two numbers `a` and `b` are equal: (`a == b`)
2. `a` is not equal to `b`: (`a != b`)
3. `a` is smaller than `b`: (`a < b`) or (`b > a`)
4. `a` is greater than `b`: (`a > b`) or (`b < a`)
5. `a` is smaller or equal to `b`: (`a <= b`) or (`b >= a`)
6. `a` is greater or equal to `b`: (`a >= b`) or (`b <= a`)



#### Take Note:

When we make the comparison that two numbers are equal we use a double equal sign (`==`). If we use a single equal sign (`=`) we will bind `a` to the object bound by `b` instead.

Python will tell us whether our comparison(s) is `True` or `False` without hesitation or error. Comparison can also be combined by using the `and` as well as the `or` statements. The syntax for the `and` statement is as follows:

`1 (comparison1) and (comparison2)`

and for the `or` statement

`1 (comparison1) or (comparison2)`

Here, `and` and `or` creates a statement about the answers of `comparison1` and `comparison2` which Python then answers. Python will first evaluate `comparison1` and `comparison2` before it evaluates the `and` or `or` statements. The evaluations of the `and` and `or` statement are summarized in Table 5.1 and Table 5.2 respectively.

comparison1	comparison2	comparison1 and comparison2
True	True	True
False	True	False
True	False	False
False	False	False

Table 5.1: Evaluation of the **and** statement.**Take Note:**

Evaluation of the **and** statement:

Both **comparison1** and **comparison2** have to be **True** for the evaluation of the **and** statement to be **True**.

comparison1	comparison2	comparison1 or comparison2
True	True	True
False	True	True
True	False	True
False	False	False

Table 5.2: Evaluation of the **or** statement.**Take Note:**

Evaluation of the **or** statement:

Either **comparison1** or **comparison2** must be **True** for the evaluation of the **or** statement to be **True**.

Clearly Python's response to the **and** and **or** statements is either **True** or **False**.

### 5.2.2 Conditions as questions

Let us consider conditions as questions. We ask a question and Python will tell us **True** (*yes*) or **False** (*no*). Keep in mind that Python only says **True** or **False**. But to consider conditions as questions we need to think of **True** as *yes* and of **False** as *no*. Again, we can only ask certain types of questions. The questions we are allowed to ask are listed:

1. Are the two numbers **a** and **b** equal?: (**a == b**)
2. Is **a** not equal to **b**? (**a != b**)

3. Is `a` smaller than `b`?: `(a < b)` or `(b > a)`
4. Is `a` greater than `b`?: `(a > b)` or `(b < a)`
5. Is `a` smaller or equal to `b`?: `(a <= b)` or `(b >= a)`
6. Is `a` greater or equal to `b`?: `(a >= b)` or `(b <= a)`

**Take Note:**

Note that when we ask the question of whether two numbers are equal we use a double equal sign (`==`). If we use a single equal sign (`=`) we will bind `a` to the object bound by `b` instead.

Python will answer our questions with without hesitation or error. Questions can also be combined by using the **and** as well as the **or** statements in our questions. The syntax for the **and** question is as follows:

`1 (question1) and (question2)`

and for the **or** statement

`1 (question1) or (question2)`

Here, **and** and **or** asks a questions about `question1` and `question2` for which Python will then answer *yes* (`True`) or *no* (`False`). Before the **and** or **or** question is asked Python answers `question1` and `question2`. Only then does Python ask the **and** or **or** questions. The evaluations of the **and** and **or** statement are summarized in Table 5.3 and Table 5.4 respectively.

<code>question1</code>	<code>question2</code>	<code>question1 and question2</code>
<code>yes (True)</code>	<code>yes (True)</code>	<code>yes (True)</code>
<code>no (False)</code>	<code>yes (True)</code>	<code>no (False)</code>
<code>yes (True)</code>	<code>no (False)</code>	<code>no (False)</code>
<code>no (False)</code>	<code>no (False)</code>	<code>no (False)</code>

Table 5.3: Answers to the **and** question.

**Take Note:**

Evaluation of the **and** statement:

Both **question1 and question2** have to be *yes* (True) for the evaluation of the **and** statement to be *yes* (True).

question1	question2	question1 or question2
<i>yes</i> (True)	<i>yes</i> (True)	<i>yes</i> (True)
<i>no</i> (False)	<i>yes</i> (True)	<i>yes</i> (True)
<i>yes</i> (True)	<i>no</i> (False)	<i>yes</i> (True)
<i>no</i> (False)	<i>no</i> (False)	<i>no</i> (False)

Table 5.4: Answers to the **or** question.



#### Take Note:

Evaluation of the **or** statement:

Either **question1 or question2** must be *yes* (True) for the evaluation of the **or** statement to be *yes* (True).

Clearly Python's response to the **and** and **or** statements is either *yes* (True) or *no* (False).

### 5.2.3 Examples of conditions

As an example, suppose we have to repeat a certain computation if either the value of **a** or **b** is less than some specified name **tolerance**. Such a condition is programmed in Python as

```
1 (a < tolerance) or (b < tolerance)
```

Similarly, suppose we have to repeat a certain computation only if both **a** and **b** are less than some specified name **tolerance**. In this case, the condition is programmed as

```
1 (a < tolerance) and (b < tolerance)
```

We may assemble any type of condition from any of the operators above (`==`, `!=`, `<`, `>`, `<=`, `>=`, `and`, `or`). We can also chain comparisons together, for example:

```
1     (10 <= a < b)
```

In this example Python works its way from left to right by first evaluating `10 <= a` and, as long as the answer is `True`, then moves to the next chain (`a < b`).



#### More Info:

Type `help('==')` in the *IPython Console* to see documentation on comparisons.

### 5.3 Simulating a for loop statement using a while loop statement

Now that we know how to write conditional statements, we can mimic the `for` loop statement using the `while` loop statement. Let us do the addition:

$$\sum_{n=1}^{100} n = 1 + 2 + 3 + \dots + 100, \quad (5.1)$$

using the `while` loop statement. Before we proceed we need to recognise that when we performed addition using the `for` loop statement, we made use of a counter (or index). However, the `while` loop statement does not have a counter. We therefore need to program our own counter. The following program mimics the addition of the `for` loop statement using the `while` loop statement:

```
1  cnt = 1
2  mysum = 0
3  while (cnt <= 100):
4      mysum += cnt
5      cnt += 1
6  print "mysum = ", mysum
7  print "cnt = ", cnt
```

We programmed our counter `cnt` by initialising it to 1 (which is the first term we want to add) and then by incrementing `cnt` with 1 so that it generates the next term of our sequence. In addition, we needed to make sure that our condition in the `while` loop

statement is `True` by the time we reach it in our program. We achieved that when we initialised  $n = 1$ , since we want to continue adding while `cnt <= 100`. We added the last line in the code so that you can see that the programs stops when `cnt > 100` since our condition then becomes `False`.



#### Take Note:

If your program does not want to break out of a `while` loop statement (i.e. infinite loop because the `while` loop condition always remains `True`), you can press the `Ctrl` button and while holding it in press the `c` button, in the *IPython Console*, to stop the program.

## 5.4 Taylor series using the `while` loop statement

Now that we know how to write conditional statements, we can re-program the Taylor series used to compute  $\pi$  using a `while` loop statement as follows:

```
1 taylor_pi = 0.0
2 accuracy = 0.001
3 taylor_term = 1      #initialize greater than accuracy
4 cnt = 0              #term counter
5 while abs(taylor_term) > accuracy:
6     taylor_term = (4.0 * (-1)**cnt) / (2*cnt + 1)
7     taylor_pi += taylor_term
8     cnt += 1
9 print "pi = ", taylor_pi
10 print "cnt = ", cnt
```

This program produces the following output:

```
1 pi =  3.14209240368
2 cnt =  2001
```

which means that we need to sum the first 2001 terms in the Taylor series to compute the value of  $\pi$  within an accuracy of 0.001.

The most important feature of the above program is the use of the `while` loop statement. Even though you as the programmer might quite easily recognise that you need to

use a `while` loop statement, it is more important (and often more challenging) to find the correct condition which must be used to decide whether or not to execute the loop again. In this particular case we will repeat the `while` loop, which adds additional terms in the Taylor series, as long as the *magnitude* of these terms are *greater than* some specified small number (I've used a name `accuracy` in this program).



#### Take Note:

Also note that I check whether or not the *absolute* value of the next term is larger than the value assigned to the name `accuracy`. A large negative term will not satisfy the `while` loop condition and the loop will terminate prematurely, although the addition of such a large negative term still has a significant impact on the accuracy of the function value approximation.

In the above program, we initialise the name `taylor_pi` to zero as before. We also initialise the name `cnt` to zero and we assign a value of 0.001 to the name `accuracy`. Next, we define a value of 1 to the `taylor_term` name. This might not make any sense, but it is required to ensure that the `while` loop condition is `True` for the very first iteration. As long as we start off with a value on `taylor_term` that is greater than `accuracy`, the `while` loop condition is `True` and the `while` loop will be executed at least once.

Two separate outputs will be printed to the *IPython Console* after the program is executed: the function value approximation contained in the name `taylor_pi` and the total number of terms required in the Taylor series, which for this particular implementation is equal to `cnt`.

Just to illustrate that there exist many solutions to the same programming problem, consider the following alternative:

```
1  accuracy = 0.001
2  cnt = 0          #term counter
3  taylor_term = (4.0 * (-1)**cnt) / (2*cnt + 1)
4  taylor_pi = taylor_term
5  while abs(taylor_term) > accuracy:
6      cnt += 1
7      taylor_term = (4.0 * (-1)**cnt) / (2*cnt + 1)
8      taylor_pi += taylor_term
9  print "pi = ", taylor_pi
10 print "cnt = ", cnt+1
```

In this implementation, the first Taylor term is computed outside the `while` loop statement. The subsequent terms are computed inside the `while` loop statement. The

benefit of this solution is that we do not need to assign some artificial initial value to `taylor_term` just to make sure the `while` loop condition is `True` for the first iteration. Also note that the number of terms added is now given by `cnt+1` instead of `cnt`. This is because we now increment the value of `cnt` at the start of the loop instead of the end of the loop.

We can also use a different condition for the `while` loop statement. I've mentioned before that we can stop the loop if the function value approximation converges (instead of checking if the next Taylor term is small enough). So how do we check for convergence? Using a computer, all we can do is to check if the function value changes from one iteration to the next is within some specified tolerance (or accuracy, as before). The following program illustrates:

```

1  tolerance = 0.001
2  cnt = 0          #term counter
3  taylor_change = 1 #initialize greater than tolerance
4  taylor_pi = 0.0
5  while abs(taylor_change) > tolerance:
6      taylor_old = taylor_pi
7      taylor_term = (4.0 * (-1)**cnt) / (2*cnt + 1)
8      taylor_pi += taylor_term
9      taylor_change = taylor_old - taylor_pi
10     cnt += 1
11  print "pi = ", taylor_pi
12  print "cnt = ", cnt

```

In this program we check if the change in the Taylor series approximation, stored in `taylor_change`, is large enough. If so, we keep on adding terms to the series. Once the change in the function value approximation becomes smaller than the tolerance, the `while` condition is no longer `True` and the program stops adding terms. If you carefully analyse the logic above, you will realise it is equivalent to the previous program: the change in the Taylor series is exactly equal to the magnitude of the new term being added. So it is equivalent to check the magnitude of the new term, or to check the magnitude of the function value change.



#### More Info:

This is a very typical usage of a `while` loop statement. Frequently we are trying to solve a problem, but we do not know in advance what the solution is. So we generate a sequence of trial solutions to the problem. Usually, trial solutions improve i.e. every new computed trial solution is superior to the previous trial solutions. If the trial solutions produce a series of approximations that tend towards a specific

fixed value, we say that the method (or solution) converges. We test for convergence by simply checking if the difference between consecutive solutions are smaller than some small specified value. I refer to this small value as the *accuracy*, or *tolerance* of the computation. This process of repeatedly computing a trial solution to a problem until some condition is met, is called *iteration*.

## 5.5 Exercises

1. Write a program that first generates a random integer  $N$ . The program must then add the numbers  $1, 2, 3, \dots$  together as long as the sum is less than the random integer  $N$ . The program must then display the sum as well as the last added value in an appropriate message to the screen.
2. Write a program that generates random numbers between 1 and 10. The program must add these random numbers together until either 21 is exceeded or five numbers where added. The program must then display the sum to the screen.
3. Write a program that simulates a single dice being thrown. The program must continue to simulate the throw of a dice until an even number is thrown. The program must then display the number of throws required to throw an even number.
4. The geometric series for 2 is given by

$$\sum_{n=0}^{\infty} 2^{-n} = 2 \quad (5.2)$$

The geometric series only equals 2 when an infinite number of terms have been summed together. Every term that you add to the geometric series gets the approximation closer to 2.

Write a program that approximates 2 using the given geometric series. Select a required accuracy to which 2 must be approximated by the geometric series. Compute the absolute error to determine whether the required accuracy of the approximation is obtained i.e. take the absolute value of the difference between the series approximation and 2 ( $|approximation - 2|$ ).

The program must stop when the absolute error is smaller than the required accuracy. If the absolute error is still bigger than required accuracy the next term in the geometric series must be added. The program must continue to add terms until the desired accuracy is obtained.

5. In the exercises for Chapter 4 you had to write a program that computed the first 20 terms of the Fibonacci series. Recall that the Fibonacci series is given by the formula

$$L_k = L_{k-2} + L_{k-1} \quad \text{for } k = 2, 3, \dots \quad (5.3)$$

with  $L_0 = 0$  and  $L_1 = 1$ . For this exercise you have to compute the limit of the ratio of two consecutive terms in the series. I.e. compute:

$$p = \lim_{k \rightarrow \infty} \frac{L_k}{L_{k-1}} \quad (5.4)$$

Of course we cannot compute  $p$  using the equation above, because we cannot generate an infinite number of terms. However, we can generate the series:

$$\frac{L_2}{L_1}, \frac{L_3}{L_2}, \frac{L_4}{L_3}, \dots \quad (5.5)$$

As we generate a new term in the Fibonacci sequence. The series above is supposed to converge to the value of  $p$ . Write a program to compute  $p$ . Also display the number of terms required in the series as part of your output.

Assume that you do not know the value to which the sequence converges, therefore make use of a “numerical error” as discussed in class i.e. use the absolute difference of two consecutive terms to compute an approximation to the error.

6. Redo Question 4 but instead of the absolute error use a numerical error i.e. compute the absolute difference of two consecutive terms of the geometric series i.e.  $n$  and  $n - 1$ . Look at the hint below:

$$|2^{-(n)} - 2^{-(n-1)}| \quad (5.6)$$

[Hint: The program must stop when the numerical error is smaller than the required accuracy. If the numerical error is still bigger than required accuracy the next term in the geometric series must be added. The program must continue to add terms until the desired accuracy is obtained.]

7. To what value does

$$\frac{1}{1} + \frac{1}{2} - \frac{1}{4} + \frac{1}{8} - \frac{1}{16} + \frac{1}{32} - \dots$$

converge? Write a program that computes (approximates) the sequence within a specified tolerance (i.e. when the absolute difference of two consecutively computed sequence approximations are less than the specified tolerance). The program must then display the converged value to the screen as well as the number of terms required.

8. What is the limit value of

$$\lim_{x \rightarrow \infty} f(x) = \lim_{x \rightarrow \infty} \frac{2x}{1+x}$$

Write a program that computes (approximates) the limit value within a user specified tolerance (i.e. when the absolute difference of two consecutively computed sequence approximations are less than the specified tolerance). The program must then display the limit value to the screen.

9. In the exercises for Chapter 4 you had to compute the first 10 terms of the Taylor series expansion for  $\sin(x)$ .

For this exercise I would like you now to write a program that approximates the `cos` function with the Taylor series expansion (given below) to within a required accuracy. Select a required accuracy as well as the  $x$  value at which you would like to approximate  $\cos(x)$ . The program then has to keep on adding terms to the Taylor series approximation until the required accuracy from the `math` module's  $\sin(x)$  function.

$$\cos(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!} x^{2n}$$



# Chapter 6

## Branching

So far, our programs have simply been executed one line at a time, from top to bottom. *All* commands are executed. In cases where we used loop statements (`for` or `while`), *all* the statements within the looping commands are repeated from top to bottom a certain number of times. These type of programs are not as flexible as you might think. Certain scenarios might exist where different strategies are necessary to perform different tasks. In such cases, we require a programming structure that allows us to perform different tasks depending on the situation. In this section, I'll illustrate the use of branching statements i.e. statements that allow the program to decide between different courses of action. No longer will all programming statements be executed, but only those that are required.

The `if` statement is based on conditions (the same boolean conditions or questions we considered for the `while` loop statement), see Section 5.2. Again the conditions can be either `True` or `False`.

### 6.1 The if statement

The most basic `if` structure is given below:

```
1   if condition_a:  
2       statements_a
```

The `if` statement checks whether `condition_a` is `True`, if it is `True` then `statements_a` are executed, otherwise `statements_a` are ignored. The program then continues with the code below the `if` statement (from line 3 onwards). Therefore this basic structure either

executes `statements_a` or ignores code after which the program continues with the code after the `if` statement (from line 3 onwards).

In some cases it is desired to choose between multiple pieces of code and execute *only one* or *none* of them. This is achieved by the more general `if-elif` structure given below:

```
1  if condition_a:  
2      statements_a  
3  elif condition_b:  
4      statements_b
```

The `if-elif` structure starts with `condition_a`. If `condition_a` is `True` then `statements_a` are executed and then the program continues with the code below the `if` statement (from line 5 onwards), therefore ignoring `condition_b` and `statements_b` whether it is `True` or `False`. Only when `condition_a` is `False` does the `if-elif` structure proceed to `condition_b`. If `condition_b` is `True` then `statements_b` is executed otherwise `statements_b` is ignored and the program continues with the code below the `if` statement (from line 5 onwards). The `if-elif` structure can have as many `elif` (short for “else if”) parts as you like e.g.

```
1  if condition_a:  
2      statements_a  
3  elif condition_b:  
4      statements_b  
5  elif condition_c:  
6      statements_c  
7  elif condition_d:  
8      statements_d
```

The same rules however apply i.e. the program only proceeds to the next `condition` if all the proceeding conditions were `False`. The only code that is executed is the code between first `condition` from the top that is `True` and the next `condition`. The program then proceeds with the code below the `if` statement (from line 9 onwards). Therefore at most **one** of statements a, b, c or d is executed but nothing is executed when all the conditions are `False`.

The `if-elif` structure is therefore not appropriate when you always want to execute `statements_a`, `statements_b`, `statements_c` and `statements_d` when their respective conditions are `True`. For that you have to use the `if` structure for each condition e.g.

```
1  if condition_a:  
2      statements_a  
3  if condition_b:  
4      statements_b  
5  if condition_c:  
6      statements_c  
7  if condition_d:  
8      statements_d
```

Here the **statements** of every **True** condition are executed. Considering the **if** structures so far either some code is executed or no code is executed depending on whether the conditions are **True** and also on which conditions are **True**.

In certain cases you might always want some code to be executed. In particular, you might want to execute some code only when all the preceding **conditions** are **False**. The **if-elif-else** structure allows for that and is given below:

```
1  if condition_a:  
2      statements_a  
3  elif condition_b:  
4      statements_b  
5  else:  
6      statements_c
```

The **if** and **elif** structures works exactly as before. Therefore if **condition\_a** is **True** then only **statements\_a** is executed. Only when **condition\_a** is **False**, **condition\_b** is considered. If **condition\_b** is **True** then only **statements\_b** is executed. Only when **condition\_b** is also **False** then the code after the **else** is executed. The program then continues with the code below the **if-elif-else** statement (from line 7 onwards).



#### Take Note:

Note that the **else** in this structure does not have a condition since the code after the **else** is only executed when *all* the preceding **conditions** are **False**.



#### Take Note:

Note the double colon (**:**) at the end of the **if**, **elif** and **else** statements (in line 1, line 3 and line 5 respectively). This is the same as the **for** loop statement and tells Python where the respective **program statements** inside the **if**, **elif** and **else**

statement start.

If the double colon (:) is left out you will see an error message.



#### Take Note:

The rules for indentation for the `if-elif-else` statements are the same as for the `for` loop, see Section 4.2.5.

Python requires 4 spaces before **all program statements** inside the `if-elif-else` statements.



#### Take Note:

A `else` statement is always at the end, i.e. it cannot come before either the `if` or the `elif` statements.

Like wise a `elif` statement cannot come before the `if` statements.



#### More Info:

Additional help is available by typing `help('if')` which will produce the following output:

```
 1 The ``if`` statement
 2 ****
 3
 4 The ``if`` statement is used for conditional execution:
 5
 6 if_stmt ::= "if" expression ":" suite
 7         ( "elif" expression ":" suite )*
 8         ["else" ":" suite]
 9
10 It selects exactly one of the suites by evaluating the
11 expressions one by one until one is found to be true
12 (see section *Boolean operations* for the definition of
13 true and false); then that suite is executed (and no
14 other part of the ``if`` statement is executed or
15 evaluated). If all expressions are false, the suite of
16 the ``else`` clause, if present, is executed.
17 (END)
```

## 6.2 Simple example

Lets consider the flowchart problem given in Chapter 1 (Section 1.3.1), where we have to compute a symbol based on a given percentage:

```
1 percentage = 40
2 if percentage >= 80:
3     symbol = "A"
4 elif percentage >= 70:
5     symbol = "B"
6 elif percentage >= 60:
7     symbol = "C"
8 elif percentage >= 50:
9     symbol = "D"
10 else:
11     symbol = "F"
12 print "symbol = ", symbol
```

You can see from this example, as Python evaluates the first `if` statement (line 2) and then the `elif` statements (lines 4, 6 and 8) the conditions evaluated are `False`. Python then finally gets to the `else` statement and the code on line 11 is executed. Python then exits the `if-elif-else` block and executes the code on line 12.

If we changed the percentage value to 70, Python would evaluate the first `if` statement (line 2) and the condition evaluated would be `False`. Then Python would evaluate the first `elif` statement (line 4) and the condition evaluated would be `True` and Python would then execute the code on line 5. Python would then exit the `if-elif-else` block and execute the code on line 12.

Would this example work correctly if we changed the order, in which we evaluate the percentage symbol, from small to large (as shown in the example below) ??

```
1 percentage = 70
2 if percentage < 50:
3     symbol = "F"
4 elif percentage >= 50:
5     symbol = "D"
6 elif percentage >= 60:
7     symbol = "C"
8 elif percentage >= 70:
```

```

9     symbol = "B"
10    else:
11        symbol = "A"
12    print "symbol = ", symbol

```



### Take Note:

The answer is in fact "No", this example would not work correctly and if you answered differently, please go back and review Section 6.1 again. Python executes the code in the first `if` or `elif` statement where the condition evaluated was `True` and skips the others.

## 6.3 Leg before wicket example

I'll use the leg before wicket (LBW) rule of the game cricket to illustrate the use of the `if` statement again. The LBW rule is as follows:

1. A batsmen can only be given out (LBW) when the batsmen's bat did not make contact with the ball and the ball hit the batsmen's leg / pad. The batsmen is then out if:
  - (a) The ball would have continued on to hit the stumps.
  - (b) The ball did not pitch on the leg side.
  - (c) If the batsmen offered a shot, the ball must hit the pad in line with the stumps.
  - (d) If the batsmen did not offer a shot, the ball may hit the pad outside the line of offstump.

We now proceed to build a flow chart from the above problem statement. The flow chart for the leg before wicket problem is depicted in Figure 6.1. You can see how the flow chart splits between various sections of the logic depending on whether the conditions are `True` (*yes*) or `False` (*no*).

We are now ready to implement the logic into a program. Here follows the program that uses the appropriate set of questions to determine whether or not a batsmen should be given out LBW:

```

1 Q1 = raw_input("Would the ball have hit the wicket? [y or n]: ")
2 if Q1 == "y":      # hit wicket?

```

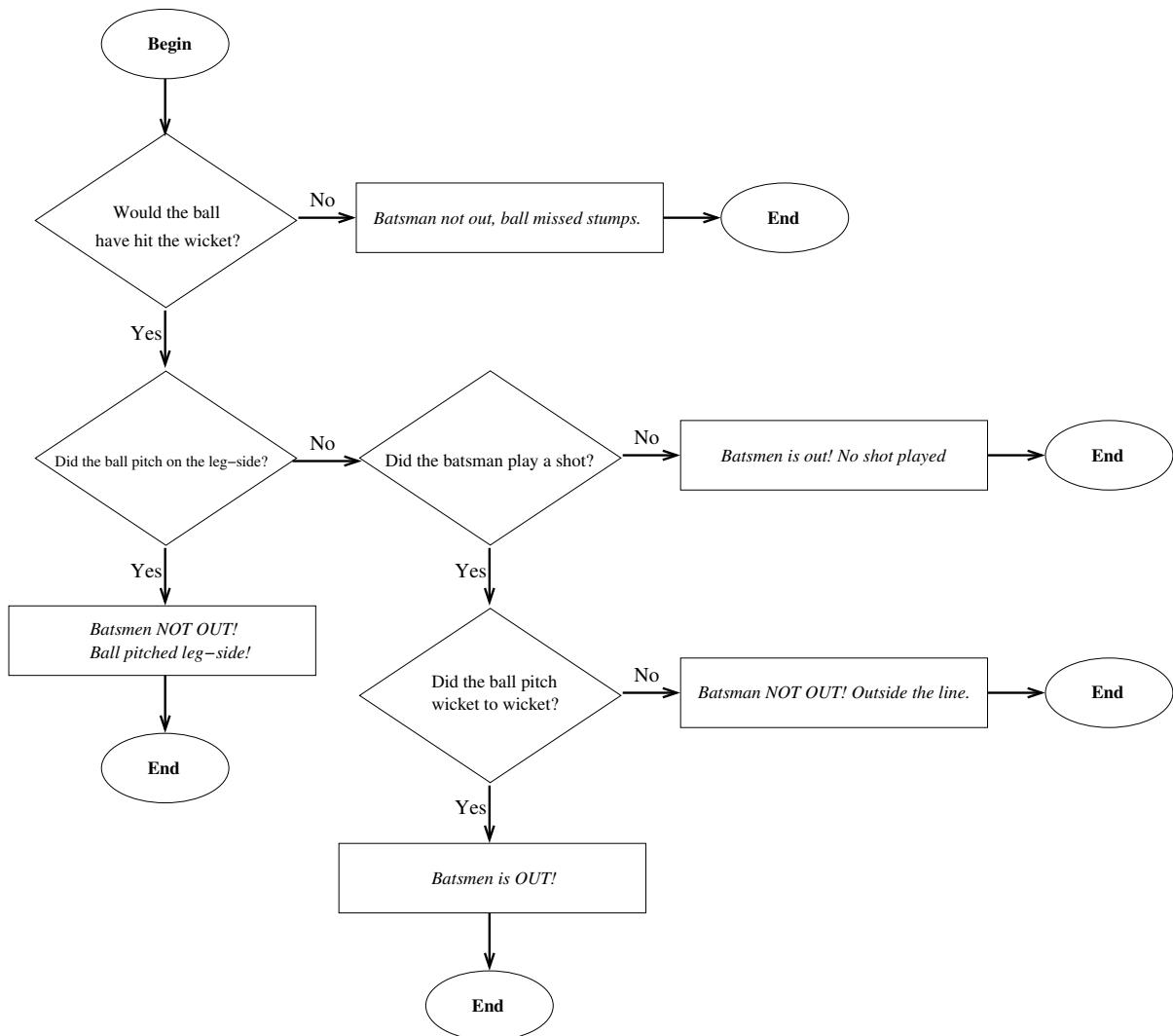


Figure 6.1: The leg before wicket flowchart logic.

```

3   Q2 = raw_input("Ball pitched on the leg-side? [y or n]: ")
4
5   if Q2 == "n":      # leg side?
6       Q3 = raw_input("Batsmen played a shot? [y or n]: ")
7
8       if Q3 == "n":      # play shot?
9           print "Batsmen is OUT! No shot played."
10      else:
11          Q4 = raw_input("Pitched wicket to wicket? [y or n]: ")
12
13         if Q4 == "y":      # in line?
14             print "Batsmen is OUT!"
15         else:
  
```

```
16         print "Batsmen NOT OUT! Outside the line."  
17  
18     else:  
19         print "Batsmen NOT OUT! Ball pitched leg-side."  
20  
21 else:  
22     print "Batsmen NOT OUT! Ball missed stumps."
```



### More Info:

The `raw_input` statement is used to get information from the user. The syntax for the `raw_input` statement is:

```
1 var1 = raw_input("Message appearing in IPython Console.")
```

The message between the quotation marks will be printed to the *IPython Console*, and Python will wait for the user to input something.

The user's input will then be assigned to name `var1` and Python will continue.

The value assigned to `var1` will always be a string !!!

This is discussed in more detail in Section 7.1.2.



### Take Note:

You should by now be familiar with why this example is indented the way it is: each nested (inner) `if-else` statement has 4 more spaces than the (outer) `if-else` statement before it.

Make sure you understand the behaviour and output of the following two examples and why they are different !!

```
1 a = 9  
2 b = 60  
3 if a == 10:  
4     b = 100  
5     if b > 50:  
6         b *= 2  
7 else:  
8     a = 100  
9 print "a = ", a  
10 print "b = ", b
```

```
1 a = 9
2 b = 60
3 if a == 10:
4     b = 100
5 if b > 50:
6     b *= 2
7 else:
8     a = 100
9 print "a = ", a
10 print "b = ", b
```

## 6.4 Number guessing game

In this section you won't learn anything new. I'll just use what you know already and write a simple number guessing program. The computer will guess an integer between 0 and 100. The user (player) then gets a chance to guess the number. The program must give feedback that the number is either too small, too large or correct. The game continues until the correct number is guessed. Once the user has guessed the right number, the program must tell the user how many guesses he or she took to guess the correct number.

By now, you should be able to tell that a `while` loop statement is required. The `while` loop statement will terminate as soon as the correct number is guessed. Inside the `while` loop, an `if` statement will check whether the number is too small, too large or correct.

How can the computer guess a number between 0 and 100? We'll make use of the `random.randint(start, end)` statement, which generates random integer numbers between `start` and `end` (inclusive of `start` and `end`).

Here's my version of this program:

```
1 import random
2
3 # Number guessing game
4 print "Welcome to the number guessing game."
5 print "I've guessed a number between 0 and 100."
6
7 # Computer guesses number between 0 and 100
8 number = random.randint(0, 100)
9
10 # User guesses a number
```

```
11 guess = int(raw_input("Enter a guess: "))

12 # First guess counter
13 count = 1

14

15 # while loop continues as long as Guess not equal to Number
16 while guess != number:
17     if guess < number:
18         print "You guessed too small."
19     else:
20         print "You guessed too large."
21     guess = int(raw_input("Try again. "))
22     count += 1 # Add 1 to the counter
23 # while loop terminated, so guess must be correct
24 print "Congratulations, you guessed right!"
25 print "You took " + str(count) + " guesses."
```



#### More Info:

Type `import random` and then `help(random)` in the *Python Console* to get more information on the available functions in the `random` module.



#### More Info:

Remember that the `raw_input` statement returns the users input as a string, so we use the `int()` statement to convert the string input into an integer.

The `int()` function converts either a float or a string representation of an integer to an integer:

```
1 In [#]: int(2.5)
2 Out[#]: 2

3
4 In [#]: int("5")
5 Out[#]: 5

6
7 In [#]:
```

String representations of float (e.g. "5.4") can not be converted to integers !!  
Type `help('int')` in the *IPython Console* for more information.

**More Info:**

The `str()` function converts either a float or an integer to a string:

```
1 In [#]: str(10)
2 Out[#]: '10'
3
4 In [#]: str(12.5)
5 Out[#]: '12.5'
6
7 In [#]:
```

Type `help('str')` in the *IPython Console* for more information.

**More Info:**

The plus sign (+) between 2 (or more) strings will join them together:

```
1 In [#]: "Logan " + "Page"
2 Out[#]: 'Logan Page'
3
4 In [#]:
```

More on this in Section [7.1.3](#).

Here is the output in the *IPython Console* for the first game I played:

```
1 Welcome to the number guessing game.
2 I've guessed a number between 0 and 100.
3 Enter a guess: 10
4 You guessed too small.
5 Try again: 50
6 You guessed too small.
7 Try again: 70
8 You guessed too large.
9 Try again: 60
10 You guessed too small.
11 Try again: 65
12 You guessed too large.
13 Try again: 62
```

```
14 You guessed too small.  
15 Try again: 63  
16 You guessed too small.  
17 Try again: 64  
18 Congratulations, you guessed right!  
19 You took 8 guesses.
```

What can we do to make the game a little more challenging? Let's extend the game to have three levels of play: easy for a number between 0 and 10, intermediate for a number between 0 and 100, and difficult for a number between 0 and 1000. All that we have to do to implement this change is to ask the user which level he or she wants to play and then create a random number between the appropriate bounds. Here's one possible version of the improved game:

```
1 import random  
2  
3 # Improved number guessing game  
4 print "Welcome to the number guessing game."  
5 print  
6 print "Choose a difficulty:"  
7 print "    1: easy [number between 0 and 10]"  
8 print "    2: medium [number between 0 and 100]"  
9 print "    3: hard [number between 0 and 1000]"  
10 level = int(raw_input("Difficulty Level: "))  
11  
12 if level == 1:  
13     # Computer guesses number between 0 and 10  
14     print "I've guessed a number between 0 and 10."  
15     number = random.randint(0, 10)  
16 elif level == 2:  
17     # Computer guesses number between 0 and 100  
18     print "I've guessed a number between 0 and 100."  
19     number = random.randint(0, 100)  
20 elif level == 3:  
21     # Computer guesses number between 0 and 1000  
22     print "I've guessed a number between 0 and 1000."  
23     number = random.randint(0, 1000)  
24  
25 # User guesses a number  
26 guess = int(raw_input("Enter a guess: "))  
27  
28 # First guess counter
```

```
29 count = 1
30
31 # while loop continues as long as Guess not equal to Number
32 while guess != number:
33     if guess < number:
34         print "You guessed too small."
35     else:
36         print "You guessed too large."
37     guess = int(raw_input("Try again: "))
38     # Add 1 to the counter
39     count += 1
40 # while loop terminated, so guess must be correct
41 print "Congratulations, you guessed right!"
42 print "You took " + str(count) +" guesses."
```

Here's the output of one of the hard games I played.

```
1 Choose a difficulty:
2     1: easy [number between 0 and 10]
3     2: medium [number between 0 and 100]
4     3: hard [number between 0 and 1000]
5 Difficulty Level: 3
6 I've guessed a number between 0 and 1000.
7 Enter a guess: 500
8 You guessed too small.
9 Try again: 750
10 You guessed too small.
11 Try again: 875
12 You guessed too large.
13 Try again. 800
14 You guessed too large.
15 Try again. 775
16 You guessed too small.
17 Try again: 790
18 You guessed too small.
19 Try again: 795
20 You guessed too large.
21 Try again. 794
22 Congratulations, you guessed right!
23 You took 8 guesses.
```

In program lines 12–20, I make use of an `if` statement. The `if` statement checks the

value of the `level` name and depending on the value, the bounds of the random number is varied. In this program, the name `level` can have only three possible values, i.e. 1, 2 or 3.

## 6.5 The if statement with combined conditions

We have already used the `if` statement inside another `if` statement. The rules as explained holds for `if-elif-else` structure. As you have seen it is important to see each structure as a unit and that the rules apply for each unit.

Remember we can combine two `conditions` using the `and` or the `or` keywords (Section 5.2). Similarly for the `if` statements we can combine two `conditions` e.g:

```
1  if (condition1) and (condition2):  
2      statements
```

When we use `if` statements, we can also write the program as follows:

```
1  if (condition1):  
2      if (condition2):  
3          statements
```

Both programs do exactly the same thing, it simply depends on the programmer how to program and implement it. Again, this illustrates that programs are not unique. Programs may look different but when you follow the logic you might find that they solve the same problem.

We can also write the combination using the `or` keyword differently. The combined `conditions` for the `or` keyword is given by

```
1  if (condition1) or (condition2):  
2      statements
```

Alternatively we could have written

```
1  if (condition1):  
2      statements  
3  elif (condition2):  
4      statements
```

Some implementations have their advantages and some their disadvantages e.g. imagine **statements** being a long list of instructions and you need to make changes to **statements**, which implementation would you prefer? Our last implementation duplicates the instructions of **statements** and you therefore have to make double the changes. As you grow in your programming ability and start maintaining your own code, consideration of such issues will become more important. Here I would just like to bring to your attention alternative issues you might want to consider when you want to decide on which implementation you want to code to solve a problem.

## 6.6 Exercises

1. Write a program that asks the user to type a number. The program must then display one of the following:
  - **Positive number**, if number is larger than zero,
  - **Negative number**, if number is smaller than zero, and
  - **Neither positive nor negative it's zero**, if number is equal to zero.
2. The colors red, blue and yellow are primary colors. By mixing two primary colors you obtain a secondary color e.g.
  - (a) Blue and red gives purple
  - (b) Yellow and blue gives green
  - (c) Red and yellow gives orange

Write a program requires two primary colours as input, from the user. Any inputs other than red, blue or yellow should display an appropriate error message. For valid user inputs the program must display the appropriate secondary colour as output.
3. Write a program that takes a number of seconds as an input from the user.
  - (a) For a value of less than 60 the program should display the number of seconds in an appropriate message.
  - (b) There are 60 seconds in a minute, for a value of greater or equal to 60 the program should display the number of minutes in an appropriate message.

- (c) There are 3600 seconds in a hour, for a value of greater or equal to 3600 the program should display the number of hours in an appropriate message.
- (d) There are 86 400 seconds in a day, for a value of greater or equal to 86400 the program should display the number of days in an appropriate message.
- (e) There are 604 800 seconds in a week, for a value of greater or equal to 604 800 the program should display the number of weeks in an appropriate message.
4. Write a program that asks the user to input the mass (*kg*) of his/her car and then the speed (*km/h*) at which he/she drives. The program must then compute the kinetic energy of the car at that speed:

$$E_{kinetic} = \frac{1}{2}mv^2 \quad (6.1)$$

The program must then display the height at which the potential energy is equal to the kinetic energy as well as the kinetic energy of the vehicle.

$$E_{potential} = mgh \quad (6.2)$$

with  $g = 9.81 \text{ m/s}^2$ .

5. Write a program that asks the user to choose one of the two options
- a) when he/she wants to compute the average velocity or
  - b) when he/she wants to compute the average acceleration,
- of an object that is moving in a straight line.
- a) If the user types V or v for the `input` statement:- the program must ask the user to type a value for the coordinate  $x_1$  at time  $t_1$ . The program must then ask the user to type a value for the coordinate  $x_2$  at time  $t_2$ . The program must also ask the user to enter a value for  $t_1$  and for  $t_2$ . The program must then compute and display the average velocity:
- $$v_{avg} = \frac{\Delta x}{\Delta t} = \frac{x_2 - x_1}{t_2 - t_1} \quad (6.3)$$
- b) If the user types A or a for the `input` statement:- the program must ask the user to type a value for the velocity  $v_1$  at time  $t_1$ . The program must then ask the user to type a value for the velocity  $v_2$  at time  $t_2$ . The program must also ask the user to enter a value for  $t_1$  and for  $t_2$ . The program must then compute and display the average acceleration:
- $$a_{avg} = \frac{\Delta v}{\Delta t} = \frac{v_2 - v_1}{t_2 - t_1} \quad (6.4)$$

6. Write a program that asks the user to enter a positive integer. The program must then display one of the following

- **Even number**, if the entered number is even (i.e. divisible by 2).
- **Odd number**, if the entered number is odd (i.e. not divisible by 2).

*Hint: A number is divisible by another number only if the remainder is equal to 0 after division. You may make use of the remainder (%) operator.*

7. Write a program that asks the user to enter a positive integer. The program must then display one of the following

- **Prime number**, if the entered number is a prime number.
- **Not a prime number**, if the entered number not a prime number.

*Hint: A prime number is a number that is only divisible by itself and 1. The number 1 is excluded and is not considered to be a prime number. You may make use of the remainder (%) operator.*

8. Write a program that asks the user two questions. Firstly, the program must ask the user to enter an *enumerator* which must be a positive integer. Secondly, the program must ask the user to enter the *denominator* which also must be a positive integer. The program must then compute the remainder when you divide the *enumerator* by the *denominator*.

You may **not** use the remainder (%) operator in your program but you can use the remainder (%) operator to test your program.

Your program must be able to handle *enumerator's* that are smaller than the *denominator*. The *denominator* is then the remainder.

I include the hint below which may be of assistance:

$$\frac{24}{7} = \frac{7}{7} + \frac{17}{7} = \frac{7}{7} + \frac{7}{7} + \frac{10}{7} = \frac{7}{7} + \frac{7}{7} + \frac{7}{7} + \frac{3}{7} \quad (6.5)$$

The remainder of the above example is 3.

9. Write a program that asks the user to enter the number of double dice throws he would like to see. For every double throw (two dices that are thrown simultaneously) the program must then generate two random integers `dice1` and `dice2` between 1 and 6. The sum of values of the two dices namely `dice1 + dice2` must then be calculated. The program must then keep count of how many times every possible sum i.e. between 2 and 12 was thrown. The program must then output in the following format:

`1 How many double dice throws would you like to make? 1000`  
`2 Two: 27 Three: 57 Four: 82 Five: 99 Six: 132 Seven: 168`  
`3 Eight: 135 Nine: 124 Ten: 89 Eleven: 57 Twelve: 30`

10. Write a program that generates two random integers between 0 and 100 and asks the user to enter the product of the two numbers. The program must continue to ask the user to guess the product of the two numbers as long as the provided answer is wrong. When the user supplies the correct answer the program must display the correct answer to the screen with an appropriate message.
11. Extend the previous program such that every time the user completes a multiplication question, the program must ask the user whether he/she would like another question. If the user enters 1 the program should continue to ask the user another multiplication question otherwise the program should stop.
12. “Dice run” is a game of chance and requires a person to throw two dices. A person continues to throw two dices until a 2 or 12 is thrown. The number of throws required to throw a 2 or 12 are then counted to give a score for the run. Write a program that computes and displays the average score over a user specified number of runs of dice run.

# Chapter 7

## Additional examples and statements

In this chapter we will cover a few additional statements, with short examples, commonly used in Python (Section 7.1). We will also look at how to manipulate strings and string objects for better output (feedback from the program). We will then apply what has been learnt thus far to solving a few additional mathematical and engineering problems (Section 7.2).

### 7.1 Additional statements

#### 7.1.1 lambda function

A very powerful Python command is the `lambda` statement. It allows us to create additional functions that we can use in Python. The general syntax for the `lambda` function is as follows:

```
function = lambda variable/s: function based on variable/s
```

This can be written more mathematically as:

```
function = lambda x, y: f(x, y)
```

**Take Note:**

The given variables must be separated by a comma and a space (', ')

Note the double colon (:) after the variables and before the function. This tells Python where the function, based on the variables, starts.

If the double colon (:) is left out you will see an error message.

From this structure you can see that you create an additional function by using the `lambda` statement, followed by the variables for the function, followed by a double colon (:), followed by the function based on the variables.

As an example, consider the statement:

```
1 quad = lambda x: x**2 - 4*x + 10
```

This statement defines a new function `quad` which is the quadratic equation  $x^2 - 4x + 10$ , which is only a function of one variable ( $x$ ). We can evaluate the new function `quad` at any desired value  $x$ . Here's an example of a program using the `lambda` statement:

```
1 In [#]: quad = lambda x: x**2 - 4*x + 10
2
3 In [#]: quad(2.0)
4 Out [#]: 6.0
5
6 In [#]: x2 = 3.0
7
8 In [#]: quad(x2)
9 Out [#]: 7.0
10
11 In [#]: x3 = 5.0
12
13 In [#]: f3 = quad(x2)
14
15 In [#]: print f3
16 15.0
17
18 In [#]:
```

We will use the `lambda` statement to expand the suite of built-in Python functions. Especially when we work with long, complicated functions which we need to evaluate often, the `lambda` statement is very useful.

As a second example consider the following function:

```
1 radius = lambda x, y: (x**2 + y**2)**0.5
```

This statement defines a new function `radius`, which is the radius of a circle ( $r^2 = x^2 + y^2$ ) and is a function of two variables ( $x$  and  $y$ ). We can evaluate the new function `radius` at any desired value of  $x$  and  $y$ . Here's an example of a program using the `lambda` statement:

```
1 In [#]: radius = lambda x, y: (x**2 + y**2) ** 0.5
2
3 In [#]: radius(0, 1)
4 Out[#]: 1.0
5
6 In [#]: radius(0.5, 0.5)
7 Out[#]: 0.7071067811865476
8
9 In [#]:
```

### 7.1.2 `raw_input` statement

The use of the `raw_input` statement was already introduced in Chapter 6 (Section 6.3). I will, however, formally cover the use of the `raw_input` statement in this section. The `raw_input` statement is used whenever our program requires input from the user. It is unlikely that every person that uses our program wants to execute it using the same settings. Consider again the Taylor Series approximation for  $\pi$  (Section 5.4):

```
1 taylor_pi = 0.0
2 # The required accuracy typed in by the programmer
3 accuracy = 0.001
4 taylor_term = 1      #initialize greater than accuracy
5 k = 0                #term counter
6 while abs(taylor_term) > accuracy:
7     taylor_term = (4.0 * (-1)**k) / (2*k + 1)
```

```
8     taylor_pi += taylor_term
9     k += 1
10    print "pi = ", taylor_pi
11    print "k = ", k
```

For example, the name `accuracy` in the program above is set to 0.001. But, some users might require a more accurate value for  $\pi$  and would like to set the name `accuracy` to 0.0001. The simplest (and also least elegant) solution is to physically change the value of `accuracy` by deleting and editing it. Then save the file and execute the program again. Should you do that, the output of the program becomes:

```
1 pi = 3.14164265109
2 k = 20001
```

A more elegant solution is to develop the program such that certain settings are not entered as fixed values, rather they are obtained from the user. The `raw_input` statement is used to do exactly this. You still type the name followed by the equal sign, but instead of some number, we use the `raw_input` statement:

```
1 name = raw_input("Message appearing in IPython Console.")
```

Whatever you type between the double quotation marks appear exactly as is in the *IPython Console*. This message is supposed to tell the user what he or she is supposed to enter. A cursor appears immediately after the message. The user types some input and presses enter. The entered value is assigned to the name at the left of the equal sign. If you omit the descriptive message, the program will still work, but the user will have no idea what to enter in the *IPython Console*. The user will be confronted by a cursor in the *IPython Console* with no instruction what to do. In the Taylor series example above, I think it makes sense to rather get the required accuracy from the user. We do this by changing the lines

```
2 # The required accuracy typed in by the programmer
3 accuracy = 0.001
```

with the lines

```
2 # Get the required accuracy from the user
3 accuracy = raw_input("Input an accuracy to calculate pi: ")
```

However if we make these change and run the program we get the following output:

```
1 Input an accuracy to calculate pi: 0.0001
2 pi = 0.0
3 k = 0
```

So what is happening? Why did the program not compute  $\pi$ ? Why did the `while` loop statement not execute (lines 6 – 9)?

This is because the `raw_input` statement returns the users input as a string. So the name `accuracy` contains the string "0.0001". For interest sake lets take a look at what the condition for `while` loop statement (line 6) returns if `accuracy` is a string:

```
1 In [#]: taylor_term = 1
2
3 In [#]: accuracy = "0.0001"
4
5 In [#]: abs(taylor_term) > accuracy
6 Out [#]: False
7
8 In [#]:
```

From this little example it is clear why the `while` loop statement did not execute. We first need to convert the string to a floating point number (real number). We do this by using the `float()` statement. The final version of the Taylor series approximation for  $\pi$  is given as:

```
1 taylor_pi = 0.0
2 # Get the required accuracy from the user
3 accuracy = float(raw_input("Input an accuracy to calculate pi: "))
4 taylor_term = 1      #initialize greater than accuracy
5 k = 0                #term counter
6 while abs(taylor_term) > accuracy:
7     taylor_term = (4.0 * (-1)**k) / (2*k + 1)
```

```
8     taylor_pi += taylor_term
9     k += 1
10    print "pi = ", taylor_pi
11    print "k = ", k
```



### Take Note:

The `raw_input` statement ALWAYS returns the users input as a string. If you want a numerical value input from the user you have to convert the string input, from the user, to either an integer (`int()`) or floating point number (`float()`).



### More Info:

The `float()` function converts either an integer or a string representation of an integer or float to an float:

```
1 In [#]: float("2.5")
2 Out[#]: 2.5
3
4 In [#]: float(5)
5 Out[#]: 5.0
6
7 In [#]:
```

Type `help('float')` in the *IPython Console* for more information.

## 7.1.3 String manipulation

String manipulation is useful for either displaying results and information to the screen (feedback from the program) or for writing results and information to a file (discussed in Chapter 12). So far we have seen one example of string manipulation: the plus sign (+), which adds two strings together (Section 6.4). Let us look at a few more:

### 7.1.3.1 The string plus (+) operator

For completeness of this section I will include the string plus (+) operator again. The plus sign (+) between two or more strings will join them together:

```
1 In [#]: "The quick brown " + "fox jumps " + "over the lazy ..."  
2 Out[#]: The quick brown fox jumps over the lazy ...'  
3  
4 In [#]:
```

You can see from this example that you can also chain multiple strings together with the plus sign (+).



#### Take Note:

The string plus operator (+) can only be used to join strings. The following will thus give an error in Python:

```
"The meaning to life is " + 42
```

#### 7.1.3.2 The string multiply (\*) operator

The string multiply operator (\*) is used by multiplying a string (or string object) with an integer and this will repeat the string by the value of the integer specified. Consider the following examples:

```
1 In [#]: a = "Some text. "  
2  
3 In [#]: a * 5  
4 Out[#]: 'Some text. Some text. Some text. Some text. Some text. '  
5  
6 In [#]: a = a + "\n"  
7  
8 In [#]: a *= 4  
9  
10 In [#]: a  
11 Out[#]: 'Some text. \nSome text. \nSome text. \nSome text. \n'  
12  
13 In [#]: print a  
14 Some text.  
15 Some text.  
16 Some text.  
17 Some text.  
18  
19 In [#]:
```

You can see from this example that multiplying a string with an integer will repeat that string to the value of the integer. You should also notice that the assignment operators ( $+=$ ) and ( $*=$ ) work in the same way.



#### Take Note:

The string multiply operator (\*) can only be used to multiply a string with an integer. The following will thus give an error in Python:

```
"Some Text. " * 2.5
```



#### More Info:

The ‘‘\n’’ character represents a newline. Whenever you `print` the newline (‘‘\n’’) character a new line is started as shown in the example above.

The string plus operator (+) and multiply operator (\*) can be combined in anyway you can imagine, as long as they adhere to the rules give in each section. E.g. ("The " + "dog. ") \* 3

#### 7.1.3.3 The string formatting operator (%)

This operator makes Python one of easiest programming languages to use when it come to string manipulation. This operator is used to insert numerical values and/or strings into an existing string template. Lets again consider the Taylor Series approximation for  $\pi$ :

```
1 taylor_pi = 0.0
2 accuracy = 0.0001
3 taylor_term = 1      #initialize greater than accuracy
4 k = 0                #term counter
5 while abs(taylor_term) > accuracy:
6     taylor_term = (4.0 * (-1)**k) / (2*k + 1)
7     taylor_pi += taylor_term
8     k += 1
9 print "pi = ", taylor_pi
10 print "k = ", k
```

If we wanted to change the output to the following:  
The Taylor Series approximation for pi is 3.14164265109

which has an accuracy of 0.0001

we could change lines 9 – 10 and use the string plus operator (+) to chain the information together, as follows:

```
9 print "The Taylor Series approximation for pi is " + str(pi)
10 print "Which has an accuracy of " + str(accuracy)
```

Or we could use the string format operator (%):

```
9 print "The Taylor Series approximation for pi is %f" % pi
10 print "Which has an accuracy of %f" % accuracy
```

The (%f) symbols in a string template are seen, by Python, as “place holders” for information that we want to insert into the string template at these locations. After the string template we use the (%) symbol to tell Python what information to add in the locations of these “place holders”.

In this example above we add a (%f) symbol (place holder) in the string template to tell Python that we will be inserting a floating point number into this location of the string template. After the string template we use (%) symbol to tell Python that the object after this symbol is what needs to be inserted into the string template (in the place holder location). So pi is inserted into the string template on line 9 and accuracy is inserted into the string template on line 10.

We can further change the formatting of the data by specifying if Python should display an integer, float, scientific number or an additional string in the string template (i.e. we can change the type of the place holder in the string template):

- (%d) - insert an integer number
- (%f) - insert a floating point number
- (%e) - insert a scientific number
- (%s) - insert another string

Consider the following examples:

```
1 In [#]: "%d is a small %s." % (12.44213, "integer")
2 Out[#]: '12 is a small integer.'
3
4 In [#]: "%f is a small %s." % (12.44213, "float")
5 Out[#]: '12.442130 is a small float.'
6
7 In [#]: "%e is a small %s." % (12.44213, "scientific number")
8 Out[#]: '1.244213e+01 is a small scientific number.'
9
10 In [#]:
```

From this example you can see that we have added two “place holders” (or insert points) in our string template and we tell Python to insert 12.44213 and a string into our template. And as you would expect, in line 1 and 2, the integer formatting for the “place holder” only inserts the integer part of the number (**12.44213**) into the string template and discards the rest.



#### Take Note:

When inserting multiple pieces of information into a string template with multiple “place holders” you have to enclose the information in round brackets and separate the pieces of information with a comma and a space (, ), as shown in the above example.

The last formatting example I’m going to show is how to adjust the number of decimal places that are displayed for either a floating point number or a scientific number. In order to do this all you need to do is add a dot and the number of decimal places you want to display (.5) in front of the letter (**f**, **e**). Consider the following examples:

```
1 In [#]: "%.3f is a small %s." % (12.44213, "float")
2 Out[#]: '12.442 is a small float.'
3
4 In [#]: "%.8f is a small %s." % (12.44213, "float")
5 Out[#]: '12.44213000 is a small float.'
6
7 In [#]: "%.3e is a small %s." % (12.44213, "scientific number")
8 Out[#]: '1.244e+01 is a small scientific number.'
9
10 In [#]: "%.8e is a small %s." % (12.44213, "scientific number")
11 Out[#]: '1.24421300e+01 is a small scientific number.'
```

```
12 | In [#]:  
13 |
```

#### 7.1.3.4 String functions

So far I have told you that only modules contain functions, that we can use in Python. Well, this is not entirely true. String objects also have many functions associated to them. Lets look at a few of these functions (with `my_str = "hello world!"`):

1. Convert the string to upper case characters:

```
1 In [#]: my_str = my_str.upper()  
2  
3 In [#]: print my_str  
4 HELLO WORLD!  
5  
6 In [#]:
```

Similarly to modules, the dot (.) tells Python that the `upper` function resides in the string object.

2. Convert the string to lower case characters:

```
1 In [#]: my_str = my_str.lower()  
2  
3 In [#]: print my_str  
4 hello world!  
5  
6 In [#]:
```

3. Capitalise the first character of the string

```
1 In [#]: my_str = my_str.capitalize()  
2  
3 In [#]: print my_str  
4 Hello world!  
5  
6 In [#]:
```

4. Split the string at any white space character/s

```
1 In [#]: my_str = my_str.split()  
2  
3 In [#]: print my_str  
4 ['Hello', 'world!']  
5  
6 In [#]:
```



#### More Info:

Type `help(str)` in the *IPython Console* to see a list of available functions associated to the string object (scroll past any function with “\_” in front or behind the name, these functions are outside the scope of this course)

Alternatively you can create a string object e.g. `a="name"` and then type `a.` (note the dot) and hit the `tab` key, IPython will then print a list of functions associated to the string object to the screen.

## 7.2 Additional examples

In this section we will look at a few additional examples. We will not learn anything new, we will simply apply what we have learnt thus far to solving a few additional mathematical and engineering problems. It is important that, in this section, you understand how the logic of solving the problem is converted into a program.

### 7.2.1 Limit of the natural logarithm

The natural logarithm can be computed using

$$e = \lim_{p \rightarrow 0} (1 + p)^{1/p}. \quad (7.1)$$

As  $p$  approaches zero the above expression approaches the natural logarithm. We can start the program by computing  $e$  for a value  $p = 0.1$ . We can perform this computation inside a `for` loop statement, but we must ensure the value for  $p$  progressively approaches zero every time the loop is repeated. The following program will do precisely that:

```
1 p_val = 0.1  
2 for i in range(10):
```

```

3     exp = (1 + p_val) ** (1 / p_val)
4     p_val /= 10
5     print "e = %.8f      p = %.3e" % (exp, p_val)

```

Do you see how I make sure that the value `p_val` decreases every time the loop is executed? I divide `p_val` by 10 in line 4. So `p_val` starts off with a value of 0.1 outside the loop and at line 4, `p_val` is divided by 10 and re-assigned to `p_val` i.e. `p_val` now has the value 0.01. This happens every time line 4 is executed until the `for` loop is completed. So my program uses a `for` loop to simulate the limit as  $p$  approaches zero. The output of the above program is:

```

1 e = 2.59374246    p = 1.000e-02
2 e = 2.70481383    p = 1.000e-03
3 e = 2.71692393    p = 1.000e-04
4 e = 2.71814593    p = 1.000e-05
5 e = 2.71826824    p = 1.000e-06
6 e = 2.71828047    p = 1.000e-07
7 e = 2.71828169    p = 1.000e-08
8 e = 2.71828180    p = 1.000e-09
9 e = 2.71828205    p = 1.000e-10
10 e = 2.71828205   p = 1.000e-11

```

Note that in this program we did not explicitly make use of the value of the index `i`. In such a program, the index is merely used as a counter i.e. the statements, inside the `for` loop, are repeated the required number of times. However, the value of the index, `i` in this case, is available inside the loop and can be used if required. In fact, the index is just another name, which just happens to control the execution of the `for` loop. The next example illustrates how the index can be used inside the `for` loop:

```

1 for i in range(1, 11):
2     p_val = 10 ** (-i)
3     exp = (1 + p_val) ** (1 / p_val)
4     print "e = %.8f      p = %.3e" % (exp, p_val)

```

In this example, we did not need to define an initial value for `p_val` outside the `for` loop, since the value of `p_val` is computed as a function of the index `i`. This program will produce exactly the same output as the one before. Again note that there is not a unique solution to a particular problem. As long as the list of instructions we give to the computer makes sense and is consistent with the required task, the program is acceptable.

You should be aware by now that if we wanted to solve for  $e$  only to a required accuracy we would need to replace the `for` loop statement with a `while` loop statement. The following program will allow the user to specify the accuracy to which  $e$  should be computed:

```

1 p_val = 0.1
2 exp = 0
3 error = 1
4 accuracy = float(raw_input("Input accuracy: "))
5 while abs(error) > accuracy:
6     e_old = exp
7     exp = (1 + p_val) ** (1 / p_val)
8     error = exp - e_old
9     p_val /= 10
10    print "e = ", exp

```

We initialise `p_val` with the value of 0.1 as before, in line 1. In line 2 we initialise `exp` to 0, this is so that when we store `exp` in `e_old` (line 6) for the very first loop Python won't give us an error message saying that "exp" is not defined.

### 7.2.2 Numerical differentiation

Limits are also used to compute the derivative of a function. The definition of the derivative  $f'(x)$  of a function  $f(x)$  is given by

$$f'(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h} \quad (7.2)$$

The problem with the above definition, if we want to use a computer to compute the derivative, is that we cannot divide by zero. We have encountered this problem before. Although we cannot divide by zero, let's try dividing by a number that approaches zero. We generate a series of approximations to  $f'(x)$  by using values of  $h$  that become progressively smaller. If such a series converges to some value, we have found the derivative.

You already know enough about programming to compute the derivative of a given function  $f(x)$  using Eq.(7.2). As an example, let's try to compute the derivative of  $f(x) = \sin(x)$  at  $x = 0$ . It is important to realise that we can only compute the derivative of  $f(x)$  at a specified value for  $x$ . We cannot get the analytical derivative, rather we compute the numerical derivative at a specified value for  $x$ . The following program approximates the derivative of  $\sin(x)$  at  $x = 0$  by using a  $h$ -value of 0.1:

```
1 from math import sin
2
3 delta_x = 0.1
4 x_val = 0.0
5 func1 = sin(x_val + delta_x)
6 func2 = sin(x_val)
7 delta_func = (func1 - func2) / delta_x
8 print "df = ", delta_func
```

and it produces the following output:

```
1 df = 0.998334166468
```

The exact derivative is  $\cos(0) = 1.0$ . The numerical derivative is already accurate to 2 digits. We can get a more accurate value for the numerical derivative by using a smaller value for  $h$ . If we used `delta_x = 0.001` instead, the output is

```
1 df = 0.999999833333
```

which is accurate to 6 digits. Let's also consider the function  $f(x) = e^x$ . The analytical derivative is  $f'(x) = e^x$ . If we use this program:

```
1 from math import exp
2
3 delta_x = 0.1
4 x_val = 1.0
5 func1 = exp(x_val + delta_x)
6 func2 = exp(x_val)
7 delta_func = (func1 - func2) / delta_x
8 print "df = ", delta_func
```

the output is

```
1 df = 2.85884195487
```

instead of the analytical value  $2.7182818284589\dots$ . This is only accurate to the first digit. If we rather use `delta_x = 0.001` the output is

```
1 df = 2.71964142253
```

which is better but only accurate to 3 digits. These two examples illustrate the difficulty in obtaining accurate numerical gradients: we do not know in advance how small  $h$  must be. Sometimes  $h = 0.001$  produces a numerical gradient accurate to 6 digits but for some other function it might only give an answer accurate to 3 digits.

These examples illustrate the general approach to developing a program that must perform some specified task. First of all, play around with specific examples in order to fully understand the problem. Then you can generalise and develop the final version. Let's investigate the behaviour of the numerical differentiation formula further. I'll use a `for` loop statement to check how the numerical gradient value changes as the value for  $h$  decreases (i.e. tend towards zero). I'll use the  $f(x) = e^x$  function again.

```
1 from math import exp
2
3 x_val = 1.0
4 for i in range(1, 17):
5     delta_x = 10 ** (-i)
6     func1 = exp(x_val + delta_x)
7     func2 = exp(x_val)
8     delta_func = (func1 - func2) / delta_x
9     print "df = %.8f" % delta_func
```

Do you understand this program? Make sure that you follow the logic. We evaluate the numerical derivative of the function  $e^x$  at  $x = 1$ . The `for` loop index `i` is used to define the value of `delta_x`. As the `index` value changes from 1 to 16, the corresponding `delta_x` value changes from  $10^{-1}$  to  $10^{-16}$ . The following output is produced (I've modified the output below to save space):

```
1 df = 2.85884195
2 df = 2.73191866
3 df = 2.71964142
4 df = 2.71841775
5 df = 2.71829542
6 df = 2.71828319
```

```
7 df = 2.71828197
8 df = 2.71828182
9 df = 2.71828204
10 df = 2.71828338
11 df = 2.71831446
12 df = 2.71871414
13 df = 2.71782596
14 df = 2.70894418
15 df = 3.10862447
16 df = 0.00000000
```

The numerical gradients approach the analytical value as `delta_x` is decreased, it then stabilises for a range of `delta_x` values ( $10^{-6}$  to  $10^{-10}$ ) that produce numerical gradients very close to the analytical value and finally the numerical gradients become inaccurate if `delta_x` becomes too small. This is known as reaching machine precision i.e. the two numbers `func1` and `func2` that we are subtracting are almost identical and the computer does not use enough digits to perform this computation accurately. In fact, a value for `delta_x` =  $10^{-16}$  is so small that `x_val` and `x_val + delta_x` are identical using 15 digits. Therefore `delta_func = 0` for this case (and all values for `delta_x` smaller than  $10^{-16}$ ). So the trick in computing accurate numerical gradients is to find the value of `delta_x` that is just right: not too big to give inaccurate values and not so small to reach machine precision.

**Take Note:**

Computations with very small numbers (around  $10^{-15}$ ) will give unexpected results or errors. This is known as reaching machine precision.

The solution I suggest makes use of a `while` loop statement. We have seen the `while` loop statement before when we computed a Taylor series. The idea then was to keep on adding terms to the series until the function value approximation converged to within some specified tolerance. I will make use of the same logic now: I'll keep on making `delta_x` smaller until the numerical derivative converges to within some specified tolerance. Here's the program:

```
1 from math import exp
2
3 accuracy = 0.001
4 x_val = 1.0
5 delta_x = 0.1
6 func1 = exp(x_val + delta_x)
```

```
7 func2 = exp(x_val)
8 delta_func = (func1 - func2) / delta_x
9 grad_change = 1
10 while grad_change > accuracy:
11     df_prev = delta_func
12     delta_x /= 10
13     func1 = exp(x_val + delta_x)
14     delta_func = (func1 - func2) / delta_x
15     grad_change = abs(delta_func - df_prev)
16 print "h = ", delta_x
17 print "df = ", delta_func
```

Do you understand the logic of the above program? Compare it to the last example of the Taylor series program that used a `while` loop statement (Section 5.4). The similarity should be obvious.

I start off by calculating a numerical gradient using the value `delta_x = 0.1` outside the `while` loop statement. Next, I have to decide on the condition which has to be satisfied (be `True`) in order to repeat the loop: this is the most important part of the program. I decided to use a name `grad_change` in which I store the change in the value of the numerical gradient each time `delta_x` is made smaller. That's why I need to compute the initial value for `delta_func` outside the loop: I need the initial value in order to calculate the change in the value. So the condition of the `while` loop statement is that I check whether or not the change in the numerical gradient value is larger than the specified accuracy. If this is true, I have to make `delta_x` smaller (I do this by dividing `delta_x` by 10 in program line 12) and repeat the numerical gradient computation. This process repeats itself until the change in the numerical gradient becomes smaller than the required accuracy.

Inside the loop, the value of `delta_x` is changed. Therefore, the function evaluated at `x_val + delta_x` will change (`func1` in my program). Since the value of `x_val` doesn't change, the function value evaluated at `x_val` doesn't change and there is no need to recompute `func2` inside the loop. You are welcome to do it, but you are just wasting valuable computer time. In my program, the value contained in `func2` was assigned outside the loop and that value remains in the name `func2` until it is replaced by some other value (which does not occur in this example). The program produces the following output:

```
1 h = 1e-05
2 df = 2.71829541996
```

Just to make things interesting, I now include a different solution to the above prob-

lem.

```
1 from math import exp
2
3 accuracy = 0.001
4 x_val = 1.0
5 delta_x = 1.0
6 delta_func = 0.0
7 grad_change = 1
8 while grad_change > accuracy:
9     df_prev = delta_func
10    delta_x /= 10
11    func1 = exp(x_val + delta_x)
12    func2 = exp(x_val)
13    delta_func = (func1 - func2) / delta_x
14    grad_change = abs(delta_func - df_prev)
15 print "h = ", delta_x
16 print "df = ", delta_func
```

In the example above, I do not compute anything outside the `while` loop statement. This means that I now have to be careful not to use anything inside the `while` loop that is not defined the very first time the loop is executed. The first statement inside the `while` loop saves the previous value for the derivative `delta_func` in the name `df_prev`. This logic is OK from the second time the `while` loop is executed (since I compute the name `delta_func` in line 13 every time the `while` loop is executed). But I need some value for `delta_func` the very first time the loop is executed. I've decided to use a value `delta_func = 0.0`, which I assign outside the `while` loop statement. This version of the program is probably a bit risky, since the value I assign to `delta_func` outside the `while` loop is not based on a formula, rather I pick some arbitrary number. Note that program line 12 makes the program less efficient than the previous solution. Since `x_val` does not change, it is unnecessary to recompute `func2`. An improved version of this program will move program line 12 outside the `while` loop, somewhere below line 4.

### 7.2.3 Solving a non-linear equation: Newton's method

I'll persist with my approach to teach programming by using yet another mathematical algorithm. Although I will now give some background to the method, this is not necessary. In general, I will expect you to be able to program some method even if you have never heard about it before. I will give you enough information to implement the method though.

Newton's method is an iterative method to find a solution to the non-linear equation

$$f(x) = 0 \quad (7.3)$$

The function  $f(x)$  is one-dimensional i.e. it depends only on the variable  $x$ . Since  $f(x)$  is non-linear, we can't simply isolate  $x$  on the left and group all constants to the right. Examples of non-linear functions are  $f(x) = x \sin(x) - x^2$ ,  $f(x) = \cos(x/2)$  and  $f(x) = x^2 - 4x + 2$ . Non-linear functions can have more than one solution, but if we use a computer to find the solutions, we will have no idea how many solutions there are. The best we can do is to find **one** of the many solutions.

As I've mentioned, Newton's method is iterative. That means that if we have a current guess for the solution, we can use that guess to find a better guess. We keep going like that until we find some  $x$  that satisfies  $f(x) = 0$ . Newton's algorithm starts with some guess  $x_0$  to the problem  $f(x) = 0$  and we then generate a series of improved solutions  $x_1$ ,  $x_2$ ,  $x_3$  etc. from the equation:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \quad \text{for } k = 0, 1, 2, 3, \dots \quad (7.4)$$

where  $f'(x)$  is the derivative of  $f(x)$ . This is sufficient information to program Newton's method. You simply have to realise that Eq.(7.4) defines the next solution ( $x_{k+1}$ ) in terms of the previous solution ( $x_k$ ). So if we have  $x_0$ , we can get  $x_1$ . If we have  $x_1$ , we can get  $x_2$ . And so on.

Just to be friendly, I'll give you a bit of background on Eq.(7.4). Consider a general non-linear function  $f(x)$  in the vicinity where  $f(x)$  goes through zero. This is depicted in Figure (7.1). Also assume that we have some current guess  $x_k$  and we want to find  $x_{k+1}$  that is an even better guess to  $f(x) = 0$ .

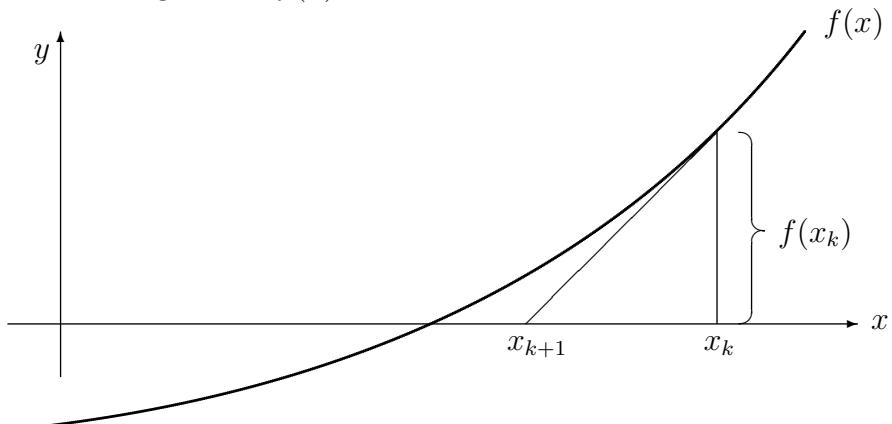


Figure 7.1: Graphical illustration of Newton's method

Newton's method is based on the following idea: get the tangent to the function at  $x_k$  and find the position where the tangent line goes through zero. We call this point  $x_{k+1}$ .

This is depicted graphically in Figure (7.1). Now we need a mathematical expression for Newton's method. The slope of the tangent line at  $x_k$  is given by  $f'(x_k)$ . This slope is also given by

$$f'(x_k) = \frac{f(x_k) - 0}{x_k - x_{k+1}} \quad (7.5)$$

If we solve Eq.(7.5) for  $x_{k+1}$ , we get

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \quad (7.6)$$

I hope that you agree Newton's method is elegant and simple. We only require three pieces of information i.e.  $x_k$ ,  $f(x_k)$  and  $f'(x_k)$  in order to find the next (improved) solution to  $f(x) = 0$ . I'll now illustrate Newton's method for the function  $f(x) = \cos(x/2)$  using the initial guess  $x_0 = 3$ . Hopefully you recognise that I'm looking for the solution  $x = \pi$ . Let's see how Newton's method produce the series  $x_1, x_2, x_3, \dots$  that converges to  $\pi$ . We first get the function value and the derivative value:

$$\begin{aligned} f(x_0) &= \cos\left(\frac{3}{2}\right) = 0.070737201 && \text{and} \\ f'(x_0) &= -\frac{1}{2} \sin\left(\frac{3}{2}\right) = -0.498747493 \end{aligned} \quad (7.7)$$

Now we use this information to calculate  $x_1$ :

$$\begin{aligned} x_1 &= x_0 - \frac{f(x_0)}{f'(x_0)} = 3 - \frac{0.0707372}{-0.4987475} \\ &= 3.141829689 \end{aligned} \quad (7.8)$$

With this new guess  $x_1$ , we can find  $x_2$ . We proceed as before.

$$\begin{aligned} f(x_1) &= \cos\left(\frac{3.1418297}{2}\right) = -0.000118517 && \text{and} \\ f'(x_1) &= -\frac{1}{2} \sin\left(\frac{3.141829689}{2}\right) = -0.5000 \end{aligned} \quad (7.9)$$

Now we use this information to calculate  $x_2$ :

$$\begin{aligned} x_2 &= x_1 - \frac{f(x_1)}{f'(x_1)} = 3.1418297 - \frac{(-0.0001185)}{(-0.5)} \\ &= 3.141592654 \end{aligned} \quad (7.10)$$

We compute  $\pi$  accurate to 10 digits within only 2 iterations using Newton's method. I hope this convinces you that Newton's method is not only elegant and simple, it is also very efficient.

You might think the above exercise is a waste of time. But, how do you expect to program a method if are not completely comfortable with the application of the method. In my opinion, this should be your approach whenever you attempt a new program. First make sure that you yourself can perform the task required, only then attempt to give a computer a list of instructions to do what you just did. So here we go. In the program below I use a computer to perform 5 iterations of Newton's method.

```

1  from math import cos, sin
2
3  x_val = 3.0
4  for i in range(5):
5      func_val = cos(x_val / 2.0)
6      deriv_val = -0.5 * sin(x_val / 2.0)
7      x_val -= func_val / deriv_val
8      print "x = %.8f      f(x) = %.8f" % (x_val, func_val)
```

The output of the above program, using `format long`, is

```

1  x = 3.14182969      f(x) = 0.07073720
2  x = 3.14159265      f(x) = -0.00011852
3  x = 3.14159265      f(x) = 0.00000000
4  x = 3.14159265      f(x) = 0.00000000
5  x = 3.14159265      f(x) = 0.00000000
```

Do you think the above program is a good one? In general, do you think that we will know in advance (i.e. before we run our program) how many iterations would be required to find the solution? The answer is No: it is unlikely that we will know how many times the `for` loop must be executed. Let's modify the program to rather make use of a `while` loop statement.

Whenever we make use of a `while` loop statement, the most important part is to figure out the condition that must be satisfied (be `True`) in order to keep on repeating the `while` loop. Or, find a condition which tells us that the loop must not be repeated any more. Can you suggest an idea? I can think of two conditions that tell me that we have found the solution:

1. If  $f(x) = 0$ , we can stop. However, when we are dealing with real numbers represented by a computer, we cannot expect to check if the function value  $f(x)$  equals zero. What we can do, is to check if  $f(x)$  is almost zero i.e.

$$-\text{tol} \leq f(x) \leq \text{tol} \quad \text{or} \quad |f(x)| \leq \text{tol} \quad (7.11)$$

where `tol` is some small value (I use terminology such as tolerance, or accuracy). If Eq. 7.11 is satisfied, it means that we have found the solution (or at least an accurate enough approximation to the solution) and the `while` loop must terminate. However, the syntax of a `while` loop statement requires that the condition must be satisfied (be `True`) to continue (not terminate) the loop. So in our program the inequality changes to:

```
1   while abs(f) > tol:
```

where `func_val` is a name that contains the function value. If we use this idea, the specified accuracy ensures that  $f(x)$  will be less than or equal to the specified accuracy once the program terminates.

2. If the absolute value of  $\Delta x = x_{k+1} - x_k = -f(x_k)/f'(x_k)$  is very small, we can stop. This is an indirect criterion that tells us we have found the solution. You should recognise from Eq. 7.4 that if  $f(x_k) = 0$ ,  $x_{k+1} = x_k$ , i.e.  $\Delta x = 0$ . If we want to use this condition in our program, the `while` loop statement will be:

```
1   while abs(delta_x) > tol:
```

where `delta_x` is the name containing  $x_{k+1} - x_k$ . If we use this idea, the specified accuracy ensures that the distance between two consecutive guesses for  $x$  will be less than or equal to the specified accuracy.

Here follows a program that makes use of the first idea (i.e. we want to make sure that  $f(x) \approx 0$ , condition 1 above):

```
1  from math import cos, sin
2
3  x_val = 3.0
4  func_val = cos(x_val / 2.0)
5  tol = 1E-6
6  while abs(func_val) > tol:
7      deriv_val = -0.5 * sin(x_val / 2)
8      delta_x = -func_val / deriv_val
9      x_val += delta_x
10     func_val = cos(x_val / 2)
11     print "x = %.8f      f(x) = %.8f" % (x_val, func_val)
```

Do you follow the logic of the above program? I define the original guess to  $x$  in line 3. Next I compute the function value in line 4 and then define a tolerance in line 5.

Lines 4 and 5 are essential when viewed in conjunction with line 6. In line 6 I test if the absolute function value is greater than the tolerance. In order to perform this check, I need the values of both `func_val` and `tol`. Inside the loop, I compute the derivative (line 7), followed by the change in  $x$  (line 8). In line 9 I increment  $x$  (the old value) with  $\Delta x$  to get the new improved guess for  $x$ . In program line 10, I compute the function value, using the  $x$  value just computed in line 9. The function value which was just computed in line 10 is compared to `tol` and a decision is made on whether or not to repeat the loop once more (line 6).

The program above produces the following output. As you can see, using the `while` loop statement we only require two iterations before  $|f(x)|$  is less than  $10^{-6}$ .

```
1 x = 3.14182969      f(x) = -0.00011852
2 x = 3.14159265      f(x) =  0.00000000
```

As an alternative, I'll now give a few examples of how we can use the condition  $|\Delta x| > \text{tol}$  to decide whether or not to repeat the loop.

### 7.2.3.1 Example 1

In this example, I make a copy of `x_val` just before I compute the new value of `x_val` (line 9). After the new value of `x_val` is computed, I subtract it from the old value (line 9) and use this difference (`delta_x`) to decide whether or not to repeat the while loop (line 6).

```
1 from math import cos, sin
2
3 x_val = 3.0
4 tol = 1E-6
5 delta_x = 1
6 while delta_x > tol:
7     func_val = cos(x_val / 2.0)
8     deriv_val = -0.5 * sin(x_val / 2.0)
9     x_old = x_val
10    x_val -= func_val / deriv_val
11    delta_x = abs(x_val - x_old)
12    print "x = %.8f      f(x) = % .8f" % (x_val, func_val)
```

### 7.2.3.2 Example 2

In this example, I use a name `x_new` to store the value of the next  $x$ . Since I don't destroy the content of the name `x_val` in this case, I compute the difference between the new estimate `x_new` and previous estimate `x_val` in line 10 and store it in `delta_x`. However, since the next value of  $x$  is now stored in `x_new`, I need one additional program line. Line 11 moves the new value of  $x$  stored in `x_new` to the name `x_val`. You'll see that if the program now returns to line 6 and decides to repeat the while loop, the name `x_val` in lines 9, 10 and 11 contains the improved guess for  $x$ .

```

1 from math import cos, sin
2
3 x_val = 3.0
4 tol = 1E-6
5 delta_x = 1
6 while delta_x > tol:
7     func_val = cos(x_val / 2.0)
8     deriv_val = -0.5 * sin(x_val / 2.0)
9     x_new = x_val - func_val / deriv_val
10    delta_x = abs(x_new - x_val)
11    x_val = x_new
12    print "x = %.8f      f(x) = % .8f" % (x_val, func_val)
```

### 7.2.3.3 Example 3

In this example, we recognise that  $\Delta x = x_{k+1} - x_k = -f(x_k)/f'(x_k)$ . Therefore, we might as well check if  $f(x_k)/f'(x_k)$  is small enough to decide whether or not to terminate the loop. By now you should be able to figure out the logic in the program below.

```

1 from math import cos, sin
2
3 x_val = 3.0
4 tol = 1E-6
5 delta_x = 1
6 while abs(delta_x) > tol:
7     func_val = cos(x_val / 2.0)
8     deriv_val = -0.5 * sin(x_val / 2.0)
9     delta_x = -func_val / deriv_val
10    x_val += delta_x
11    print "x = %.8f      f(x) = % .8f" % (x_val, func_val)
```

#### 7.2.3.4 Example 4

In this last example I will use the combined condition to test whether or not to continue the `while` loop statement: if  $|\Delta x| > \text{tol}$  and  $|f(x)| > \text{tol}$  continue the `while` loop. By now you should be able to figure out the logic in the program below.

```
1 from math import cos, sin
2
3 x_val = 3.0
4 tol = 1E-6
5 delta_x = 1
6 func_val = cos(x_val / 2.0)
7 while (abs(delta_x) > tol) and (abs(func_val) > tol):
8     func_val = cos(x_val / 2.0)
9     deriv_val = -0.5 * sin(x_val / 2.0)
10    delta_x = -func_val / deriv_val
11    x_val += delta_x
12    print "x = %.8f      f(x) = %.8f" % (x_val, func_val)
```

#### 7.2.3.5 Example 5

In this example I will ask the user to enter:

- The initial guess for  $x$  and
- The required accuracy for computing  $x$ .

The program will then apply Newton's method to find  $x$  which satisfies  $f(x) = 0$ .

```
1 from math import cos, sin
2
3 x_val = float(raw_input("Enter a starting guess for x: "))
4 tol = float(raw_input("Enter a required accuracy: "))
5
6 func = lambda x: cos(x / 2.0)
7 deriv = lambda x: -0.5 * sin(x / 2.0)
8
9 print # print a blank line
10
```

```
11 func_val = func(x_val)
12 delta_x = 1
13 while abs(delta_x) > tol:
14     deriv_val = deriv(x_val)
15     delta_x = -func_val / deriv_val
16     x_val += delta_x
17     func_val = func(x_val)
18     print "x = %.8f      f(x) = %.8f" % (x_val, func_val)
```

The output for the program above is:

```
1 Enter a starting guess for x: 3.0
2 Enter a required accuracy: 1e-8
3
4 x = 3.14182969      f(x) = -0.00011852
5 x = 3.14159265      f(x) =  0.00000000
6 x = 3.14159265      f(x) =  0.00000000
```

If the function and derivative are change in program lines 6 and 7 respectively:

```
6 func = lambda x: x**2.0 - 3.0
7 deriv = lambda x: 2.0 * x
```

The following output is obtained:

```
1 Enter a starting guess for x: 1.0
2 Enter a required accuracy: 1e-8
3
4 x = 2.00000000      f(x) =  1.00000000
5 x = 1.75000000      f(x) =  0.06250000
6 x = 1.73214286      f(x) =  0.00031888
7 x = 1.73205081      f(x) =  0.00000001
8 x = 1.73205081      f(x) = -0.00000000
```

### 7.2.4 Newton's method using numerical gradients

In the previous example we developed a program that uses Newton's method to solve a non-linear equation  $f(x) = 0$ . One of the requirements for the program is the derivative of the function  $f(x)$ . Which is easy enough to add to the program if you either have simple functions (e.g.  $f(x) = 2x^5 + 4$ ) or you remember the derivative of the function  $f(x)$ . However, the more likely scenario is that you will either have very complicated functions or you can't differentiate them.

Fortunately, we have already developed a program that can calculate numerical gradients. So instead of adding the derivative of  $f(x)$  to the program, we can compute a numerical derivative. A program using Newton's method to solve  $f(x) = 0$  using numerical gradients might look like this:

```
1  from math import cos
2
3  x_val = float(raw_input("Enter a starting guess for x: "))
4  tol = float(raw_input("Enter a required accuracy: "))
5
6  print "# print a blank line"
7
8  func = lambda x: cos(x / 2.0)
9
10 delta_x = 2.0 * tol
11 while abs(delta_x) > tol:
12     func_val = func(x_val)
13
14     h_val = 0.1
15     diff = 2.0 * tol
16     deriv_val = (func(x_val + h_val) - func_val) / h_val
17     while diff > tol:
18         h_val /= 10
19         df_previous = deriv_val
20         deriv_val = (func(x_val + h_val) - func_val) / h_val
21         diff = abs(deriv_val - df_previous)
22
23     delta_x = -func_val / deriv_val
24     x_val += delta_x
25     print "x = %.8f      f(x) = %.8f" % (x_val, func_val)
```

In the example program above, Line 14 – Line 21 is used to compute an accurate

numerical gradient. When we first programmed Newton's method, these 8 lines were reduced to a single program line:

```
1 deriv_val = deriv(x)
```

The same examples I used to illustrate Newton's method, using analytical derivatives, now follow. Compare the results and you'll see the subtle differences in the computed values.

`func = lambda x: cos(x / 2.0)` (line 8):

```
1 Enter a starting guess for x: 3.0
2 Enter a required accuracy: 1e-8
3
4 x = 3.14182969      f(x) =  0.07073720
5 x = 3.14159265      f(x) = -0.00011852
6 x = 3.14159265      f(x) =  0.00000000
```

`func = lambda x: x**2 - 3` (line 8):

```
1 Enter a starting guess for x: 1.0
2 Enter a required accuracy: 1e-8
3
4 x = 1.99999992      f(x) = -2.00000000
5 x = 1.75000002      f(x) =  0.99999967
6 x = 1.73214286      f(x) =  0.06250007
7 x = 1.73205080      f(x) =  0.00031888
8 x = 1.73205081      f(x) = -0.00000003
```

The above implementation of Newton's method is not very efficient since we spend a lot of computational effort inside the inner `while` loop statement to find an appropriate value for `h_val`. I'm of the opinion that once we obtain a value for `h_val` that gives accurate numerical derivatives, we do not need to repeat this calculation for every Newton iteration. (However, if we change the function  $f(x)$ , we might need to find a new `h_val` value). We can simplify Newton's method, using numerical gradients, by using two consecutive `while` loop statements instead of two nested `while` loop statements. In the program that follows, program lines 10–18 are used to compute an appropriate value of

`h_val` that provides an accurate numerical gradient. This value for `h_val` is then used in all subsequent computations.

**More Info:**

Nested statements refers to statements within statements:

E.g.: nested `for` loop statements refers to `for` loop statements within `for` loop statements

```
1 from math import cos
2
3 x_val = float(raw_input("Enter a starting guess for x: "))
4 tol = float(raw_input("Enter a required accuracy: "))
5
6 print "# print a blank line"
7
8 func = lambda x: cos(x / 2.0)
9
10 h_val = 0.1
11 diff = 2.0 * tol
12 func_val = func(x_val)
13 deriv_val = (func(x_val + h_val) - func_val) / h_val
14 while diff > tol:
15     h_val /= 10
16     df_previous = deriv_val
17     deriv_val = (func(x_val + h_val) - func_val) / h_val
18     diff = abs(deriv_val - df_previous)
19
20 delta_x = 2.0 * tol
21 while abs(delta_x) > tol:
22     func_val = func(x_val)
23     deriv_val = (func(x_val + h_val) - func_val) / h_val
24     delta_x = -func_val / deriv_val
25     x_val += delta_x
26     print "x = %.8f      f(x) = % .8f" % (x_val, func_val)
```

The two previous programs again illustrate how different solutions can exist to the same problem. It's up to you as the programmer to decide on a particular implementation.

### 7.2.5 The bisection method

Here we go again. I'll use a simple numerical algorithm to highlight some programming ideas. To solve an equation  $f(x) = 0$ , we have already used Newton's method. Another method, which is based on 'common sense' rules is called the bisection method.

The idea is simple. First of all, we have to start with a function  $f(x)$  as well as an interval  $x \in [x_l, x_u]$  such that the function undergoes a sign change between  $x_l$  and  $x_u$ .  $x_l$  refers to the lower bound of the interval and  $x_u$  refers to the upper bound of the interval. The sign change implies that either  $f(x_l) < 0$  and  $f(x_u) > 0$ , or  $f(x_l) > 0$  and  $f(x_u) < 0$ . If the function is continuous and it changes sign somewhere between  $x_l$  and  $x_u$  there must be a point  $x$  between  $x_l$  and  $x_u$  where  $f(x) = 0$ .

The interval is now divided into two equal parts. The function is also evaluated at this midpoint  $x_m = \frac{1}{2}(x_l + x_u)$ . All that we have to do now is to decide in which of the two new intervals the solution lies. Either the solution lies between the lower bound  $x_l$  and the midpoint  $x_m$ , or the solution lies between the midpoint  $x_m$  and the upper bound  $x_u$ .

We want to implement the bisection algorithm to find a solution to  $f(x) = 0$ . Recall that we have a lower bound  $x_l$ , midpoint  $x_m$  and upper bound  $x_u$  and we have to decide in which of the two possible intervals the solution lies (i.e. in which of the two intervals a sign change occurs). An elegant method is to check if  $f(x_l) \times f(x_m) < 0$ ? If this condition is true, it means that there is a sign change between the lower bound and the midpoint. Therefore the solution must lie in this lower interval, and we discard the upper interval.

If the condition is not true, the sign change does not occur in the lower interval and it must therefore occur in the upper interval. The solution must lie in the upper interval, and (in this case) the lower interval is discarded. We then repeat the process on the retained interval until the solution is found.

So when do we stop dividing intervals in half? There are two possible criteria we can use to decide when to stop the algorithm.

- As soon as  $|f(x_m)| < \varepsilon$ , where  $\varepsilon$  is some very small number, we can assume we have found the solution. There is no way of knowing how many times the initial interval has to be divided in two to reach the point where  $|f(x)| < \varepsilon$ , so we will have to make use of a `while` loop statement.
- As soon as the interval size is very small (i.e.  $x_m - x_l < \varepsilon$  or  $x_u - x_m < \varepsilon$ ) we know that we have found the solution to  $x$  within an accuracy of  $\varepsilon$ . We can also use a `while` loop statement here, but is it really necessary? We know the size of

the initial interval and we know that this initial interval is repeatedly divided in two equal parts. If the initial interval size is  $\Delta x$ , the interval size after 1 iteration is  $\frac{1}{2}\Delta x$ . After 2 iterations, the interval size is equal to  $\frac{1}{2}(\frac{1}{2}\Delta x) = (\frac{1}{2})^2\Delta x$ . After  $n$  iterations, the interval size will be  $(\frac{1}{2})^n\Delta x$ . So we can determine the required number of iterations  $n$  if we require the final interval size to be equal to some specified size  $\varepsilon$ , i.e.

$$(\frac{1}{2})^n \Delta x = \varepsilon \quad (7.12)$$

We can solve Eq. 7.12 for  $n$ :

$$n = \frac{\ln(\frac{\varepsilon}{\Delta x})}{\ln(\frac{1}{2})} \quad (7.13)$$

Here follows both above implementations of the bisection method. First the version that uses a `while` loop statement. The comment statements in the program explains the logic.

```

1 # Create the function to evaluate
2 func = lambda x: 1.2 - x
3
4 # Get the lower and upper bound of the initial interval
5 x_lower = float(raw_input("Enter lower bound: "))
6 x_upper = float(raw_input("Enter upper bound: "))
7
8 # Get the required accuracy
9 acc = float(raw_input("Enter required accuracy: "))
10
11 # Compute the midpoint
12 x_mid = 0.5 * (x_lower + x_upper)
13 print "x_m = %.8f" % x_mid
14
15 # Compute function value at lower, upper and midpoint
16 f_lower = func(x_lower)
17 f_upper = func(x_upper)
18 f_mid = func(x_mid)
19
20 # Start of main conditional loop: check if |f(x)| > acc
21 while abs(f_mid) > acc:
22     # if f_lower * f_mid < 0, then sign change
23     # is in lower interval
24     if (f_lower * f_mid) < 0.0:
25         # Midpoint becomes new upper bound
26         x_upper = x_mid
27         f_upper = f_mid

```

```
28     else:  
29         # sign change must be in upper interval  
30         # midpoint becomes new lower bound  
31         x_lower = x_mid  
32         f_lower = f_mid  
33         # At the end of the if statement, we either updated  
34         # the lower or the upper bound and now we must update  
35         # the midpoint and associated function value  
36         x_mid = 0.5 * (x_lower + x_upper)  
37         f_mid = func(x_mid)  
38         print "x_m = %.8f" % x_mid
```

Many students understand the use of the `while` loop statement (program line 21) as well as the `if` statement (program line 24), but they cannot understand program lines 36 and 37. I'll do my best to explain:

All that the `if` statement does is to decide which of the two possible intervals to retain (upper or lower). Either the upper bound is replaced with the midpoint (program line 26) or the lower bound is replaced with the midpoint (program line 31). At the completion of the `if` statement (program line 33), the new bounds have to be used to compute the new midpoint (program line 36) and the associated function value (program line 37).

The new midpoint that is calculated, every time the `while` loop is executed, is computed in program line 36. Here's an example of using the bisection method to find a root to the equation  $f(x) = 1.2 - x$  in the interval  $[1.0; 2.0]$  (Of course we know the solution is  $x = 1.2$ , but let's see how the bisection algorithm finds this solution).

```
1 Enter lower bound: 1.0  
2 Enter upper bound: 2.0  
3 Enter required accuracy: 0.0001  
4 x_m = 1.50000000  
5 x_m = 1.25000000  
6 x_m = 1.12500000  
7 x_m = 1.18750000  
8 x_m = 1.21875000  
9 x_m = 1.20312500  
10 x_m = 1.19531250  
11 x_m = 1.19921875  
12 x_m = 1.20117188  
13 x_m = 1.20019531  
14 x_m = 1.19970703  
15 x_m = 1.19995117
```

Now for the bisection algorithm version that makes use of a `for` loop statement:

```
1  from math import log
2
3  # Create the function to evaluate
4  func = lambda x: 1.2 - x
5
6  # Get the lower and upper bound of the initial interval
7  x_lower = float(raw_input("Enter lower bound: "))
8  x_upper = float(raw_input("Enter upper bound: "))
9
10 # Get the required accuracy
11 acc = float(raw_input("Enter required accuracy: "))
12
13 # Compute the midpoint
14 x_mid = 0.5 * (x_lower + x_upper)
15 print "x_m = %.8f" % x_mid
16
17 # Compute function value at lower, upper and midpoint
18 f_lower = func(x_lower)
19 f_upper = func(x_upper)
20 f_mid = func(x_mid)
21
22 # Compute the required number of iterations
23 n = log(acc / (x_upper - x_lower)) / log(0.5)
24 n = int(n)      # convert float to integer
25
26 # Start of for loop
27 for i in range(n):
28     # if F_lower*F_mid < 0, then sign change is in lower interval
29     if (f_lower * f_mid) < 0.0:
30         # Midpoint becomes new upper bound
31         x_upper = x_mid
32         f_upper = f_mid
33     else:
34         # sign change must be in upper interval
35         # midpoint becomes new lower bound
36         x_lower = x_mid
37         f_lower = f_mid
38     # At the end of the if statement, we either updated
39     # the lower or the upper bound and now we must update
40     # the midpoint and associated function value
41     x_mid = 0.5 * (x_lower + x_upper)
```

```
42     f_mid = func(x_mid)
43     print "x_m = %.8f" % x_mid
```

This algorithm is also illustrated using the example  $f(x) = 1.2 - x$ .

```
1 Enter lower bound: 1.0
2 Enter upper bound: 2.0
3 Enter required accuracy: 0.0001
4 x_m = 1.50000000
5 x_m = 1.25000000
6 x_m = 1.12500000
7 x_m = 1.18750000
8 x_m = 1.21875000
9 x_m = 1.20312500
10 x_m = 1.19531250
11 x_m = 1.19921875
12 x_m = 1.20117188
13 x_m = 1.20019531
14 x_m = 1.19970703
15 x_m = 1.19995117
16 x_m = 1.20007324
17 x_m = 1.20001221
```



# Chapter 8

## Numerical integration

Recall from the Riemann sum in Calculus that integration implies finding the area under a curve, as depicted in Figure 8.1.

Let's try to develop a program that can be used to integrate any function numerically. A simple method to approximate the integral of a function  $f(x)$  between the lower bound  $a$  and upper bound  $b$ , is by constructing rectangles over intervals between  $a$  and  $b$ , as depicted Figure 8.1. We can then easily calculate and add the areas of the rectangles.

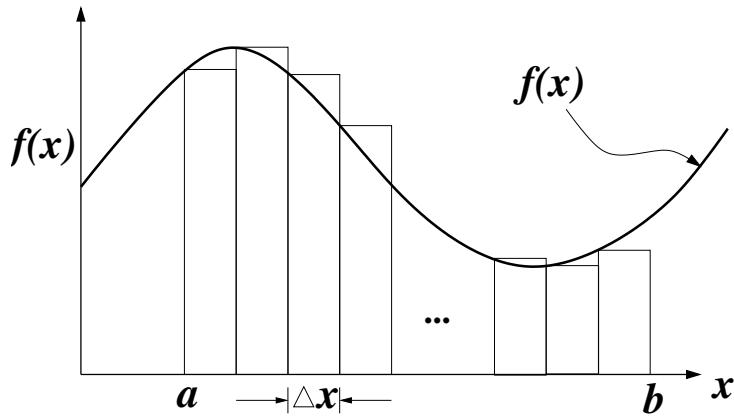


Figure 8.1: Numerical integration of  $\int_a^b f(x)dx$  using a left point method.

We can simply make the width of each rectangle the same i.e.  $\Delta x$  but then we still need to calculate the height of each rectangle. We can compute the height of the rectangles in various ways. I can think of at least 3 methods:

1. A left-point method, where the interval  $[a, b]$  is divided in  $n$  equal intervals of width  $\Delta x$  and the height of each rectangle is computed at the left point of each interval.

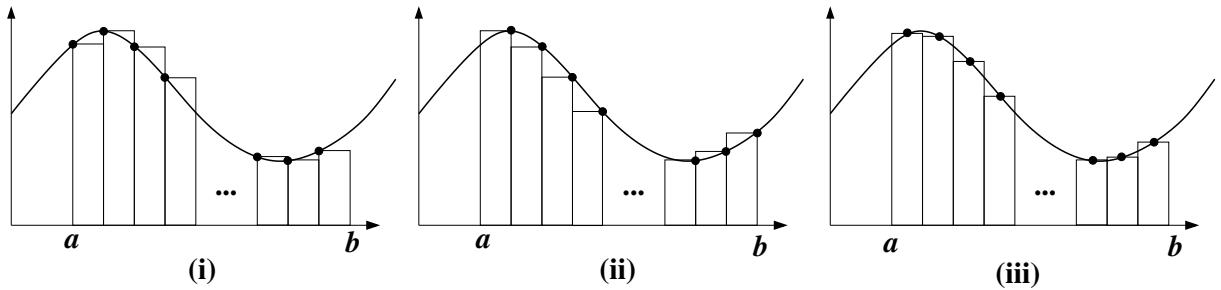


Figure 8.2: Numerical integration of  $\int_a^b f(x)dx$  using a (i) left point method, (ii) right point method and (iii) midpoint method.

I.e. the first rectangle's height is  $f(a)$ , the next rectangles height is  $f(a + \Delta x)$ , and so on. The last rectangle's height is given by  $f(b - \Delta x)$ . This method is depicted in Figure 8.2 (i).

2. A right point method is where the rectangle heights are computed by  $f(a + \Delta x), f(a + 2\Delta x), \dots, f(b)$  as depicted in Figure 8.2 (ii).
3. A midpoint method is where the rectangle heights are computed by  $f(a + \frac{1}{2}\Delta x), f(a + \frac{3}{2}\Delta x), \dots, f(b - \frac{1}{2}\Delta x)$  as depicted in Figure 8.2 (iii).

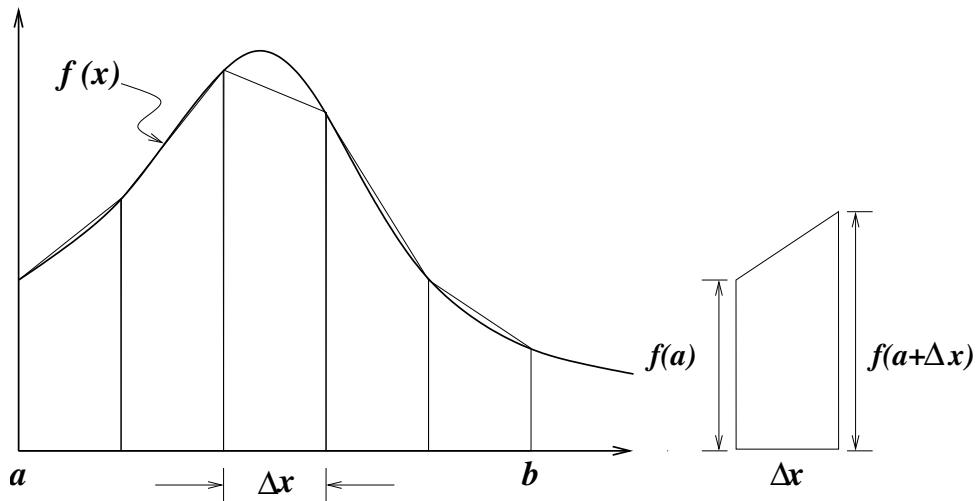


Figure 8.3: Numerical integration using the trapezium method.

Instead of approximating the integral with rectangles, the area can also be approximated by trapeziums. The area of a trapezium is given by the product of the average of the two parallel sides with the perpendicular height. The area for the trapezium between  $a$  and  $a + \Delta x$ , as depicted in Figure 8.3, is

$$\text{Area} = \Delta x \frac{1}{2}(f(a) + f(a + \Delta x)), \quad (8.1)$$

Should we approximate the total area using trapeziums, the integral is given as

$$\int_a^b f(x)dx \approx \frac{1}{2}\Delta x [f(a) + f(a + \Delta x)] + \frac{1}{2}\Delta x [f(a + \Delta x) + f(a + 2\Delta x)] + \dots + \frac{1}{2}\Delta x [f(b - \Delta x) + f(b)] \quad (8.2)$$

$$= \Delta x \left[ \frac{1}{2}f(a) + f(a + \Delta x) + f(a + 2\Delta x) + \dots + f(b - 2\Delta x) + f(b - \Delta x) + \frac{1}{2}f(b) \right] \quad (8.3)$$

Now that we have a few ideas of how to integrate a function numerically, let's develop the program. First of all, I'll list the information that has to be available to perform numerical integration:

1. The function  $f(x)$
2. The bounds of integration
3. The number of intervals
4. The method i.e. left-point, right-point, mid-point or trapezium method

After we get the above information from the user (using a number of `raw_input` statements), I'll compute the interval size  $\Delta x$  and then use a `for` loop to add all the areas. Here's my implementation of a numerical integration program:

```

1  from math import sin, pi
2
3
4  func = lambda x: sin(x)
5  func_str = "sin(x)"
6
7  lower = 0
8  upper = pi / 2
9  num_int = 20
10
11 # Display integration options
12 print ("\n" +
13     "1: Left-point 2: Right-point 3: Mid-point \n" +
14     "4: Trapezium 5: Quit program")
15 method = int(raw_input("Please enter your option: "))
16
17 # My program repeats until the user inputs option 5
18 while method != 5:

```

```
19 # Initialize the integral at zero
20 integral = 0
21 # Compute the interval size Delta_x
22 delta_x = (upper - lower) / num_int
23 if method == 1:      # left-point method
24     # For loop to compute area of each rectangle and
25     # add it to integral
26     for i in range(num_int):
27         x_val = lower + i * delta_x
28         integral += delta_x * func(x_val)
29
30 elif method == 2:      # right-point method
31     for i in range(num_int):
32         x_val = lower + (i + 1) * delta_x
33         integral += delta_x * func(x_val)
34
35 elif method == 3:      # mid-point method
36     for i in range(num_int):
37         x_val = lower + i * delta_x + 0.5 * delta_x
38         integral += delta_x * func(x_val)
39
40 elif method == 4:      # trapezium method
41     for i in range(num_int):
42         x_L = lower + i * delta_x
43         x_R = x_L + delta_x
44         integral += 0.5 * delta_x * (func(x_L) + func(x_R))
45
46 # Display answer
47 print ("Integrating %s between " +
48       "%.3f and %.3f = %.6f") % (func_str,
49                               lower,
50                               upper,
51                               integral)
52
53 # Display options again
54 print ("\n" +
55       "1: Left-point 2: Right-point 3: Mid-point \n" +
56       "4: Trapezium 5: Quit program")
57 method = int(raw_input("Please enter your option: "))
58
59 # Outside while loop, user entered option 5 = Quit
60 # Displays a friendly message.
61 print "\nThanks for using the program. Have a good day."
```

Do you notice the use of an outer `while` loop statement that keeps on repeating the numerical integration program as long as the name `method` is not equal to 5. The program listed above is used to numerically integrate the `sin` function between 0 and  $\pi/2$ . The analytical answer is 1, so let's see how the program performs using 20 intervals. I ran the program once and entered option 1 to 5 in that order. The output follows:

```
1 1: Left-point 2: Right-point 3: Mid-point
2 4: Trapezium 5: Quit program
3 Please enter your option: 1
4 Integrating sin(x) between 0 and 1.571 = 0.960216
5
6 1: Left-point 2: Right-point 3: Mid-point
7 4: Trapezium 5: Quit program
8 Please enter your option: 2
9 Integrating sin(x) between 0 and 1.571 = 1.038756
10
11 1: Left-point 2: Right-point 3: Mid-point
12 4: Trapezium 5: Quit program
13 Please enter your option: 3
14 Integrating sin(x) between 0 and 1.571 = 1.000257
15
16 1: Left-point 2: Right-point 3: Mid-point
17 4: Trapezium 5: Quit program
18 Please enter your option: 4
19 Integrating sin(x) between 0 and 1.571 = 0.999486
20
21 1: Left-point 2: Right-point 3: Mid-point
22 4: Trapezium 5: Quit program
23 Please enter your option: 5
24
25 Thanks for using the program. Have a good day.
```

## 8.1 Numerical integration using a `while` loop statement

As you can see from the example above 20 intervals is not very accurate for using either the left-point or right-point methods, but it seems adequate for the mid-point and trapezium methods. So how do we know in advance how many intervals we need to get an accurate numerically integrated value? We don't, so therefore I suggest the use of a `while` loop

statement to keep increasing the number of intervals until the numerically integrated result converges to within some specified tolerance. Here follows the modified program:

```
1  from math import sin, pi
2
3
4  func = lambda x: sin(x)
5  func_str = "sin(x)"
6
7  lower = 0
8  upper = pi / 2
9  num_init = 20
10 acc = float(raw_input("Enter required accuracy: "))
11
12 # Display integration options
13 print ("\n" +
14     "1: Left-point 2: Right-point 3: Mid-point \n" +
15     "4: Trapezium 5: Quit program")
16 method = input("Please enter your option: ")
17
18 # My program repeats until the user inputs option 5
19 while method != 5:
20     # Define initial Error greater as acc,
21     # to enter while loop
22     error = 1
23     # Since the number of intervals change during program
24     # execution, reset N to the initial number of intervals
25     num = num_init
26     # Set Int to zero to have a value the first time
27     # the while loop is executed
28     integral = 0
29
30     # Start while loop that checks if consecutive values
31     # converged
32     while error > acc:
33         # Make a copy of the previous integral just before the
34         # program section starts that computes the new value
35         integral_old = integral
36         # Initialize the integral at zero
37         integral = 0
38         # Compute the interval size Delta_x
39         delta_x = (upper - lower) / num
40         if method == 1:      # left-point method
```

```
41      # For loop to compute area of each rectangle and
42      # add it to integral
43      for i in range(num):
44          x = lower + i * delta_x
45          integral += delta_x * func(x)
46
47      elif method == 2:      # right-point method
48          for i in range(num):
49              x = lower + (i + 1) * delta_x
50              integral += delta_x * func(x)
51
52      elif method == 3:      # mid-point method
53          for i in range(num):
54              x = lower + i * delta_x + 0.5 * delta_x
55              integral += delta_x * func(x)
56
57      elif method == 4:      # trapezium method
58          for i in range(num):
59              x_L = lower + i * delta_x
60              x_R = x_L + delta_x
61              integral += 0.5 * delta_x * (func(x_L) + func(x_R))
62
63      # Compute Error i.e. difference between consecutive
64      # integrals
65      error = abs(integral - integral_old)
66      # Double the number of intervals
67      num *= 2
68
69      # Display answer
70      print ("Integrating %s between " +
71             "%.3f and %.3f = %.6f") % (func_str,
72                                         lower,
73                                         upper,
74                                         integral)
75      print "%d equal intervals needed" % num / 2
76      # Display options again
77      print ("\n" +
78             "1: Left-point 2: Right-point 3: Mid-point \n" +
79             "4: Trapezium 5: Quit program")
80      method = input("Please enter your option: ")
81
82      # Outside while loop, user entered option 5 = Quit
83      # Displays a friendly message.
```

```
84 print "\nThanks for using the program. Have a good day."
```

I'll only discuss the changes I've made. In program line 8, I now call the number of intervals `num_init`, to indicate that this is the initial number of intervals. As the program proceeds, the number of intervals increase. In program line 9 I also ask the user to enter the required accuracy, which will be used in the condition for the `while` loop statement.

In program line 21 I define a name `error`, which is the difference between consecutive numerical integrals (using a different number of intervals). The number of intervals is initially set to the value specified by the user in program line 24.

In line 27 I set the name `integral` to zero. This statement is required because I assign the value of `integral` to the name `integral_old` in program line 34. The very first time the `while` loop is executed (line 31) no calculations have been performed and therefore no value is available in the name `integral`. Hence I require line 27.

Program lines 39–60 compute the numerical integral, based on the chosen method. This part of the program is unchanged. In line 64 I compute the difference between the integral and the value obtained previously. I then double the number of intervals in line 66 and the program returns again to the `while` loop in line 31. If the difference between consecutive numerical integration values (computed in line 64) is smaller than `acc` the `while` loop terminates and the program continues at program line 68. Answers are displayed in the *IPython Console* and the options are re-displayed. Depending on the user's input, the program is either re-run (`Method = 1 to 4`), or terminates with a friendly message (`Method = 5`). If `error > acc`, the `while` loop (program lines 31–66) is repeated again.

Here follows the output for the same example I presented earlier. An accuracy of 0.0001 is specified. As you can see, the different methods require a different number of intervals to achieve this specified accuracy. Clearly the mid-point and trapezium methods (`num=80`) is much more efficient than the left-point and right-point methods (`num=10240`).

```
1 Enter required accuracy: 0.0001
2
3 1: Left-point 2: Right-point 3: Mid-point
4 4: Trapezium 5: Quit program
5 Please enter your option: 1
6 Integrating sin(x) between 0.000 and 1.571 = 0.999923
7 10240 equal intervals needed
8
9 1: Left-point 2: Right-point 3: Mid-point
10 4: Trapezium 5: Quit program
```

```
11 Please enter your option: 2
12 Integrating sin(x) between 0.000 and 1.571 = 1.000077
13 10240 equal intervals needed
14
15 1: Left-point 2: Right-point 3: Mid-point
16 4: Trapezium 5: Quit program
17 Please enter your option: 3
18 Integrating sin(x) between 0.000 and 1.571 = 1.000016
19 80 equal intervals needed
20
21 1: Left-point 2: Right-point 3: Mid-point
22 4: Trapezium 5: Quit program
23 Please enter your option: 4
24 Integrating sin(x) between 0.000 and 1.571 = 0.999968
25 80 equal intervals needed
26
27 1: Left-point 2: Right-point 3: Mid-point
28 4: Trapezium 5: Quit program
29 Please enter your option: 5
30
31 Thanks for using the program. Have a good day.
```

Here follows one more version of this program, see if you can understand the differences.

```
1 from math import sin, pi
2
3
4 func = lambda x: sin(x)
5 func_str = "sin(x)"
6
7 lower = 0
8 upper = pi / 2
9 num_init = 20
10 acc = float(raw_input("Enter required accuracy: "))
11
12 while True:
13     print ("\n" +
14             "1: Left-point 2: Right-point 3: Mid-point \n" +
15             "4: Trapezium 5: Quit program")
16     method = int(raw_input("Please enter your option: "))
17     if method == 5:
```

```
18     break
19
20     num = num_init
21     error = 2 * acc
22     integral = 0
23     while error > acc:
24         integral_old = integral
25         integral = 0
26         delta_x = (upper - lower) / num
27         for i in range(num):
28             if method == 1:
29                 x_val = lower + i * delta_x
30                 integral += delta_x * func(x_val)
31
32             elif method == 2:
33                 x_val = lower + (i + 1) * delta_x
34                 integral += delta_x * func(x_val)
35
36             elif method == 3:
37                 x_val = lower + i * delta_x + 0.5 * delta_x
38                 integral += delta_x * func(x_val)
39
40             elif method == 4:
41                 x_L = lower + i * delta_x
42                 x_R = x_L + delta_x
43                 integral += 0.5 * delta_x * (func(x_L) + func(x_R))
44
45             error = abs(integral - integral_old)
46             num *= 2
47
48             print ("Integrating %s between " +
49                   "%.3f and %.3f = %.6f") % (func_str, lower,
50                                         upper, integral)
51             print "%d equal intervals needed" % (num / 2)
52
53     print "\nThanks for using the program. Have a good day."
```



#### More Info:

The `break` statement is used to break out of a looping statement (`for` or `while`). Thus in the example above if the user enters the option 5: Python executes the `break` statement on line 18, exits the `while` loop statement (line 12) and continues to execute code after the `while` loop statement (line 53).

## 8.2 Exercises

1. Write a program that computes the numerical integral of a selected function  $f(x)$  between the user specified upper bound  $x_{UB}$  and lower bound  $x_{LB}$  as depicted in Figure 8.4.

The user has to enter the number of intervals  $N$  he requires for the numerical integration scheme. The interval size can then be computed using

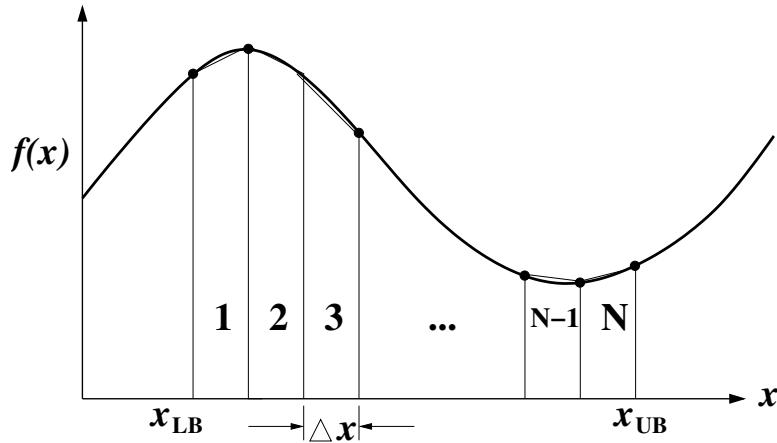


Figure 8.4: Numerical integration using the trapezium method.

$$\Delta x = \frac{x_{UB} - x_{LB}}{N} \quad (8.4)$$

The area of each interval must then be approximated using trapeziums. The area of the first trapezium  $A_1$  is given by

$$A_1 = \Delta x \frac{f(x_{LB}) + f(x_{LB} + \Delta x)}{2} \quad (8.5)$$

The area of the second trapezium  $A_2$  is then given by

$$A_2 = \Delta x \frac{f(x_{LB} + \Delta x) + f(x_{LB} + 2\Delta x)}{2} \quad (8.6)$$

The process is repeated and the last trapezium  $A_N$  is computed by

$$A_N = \Delta x \frac{f(x_{LB} + (N-1)\Delta x) + f(x_{LB} + N\Delta x)}{2} \quad (8.7)$$

You should be able to derive the above formula on your own! If you consider the calculation of the individual trapeziums e.g.  $A_1$  and  $A_2$ , you'll see some of the function evaluations are repeated e.g.  $f(x_{LB} + \Delta x)$  is evaluated in both  $A_1$  and  $A_2$ . Can you come up with a better algorithm where you only evaluate the function at each point once without looking at the notes. Test then your expression and see if it gives you the same answer as the algorithm above.

2. Write a numerical integration program that asks the user to select a numerical integration scheme by typing `left-point` for a left-point scheme, `right-point` for a right-point scheme and `midpoint` for a midpoint scheme. Choose any function  $f(x)$  you like for the program. The program must then ask the user to define the upper bound  $x_{UB}$ , lower bound  $x_{LB}$  and accuracy  $\epsilon$ . The program must then start with  $N = 2$  intervals and numerically integrate  $f(x)$ . The program must then double the  $N$  and recalculate the numerical integral. The program must continue to do so until the difference of two consecutive calculated numerical integrals is less than the user specified accuracy  $\epsilon$ .

The left-point scheme, right-point scheme and midpoint scheme is illustrated in Figure 8.2 and given below:

- For a left-point scheme the interval  $[x_{LB}, x_{UB}]$  is divided in  $n$  equal intervals of width  $\Delta x$  and the height of each rectangle is computed at the left point of each interval. I.e. the first rectangle's height is  $f(x_{LB})$ , the next rectangles height is  $f(x_{LB} + \Delta x)$ , and so on. The last rectangle's height is given by  $f(x_{UB} - \Delta x)$ . This scheme is depicted in Figure 8.2 (i).
- A right point method is where the rectangle heights are computed by  $f(x_{LB} + \Delta x), f(x_{LB} + 2\Delta x), \dots, f(x_{UB})$  as depicted in Figure 8.2 (ii).
- A midpoint method is where the rectangle heights are computed by  $f(x_{LB} + \frac{1}{2}\Delta x), f(x_{LB} + \frac{3}{2}\Delta x), \dots, f(x_{UB} - \frac{1}{2}\Delta x)$  as depicted in Figure 8.2 (iii).

# Chapter 9

## Data containers

So far, we have only dealt with names that contain a single value i.e. scalar names. But how do we represent a quantity that contains more than a single value i.e. a collection of values / information. We could use scalar names, one for each value, but if we have a large collection of values this becomes very difficult to work with and very cumbersome.

### 9.1 Available Data Containers in Python

In Python there are several “data containers” in which we can group a collection of values together under one name. I’ll list the “data containers” (available in Python) below and discuss each one in a different section of this chapter:

- List Object
- Tuple Object
- Dictionary Object

#### 9.1.1 Lists

Lists are the most commonly used “data container” in Python for grouping a collection of values or information together under one name. I’ll start off by showing an example of how you create a list in Python. A list is defined by enclosing the values (or components) in square brackets ([]) as follows:

```
1 a = [1, 12, 32, 87, 5]
```

The values of the list are separated by a comma and then a space (', '), as shown above. If the comma is left out Python will give you a syntax error. You can access each value of the list object by using an indexing integer, enclosed by square brackets ([]), after the list name:

```
1 In [#]: a = [1, 12, 32, 87, 5]
2
3 In [#]: a[0]
4 Out[#]: 1
5
6 In [#]: a[1]
7 Out[#]: 12
8
9 In [#]: a[2]
10 Out[#]: 32
11
12 In [#]:
```



#### Take Note:

All “data containers” start with a base index of zero (0) !!!  
So the very first value / component in a data container is accessed with index = 0  
(See lines 3 – 4 above).  
It is easier just to remember to start counting from zero (0) and not one (1).



#### Take Note:

All “data containers” are indexed using square brackets after the object name.

The values / components of a list object can also be changed using an indexing integer:

```
1 In [#]: a = [1, 12, 32, 87, 5]
2
3 In [#]: a[0] = 100
4
5 In [#]: a
```

```
6 Out[#]: [100, 12, 32, 87, 5]
7
8 In [#]: a[2] += 265
9
10 In [#]: a
11 Out[#]: [100, 12, 297, 87, 5]
12
13 In [#]:
```



### Take Note:

List values can only be changed one at a time (i.e. each individual component at a time). See Section 9.1.2.5 on how multiple values can be changed using a `for` loop statement.

It is worth noting that you have already encountered lists before with the `range` function (Section 4.1). Recall that the `range` function is used to create a list of integers:

```
1 In [#]: a = range(5)
2
3 In [#]: a
4 Out[#]: [0, 1, 2, 3, 4]
5
6 In [#]:
```

## 9.1.2 Memory model

The memory model for a list object is shown in figure 9.1. When a list object is created (either using the `range` statement or manually `a = [0, 1, 2]`), the individual components are created in memory first and each index / element of the list object is then bound (“points”) to each of the objects. Lets try explain each statement in figure 9.1 in more detail:

1. Line 1: The integer objects (0, 1 and 2) are created in memory first. Each index / element of the list object is then bound (“points”) to each of these integer objects. Finally the name `a` is bound to the list object.
2. Line 2: The name `b` is bound to the first integer object 0. It is important to note that the name `b` is not bound to the zero index of the list object but is bound to the integer object referenced by the zero index of the list object.

3. Line 3: The new integer object 5 is first created in memory. The zero index of the list object is then bound (“points”) to this new integer object. The name **b** is still bound to the 0 integer object and the first and second indexes of the list object is still bound to the 1 and 2 integer objects respectively.

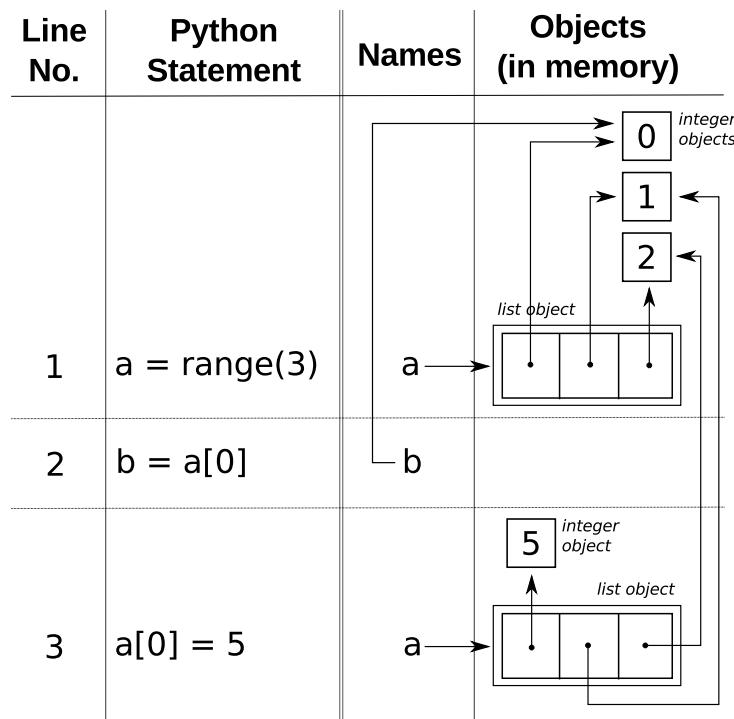


Figure 9.1: Memory model for list objects

### 9.1.2.1 Joining lists – the plus operator (+)

Separate lists can be joined by using the plus symbol (+) between two or more lists:

```

1 In [#]: a = [1, 2, 3, 4, 5]
2
3 In [#]: b = [6, 7, 8, 9, 10]
4
5 In [#]: a + b
6 Out[#]: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
7
8 In [#]: a
9 Out[#]: [1, 2, 3, 4, 5]
10
11 In [#]: b

```

```
12 Out[#]: [6, 7, 8, 9, 10]
13
14 In [#]: a += b
15
16 In [#]: a
17 Out[#]: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
18
19 In [#]:
```

### 9.1.2.2 Indexing a list

I showed above that you can access each values of a list by using an indexing integer, enclosed by square brackets ([]), after the name. However you can also access each values in reverse order by using a negative indexing integer. The last component in a list is accessed using an index of -1, the second last component with -2, the third last component with -3, and so forth:

```
1 In [#]: a = [1, 12, 32, 87, 5]
2
3 In [#]: a[4]
4 Out[#]: 5
5
6 In [#]: a[-1]
7 Out[#]: 5
8
9 In [#]: a[3]
10 Out[#]: 87
11
12 In [#]: a[-2]
13 Out[#]: 87
14
15 In [#]:
```



#### Take Note:

You cannot access, or change the value of, a component / entry that does not exist !!!

The following examples will thus give error messages in Python:

```
1 my_list = [10, 50, 54, 12, 132]
2 print my_list[5]           #IndexError: list index out of range
```

```
3 my_list[4] += my_list[5] #IndexError: list index out of range
4 my_list[5] = 100          #IndexError: list index out of range
```

### 9.1.2.3 The append and insert functions

“Data containers”, like strings, also have certain functions associated with them. Lets consider the example where we want to add a single value to the end of a list.

We could use the plus operator (+) as follows:

```
1 In [#]: a = [1, 2, 3, 4, 5]
2
3 In [#]: a += [6]
4
5 In [#]: a += [7]
6
7 In [#]: a
8 Out[#]: [1, 2, 3, 4, 5, 6, 7]
9
10 In [#]:
```

However, this method, is prone to us making mistakes, because we need to remember to add square brackets around each value being added to the end of the list and is somewhat less readable. A much better way of doing this is to use the `append` function associated with the list object:

```
1 In [#]: a = [1, 2, 3, 4, 5]
2
3 In [#]: a.append(6)
4
5 In [#]: a.append(7)
6
7 In [#]: a
8 Out[#]: [1, 2, 3, 4, 5, 6, 7]
9
10 In [#]:
```

With this example you can see that the code is much easier to understand and less prone to us making mistakes. Similarly to modules, the dot (.) tells Python that the

append function resides in the list object.

**Take Note:**

The second method used here (using the `.append()` function) is the correct way of adding a single value to the end of a list and the first method is for explanation purposes only.

The `insert` function, associated with a list object, can be used to add a single value into the list at a specified index:

```
1 In [#]: a = [1, 2, 3, 4, 5]
2
3 In [#]: a.insert(2, 100)
4
5 In [#]: a
6 Out[#]: [1, 2, 100, 3, 4, 5]
7
8 In [#]: a.insert(0, 800)
9
10 In [#]: a
11 Out[#]: [800, 1, 2, 100, 3, 4, 5]
12
13 In [#]:
```

You use the `insert` function by first specifying the index (where you want to insert the value) and then the value (what you want to insert). The rest of the entries / components are shifted to the right.

**Take Note:**

All data containers start with a base index of zero (0) !!!  
Thus in the example above `a.insert(2, 100)` (line 3) will insert the value 100 into the third position of the list.  
It is easier just to remember to start counting from zero (0) and not one (1).

In the following example I will show you how to create an empty list object and, using a `for` loop statement, we will `append` values to it. Say, for example, we had a function that we evaluate a few times and we want to store the results for each evaluation in a list object:

```
1 func = lambda x: (x ** 2) / 16
2
3 x_val = 1
4 results = []
5 for i in range(5):
6     x_val += 3**i
7     func_val = func(float(x_val))
8     results.append(func_val)
9
10 print "f(x) = ", results
```

I create an empty list object (line 4) by only specifying the square brackets ([]). In line 7 I evaluate the function value, for `x_val`, and temporarily store that in the variable `func_val`. In line 8 I then append this function value to my list of results `results`. The output of this example is the following:

```
1 f(x) = [0.25, 1.5625, 12.25, 105.0625, 930.25]
```



#### Take Note:

The `float(x_val)` in the above example (line 7) is used to convert `x_val` from an integer to a floating point number (See section 3.1.4).

Alternatively line 1 could have been changed to the following to ensure that a floating point number is returned from the `lambda` function:

```
1 func = lambda x: (x ** 2) / 16.0
```



#### More Info:

Type `help(list)` in the *IPython Console* to see a list of available functions associated to a list container (scroll past any function with “`_`” in front or behind the name, these functions are outside the scope of this course).

Alternatively you can type `list.` (note the dot) and hit the `tab` key, IPython will then print a list of functions associated to the list container to the screen.

### 9.1.2.4 Looping through a list

We can loop through a list in the same way we did with the `range` function (Section 4.2):

```
1 my_list = [10, 50, 54, 12, 132]
2 cnt = 0
3 for val in my_list:
4     print "Entry %d in 'my_list' is %d" % (cnt, val)
5     cnt += 1
```

This produces the following output:

```
1 Entry 0 in 'my_list' is 10
2 Entry 1 in 'my_list' is 50
3 Entry 2 in 'my_list' is 54
4 Entry 3 in 'my_list' is 12
5 Entry 4 in 'my_list' is 132
```

You can see from this example that the name `val` is bound to each consecutive integer object (one after the other) of `my_list` for each loop.



#### Take Note:

It is important to understand the `val` (in the example above) is bound to each consecutive object of `my_list` for each loop and it is NOT an indexing integer. In this example `cnt` has been added to act as the indexing integer.

### 9.1.2.5 Disadvantage of using lists

The disadvantage of using lists is that you cannot do computations on all values / components at once. You have to use a `for` loop statement and do the computations on each component (one at a time). Lets consider the example where we want to multiply each component in our list by 2. We need to multiple each value, in our list, by 2 using a `for` loop statement:

```
1 my_list = [10, 50, 54, 12, 132]
2 for index in range(len(my_list)):
3     my_list[index] *= 2
4 print my_list
```

**Take Note:**

The `len()` function returns the length (number of entries) in any data container. Note that `len()` starts counting from 1:

```
1 In [#]: my_list = [10, 50, 54, 12, 132]
2
3 In [#]: len(my_list)
4 Out [#]: 5
5
6 In [#]:
```

This is so that the `len()` function can easily be used with the `range()` statement to create a list of indexes for accessing the components of any data container (as done in the example above):

```
1 In [#]: my_list = [10, 50, 54, 12, 132]
2
3 In [#]: len(my_list)
4 Out [#]: 5
5
6 In [#]: range(5)
7 Out [#]: [0, 1, 2, 3, 4]
8
9 In [#]:
```

**Take Note:**

If we tried to multiply the entire list by 2 we would get the following:

```
1 In [#]: my_list = [10, 50, 54, 12, 132]
2
3 In [#]: my_list *= 2
4
```

```
5 In [#]: my_list
6 Out[#]: [10, 50, 54, 12, 132, 10, 50, 54, 12, 132]
7
8 In [#]:
```

Not exactly what we want. The multiplication sign (\*) causes the components in the list to be repeated.

#### 9.1.2.6 Working with lists – examples

Consider the example shown earlier, where we want to multiple each component in our list by 2:

```
1 my_list = [10, 50, 54, 12, 132]
2 for index in range(len(my_list)):
3     my_list[index] *= 2
4 print my_list
```

In this example above we use the `for` loop statement (along with the `range()` and `len()` statements) to generate the index number used to access each component in `my_list` and multiple that component by 2.

We could also use the `for` loop statement to simple loop through each component in our list, as follows:

```
1 my_list = [10, 50, 54, 12, 132]
2 ind = 0
3 for val in my_list:
4     my_list[ind] = val * 2
5     ind += 1
6 print my_list
```

In this example above the name `val` (in the `for` loop statement) is bound to each consecutive object of `my_list` for each loop. I thus need to create an index name `ind` (line 2) that gets updated for each loop (line 5). `ind` is the indexing integer used to access the individual components of `my_list` and `val` is used to calculate the new value for that component (line 4).

Python has a function (`enumerate()`) that incorporates the above functionality into one line in the `for` loop statement:

```
1 my_list = [10, 50, 54, 12, 132]
2 for index, val in enumerate(my_list):
3     my_list[index] = val * 2
4 print my_list
```

In this above example the statement `enumerate(my_list)` returns both the index number and the consecutive object of `my_list` in the `for` loop statement.



#### Take Note:

Note that two names are specified in the `for` loop statement in the example above.

- one name (`index`) for the index number of `my_list`
- and another name for the consecutive object of `my_list`

If only one name is given then both the index number and the consecutive object of `my_list` will be stored in a tuple and be bound by that name:

```
1 my_list = [10, 50, 54, 12, 132]
2 for var in enumerate(my_list):
3     print var
```

Gives the output:

```
1 (0, 10)
2 (1, 50)
3 (2, 54)
4 (3, 12)
5 (4, 132)
```



#### More Info:

List objects can contain any type of object (integer, float, string, boolean) and these types of objects can also be jumbled up inside one list, making Python extremely flexible:

```
1 my_list = [10, "logan", 54.123123, True]
```

**Take Note:**

It is important to note that because a data container can contain different object types, the word “value” in this chapter not be mistaken for only an integer or floating point number. The word “value” can refer to either an integer value, float value, string value or boolean value.

### 9.1.3 Tuples

A tuple object is almost exactly the same as a list object and it is defined by enclosing the values (or components) in round brackets “()” as follows:

```
1 a = (1, 12, 32, 87, 5)
```

The following two differences separate a tuple object from a list object:

1. The values / components cannot be changed after the tuple is created:

```
1 a = (1, 12, 32, 87, 5)
2 a[0] += 10      # TypeError: 'tuple' object does not
3                      # support item assignment
```

This can be useful for when you have a collection of values that wont change or you don't want them to accidentally be changed somewhere in your program.

2. A tuple does not have the `append` and `insert` functions associated to it (it does however have the same behaviour as a list when using the plus sign (+) operator):

```
1 a = (1, 12, 32, 87, 5)
2
3 a.append(10)    # AttributeError: 'tuple' object has no
4                      # attribute 'append'
5
6 a.insert(0, 10) # AttributeError: 'tuple' object has no
7                      # attribute 'insert'
```

```
8
9 a += (132, 23) # this is allowed
```

Everything else (behaviour, indexing, looping through, etc.) is the same as a list object.

### 9.1.3.1 Disadvantages of using a tuple

The one disadvantage of using a tuple object is when trying to create a tuple object with only one component. Consider the following example:

```
1 In [#]: a = (10)
2
3 In [#]: a
4 Out[#]: 10
5
6 In [#]: a = (10,)
7
8 In [#]: a
9 Out[#]: (10,)
10
11 In [#]:
```

You would expect to simple enclosing the single value in round brackets as shown in line 1 above. However this is seen by Python as a separation of mathematical operations and not the creation of a tuple object. To create a tuple object with just one component you have to add a comma (,) after the single value entry, as shown in line 6.



#### More Info:

Type `help(tuple)` in the *IPython Console* to see a list of available functions associated to a tuple container (scroll past any function with “\_” in front or behind the name, these functions are outside the scope of this course).

Alternatively you can type `tuple.` (note the dot) and hit the `tab` key, IPython will then print a list of functions associated to the tuple object to the screen.

### 9.1.3.2 Working with tuples – examples

Lets say that we want to evaluate a second order polynomial function ( $f(x) = ax^2+bx+c$ ). The simplest solution would be to add the coefficients into the function when creating it (lines 3–5 below), however for the sake of explanation lets store these coefficients in a tuple object.

```

1 coeffs = (10, 5, 50)
2
3 func = lambda x: (coeffs[0] * (x ** 2) +
4                     coeffs[1] * x +
5                     coeffs[2])
6
7 x_values = [0.3, 0.9, 1.2, 0.1]
8 f_values = []
9 for val in x_values:
10     f_values.append(func(val))
11 print f_values

```

Because the coefficients for the polynomial function will remain unchanged in our program it makes sense to store them in a tuple object (line 1) and then use the tuple object in the equation for the function (line 3–5). This is of course a very simple example, but what if we also wanted to evaluate the function derivative and add a few more degrees to the polynomial function:

```

1 coeffs = (0, 3, 10, 5, 50)
2
3 func = lambda x: ( coeffs[0] * (x ** 4) +
4                     coeffs[1] * (x ** 3) +
5                     coeffs[2] * (x ** 2) +
6                     coeffs[3] * (x ** 1) +
7                     coeffs[4] * (x ** 0) )
8
9 deriv = lambda x: ( 4 * coeffs[0] * (x ** 3) +
10                      3 * coeffs[1] * (x ** 2) +
11                      2 * coeffs[2] * (x ** 1) +
12                      coeffs[3] * (x ** 0) )
13
14 x_values = [0.3, 0.9, 1.2, 0.1]
15 f_values = []
16 deriv_values = []

```

```
17 for val in x_values:  
18     f_values.append(func(val))  
19     deriv_values.append(deriv(val))  
20 print f_values  
21 print deriv_values
```

You can now see from this example how a tuple object starts becoming useful, if we later wanted to change the values of the coefficients and re-run the program we would only have to change one line of code in place of several lines (and possibly miss a few).



#### More Info:

As with list objects, tuple objects can contain any type of object (integer, float, string, boolean) and these types of objects can also be jumbled up inside one tuple object:

```
1 my_tuple = (10, "logan", 54.123123, True)
```

#### 9.1.4 Dictionaries

A dictionary is defined by enclosing the values (or components) in curly brackets. Each component is first given a keyword, followed by a double colon(:) and then followed by the component:

```
1 a = {"score": 12,  
2      "age": 45,  
3      "year": 2013}
```

Dictionaries allow you to associate values or information with specified keywords. So in the example above the value 12 is associated with the keyword "score", 45 with the keyword "age", etc. You can then access these values / components using the keywords. This is done by enclosing the keyword in square brackets ([])) after the name:

```
1 In [#]: a = {"score": 12,  
2      ....:      "age": 24,  
3      ....:      "year": 2013}  
4
```

```
5 In [#]: a
6 Out[#]: {'age': 24, 'score': 12, 'year': 2013}
7
8 In [#]: a["age"]
9 Out[#]: 24
10
11 In [#]: a["year"]
12 Out[#]: 2013
13
14 In [#]:
```

A dictionary object is often used for storing a collection of more “real” or “physical” information about a certain aspect or category (e.g. the make, model and year of a car; or the name, surname, age and email address of a student; etc.) The dictionary object allows for easy grouping of this information and giving each component an easy to associate keyword.



#### Take Note:

The values or components of a dictionary object can only be accessed by using the keyword, as shown above, the following will give an error in Python:

```
1 In [#]: a = {"score": 12,
2 .....: "age": 24,
3 .....: "year": 2013}
4
5 In [#]: a[0] # KeyError: 0
6
7 In [#]: a["name"] # KeyError: 'name'
8
9 In [#]: a["ScoRe"] # KeyError: 'ScoRe'
10
11 In [#]:
```

Keywords are case sensitive !!!

Dictionary components can also be changed using the keyword:

```
1 In [#]: a = {"make": "Toyota",
2 .....: "model": "Corolla",
```

```
3     ....:         "year": 2009}

4
5 In [#]: a
6 Out[#]: {'make': 'Toyota', 'model': 'Corolla', 'year': 2009}

7
8 In [#]: a["year"] = 2006

9
10 In [#]: a
11 Out[#]: {'make': 'Toyota', 'model': 'Corolla', 'year': 2006}

12
13 In [#]: a["year"] -= 5

14
15 In [#]: a
16 Out[#]: {'make': 'Toyota', 'model': 'Corolla', 'year': 2001}
```

#### 9.1.4.1 Adding to a dictionary

You can add new “keyword: value” entries to a dictionary object by simply indexing it with a new keyword and a new value:

```
1 In [#]: a = {"make": "Toyota",
2     ....:         "model": "Corolla",
3     ....:         "year": 2009}

4
5 In [#]: a
6 Out[#]: {'make': 'Toyota', 'model': 'Corolla', 'year': 2009}

7
8 In [#]: a["colour"] = "red"

9
10 In [#]: a
11 Out[#]:
12 {'colour': 'red',
13 'make': 'Toyota',
14 'model': 'Corolla',
15 'year': 2001}

16
17 In [#]: a["code"] = "133L"

18
19 In [#]: a
20 Out[#]:
21 {'code': '133L',
```

```
22 'colour': 'red',
23 'make': 'Toyota',
24 'model': 'Corolla',
25 'year': 2001}
26
27 In [#]:
```



#### Take Note:

Dictionary objects do not support the plus operator (+) and thus cannot be joined in this fashion, as tuple or list objects could !!!

#### 9.1.4.2 Looping through a dictionary

Dictionary objects behave very differently to tuple and list objects when it come to looping through them. Looping through a dictionary will thus be outside the scope of this course, however please feel free to “Google” this topic if you are interested in knowing how it is done. There is a lot of information and examples on the web covering this.

#### 9.1.4.3 Disadvantage of using dictionaries

The only disadvantage I am going to mention is that of misspelling the keyword when trying to change the value of a certain entry.

```
1 In [#]: a = {"make": "Toyota",
2 .....: "model": "Corolla",
3 .....: "year": 2009}
4
5 In [#]: a
6 Out[#]: {'make': 'Toyota', 'model': 'Corolla', 'year': 2009}
7
8 In [#]: a["Year"] = 2006
9
10 In [#]: a
11 Out[#]:
12 {'Year': 2006,
13   'make': 'Toyota',
14   'model': 'Corolla',
15   'year': 2009}
```

16  
17 In [#] :

As you can see from this example there are now 2 “year” keywords, one with a lower case “y” and one with an upper case “Y”, take care in making sure you type the keyword exactly the same each time.

#### 9.1.4.4 Working with dictionaries – examples

By now you should be able to figure out the logic of the following program:

```
1 student = {"name": "John",
2             "surname": "Doe",
3             "mark": 50}
4
5 if student["mark"] >= 50:
6     template_info = (student["surname"],
7                       student["name"],
8                       "passed")
9 else:
10    template_info = (student["surname"],
11                      student["name"],
12                      "failed")
13
14 print "%s, %s %s." % template_info
```

You should now notice that the string formatting operator (%) requires a tuple object for the information being inserted into the string template, when multiple pieces of information are being inserted.



#### More Info:

Type `help(dict)` in the *IPython Console* to see a list of available functions associated to a dictionary object (scroll past any function with “\_” in front or behind the name, these functions are outside the scope of this course).

Alternatively you can type `dict.` (note the dot) and hit the `tab` key, IPython will then print a list of functions associated to the dictionary object to the screen.



#### More Info:

As with list objects, dictionary objects can contain any type of object (integer, float, string, boolean) and these types of objects can also be jumbled up inside one dictionary object. This also holds for the keywords (the following is thus allowed):

```
1 my_dict = {1: True
2     2: "yes"
3     "a": 12.445}
```

### 9.1.5 Summary

So far I have shown how flexible Python can be with three different data containers, each of which, not restricted to only one object type. This allows you, as the programmer, to choose how you want to group information together in order to make your program easier to use and easier to understand. There is one additional data container available to Python, which will be covered extensively in Chapter 10. This additional data container is a `numpy` array which is generally used for vector and matrix algebra.

In the following sections I will go through a few examples showing the use of these data containers. It should be even clearer now, with the vast flexibility of Python, that there is no one solution for a program and as the programmer you will need to fully understand the problem being solved and decide how you want to solve it in Python.



#### More Info:

As you saw in section 3.5 and related examples in the following chapters; object types can be converted from one type to another. Data containers can also be converted from one type to another using the following statements:

1. `tuple()` - Convert to a tuple container
2. `list()` - Convert to a list container
3. `dict()` - Convert to a dictionary container

More information on each of these will be given as and when they are used in the examples in these notes.

## 9.2 Revisiting previous programs

I'll give a few more examples of the use of data containers. I'll first use lists to compute a Taylor series approximation. Then I'll illustrate how lists can be used to perform numerical integration.

### 9.2.1 Taylor series using lists

I discussed Taylor series analyses in Chapter 4 (Section 4.6) and Chapter 5 (Section 5.4) of the lecture notes. I used both `for` loop statements and `while` loop statements to perform the required computations. Revisit these sections before you continue.

You should have noticed that when we computed a Taylor series approximation at the start of the year, we only used scalar variables. We kept on adding new terms in the series until the series converged.

Let's compute a Taylor series approximation again, but now making use of two list. In the first list, called `terms`, I will save the Taylor series terms i.e. the first term in the series is saved in the first component, the second term in the series is saved in the second component, and so on. In the second list, called `taylor_approx`, I will save the current value of the Taylor series approximation. The first component will contain the first term, the second component will contain the sum of the first two terms, and so on. We proceed like this and save the `n` term Taylor series approximation in the `n`-th component of the list `taylor_approx` (where  $n = 0, 1, 2, \dots$ ). When the complete lists are displayed, we should clearly see how the components of the list `terms` become smaller, while the components of the list `taylor_approx` should converge to the analytical result.

As an example, lets compute the Taylor series approximation to the exponential function, which is given by

$$e^x = 1 + x + \frac{1}{2}x^2 + \frac{1}{6}x^3 + \dots = \sum_{k=0}^{\infty} \frac{x^k}{k!} \quad (9.1)$$

Here's a program that uses a `while` loop statement to compute the Taylor series approximation to  $e^x$  within a specified accuracy.

```

1 from math import factorial
2
3 # Get value for x at which to compute exp(x)
4
```

```

5 x_val = float(raw_input("Compute exp(x) at x = "))
6 # Get required accuracy
7 acc = float(raw_input("Enter the numerical accuracy: "))
8
9 terms = []
10 taylor_approx = [0.0]
11 # Set term counter to 0
12 cnt = 0
13 # Set initial Error > acc to make sure while loop is entered
14 error = 1
15 while error > acc:
16     # Compute next term in series
17     terms.append(x_val**cnt / factorial(cnt))
18     # Next component in list equal to previous component
19     # plus new term
20     taylor_approx.append(taylor_approx[cnt] + terms[cnt])
21     # Error = difference between consecutive components
22     error = abs(taylor_approx[cnt+1] - taylor_approx[cnt])
23     # Increment the counter
24     cnt += 1
25
26 print "Terms: \t Taylor Approx:"
27 for i in range(len(terms)):
28     print "%.8f \t %.8f \n" % (terms[i], taylor_approx[i]),

```

Here follows the output of the program listed above. You can clearly see how the Taylor series converges to  $e = 2.718281828459\dots$  as the number of terms in the series increases.

```

1 Compute exp(x) at x = 1.0
2 Enter the numerical accuracy: 1e-7
3 Terms:      Taylor Approx:
4 1.00000000  0.00000000
5 1.00000000  1.00000000
6 1.00000000  2.00000000
7 0.50000000  2.50000000
8 0.16666667  2.66666667
9 0.04166667  2.70833333
10 0.00833333  2.71666667
11 0.00138889  2.71805556
12 0.00019841  2.71825397
13 0.00002480  2.71827877

```

```

14 0.00000276 2.71828153
15 0.00000028 2.71828180
16 0.00000003 2.71828183

```

Modify the above program to compute the Taylor series approximation to  $\pi$ . Be careful not to specify a very small accuracy. You'll see that the  $\pi$  Taylor series converges very slowly, so you need a large number of terms for an accurate approximation.

### 9.2.2 Numerical integration using lists

Let's revisit the numerical integration program developed in Section 8. In that section, the name `integral_old` is used to have the previous value of the numerical integral available after the new value is computed. Now I propose that we not only save the most recent value of the numerical integral, but that we save the numerical integral for every case analysed. We can save all these values in a single list.

I'm going to create two lists. In the first list, called `num_int`, I'll save the number of intervals. As the number of intervals change, I'll save the new number of intervals in the next component of the list `num_int`. The second list, called `integral`, will contain the numerical integrals associated with the number of intervals in the list `num_int`.

Here's the modified numerical integration program:

```

1 from math import sin, pi
2
3
4 func = lambda x: sin(x)
5 func_str = "sin(x)"
6
7 lower = 0
8 upper = pi / 2
9 acc = float(raw_input("Enter required accuracy: "))
10
11 # Display integration options
12 print ("\n" +
13     "1: Left-point 2: Right-point 3: Mid-point \n" +
14     "4: Trapezium 5: Quit program")
15 method = input("Please enter your option: ")
16
17 # My program repeats until the user inputs option 5

```

```
18 while method != 5:
19     # Define initial Error greater as acc,
20     # to enter while loop
21     error = 1
22     #Since the number of intervals change during program
23     # execution, reset N to the initial number of intervals
24     num_int = [0, 10]
25     num = num_int[-1]
26     # Set Int to zero to have a value the first time
27     # the while loop is executed
28     integral = [0.0]
29
30     # Start while loop that checks if consecutive values
31     # converged
32     while error > acc:
33         # Make a copy of the previous integral just before the
34         # program section starts that computes the new value
35         integral_old = integral[-1]
36         # Initialize the integral at zero
37         integral.append(0.0)
38         # Compute the interval size Delta_x
39         delta_x = (upper - lower) / num
40         if method == 1:      # left-point method
41             # For loop to compute area of each rectangle and
42             # add it to integral
43             for i in range(num):
44                 x = lower + i * delta_x
45                 integral[-1] += delta_x * func(x)
46
47         elif method == 2:    # right-point method
48             for i in range(num):
49                 x = lower + (i + 1) * delta_x
50                 integral[-1] += delta_x * func(x)
51
52         elif method == 3:    # mid-point method
53             for i in range(num):
54                 x = lower + i * delta_x + 0.5 * delta_x
55                 integral[-1] += delta_x * func(x)
56
57         elif method == 4:    # trapezium method
58             for i in range(num):
59                 x_L = lower + i * delta_x
60                 x_R = x_L + delta_x
61                 integral[-1] += ( 0.5 * delta_x *
```

```

62                                     (func(x_L) + func(x_R)) )
63
64     # Compute Error i.e. difference between consecutive
65     # integrals
66     error = abs(integral[-1] - integral_old)
67     # Double the number of intervals
68     num *= 2
69     num_int.append(num)
70
71     # Display answer
72     print "Integrating %s between %.3f and %.3f:" % (func_str,
73                                         lower,
74                                         upper)
75     print "Intervals \t Integral"
76     for i in range(len(integral)):
77         print "%d \t %.8f" % (num_int[i], integral[i])
78
79     # Display options again
80     print ("\n" +
81             "1: Left-point 2: Right-point 3: Mid-point \n" +
82             "4: Trapezium 5: Quit program")
83     method = input("Please enter your option: ")
84
85     # Outside while loop, user entered option 5 = Quit
86     # Displays a friendly message.
87     print "\nThanks for using the program. Have a good day."

```

Very few changes were made compared to the previous version. Since the number of intervals  $N$  is now a list, I need to specify which component of  $N$  I need e.g. program lines 25, 68 and 69. Similarly the numerical integrals are now stored in the list `integral`, so I need to refer to a specific component of `integral`, e.g. program lines 45, 50 and 61.

I've also decided to change the output. Now I display the first `k` components of the lists `num_int` and `integral`. Program lines 75–77 displays these lists. I make use of a `for` loop statement to `print` each component of the lists `num_int` and `integral` to the screen.



### More Info:

The tab (`\t`) character is used to add white space in its place when printed to the screen.

Here follows the output from the program. I only executed option 2 (right-point) and option 4 (trapezium) before I quit the program (option 5). Note the tabular output, where

we now clearly see the progress towards the analytical result as the number of intervals is increased.

```
1 Enter required accuracy: 0.0001
2
3 1: Left-point 2: Right-point 3: Mid-point
4 4: Trapezium 5: Quit program
5 Please enter your option: 2
6 Intergrating sin(x) between 0.000 and 1.571:
7 Intervals Integral
8 0 0.00000000
9 10 1.07648280
10 20 1.03875581
11 40 1.01950644
12 80 1.00978535
13 160 1.00490071
14 320 1.00245236
15 640 1.00122668
16 1280 1.00061347
17 2560 1.00030676
18 5120 1.00015339
19 10240 1.00007670
20
21 1: Left-point 2: Right-point 3: Mid-point
22 4: Trapezium 5: Quit program
23 Please enter your option: 4
24 Intergrating sin(x) between 0.000 and 1.571:
25 Intervals Integral
26 0 0.00000000
27 10 0.99794299
28 20 0.99948591
29 40 0.99987149
30 80 0.99996787
31
32 1: Left-point 2: Right-point 3: Mid-point
33 4: Trapezium 5: Quit program
34 Please enter your option: 5
35
36 Thanks for using the program. Have a good day.
```

### 9.3 Sorting algorithms

Let's further develop our programming skills by trying to write a program that can sort the components of a list in ascending order (small to large). In order to develop a sorting algorithm, you have to think of a simple operation that a computer can perform that will enable a list be sorted. If required, this operation can be repeated a large number of times (which will be the case).

The operation I'm referring to is to compare two numbers. If the number pair is in the correct order (1st value smaller than the 2nd), then move to a different number pair. If the number pair is not in the correct order, swap the 2 numbers. This simple operation is sufficient to order a list.

I can think of a few sorting algorithms. I'll start off with a method that is easy to explain and implement, but not very efficient. Let's compare the first component of the list with all the other components, one by one (starting at component 2, ending at the last component), i.e. compare component 1 with 2, then component 1 with 3 and so on. Whenever the 1st component is larger than the component it is compared to, we swap the components. If not, we simply move to the next component of the list. When we reach the end of the list, the smallest number will be in component 1. We start at the top again but now we compare the 2nd component of the list with all the components from 3 to the end (It's no use to start at 1 again, it already contains the smallest number). We repeat this process till we compare the last two numbers. This process is implemented below. We need two nested `for` loops: the first loop provides the index of the 1st number in the number pair, while the second loop provides the index of the 2nd number in the number pair.

```
1 # Create the list a
2 my_list = [5, 4, 3, 2, 1]
3 # Get how many components the list has
4 num = len(my_list)
5 # First for loop: position of first number in pair
6 for i in range(num-1):
7     # Second for loop: position of second number in pair
8     for j in range(i+1, num):
9         # Check if number pair is in ascending order
10        if my_list[j] < my_list[i]:
11            # Swap numbers if they are in the wrong order
12            copy_comp = my_list[i]
13            my_list[i] = my_list[j]
14            my_list[j] = copy_comp
15 # Display the list each time the outer for loop is
```

```
16     # complete  
17     print "list = ", my_list, "\n",
```

In program line 2 I define the list `my_list`. I've decided to use a list that is sorted from large to small. Such a list will really test if my program can sort in ascending order. Program line 4 gets the number of components of the list. The first `for` loop statement is coded in program line 6: do you see that the index of the first number in my number pair starts at 0 and ends at `n-1` (one from the end). The second `for` loop statement is coded in program line 8. The 1st number in the number pair is given by index `i`. I now have to compare this number with the remaining numbers below position `i` in the list. That is why the second index `j` starts at `i+1` (the very next component in the list) and ends at `n`, the last entry of the list.

In program line 10 I check if the number pair is in the correct order. If the `i`-th component (1st number of the pair) is greater than the `j`-th component, I swap the pair of numbers. This is done by first making a copy of the `i`-th component (program line 12), then assigning the old `j`-th component to the new `i`-th component (program line 13). Finally the copy of the old `i`-th component is assigned to the new `j`-th component (program line 14). Program line 17 simply displays the list every time the outer `for` loop is complete. This will provide output that will illustrate how the program proceeds to sort the list.

Here follows the output of the above program:

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

As you can see, the program correctly sorts the descending list into ascending order. After the `for` loop is executed once, the smallest number occupies the first list position. After the `for` loop is executed again, the first 2 numbers are in their correct positions. This process repeats until all the numbers are correct after the `for` loop has been executed 4 times.

Another version of this program is given below to illustrate the use of the `continue` statement. The `continue` statement is used to tell Python to continue with a looping statement (`for` or `while`) and skip any code after the `continue` statement is encountered. So in the example below if the `j`-th component is larger than the `i`-th component (i.e. in the correct order) then the `continue` statement, on line 11, is executed and lines 12–15 are skipped. The `for` loop statement (line 8) then continues by incrementing `j`.

```

1 # Create the list a
2 my_list = [5, 4, 3, 2, 1]
3 # Get how many components the list has
4 num = len(my_list)
5 # First for loop: position of first number in pair
6 for i in range(num-1):
7     # Second for loop: position of second number in pair
8     for j in range(i+1, num):
9         # Check if number pair is in ascending order
10        if my_list[j] > my_list[i]:
11            continue
12        # Swap numbers if they are in the wrong order
13        copy_comp = my_list[i]
14        my_list[i] = my_list[j]
15        my_list[j] = copy_comp
16        # Display the list each time the outer for loop is
17        # complete
18        print "list = ", my_list, "\n",

```

### 9.3.1 Bubble sort algorithm

Let's also look at an alternative sorting algorithm. It will help to illustrate that many solutions exist to the same problem. The bubble sorting algorithm also checks if a number pair is in the correct order. The only difference with the algorithm above is that the bubble sort algorithm always compares a component with the very next component in the list, i.e. compare component 1 with 2, then component 2 with 3, then component 3 with 4 and so on. After the last number pair has been compared, the largest number will occupy the last position in the list. Now we have to start at the top again and compare component 1 with 2, 2 with 3 and so on. This time we stop 1 pair from the bottom, because we already know the largest number is in the correct position. This process is repeated till only components 1 and 2 are compared. Here's my implementation of the bubble sort algorithm:

```

1 # Create the list my_list
2 my_list = [5, 4, 3, 2, 1]
3 # Get how many components the list has
4 num = len(my_list)
5 # First for loop: how many times the list
6 # has to be scanned from top to bottom
7 for i in range(num-1):

```

```

8   # The index to the first number of the pair
9   for j in range(num-i-1):
10      # Compare the j-th and j+1-th list components
11      if my_list[j] > my_list[j+1]:
12          # Swap component j and j+1 if in wrong order
13          copy_comp = my_list[j]
14          my_list[j] = my_list[j+1]
15          my_list[j+1] = copy_comp
16      # Display the list each time the outer for loop is
17      # complete
18      print "list = ", my_list, "\n",

```

The output of the bubble sort algorithm, using the same 5 component list as before, follows:

```

1 list = [4, 3, 2, 1, 5]
2 list = [3, 2, 1, 4, 5]
3 list = [2, 1, 3, 4, 5]
4 list = [1, 2, 3, 4, 5]

```

As you can see the largest entry occupies the last list position after the outer `for` loop is executed once. The largest two entries are in their correct positions after the outer `for` loop is executed again. This process repeats until all entries are correct after outer the `for` loop is executed four times.

I'll now run the bubble sort program again, but this time I replace program lines 1–2 with

```

1 from random import randint
2
3
4 # Create the list my_list
5 my_list = []
6 for i in range(8):
7     my_list.append(randint(1, 10))

```

which will create a random 8 component list `my_list`. I include the output of the bubble sort algorithm for one of the cases I ran after the above change:

```
1 list = [1, 5, 5, 5, 3, 1, 3, 8]
2 list = [1, 5, 5, 3, 1, 3, 5, 8]
3 list = [1, 5, 3, 1, 3, 5, 5, 8]
4 list = [1, 3, 1, 3, 5, 5, 5, 8]
5 list = [1, 1, 3, 3, 5, 5, 5, 8]
6 list = [1, 1, 3, 3, 5, 5, 5, 8]
7 list = [1, 1, 3, 3, 5, 5, 5, 8]
```

The algorithm clearly works. For the 8 component list above, the outer `for` loop is executed 7 times. However, the output for this particular example shows that the list is in the correct order after only 4 outer `for` loop executions. So how can we improve the bubble sort algorithm to stop once the list is in the correct order?

The answer is quite simple. We have to keep track of number pair swapping. If we find that we start at the top of the list and scan all the way to the bottom and never find a number pair in the wrong order, we know the list is sorted and the algorithm can stop. This type of logic requires a `while` loop.

```
1 from random import randint
2
3
4 # Set how many components the list has
5 num = 10
6 # Create the list my_list
7 my_list = []
8 for k in range(num):
9     my_list.append(randint(1, 100))
10
11 # Start index at zero
12 ind = 0
13
14 print "list = ", my_list, "\n",
15 # Set is_sorted to false i.e. assume list is not ordered
16 is_sorted = False
17 # Start while loop: repeat loop while index i<n-1
18 # or the is_sorted not equal to false
19 while (not is_sorted) and (ind < num-1):
20     # Set is_sorted to true i.e. assume list is ordered
21     is_sorted = True
22     # The index to the first number of the pair
23     for j in range(num-ind-1):
```

```

24      # Compare the j-th and j+1-th list components
25      if my_list[j] > my_list[j+1]:
26          # Swap component j and j+1 if in wrong order
27          copy_comp = my_list[j]
28          my_list[j] = my_list[j+1]
29          my_list[j+1] = copy_comp
30          # Set is_sorted to false to indicate list is not
31          # ordered yet
32          is_sorted = False
33      # Increment index with 1
34      ind += 1
35      # Display the number of times the while loop is
36      # executed
37      print "list = ", my_list,

```

I'll only explain the program lines that are new. The index `ind` is set equal to zero in program line 12 and is incremented by one inside the `while` loop in program line 34. This replaces the `for` loop that was repeated a predetermined number of times (`n-1`). I also use a new name called `is_sorted` to indicate whether the list is sorted or not. I set `is_sorted` equal to `True` in program line 16, before the `while` loop starts. The `while` loop condition in program line 19 states that the sorting loop is repeated as long as the name `is_sorted` equals `False` and the index `ind` is less than `n-1` (We know that the list will be sorted once `i = n-1`). As soon as the loop starts, I set the name `is_sorted` equal to `True` in program line 21 i.e. I assume that the list is already in the correct order. Should I find a pair of numbers that is not in the correct order, I change the value of the name `is_sorted` to `False` (program line 32). The rest of the program is unchanged.

Here follows an example of the modified bubble sort algorithm. I chose a case in which the original list `my_list` was almost sorted already, so the `while` loop is repeated only 5 times for a list of length 8. As you can see the program now terminates as soon as the list is scanned from top to bottom without a number pair swapping. In fact, the `while` loop is repeated only once for any list that is already sorted in ascending order.

```

1  list = [8, 4, 4, 8, 7, 6, 4, 8]
2  list = [4, 4, 8, 7, 6, 4, 8, 8]
3  list = [4, 4, 7, 6, 4, 8, 8, 8]
4  list = [4, 4, 6, 4, 7, 8, 8, 8]
5  list = [4, 4, 4, 6, 7, 8, 8, 8]
6  list = [4, 4, 4, 6, 7, 8, 8, 8]

```

### 9.3.2 Containers inside containers

So far I have shown how flexible Python is with three different data containers, each of which, not restricted to only one object type. Python can be even more flexible by being able to have data containers inside data containers. The most common of which, for example, is a list object of list objects:

```
1 my_list = [ [1, 2, 3],  
2             [4, 6, 1],  
3             [0, 1, 6] ]
```

From the example above you can see I have created a list of three lists. The lists inside `my_list` are separated with a comma (,). The individual components of the list can be indexed in the same manner as in section 9.1.1. `my_list[0]` returns a reference to the first list object ([1, 2, 3]) and thus `my_list[0][0]` returns a reference to the first integer object (1) in the first list object ([1, 2, 3]). Consider the following examples:

```
1 In [#]: a = [[0, 1, 2],  
2 ...:      [1, 2, 3]]  
3  
4 In [#]: a  
5 Out[#]: [[0, 1, 2], [1, 2, 3]]  
6  
7 In [#]: a[0]  
8 Out[#]: [0, 1, 2]  
9  
10 In [#]: a[0][0]  
11 Out[#]: 0  
12  
13 In [#]: a[0][-1] += 100  
14  
15 In [#]: a  
16 Out[#]: [[0, 1, 102], [1, 2, 3]]  
17  
18 In [#]:
```

Here is where understanding the memory model becomes quite important. Every thing in Python (at least from a memory point of view) is an object and names are merely bound (or point) to that object. The same is true for lists, and the indexes / elements of a list

object are merely bound (or point) to other objects. So in the example above, the element `a[0]` is bound to the list object created in memory by `[1, 2, 3]` and the element `a[0][0]` is bound to the integer object 0. Consider the following examples:

```

1 In [#]: a = [[0, 1, 2],
2     ....:           [1, 2, 3]]
3
4 In [#]: a
5 Out[#]: [[0, 1, 2], [1, 2, 3]]
6
7 In [#]: b = a[0]
8
9 In [#]: b
10 Out[#]: [0, 1, 2]
11
12 In [#]: a
13 Out[#]: [[0, 1, 2], [1, 2, 3]]
14
15 In [#]: b[0] = 5
16
17 In [#]: b
18 Out[#]: [5, 1, 2]
19
20 In [#]: a
21 Out[#]: [[5, 1, 2], [1, 2, 3]]
22
23 In [#]:

```

In the example above it is important to notice that when changing `b[0] = 5` (line 15) that the binding of the first element of the list object `[0, 1, 2]` is changed. `b` is the same list object as `a[0]` and `b[0]` is the same integer object as `a[0][0]`; so changing `b[0]` to 5 only changes the binding of the first element of **one** list object (`[0, 1, 2]`), but both `a[0]` and `b` are bound to the same list object. See if you can understand the effect of line 15, in the example above, on the memory model shown in figure 9.2

Lets consider the case where we want to store a list of student information. Here is an example of a list of dictionary objects, where each dictionary object stores one students information. You should by now be able to figure out the logic of this example.

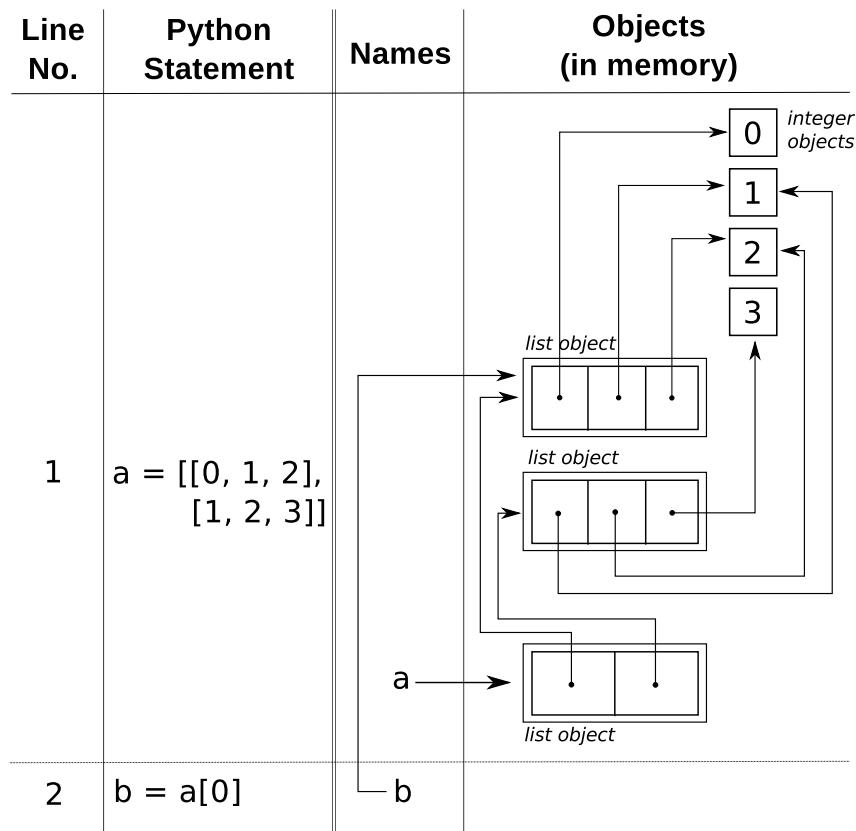


Figure 9.2: Memory model for list objects

```

1 all_students = []
2 name_template = "\nEnter student %d's name [q to quit]: "
3
4 cnt = 1
5 while True:
6     name = raw_input(name_template % cnt)
7     if name == 'q':
8         break
9     mark = float(raw_input("Enter student %d's mark: " % cnt))
10    student = {'name': name,
11                'mark': mark}
12    all_students.append(student)
13    cnt += 1
14
15
16 print '\nStudent Info: '
17 for student in all_students:
18     print 'name: %s \t mark: %d' % (student['name'],

```

<sup>19</sup>

student[ 'mark' ])

## 9.4 Exercises

1. Write a program that displays each element of a list or tuple.
2. Write a program that asks the user to enter the length of a list that the user would like to input. The program must then ask the user to enter a value for each entry of the list.
3. Write a program that sums all the elements of list or tuple.  
Check your answer by adding the terms individually on a calculator.
4. Write a program that asks the user to enter two vectors  $\mathbf{x}$  and  $\mathbf{y}$ . The program must then add the two vectors component for component if they have the same length and display the result to the screen. If the vectors have different lengths the program must display Dimension miss match to the screen.
5. Write a program that asks the user to input a list. The program must then determine the smallest positive component of the list and display it's value and position (index) in an appropriate message to the screen. If the list contains only negative values the program must display 0 to the screen. For example the smallest positive component the following list is 2

$$[4, 2, -5, 6, -1]$$

6. Write a program that determines the infinity or maximum norm  $\|\mathbf{x}\|_\infty$  of a list  $\mathbf{x}$  (i.e. the entry of a list with the largest absolute value e.g.)

$$\mathbf{x} = [1 \ -5 \ 2 \ 7 \ 13 \ -14 \ 8] \quad (9.2)$$

the index with the largest absolute value is index number 5 where the entry has an absolute value of  $|-14| = 14$ . Therefore

$$\|\mathbf{x}\|_\infty = 14 \quad (9.3)$$

7. Write a program that simulates a user specified amount of single player dice battle games. The program must then return the score for each game in a list.

**The dice battle game works as follows:**

First a single dice is thrown. If the number thrown is

- (a) an even number then two additional dices are thrown,
- (b) an odd number then three additional dices are thrown.

The score is calculated by summing the values of all the dices thrown. If the score is below 8 then an additional dice is thrown and added to the score. If the score exceeds 13 the player gets a score of 0 otherwise the player obtains the calculated score.

8. Write a program that simulates a user specified amount of dice battle games between two players. The results of the games must be stored in lists as follows:
  - (a) First list: the score for player 1 for each game,
  - (b) Second list: the score for player 2 for each game, and
  - (c) Third list: a 1 or 2 indicating the number of the player that one that round for each game.

# Chapter 10

## Vectors and matrices

As mentioned in chapter 9 (section 9.1.5) there is one additional data container available in Python: the `numpy` array object. `numpy` arrays are used for either vector or matrix algebra (i.e. dot product or two vectors or matrices, inverse of a matrix, transpose of a matrix, etc). `numpy` itself is actually a module containing the `numpy` array object and a multitude of functions for vector and matrix algebra. We will thus be importing the required data container and functions, as we need them, from the `numpy` module.

I'll start off by showing as example of how we create a row vector in Python. A row vector can be seen as almost being the same as a list and in fact we will use lists to create an array (an array is a generic programming term use for both vectors and matrices).

```
1 from numpy import array  
2  
3  
4 vec = array([1, 2, 3])  
5 print vec
```

In line 1 we `import` the `array` object from `numpy` and use it to create a row vector in line 4. Now lets move on to creating a column vector. A column vector is created by using a list of lists, shown in the following example:

```
1 from numpy import array  
2  
3  
4 vec = array([[1],
```

```
5     [2],  
6     [3]])  
7 print vec
```

This example defines a column vector with the same three components. As you can see each value is stored in a list and that list is in turn stored in another list. Lines 4–6 can also be re-written as follows to create the same column vector:

```
4 vec = array([[1], [2], [3]])
```

Row and column vectors are indexed (values are accessed) in exactly the same way you would with a list. A matrix is also defined by using a list of lists e.g.

```
1 from numpy import array  
2  
3  
4 mat = array([[1, 2, 3],  
5             [2, 3, 4],  
6             [3, 4, 5]])  
7 print mat
```

This example creates a  $3 \times 3$  symmetric matrix `mat`. Matrices are indexed using two indexing integers enclosed by square brackets ([]), one index for the row and one index for the column, separated by a comma and a space:

```
1 In [#]: mat = array([[1, 2, 3],  
2     ....:                 [2, 3, 4],  
3     ....:                 [3, 4, 5]])  
4  
5 In [#]: mat  
6 Out[#]:  
7 array([[1, 2, 3],  
8         [2, 3, 4],  
9         [3, 4, 5]])  
10  
11 In [#]: mat[0, 2]  
12 Out[#]: 3
```

```
13 In [#]: mat[2, 0]
14 Out[#]: 3
15
16
17 In [#]: mat[1, 1]
18 Out[#]: 3
19
20 In [#]:
```

As you can see the first index corresponds to the row and the second index corresponds to the column (going from the top down and left to right). Matrices can also be indexed by enclosing each indexing integer in its own square brackets ([])(as you would in the case of a list of lists) as follows:

```
1 In [#]: mat = array([[1, 2, 3],
2     ....:                 [2, 3, 4],
3     ....:                 [3, 4, 5]])
4
5 In [#]: mat
6 Out[#]:
7 array([[1, 2, 3],
8         [2, 3, 4],
9         [3, 4, 5]])
10
11 In [#]: mat[0][2]
12 Out[#]: 3
13
14 In [#]: mat[2][0]
15 Out[#]: 3
16
17 In [#]: mat[1][1]
18 Out[#]: 3
19
20 In [#]:
```

Reverse indexing (negative indexing integers) can also be used for both vectors and matrices (e.g. `mat[-1][-1]`, `mat[-1][0]`, etc).



**More Info:**

Import the `numpy` module into the *IPython Console* and type `numpy?` for available documentation. The following is a snippet taken from this documentation.

```
1 .
2 .
3 .
4 Available subpackages
-----
5 doc
6     Topical documentation on broadcasting, indexing, etc.
7 lib
8     Basic functions used by several sub-packages.
9 random
10    Core Random Tools
11 linalg
12    Core Linear Algebra Tools
13 fft
14    Core FFT routines
15 polynomial
16    Polynomial tools
17 testing
18    Numpy testing tools
19 f2py
20    Fortran to Python Interface Generator.
21 distutils
22    Enhancements to distutils with support for
23    Fortran compilers support and more.
24 .
25 .
26 .
27 .
```

Many of the functions available in the `numpy` module are sorted into sub-packages (or sub-modules) as shown by the documentation snippet above. Remember that you can get documentation or information on any sub-packages, functions, etc. by typing `help(object)` or `object?.`. For example the following documentation snippet is taken for typing `numpy.linalg?`

```
1 .
2 .
3 .
4 Core Linear Algebra Tools
-----
```

```
6  Linear algebra basics:  
7  
8  - norm          Vector or matrix norm  
9  - inv           Inverse of a square matrix  
10 - solve         Solve a linear system of equations  
11 - det           Determinant of a square matrix  
12 - lstsq          Solve linear least-squares problem  
13 - pinv          Pseudo-inverse (Moore-Penrose) calculated  
14                  using a singular value decomposition  
15 - matrix_power   Integer power of a square matrix  
16 .  
17 .  
18 .
```



#### More Info:

For more information regarding the `numpy` module and available functions please see the following sources:

<http://docs.scipy.org/doc/numpy/user/>  
[http://www.scipy.org/Tentative\\_NumPy\\_Tutorial](http://www.scipy.org/Tentative_NumPy_Tutorial)  
<http://docs.scipy.org/doc/numpy/reference/>  
<http://mathesaurus.sourceforge.net/numeric-numpy.html>

## 10.1 Simple Computations with vectors and matrices

The one big advantage of `numpy` arrays is that you can do computations on all components of a vector or matrix at once, without having to use a `for` loop statement:

```
1  In [#]: mat = array([[1, 2, 3],  
2      ....:                 [2, 3, 4],  
3      ....:                 [3, 4, 5]])  
4  
5  In [#]: mat * 2  
6  Out[#]:  
7  array([[ 2,  4,  6],  
8      [ 4,  6,  8],  
9      [ 6,  8, 10]])  
10  
11 In [#]: mat
```

```
12 Out[#]:  
13 array([[1, 2, 3],  
14         [2, 3, 4],  
15         [3, 4, 5]])  
16  
17 In [#]: mat *= 10  
18  
19 In [#]: mat  
20 Out[#]:  
21 array([[10, 20, 30],  
22         [20, 30, 40],  
23         [30, 40, 50]])  
24  
25 In [#]:
```

Vectors and matrices of the same size (shape) can be added together or subtracted from one another, on a component by component basis, simply by using the plus or minus operators (+ -):

```
1 In [#]: mat1 = array([[1, 2, 3],  
2     ....:             [2, 3, 4],  
3     ....:             [3, 4, 5]])  
4  
5 In [#]: mat2 = array([[10, 20, 30],  
6     ....:             [20, 30, 40],  
7     ....:             [30, 40, 50]])  
8  
9 In [#]: mat1 += mat2  
10  
11 In [#]: mat1  
12 Out[#]:  
13 array([[11, 22, 33],  
14         [22, 33, 44],  
15         [33, 44, 55]])  
16  
17 In [#]:
```



### Take Note:

numpy arrays should only be used for integer values or floating point values !!

numpy arrays can only contain one object type (i.e. all components are either integers or all components are floating point numbers)



### Take Note:

numpy arrays also behave differently for integer values and floating point values, what I mean by this is that if you specify all the components of a vector or matrix as integers then the array will be an integer type array. At least one component must be a floating point value for the array to be a float type array. Even I got caught out by this will compiling these notes so please be aware of the following examples:

```
1 In [#]: from numpy import array
2
3 In [#]: vec1 = array([4, 5, 6])
4
5 In [#]: vec2 = array([2, 2, 2])
6
7 In [#]: vec1.dtype
8 Out[#]: dtype('int64')
9
10 In [#]: vec1 / vec2
11 Out[#]: array([2, 2, 3])
12
13 In [#]: vec3 = array([2, 2.0, 2])
14
15 In [#]: vec3
16 Out[#]: array([ 2.,  2.,  2.])
17
18 In [#]: vec3.dtype
19 Out[#]: dtype('float64')
20
21 In [#]: vec1 / vec3
22 Out[#]: array([ 2. ,  2.5,  3. ])
23
24 In [#]:
```

The `.dtype` function, associated to the array container, is used to find out the data type of the array. Alternatively you can force the type of the array to be either a float type or integer type when you create the array:

```
1 In [#]: from numpy import array
2
3 In [#]: vec1 = array([4, 5, 6], dtype=float)
4
5 In [#]: vec2 = array([2.5, 3.7, 9.8], dtype=int)
6
7 In [#]: vec1
8 Out[#]: array([ 4.,  5.,  6.])
9
10 In [#]: vec2
11 Out[#]: array([2, 3, 9])
12
13 In [#]:
```

You do this by adding a comma and a space after the list, and then the `dtype=<type>` keyword as shown above. In my opinion it is good practice to create all vectors or matrices with the optional `dtype=<type>` keyword added.



#### More Info:

Remember you can use Python's future division to avoid this integer division behaviour:

```
from __future__ import division
```

## 10.2 Displaying the components of a vector

As shown in Section 9.1.2.4 (for a list) we can make use of a `for` loop statement that starts at the first entry and displays it to the screen, and then goes to the second entry and displays it, etc. (until we have displayed all the entries). The following example illustrates the use of a `for` loop statement to display the entries of a vector one at a time:

```
1 from numpy import array
2
3
4 my_vector = array([10, 9, 8, 7, 6, 5], dtype=int)
5 for i in range(6):
6     print "Entry %d of the vector is %d" % (i, my_vector[i])
```

We can also use the other looping techniques we learnt for a list (Section 9.1.2.4) for a vector:

```
1 from numpy import array
2
3
4 my_vector = array([10, 9, 8, 7, 6, 5], dtype=int)
5
6 # ---- Method 1 ----
7 print "\nMethod 1"
8 ind = 0
9 for val in my_vector:
10     print "Entry %d of vector is %d" % (ind, val)
11     ind += 1
12
13 # ---- Method 2 ----
14 print "\nMethod 2"
15 for ind, val in enumerate(my_vector):
16     print "Entry %d of vector is %d" % (ind, val)
```

Each of the looping methods in the example above gives the same output:

```
1 Entry 0 of the vector is 10
2 Entry 1 of the vector is 9
3 Entry 2 of the vector is 8
4 Entry 3 of the vector is 7
5 Entry 4 of the vector is 6
6 Entry 5 of the vector is 5
```

The methods (shown above) for displaying the components of a vector will also work for displaying the components of a list or tuple.

### 10.3 Displaying the components of a matrix

Now that we have seen how to display the components of a vector to the screen let us display the components of a matrix to the screen. As we have seen we required one **for** loop statement to display the entries of a vector to the screen.

We can basically consider each row (or column) of a matrix as a vector. As we know we need a `for` loop statement to display the entries of the vector to the screen. We also require an additional `for` loop statement to move between the rows (or columns) of matrix. The `for` loop statements have to be inside each other (nested) because for every row (or column) we want to display all the entries of the respective row (or column) to the screen.

Here, is an example program that displays each component of a matrix one at a time:

```
1 from numpy import array
2
3
4 str_template = "Row %d and Col %d of the matrix is %d"
5 my_mat = array([[10, 9, 8, 7, 6, 5],
6                 [20, 30, 40, 50, 60, 70],
7                 [11, 9, 13, 7, 15, 5]], dtype=int)
8
9 # ---- Method 1 ----
10 print "\nMethod 1"
11 for i in range(3):
12     for j in range(6):
13         print str_template % (i, j, my_mat[i][j])
14
15 # ---- Method 2 ----
16 print "\nMethod 2"
17 i = 0
18 for row_vector in my_mat:
19     for j in range(6):
20         print str_template % (i, j, row_vector[j])
21     i += 1
22
23 # ---- Method 3 ----
24 print "\nMethod 3"
25 i = 0
26 for row_vector in my_mat:
27     j = 0
28     for val in row_vector:
29         print str_template % (i, j, val)
30         j += 1
31     i += 1
32
33 # ---- Method 4 ----
34 print "\nMethod 4"
```

```
35 for i, row_vector in enumerate(my_mat):
36     for j, val in enumerate(row_vector):
37         print str_template % (i, j, val)
```

Each of the looping methods in the example above gives the same output:

```
1 Row 0 and Col 0 of the matrix is 10
2 Row 0 and Col 1 of the matrix is 9
3 Row 0 and Col 2 of the matrix is 8
4 Row 0 and Col 3 of the matrix is 7
5 Row 0 and Col 4 of the matrix is 6
6 Row 0 and Col 5 of the matrix is 5
7 Row 1 and Col 0 of the matrix is 20
8 Row 1 and Col 1 of the matrix is 30
9 Row 1 and Col 2 of the matrix is 40
10 Row 1 and Col 3 of the matrix is 50
11 Row 1 and Col 4 of the matrix is 60
12 Row 1 and Col 5 of the matrix is 70
13 Row 2 and Col 0 of the matrix is 11
14 Row 2 and Col 1 of the matrix is 9
15 Row 2 and Col 2 of the matrix is 13
16 Row 2 and Col 3 of the matrix is 7
17 Row 2 and Col 4 of the matrix is 15
18 Row 2 and Col 5 of the matrix is 5
```

The `len` command can also be used for vectors to determine how many entries they have. For matrices if we need to know the number of rows and columns and in the matrix we can use the `.shape` (note the dot) command. The `.shape` command is associated to the `numpy` array data container and it returns a tuple with two values. The first value is the number of rows and the second value the number of columns of the matrix e.g.

```
1 from numpy import array
2
3
4 my_mat = array([[10, 9, 8, 7, 6, 5],
5                 [20, 30, 40, 50, 60, 70],
6                 [11, 9, 13, 7, 15, 5]], dtype=int)
7
8 num_rows, num_cols = my_mat.shape
9 print num_rows, num_cols
```

Here, `num_rows` is equal to 3 and `num_cols` equal to 6, since `my_mat` has 3 rows and 6 columns. If only specify a single name for the output of the `.shape` command as in the following example

```
1 from numpy import array
2
3
4 my_mat = array([[10,  9,  8,  7,  6,  5],
5                 [20, 30, 40, 50, 60, 70],
6                 [11,  9, 13,  7, 15,  5]], dtype=int)
7
8 mat_shape = my_mat.shape
9 print mat_shape
```

then `mat_shape` is a tuple with two entries. The first entry of `mat_shape` is the number of rows of the specified matrix `my_mat` and the second entry of `mat_shape` is the number of columns of `my_mat`.

The following example uses the `.shape` command to determine the dimensions i.e. number of rows and columns of `my_mat`:

```
1 from numpy import array
2
3
4 str_template = "Row %d and Col %d of the matrix is %d"
5 my_mat = array([[10,  9,  8,  7,  6,  5],
6                 [20, 30, 40, 50, 60, 70],
7                 [11,  9, 13,  7, 15,  5]], dtype=int)
8
9 num_rows, num_cols = my_mat.shape
10 for i in range(num_rows):
11     for j in range(num_cols):
12         print str_template % (i, j, my_mat[i][j])
```

## 10.4 Defining a matrix using the `raw_input` statement

A student asked me whether or not a matrix can be read in using the `raw_input` statement. As we've seen before, the `raw_input` statement is very convenient to read a single value

from the *IPython Window*. But how can we read all the entries of a matrix using the `raw_input` statement? I can think of a simple program making use of two `for` loops statement:

```

1 from numpy import zeros
2
3
4 # Ask the user how many rows the matrix has
5 num_rows = int(input("How many rows in the matrix? "))
6 # Ask the user how many columns the matrix has
7 num_cols = int(input("How many columns in the matrix? "))
8
9 # Initialize the matrix to all zeros
10 user_mat = zeros((num_rows, num_cols), dtype=float)
11
12 str_template = "Enter component [%d, %d]: "
13 # For loop over the rows
14 for i in range(num_rows):
15     # For loop over the columns
16     for j in range(num_cols):
17         # User input statement to read the i,j component
18         user_mat[i][j] = float(raw_input(str_template % (i, j)))
19
20 print user_mat

```



#### More Info:

The `numpy.zeros()` function is used to create a vector or matrix containing only (0) values. Import the `numpy` module into the *IPython Console* and type `numpy.zeros?` for available documentation.

## 10.5 Example: Norm of a vector

Other than referring to a single matrix entry at a time or doing simple computations on a matrix, many other matrix manipulations and computations are possible in Python (through the `numpy` module). To illustrate the use of vectors, let's develop a program that computes the norm (magnitude or length) of a vector  $\mathbf{a}$  (name `my_vec`), defined as:

$$\|\mathbf{a}\| = \sqrt{\sum_{i=1}^n a_i^2}, \quad (10.1)$$

where  $n$  is the number of components. As a first example, let's assume the vector has three components and lets compute the norm manually in a program:

```
1 from numpy import array
2
3
4 # set the norm = 0
5 my_norm = 0.0
6 # define a vector my_vec
7 my_vec = array([1, 2, 3], dtype=float)
8
9 for val in my_vec:
10     # add the square of each component to norm
11     my_norm += val ** 2
12 # take the square root of all components
13 my_norm **= 0.5
14 # display the result
15 print "Norm of vector = ", my_norm
```

This example give the following output:

```
1 Norm of vector =  3.74165738677
```

The norm of a vector can also be calculated using the built-in `norm` function in the `numpy.linalg` module as shown in the following example:

```
1 from numpy import array
2 from numpy.linalg import norm
3
4
5 # define a vector my_vec
6 my_vec = array([1, 2, 3], dtype=float)
7
8 # display the result
9 print "Norm of vector = ", norm(my_vec)
```

**More Info:**

Import the `numpy` module into the *IPython Console* and type `numpy.linalg.norm?` for available documentation.

## 10.6 Example: Dot product

The dot product of two vectors  $\mathbf{x}$  and  $\mathbf{y}$  is defined by the scalar

$$c = \mathbf{x} \cdot \mathbf{y} = \sum_{i=0}^n x_i y_i \quad (10.2)$$

where  $n$  is the dimension (or length) of vector  $\mathbf{x}$  and  $\mathbf{y}$ . The dot product is equivalent to multiplying a row vector ( $1 \times n$ ) with a column vector ( $n \times 1$ ) which gives a  $1 \times 1$  matrix or scalar value.

The following program computes the dot product between two vectors. The program first checks whether the vectors have the same length before it starts computing the dot product.

```

1  from numpy import array
2
3
4  # define 2 vectors vec_a and vec_b
5  vec_a = array([1, 1, 1], dtype=float)
6  vec_b = array([[1],
7                 [0],
8                 [0]], dtype=float)
9
10 # Dot product between two vectors is a scalar value. We need to
11 # initialise our memory name
12 dot_product = 0.0
13 # Check whether vector a and b have the same dimension (length)
14 if len(vec_a) == len(vec_b):
15     # Loop over the all the components of vector a and b
16     for i in range(len(vec_a)):
17         # Compute the dot product by multiplying the respective
18         # components of vectors a and b, and by summing them
19         # together
20         dot_product += vec_a[i] * vec_b[i]
```

```
21     print "The dot product of the 2 vectors is ", dot_product
22 else:
23     print "Vector sizes do not match"
```

This example give the following output:

```
1 The dot product of the 2 vectors is [ 1.]
```

The dot product of two vectors can also be calculated using the built-in `dot` function in the `numpy` module as shown in the following example:

```
1 from numpy import array
2 from numpy import dot
3
4
5 # define 2 vectors vec_a and vec_b
6 vec_a = array([1, 1, 1], dtype=float)
7 vec_b = array([[1],
8               [0],
9               [0]], dtype=float)
10
11 # Check whether vector a and b have the same dimension (length)
12 if len(vec_a) == len(vec_b):
13     dot_product = dot(vec_a, vec_b)
14     print "The dot product of the 2 vectors is ", dot_product
15 else:
16     print "Vector sizes do not match"
```



#### More Info:

Import the `numpy` module into the *IPython Console* and type `numpy.dot?` for available documentation.

## 10.7 Example: Matrix-vector multiplication

Let's develop a program that can multiply a matrix with a vector. Recall from your linear algebra course that matrix-vector multiplication is defined as

$$c_i = \sum_{j=0}^n A_{ij}a_j \quad \text{for } i = 0, 2, \dots, m \quad (10.3)$$

where the  $m \times n$  matrix  $\mathbf{A}$  is multiplied with the  $n \times 1$  vector  $\mathbf{a}$  to give the  $m \times 1$  vector  $\mathbf{c}$ . I'll also illustrate Eq.(10.3) with a numerical example. Let's multiply a  $2 \times 3$  matrix  $\mathbf{A}$  with a  $3 \times 1$  vector  $\mathbf{a}$  to obtain a  $2 \times 1$  vector  $\mathbf{c}$ :

$$\begin{bmatrix} 3 & 2 & 5 \\ 2 & 3 & 1 \end{bmatrix} \begin{Bmatrix} 1 \\ 2 \\ 3 \end{Bmatrix} = \begin{Bmatrix} 3 \times 1 + 2 \times 2 + 5 \times 3 \\ 2 \times 1 + 3 \times 2 + 1 \times 3 \end{Bmatrix} = \begin{Bmatrix} 22 \\ 11 \end{Bmatrix} \quad (10.4)$$

Now let's try to write a program that can multiply a  $m \times n$  matrix  $\mathbf{A}$  with a  $n \times 1$  vector. First of all, you must recognise that we will use `for` loop statements for our computations. This is because before we start multiplying a matrix with a vector, the matrix and vector must be known. So we will know how many rows and columns the matrix has. Since the number of rows and columns are known, we will use `for` loop statements.

We also need to know how many `for` loop statements are required. We take guidance directly from Eq.(10.3). We see that two distinct counters are required, namely  $i = 0, 2, \dots, m$  and  $j = 0, 2, \dots, n$ . The  $i$  counter refers to the current component of the  $\mathbf{c}$  vector I'm computing, while the  $j$  counter is used to sum the appropriate products.

The order in which these two `for` loop statements must appear is not unique, as long as the correct logic is used. However, I prefer to use the `for` loop over the  $i$  counter as the outer loop, within which I use the `for` loop over the  $j$  counter to perform the addition of terms. The reason for this choice is that my program now reflects the same method I use when I multiply a matrix with a vector using pen and paper. I first decide which component I want to compute (a specific value for  $i$ ), then I compute this component by adding the appropriate products (using the second `for` loop statement with  $j = 0, 2, \dots, n$ ). If I use the  $j$  counter in the outer loop and the  $i$  counter in the inner loop, I'm computing the first contribution to all the components of  $\mathbf{c}$ , then adding the second contribution to all the components of  $\mathbf{c}$  and so on. I find such a program confusing, because this does not reflect the usual method of multiplying a matrix and vector.

One last statement we need before we can write the program: the `.shape` statement is used to get the number of rows and columns of a matrix. Recall that the usage is

```
1 num_rows, num_cols = matrix.shape
```

where the function `.shape` assigns the number of rows and columns of the matrix `matrix` to the names `num_rows` and `num_cols`. Now we are ready to write a program that multiplies a matrix with a vector:

```
1 from numpy import array
2 from numpy.random import rand
3
4
5 # create a random matrix
6 mat = rand(10, 8)
7 # create a random column vector
8 vec = rand(8, 1)
9
10 # get the shape of the matrix
11 num_rows, num_cols = mat.shape
12 # initialise an empty list for storing the i-th entry
13 # of multiplication
14 multi_vec = []
15 for i in range(num_rows):
16     multi_vec.append(0.0)
17     for j in range(num_cols):
18         # add the appropriate term to the c-vector
19         multi_vec[-1] += mat[i][j] * vec[j]
20 # convert the list to a vector
21 multi_vec = array(multi_vec)
22 # display the results
23 print "The product of matrix A with vector a is:"
24 print multi_vec
```



#### More Info:

The `numpy.random.rand()` function is used to create a vector or matrix containing random values between 0 and 1. Import the `numpy` module into the *IPython Console* and type `numpy.random.rand?` for available documentation.

This example (for a selected case) gives the following output:

```
1 The product of matrix A with vector a is:
2 [[ 2.40994204]
```

```
3 [ 1.59491837]
4 [ 2.29118772]
5 [ 1.88103115]
6 [ 2.84548976]
7 [ 2.19079554]
8 [ 1.34920096]
9 [ 1.25477803]
10 [ 3.39309945]
11 [ 2.80552001]]
```

Program lines 6–8 creates a random matrix `mat` and a random vector `vec`, just to test our program. Program lines 13–19 contain the actual matrix-vector multiplication and program lines 23–24 creates the output to the *IPython Console*. In the program above it is not strictly necessary to use the `.shape` command, since we know that the size of the matrix is 10 by 8. It is however sensible to use the `.shape` statement to prevent changing the code in lines 10–24 should we choose to change the matrix `mat`.

Here follows another version of the example shown above. You should be familiar with the programming logic used.

```
1 from numpy import zeros
2 from numpy.random import rand
3
4
5 # create a random matrix
6 mat = rand(10, 8)
7 # create a random column vector
8 vec = rand(8, 1)
9
10 # get the shape of the matrix
11 num_rows, num_cols = mat.shape
12 # initialise an empty list for storing the i-th entry
13 # of multiplication
14 multi_vec = zeros((num_rows, 1))
15 for i in range(num_rows):
16     for j in range(num_cols):
17         # add the appropriate term to the c-vector
18         multi_vec[i] += mat[i][j] * vec[j]
19 # display the results
20 print "The product of matrix A with vector a is:"
21 print multi_vec
```

The matrix-vector multiplication can also be calculated using the built-in `dot` function in the `numpy` module as shown in the following example:

```

1  from numpy import dot
2  from numpy.random import rand
3
4
5  # create a random matrix
6  mat = rand(10, 8)
7  # create a random column vector
8  vec = rand(8, 1)
9
10 multi_vec = dot(mat, vec)
11
12 # display the results
13 print "The product of matrix A with vector a is:"
14 print multi_vec

```

## 10.8 Example: General vector-vector multiplication

Let us try and write a program that would allow us to multiply two vectors with each other. We have already looked at multiplying a row vector ( $1 \times n$ ) with a column vector ( $n \times 1$ ) which results in a ( $1 \times 1$ ) matrix or scalar value.

$$\begin{bmatrix} 2 & 3 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} = [2 \times 1 + 3 \times 2 + 1 \times 3] = [11] \quad (10.5)$$

When we multiply a column vector ( $m \times 1$ ) with a row vector ( $1 \times n$ ) we obtain a ( $m \times n$ ) matrix. Remember from Linear Algebra that the inner dimensions have to be the same when multiplying. Here, the inner dimension of the vector-vector multiplication is 1 and therefore the lengths of the vectors can be different.

$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \begin{bmatrix} 2 & 3 & 1 & 4 \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 4 \\ 4 & 6 & 2 & 8 \\ 6 & 9 & 3 & 12 \end{bmatrix} \quad (10.6)$$

Here follows a general vector-vector multiplication program.

```
1 # General vector-vector multiplication
2 # Multiply row vector (mx1) with a column vector (1xn)
3 # which results in a matrix (mxn)
4 # As well as a row vector (1xn) with a column vector (nx1)
5 # which result in a scalar value or matrix (1x1)
6
7 from numpy import array
8 from numpy import zeros
9
10
11 vector1 = array([[1],
12                  [2],
13                  [3]], dtype=int)
14 vector2 = array([[2, 3, 1, 4]], dtype=int)
15
16 # Here we use the shape function instead of length as we want
17 # to distinguish between row and column vectors.
18 nrows1, ncols1 = vector1.shape
19 nrows2, ncols2 = vector2.shape
20 # The column dimension of vector 1 must match the row dimension
21 # of vector 2 -> (mx1)*(1xn) or (1xn)*(nx1) also refered to as
22 # the inner dimensions that have to match.
23 if ncols1 == nrows2:
24     # Initialize ith row and jth column entry of the resulting
25     # vector product which could be a matrix or scalar value
26     # (then the two outer loops are only performed once).
27     vector_product = zeros((nrows1, ncols2))
28     # The number of rows of vector 1 determine the number of
29     # rows of the resulting matrix.
30     for i in range(nrows1):
31         # The number of columns of vector 2 determine the number
32         # of columns of the resulting matrix.
33         for j in range(ncols2):
34             # The inner dimensions i.e. number of columns of
35             # vector 1 (or the number of rows of vector 2)
36             # determine the number of elements we need to sum
37             # together.
38             for k in range(ncols1):
39                 # We sum over the inner dimensions (k counter)
40                 # of the ith row of vector 1 and the jth
41                 # column of vector 2.
42                 vector_product[i][j] += (vector1[i][k] *
43                                         vector2[k][j])
```

```

44     # Display the vector product to the screen
45     print vector_product
46 else:
47     print "Inner dimensions of vectors do not match"

```

What about matrix-matrix multiplication ? Would the above program work?

As you have probably guessed by now can also be calculated using the built-in dot function in the numpy module as shown in the following example:

```

1 from numpy import array, dot
2
3
4 vector1 = array([[1],
5                  [2],
6                  [3]], dtype=int)
7 vector2 = array([[2, 3, 1, 4]], dtype=int)
8
9 # Here we use the shape function instead of length as we want
10 # to distinguish between row and column vectors.
11 nrows1, ncols1 = vector1.shape
12 nrows2, ncols2 = vector2.shape
13 # The column dimension of vector 1 must match the row dimension
14 # of vector 2 -> (mx1)*(1xn) or (1xn)*(nx1) also referred to as
15 # the inner dimensions that have to match.
16 if ncols1 == nrows2:
17     print dot(vector1, vector2)
18 else:
19     print "Inner dimensions of vectors do not match"

```

## 10.9 Gauss elimination

The solution of a system of linear equations remains one of the numerical computations performed most often in all engineering disciplines. In your linear algebra course, you have already mastered Gauss elimination. Gauss elimination remains one of the most efficient direct methods to solve large systems of linear equations. Let's revisit the method.

During the forward reduction step of Gauss elimination, the original system matrix is reduced to a system that contains zeros below the diagonal. This is achieved by subtract-

ing a fraction of one row from the next. I'll illustrate with a  $4 \times 4$  system:

$$\begin{bmatrix} 2 & 3 & 4 & 1 \\ 1 & 1 & 2 & 1 \\ 2 & 4 & 5 & 2 \\ 1 & 2 & 3 & 4 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{Bmatrix} = \begin{Bmatrix} 10 \\ 5 \\ 13 \\ 10 \end{Bmatrix} \quad (10.7)$$

Replace Row 2 of the system with (Row 2 -  $(\frac{1}{2}) \times$  Row 1) i.e.

$$\begin{bmatrix} 2 & 3 & 4 & 1 \\ 0 & -0.5 & 0 & 0.5 \\ 2 & 4 & 5 & 2 \\ 1 & 2 & 3 & 4 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{Bmatrix} = \begin{Bmatrix} 10 \\ 0 \\ 13 \\ 10 \end{Bmatrix} \quad (10.8)$$

Then replace Row 3 with (Row 3 -  $(\frac{2}{2}) \times$  Row 1) i.e.

$$\begin{bmatrix} 2 & 3 & 4 & 1 \\ 0 & -0.5 & 0 & 0.5 \\ 0 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{Bmatrix} = \begin{Bmatrix} 10 \\ 0 \\ 3 \\ 10 \end{Bmatrix} \quad (10.9)$$

Finally, we replace Row 4 with (Row 4 -  $(\frac{1}{2}) \times$  Row 1) i.e.

$$\begin{bmatrix} 2 & 3 & 4 & 1 \\ 0 & -0.5 & 0 & 0.5 \\ 0 & 1 & 1 & 1 \\ 0 & 0.5 & 1 & 3.5 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{Bmatrix} = \begin{Bmatrix} 10 \\ 0 \\ 3 \\ 5 \end{Bmatrix} \quad (10.10)$$

We have succeeded in reducing the first column below the diagonal to zeros. Now we proceed to reduce the second column below the diagonal to zeros. This is achieved by replacing Row 3 with (Row 3 -  $(\frac{1}{-0.5}) \times$  Row 2) i.e.

$$\begin{bmatrix} 2 & 3 & 4 & 1 \\ 0 & -0.5 & 0 & 0.5 \\ 0 & 0 & 1 & 2 \\ 0 & 0.5 & 1 & 3.5 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{Bmatrix} = \begin{Bmatrix} 10 \\ 0 \\ 3 \\ 5 \end{Bmatrix} \quad (10.11)$$

The last entry in column 2 is reduced to zero by replacing Row 4 with (Row 4 -  $(\frac{0.5}{-0.5}) \times$  Row 2) i.e.

$$\begin{bmatrix} 2 & 3 & 4 & 1 \\ 0 & -0.5 & 0 & 0.5 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 1 & 4 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{Bmatrix} = \begin{Bmatrix} 10 \\ 0 \\ 3 \\ 5 \end{Bmatrix} \quad (10.12)$$

We complete the forward reduction step by reducing the column 3 entries below the diagonal (there is only 1) to zero: Row 4 = (Row 4 -  $(\frac{1}{1}) \times$  Row 3) i.e.

$$\begin{bmatrix} 2 & 3 & 4 & 1 \\ 0 & -0.5 & 0 & 0.5 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 2 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{Bmatrix} = \begin{Bmatrix} 10 \\ 0 \\ 3 \\ 2 \end{Bmatrix} \quad (10.13)$$

We are now done with the forward reduction step. All the entries of the matrix below the diagonal is zero. Now we proceed to solve the unknowns  $x_4$ ,  $x_3$ ,  $x_2$  and  $x_1$ , in that order. This part of the Gauss elimination process is called back-substitution.

The 4th equation in the above system reads

$$2x_4 = 2 \quad (10.14)$$

which is solved to provide  $x_4 = 1$ . The 3rd equation reads

$$x_3 + 2x_4 = 3 \quad (10.15)$$

which is used to solve  $x_3$  as

$$x_3 = 3 - 2x_4 = 1 \quad (10.16)$$

We continue with this process to solve  $x_2$  from the 2nd equation:

$$x_2 = \frac{0 - 0x_3 - 0.5x_4}{-0.5} = 1 \quad (10.17)$$

Finally,  $x_1$  is solved from the 1st equation:

$$x_1 = \frac{10 - 3x_2 - 4x_3 - x_4}{2} = 1 \quad (10.18)$$

### 10.9.1 Gauss elimination algorithm

Now that we have reviewed the Gauss elimination process, let's attempt to program it. You will find this program quite challenging, and will have to review the steps numerous times before you finally understand them all.

To perform Gauss elimination, we require a linear system to solve. I'll use the same linear system as above during the program development.

First of all, we have to decide what type of computations are required during Gauss elimination. You should be able to recognise that a predetermined number of operations are performed, based on the size of the linear system. So we'll use `for` loop statements, since the size of the matrix has to be known.

The next challenge is to decide how many `for` loop statements are required. This is not a trivial decision. We must use the numerical example of the previous section to help us out. Try to identify processes / computations that repeat and attempt to write these computations as a repeating `for` loop statement.

#### 10.9.1.1 Forward reduction step

Gauss elimination required three nested `for` loop statements during the forward reduction step. Here is what each loop will do:

1. One index (loop) tracks the diagonal term we are currently using to make all other terms below it zero. This row is called the pivot row and the entry on the diagonal is called the pivot element.
2. A second index (loop) will track which row we are currently working on. All the rows below the pivot row will be changed by adding or subtracting an appropriate fraction of the pivot row.
3. The last index (loop) is used to perform the required computation for all the columns of the rows below the pivot row.

Let's implement the forward reduction step of Gauss elimination. This will illustrate the necessity of the three required loop statements more clearly.

```
1 from numpy import array
2
3
4 #----- forward reduction step-----
5 # Gauss elimination program that solves x
6 # in the linear system Ax = a.
7 # Create left hand side matrix A
8 A = array([[2, 3, 4, 1],
9             [1, 1, 2, 1],
10            [2, 4, 5, 2],
11            [1, 2, 3, 4]], dtype=float)
12 # Create right hand side vector a
```

```

13  a = array([[10],
14      [5],
15      [13],
16      [10]], dtype=float)
17  # Compute size of matrix
18  n = A.shape[0]
19  # Start of 1st loop: pivot row counter
20  for i in range(n-1):
21      # Start of second loop: which row to reduce
22      for j in range(i+1, n):
23          # Compute the multiplier i.e. which fraction of the
24          # pivot row i has to be subtracted from row j
25          mult = A[j][i] / A[i][i]
26          # Start of 3rd loop: re-assign all columns of row j
27          # i.e. Row_j = Row_j - mult*Row_i for columns 1 to n
28          for k in range(n):
29              A[j][k] -= mult * A[i][k]
30          # Also modify the RHS vector
31          a[j] -= mult * a[i]
32
33  print "A = \n", A
34  print "a = \n", a

```

In program lines 8–16, the matrix  $A$  and vector  $a$  are defined. Program line 18 gets the number of rows of the matrix  $A$  and this is assigned to the name  $n$ . The actual forward reduction step of Gauss elimination starts at program line 20.

Since we use the index  $i$  to indicate the pivot element, what must the bounds be on this index? You should be able to recognise that we will start this index at 0 (recall that we make all the entries of the matrix below the  $[0,0]$  entry equal to zero?). After all the entries below the  $[0,0]$  entry are zero, we make all the entries below the  $[1,1]$  entry zero (so the pivot element index = 1). The pivot element index keeps on incrementing by one till it reaches the value  $n-1$ . Does this make sense to you? We have to stop at this value, because a pivot element of  $[n,n]$  makes no sense: there are no entries in the matrix below the  $[n,n]$  entry to make zero. I hope you now understand program line 18 which starts the pivot element index at 0 and increments with one till the final pivot element index of  $n-1$ .

Program line 22 defines the row index  $j$ . Its initial value is set to  $i+1$ , and it increments with one till a final row index of  $n$ . Does this make sense? If the pivot element index is  $i$ , we must reduce all entries below the  $[i,i]$  entry to zero. So we start with row  $i+1$  and continue to row  $n$ .

Program line 25 defines the multiplier i.e. which fraction of the  $i$ -th row must be subtracted from the  $j$ -th row in order to set the  $A[j, i]$  term to zero. The multiplier is simply the  $A[j, i]$  term divided by the pivot element  $A[i, i]$ .

Program lines 28–29 simply subtracts the appropriate fraction of row  $i$  from row  $j$  and assigns this to row  $j$ . A **for** loop is required to repeat this for all columns (1 to  $n$ ). Program line 30 performs the same computation on the right hand side vector  $a$ .

Note that the order of the three nested **for** loop statements is fixed. The outer **for** loop statement sets the pivot row counter ( $i$ ). Once  $i$  is known, the next **for** loop statement sets the lower bound of the row counter  $j$  in terms of  $i$ . Once  $i$  and  $j$  are known, the forward reduction step can be performed on row  $j$  for all columns  $k$  (the third and final **for** loop statement).

This example above will give the following output:

```

1 A =
2 [[ 2.   3.   4.   1. ]
3   [ 0.  -0.5  0.   0.5]
4   [ 0.   0.   1.   2. ]
5   [ 0.   0.   0.   2. ]]
6 a =
7 [[ 10.]
8   [ 0.]
9   [ 3.]
10  [ 2.]]
```

As you can see, the forward reduction step is implemented correctly. The matrix  $A$  and vector  $a$  agree with Eq.(10.13).

### 10.9.1.2 Back-substitution step

After we reduced the system to an upper triangular system, we are ready to start the back-substitution process. As before, we first have to decide what type of computations are required. **for** loop statements are again required, because the number of operations we have to perform is directly related to the size of the matrix (which is known).

Just to illustrate the required steps more clearly, consider a general  $4 \times 4$  system that

is already upper-triangular:

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ 0 & A_{22} & A_{23} & A_{24} \\ 0 & 0 & A_{33} & A_{34} \\ 0 & 0 & 0 & A_{44} \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{Bmatrix} = \begin{Bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{Bmatrix} \quad (10.19)$$

The four unknowns  $x_4$ ,  $x_3$ ,  $x_2$  and  $x_1$  are solved in this order from

$$x_4 = \frac{a_4}{A_{44}} \quad (10.20)$$

$$x_3 = \frac{a_3 - A_{34}x_4}{A_{33}} \quad (10.21)$$

$$x_2 = \frac{a_2 - A_{23}x_3 - A_{24}x_4}{A_{22}} \quad (10.22)$$

$$x_1 = \frac{a_1 - A_{12}x_2 - A_{13}x_3 - A_{14}x_4}{A_{11}} \quad (10.23)$$

You should be able to recognise the pattern in Eqs.(10.20)–(10.23):

$$x_n = \frac{a_n}{A_{nn}} \quad (10.24)$$

$$x_i = \left[ \frac{a_i - \sum_{j=i+1}^n A_{ij}x_j}{A_{ii}} \right] \quad \text{for } i = n-1, n-2, \dots, 0. \quad (10.25)$$

From Eq.(10.25) you should be able to recognise the need for two `for` loops:  $i = n-1, n-2, \dots, 0$  and  $j = i+1, i+2, \dots, n$ . The  $i$  counter points to the current component of the vector  $\mathbf{x}$  we are solving, while the  $j$  counter keeps track of the sum of products we must subtract from  $a_i$ . The order in which the `for` loop statements appear is fixed: the outer `for` loop statement must contain the name  $i$ , and the inner `for` loop statement the name  $j$ . This is because the  $j$  `for` loop lower bound is defined in terms of the current value of the  $i$  counter.

I'll go ahead and program the back-substitution step of the Gauss elimination algorithm.

```

1  from numpy import array, zeros
2
3

```

```
4 #----- forward reduction step-----
5 .
6 .
7 .
8 .
9 .
10 #----- backward reduction step-----
11 # Define a n x 1 vector of zeros
12 x = zeros((n, 1))
13 # Solve the last unknown
14 x[-1][0] = a[-1] / A[-1][-1]
15 # Start for loop: solve from row n-1, using
16 # increments of -1 all the way to row 1
17 for i in range(n-1, -1, -1):
18     # Start computation with the right hand side vector value
19     x[i][0] = a[i]
20     # Now subtract from that the product of A and the
21     # x terms that are already solved: use a for loop
22     for j in range(i+1, n):
23         x[i][0] -= A[i][j] * x[j]
24     # Now divide by the diagonal A(i,i) to get the x term
25     x[i][0] /= A[i][i]
26
27 print "x = \n", x
```

Add the back-substitution part below the forward-substitution part and run the program. The following output should appear:

```
1 x =
2 [[ 1.]
3  [ 1.]
4  [ 1.]
5  [ 1.]])
```

which is the correct solution to this  $4 \times 4$  linear system of equations.

## 10.10 Solving linear systems using Python

We've just developed our own implementation of Gauss elimination. It can solve any  $n \times n$  system of linear equations. I believe that Gauss elimination is quite a challenging algorithm to understand, so if you manage you are well on your way to becoming a proficient programmer.

Since the `numpy` module in Python was originally developed to perform linear algebra, you probably expect that `numpy` has built-in capability to solve large linear systems. This is in fact the case, more than one method exists to solve linear systems. If the linear system

$$\mathbf{A}\mathbf{x} = \mathbf{a} \quad (10.26)$$

has a unique solution, the inverse of  $\mathbf{A}$  exists and we can pre-multiply Eq.(10.26) with  $\mathbf{A}^{-1}$  to obtain

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{a} \quad (10.27)$$

This operation is performed in Python by using the the `numpy.linalg.inv` and `numpy.dot` statements i.e.

```

1 from numpy import dot
2 from numpy.linalg import inv
3
4 x = dot(inv(A), a)

```

Another Python method to solve linear systems is the `numpy.linalg.solve` function, which was created specifically for solving a set of linear equations i.e.

```

1 from numpy.linalg import solve
2
3
4 x = solve(A, a)

```

This solves  $\mathbf{x}$  using a factorisation and back-substitution algorithm similar to the Gauss elimination algorithm we implemented.

**More Info:**

Import the `numpy` module into the *IPython Console* and type

1. `numpy.linalg.inv?`
2. `numpy.linalg.solve?`

for the respective documentation available.

So let's compare the efficiency of our algorithm with that of `numpy`'s built-in functions. So here follows a complete program that solves a  $100 \times 100$  system using the three methods:

- Our Gauss elimination algorithm,
- `numpy`'s `dot(inv(A), a)` method and
- `numpy`'s `solve(A, a)` function.

```
1 from numpy import array, copy, dot, zeros
2 from time import time
3
4 from numpy.random import rand
5 from numpy.linalg import solve, inv
6
7
8 #----- forward reduction step-----
9 # Gauss elimination program that solves x
10 # in the linear system Ax = a.
11 # Create left hand side matrix A
12 A = rand(200, 200)
13 # Create right hand side vector a
14 a = rand(200, 1)
15
16 A_copy = copy(A)
17 a_copy = copy(a)
18
19 start = time()
20 # Compute size of matrix
21 n = A.shape[0]
22 # Start of 1st loop: pivot row counter
23 for i in range(n-1):
```

```
24     # Start of second loop: which row to reduce
25     for j in range(i+1, n):
26         # Compute the multiplier i.e. which fraction of the
27         # pivot row i has to be subtracted from row j
28         mult = A[j][i] / A[i][i]
29         # Start of 3rd loop: re-assign all columns of row j
30         # i.e. Row_j = Row_j - mult*Row_i for columns 1 to n
31         for k in range(n):
32             A[j][k] -= mult * A[i][k]
33             # Also modify the RHS vector
34             a[j] -= mult * a[i]
35
36
37 #----- backward reduction step-----
38 # Define a n x 1 vector of zeros
39 x = zeros((n, 1))
40 # Solve the last unknown
41 x[-1][0] = a[-1] / A[-1][-1]
42 # Start for loop: solve from row n-1, using
43 # increments of -1 all the way to row 1
44 for i in range(n-1, -1, -1):
45     # Start computation with the right hand side vector value
46     x[i][0] = a[i]
47     # Now subtract from that the product of A and the
48     # x terms that are already solved: use a for loop
49     for j in range(i+1, n):
50         x[i][0] -= A[i][j] * x[j]
51     # Now divide by the diagonal A(i,i) to get the x term
52     x[i][0] /= A[i][i]
53
54 stop = time()
55 print "Elapsed time = %f" % (stop - start)
56 start = stop
57
58 # Solve using numpy's dot(inv(A), a)
59 dot(inv(A_copy), a_copy)
60
61 stop = time()
62 print "Elapsed time = %f" % (stop - start)
63 start = stop
64
65 # Solve using numpy's solve function
66 solve(A_copy, a_copy)
67
```

```
68 stop = time()  
69 print "Elapsed time = %f" % (stop - start)
```

**More Info:**

The `time.time` function returns the current time in seconds. Import the `time` module into the *IPython Console* and type `time?` and `time.time?` for available documentation.

The output of the above program for matrix dimension of  $100 \times 100$  is

```
1 Elapsed time = 0.755095  
2 Elapsed time = 0.001345  
3 Elapsed time = 0.000379
```

for computations on a “Intel® Core™ i7 @ 2.80GHz” CPU with 8GB RAM running Python 2.7 under the Ubuntu 12.04 operating system. As you can see the inverse method is about 2 magnitudes times faster than our method and the solve method is about 3 magnitudes times faster than our method. The output of the same program on the same computer, but with matrix of dimension  $200 \times 200$ :

```
1 Elapsed time = 5.946412  
2 Elapsed time = 0.009324  
3 Elapsed time = 0.002996
```

As you can see the inverse method is now about 3 magnitudes times faster than our method and the solve method is also about 3 magnitudes times faster than our method.

Let us attempt to improving the performance of our algorithm. We can make a dramatic improvement in our algorithm’s efficiency by making a subtle change during the forward reduction step. Refer to program line 31 above: you’ll see that we subtract a fraction of line  $i$  from line  $j$ , for columns 1 to  $n$ . But is this really necessary? As we perform the forward reduction step of Gauss elimination, we know that the entries to the left of the diagonal will become zeros. And we never use these zeros during the back-substitution step of the algorithm. So why do we bother computing them at all? Computers take the same amount of time to multiply zeros with some other number as compared to non-zero number multiplication, so we are wasting valuable time. All that we have to do to improve our algorithm, is to change program line 31 to

```
31 for k in range(i+1, n):
```

This new program line will only subtract those columns that will end up as non-zeros, so we aren't computing the zeros any more. You can check that this change doesn't affect the solution, and the times required to solve a  $100 \times 100$  system now becomes

```
1 Elapsed time = 0.616382
2 Elapsed time = 0.001385
3 Elapsed time = 0.000380
```

and for the  $200 \times 200$  system it becomes

```
1 Elapsed time = 3.984113
2 Elapsed time = 0.009447
3 Elapsed time = 0.004280
```

This simple change has improved the speed of our algorithm by approximately 35%! In the weeks to come we'll make more changes to improve the performance of our algorithm.

## 10.11 Exercises

1. Write a program that displays each element of a  $3 \times 1$  vector.
2. Write a program that displays each element of a  $3 \times 3$  matrix. The program must display all the entries of the first column before moving on to display the entries in the second column and then lastly the third column.
3. Write a program that displays each element of a  $3 \times 3$  matrix. The program must display all the entries of the first row before moving on to display the entries in the second row and then lastly the third row.
4. Write a program that sums all the elements of a  $3 \times 3$  matrix, without `sum` function from `numpy`. You can check your answer by using the `sum` function from `numpy`: `from numpy import sum`.

5. Write a program that asks the user to input a vector. The program must then compute the average value of vector components and display it to the screen, without using the `average` function from `numpy`. You can check your answer using the `average` function from `numpy`: `from numpy import average`.
6. Write a program that asks the user to enter a matrix. The program must then display a matrix that contains the absolute value of each component of the matrix, without using the `abs` function from `numpy`. You can check your answer using the `abs` function from `numpy`: `from numpy import abs`.
7. Write a program that asks the user to enter a vector  $\mathbf{u}$ . The program must then look at every component of vector  $\mathbf{u}$  and check whether the component is divisible by 2. If it divisible by 2 the program must store a 1 in the position of the even component. Otherwise the program must store a zero in the position of the respective component. The number 0 is an even number.

Here is an example:

$$\begin{aligned}\mathbf{u} &= [ 2 \ 0 \ -6 \ 7 \ 10 \ -4 \ -11 ] \\ \mathbf{u}_{even} &= [ 1 \ 1 \ 1 \ 0 \ 1 \ 1 \ 0 ]\end{aligned}\tag{10.28}$$

8. The trace of a square ( $n \times n$ ) matrix  $\mathbf{A}$  is defined as the sum of the diagonal entries of  $\mathbf{A}$ . Write a program that asks the user to enter a matrix. The program must then check whether the matrix is square i.e. whether the number of rows is equal to the number of columns. If the matrix is square the program must compute the trace of the matrix (without using the `numpy.trace` function) which is given by

$$\text{tr}(\mathbf{A}) = \sum_{i=0}^n A(i, i)\tag{10.29}$$

You can check your answer by using the the `numpy.trace` function. Just for interesting sake: The trace of a matrix is the sum of its eigenvalues. The trace is an invariant of a matrix. Similarly, the length of a vector is an invariant as the length of a vector does not change length when the basis (or reference frame or axis system) changes.

9. Matrix  $\mathbf{B}$  is a antisymmetric (skew-symmetric) if matrix  $\mathbf{B}$  is square and the following holds for all  $i = 0, 1, 2, 3, \dots, n$  and  $j = 0, 1, 2, 3, \dots, n$ :

$$B(i, j) = -B(j, i)\tag{10.30}$$

Write a program that asks the user to enter a matrix. The program must then tell the user whether the matrix entered is skew-symmetric or not. Here is an example of an anti-symmetric matrix:

$$\begin{bmatrix} 0 & 1 & -2 & 3 \\ -1 & 0 & 4 & -5 \\ 2 & -4 & 0 & 6 \\ -3 & 5 & -6 & 0 \end{bmatrix}\tag{10.31}$$

10. Write a program that computes the transpose of an  $(m \times n)$  matrix  $\mathbf{Y}$  (without using the `numpy.transpose` function). The transpose of an  $(m \times n)$  matrix  $\mathbf{Y}$  is a  $(n \times m)$  matrix  $\mathbf{Y}^T$  where the indexes are given by

$$A^T(j, i) = A(i, j) \quad (10.32)$$

for all  $i = 0, 1, 2, 3, \dots, m$  and  $j = 0, 1, 2, 3, \dots, n$ . Here is an example:

$$\mathbf{A} = \begin{bmatrix} 5 & 4 & 3 & 2 \\ 6 & 7 & 8 & 9 \\ 1 & 0 & 12 & 11 \end{bmatrix} \quad \mathbf{A}^T = \begin{bmatrix} 5 & 6 & 1 \\ 4 & 7 & 0 \\ 3 & 8 & 12 \\ 2 & 9 & 11 \end{bmatrix} \quad (10.33)$$

You can check your answer by using the `numpy.transpose` function.

11. Write a program that asks the user to enter a square matrix  $\mathbf{X}$ . The program must then compute the antisymmetric (skew-symmetric) part of matrix  $\mathbf{X}$  component by component.

The antisymmetric part of a square matrix  $\mathbf{X}$  is given by the following:

$$\frac{1}{2}(\mathbf{X} - \mathbf{X}^T) \quad (10.34)$$

where  $\mathbf{X}^T$  is the transpose of matrix  $\mathbf{X}$ . The decomposition of a matrix into its symmetric and skew-symmetric parts are frequently used in computational mechanics as well as using skew-symmetric matrices to compute orthogonal matrices (rotation matrices) using exponential maps. The symmetric part of a matrix is given by

$$\frac{1}{2}(\mathbf{X} + \mathbf{X}^T). \quad (10.35)$$

Verify numerically that the sum of the symmetric and skew part of a matrix is equal to the matrix. It is easy to verify analytically.

12. Write a program that computes the element stiffness matrix of a truss member. The inputs of the stiffness matrix are the truss member's material property E, the member's area A, and the member's coordinates  $x_1, y_1, x_2$  and  $y_2$ . The member's stiffness matrix is given by:

$$\mathbf{k} = \frac{EA}{l} \begin{bmatrix} n^2 & nm & -n^2 & -nm \\ nm & m^2 & -nm & -m^2 \\ -n^2 & -nm & n^2 & nm \\ -nm & -m^2 & nm & m^2 \end{bmatrix} \quad (10.36)$$

where

$$l = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \quad (10.37)$$

with

$$n = \frac{x_2 - x_1}{l} \quad (10.38)$$

and

$$m = \frac{y_2 - y_1}{l} \quad (10.39)$$

13. Some square matrices can be factored into an upper triangular matrix  $\mathbf{U}$  and a lower triangular matrix  $\mathbf{L}$ . Write a program that will compute the  $\mathbf{LU}$  factorization of a square  $(n \times n)$  matrix  $\mathbf{A}$ .

First let us compute the upper triangular matrix  $\mathbf{U}$  of matrix  $\mathbf{A}$ . We start by setting  $\mathbf{U} = \mathbf{A}$  component for component.

Then for each column  $i = 0, 1, 2, \dots, n - 1$  we need to compute the following for the rows  $j = i + 1, i + 2, i + 3, \dots, n$  of matrix  $\mathbf{U}$ :

- (a) Set  $L(i, i)$  equal to 1.
- (b) Compute a scale factor and store it in matrix  $\mathbf{L}$  given by:

$$L(j, i) = \frac{U(j, i)}{U(i, i)}$$

- (c) Each  $j^{\text{th}}$  row of matrix  $\mathbf{U}$  must be replaced by

$$U(j, k) = U(j, k) - L(j, i) * U(i, k) \text{ for/vir } k = 0, 1, 2, \dots, n.$$

14. Once the above procedure is conducted for each column  $i = 0, 1, 2, \dots, n - 1$  of  $\mathbf{U}$ ,  $\mathbf{U}$  will be an upper triangular matrix and  $\mathbf{L}$  a lower triangular matrix such that  $\mathbf{LU}$  is equal to  $\mathbf{A}$ .

Here is an example for the matrix:

$$\mathbf{A} = \begin{bmatrix} 1 & 1 & 0 & 3 \\ 2 & 1 & -1 & 1 \\ 3 & -1 & -1 & 2 \\ -1 & 2 & 3 & -1 \end{bmatrix} \quad (10.40)$$

we obtain the lower  $\mathbf{L}$  and upper  $\mathbf{U}$  matrices:

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ 3 & 4 & 1 & 0 \\ -1 & -3 & 0 & 1 \end{bmatrix} \quad \mathbf{U} = \begin{bmatrix} 1 & 1 & 0 & 3 \\ 0 & -1 & -1 & -5 \\ 0 & 0 & 3 & 13 \\ 0 & 0 & 0 & -13 \end{bmatrix} \quad (10.41)$$

You can also test your answer using the `lu` function from the `scipy` module: `from scipy.linalg import lu`

15. Write your own matrix-matrix multiplication program that works on a component by component level, and test the answer of the previous question. Refrain from looking at the notes attempt this on your own.

Start by doing matrix-matrix multiplication by hand and then proceed to put the logic in place.

16. Matrix  $\mathbf{A}$  is strictly row diagonally dominant if for every row, the absolute value of the diagonal term is greater than the sum of absolute values of the other terms. That is if

$$|A(i, i)| > \sum_{j=1; j \neq i}^n |A(i, j)| \quad (10.42)$$

holds for every row  $i = 1, 2, 3, \dots, n$ .

Write a program that tests if a matrix is diagonally dominant.

The following matrix is diagonal row dominant

$$\begin{bmatrix} 5 & -3 & 0 & 1 \\ 3 & 8 & -1 & 2 \\ 4 & 6 & 15 & 3 \\ 0.5 & -1.5 & 3.25 & 7 \end{bmatrix} \quad (10.43)$$

The following matrix is not

$$\begin{bmatrix} 5 & -3 & 0 & 1 \\ 3 & 2 & -1 & 2 \\ 4 & 6 & 15 & 3 \\ 0.5 & -1.5 & 3.25 & 7 \end{bmatrix} \quad (10.44)$$

since the diagonal entry of the second row is 2 which is less than the sum of the absolute values of the rest of the entries in the second row.

17. Redo the above exercise only using the `sum` and `diag` functions from `numpy`: `from numpy import diag, sum`.
18. Carl Gustav Jakob Jacobi came up with an iterative method to solve a linear system of equations  $\mathbf{Ax} = \mathbf{b}$ , where  $\mathbf{A}$  is an  $n \times n$  matrix and both  $\mathbf{b}$  and  $\mathbf{x}$  are  $n \times 1$  vectors. Jacobi's strategy starts with an initial guess  $\mathbf{x}_0$  for the solution  $\mathbf{x}$  at iteration 0. The solution for the  $(k + 1)$ <sup>th</sup> iteration is then computed using

$$x^{k+1}(i) = \frac{1}{A(i, i)} \left( b(i) - \sum_{j=1; j \neq i}^n A(i, j) x^k(j) \right), \quad i = 1, 2, 3, \dots, n \quad (10.45)$$

Jacobi's strategy continues to compute  $x^k + 1$  as long as the solution has not converged i.e.

$$\|x^k + 1 - x^k\| \leq \epsilon. \quad (10.46)$$

Write a program that computes the the solution to a linear system of equations using Jacobi's method above.

# Chapter 11

## Functions

A function is a collection of program statements that perform a specific task. Functions are called from other programs, or from other functions. We have encountered a number of Python functions so far. The first is the `range` function, which is a function that generates a list of integers. Do you recall the syntax of the `range` statement? The statement

```
1 my_list = range(num_ints)
```

generates a list (`my_list`) of integers that has `num_ints` number of integers in the list (starting from 0). As you can see, the `range` function requires at least 1 scalar input (in the example above `num_ints`) and it generates a single list as the output (`my_list` in the example above).

We have also encountered the `numpy.linalg.inv` function, which generates the inverse of a given matrix. The `numpy.linalg.inv` function requires a single square matrix as input and it generates the inverse of this matrix as output. The statement

```
1 from numpy.linalg import inv  
2  
3 mat_inv = inv(mat)
```

will generate the matrix `mat_inv` which is the inverse of the matrix `mat`. The trigonometric functions `sin`, `cos`, `tan` and the exponential function `exp` (from the `math` module)

are other examples of functions that we have used so far. All of these functions require a single object as input and it produces a single object as output.

A special type of function we have used is the `lambda` function, which a function that generates other functions. The `lambda` function requires a few variables as inputs and it generates a function that will return a single value output. `lambda` functions can be useful in many cases, however often you will find the `lambda` function to be very limited when it come to data containers; or when you need more than one output from the function; or when the function needs to be more than 1 line of code.

## 11.1 Creating new functions

Programming will be a very difficult task if we could only make use of existing functions. Not only will our programs become very long and difficult to keep track of, but it will also be impossible for a group of programmers to develop different components of a new program simultaneously. Fortunately, all programming languages allow for creating new functions.

In Python, new functions are created by defining a function with the following syntax:

```
1 def function_name(Input1, Input2):  
2     function statements  
3     return Output1, Output2
```

The `def` statement is used to define a new function with a specified function name. The set of objects (inputs) sent to the function are enclosed in round brackets after the function name. These inputs are separated by a comma and a space (, ). Then the set of objects (outputs) that the function will generate are returned to the main program by the `return` statement. Different outputs are separated by a comma and a space (, ). These input and output objects can have any name, as long as you obey the object name rules. The function name must also obey the usual object and program name rules.



### Take Note:

Note the double colon (:) at the end of the `def` statement in line 1. This tells Python where the `program statements` inside the function start.  
If the double colon (:) is left out you will see an error message.

**Take Note:**

The rules for indentation for a function are the same as for the `for` loop, see Section 4.2.5.

Python requires 4 spaces before **all program statements** inside the function.

**Take Note:**

The `return` statement tells Python what information must be returned from the function as outputs. Without the `return` no information will be returned from the function. The `return` statement exits the function immediately and no code (in the function) after the `return` statement will be executed.

**More Info:**

PEP8 standard for function names:

- “lowercase with words separated by underscores as necessary to improve readability”, e.g. `my_function` and not `My_Function`.
- names should be at least 3 characters in length and descriptive.

The PEP8 standard is available at:

<http://www.python.org/dev/peps/pep-0008/>

The objects that you use within the function are local objects, i.e. these object values are only known inside the function. The values of the objects used in your function will not be available to the calling program (or in the *IPython Console*). If you want a object value to be available to a calling program, send the object back to the calling program by adding this object to the set of output objects.

I’ll now give numerous examples of functions. I’ll start of by writing a function that can add two numbers. My functions will need 2 inputs (the two numbers I want to add) and it will produce a single output (the sum of these two numbers). So I decide that the syntax of my function is:

```
1 num3 = add(num1, num2)
```

where the name `num3` is the sum of the names `num1` and `num2`. Now I need to write the function `add`:

```
1 def add(num1, num2):  
2     num3 = num1 + num2  
3     return num3
```

For this example, the function `add` contains only 3 program lines. The first line defines the inputs and the name of the function. The remaining program line in a function computes the sum and returns the results as an output. In this case a single program line is sufficient. Notice that inside the function, you assume that the input objects are available (i.e. these values will be passed in from the calling program, so you should not define these values). Any other names and objects that you require during computations have to be defined within the function. Also note that the names you use inside the function are independent from the names you use when calling the function.

New functions can be saved in the same file as your main program or in a completely separate file, it is up to you as the programmer how you want to consolidate / group different function and code. You can call the new function `add` from any program or other functions. You can also call the function from the *IPython Console*. I'll illustrate calling the function `add` from a program.

### 11.1.1 Example 1

In this example I will assume that the function `add` is being called from the same file as the main program:

```
1 # define the function in the same file  
2 # above the main program  
3 def add(num1, num2):  
4     return num1 + num2  
5  
6  
7 # ----- main program -----  
8 # Define the inputs that the function add requires  
9 var1 = 4.0  
10 var2 = 5.0  
11 # Call the function, passing the input names  
12 # var1 and var2 into the function and getting the  
13 # output var3 back  
14 var3 = add(var1, var2)
```

```
15 print ("The sum of " + str(var1) + " and " +
16     str(var2) + " is ", str(var3))
```

The function `add` is called in program line 14 and the output of the program listed above is:

```
1 The sum of 4 and 5 is 9
```

### 11.1.2 Example 2

In this example I will assume that the function `add` is being called from the main program in a different file. For this example lets assume that the function `add` has been saved in a file called `my_functions.py`:

```
1 # add function saved in my_functions.py
2 def add(num1, num2):
3     return num1 + num2
```

Then in the main program we need to import the `add` function, like we did with the `math` module:

```
1 # ----- main program -----
2 from my_functions import add
3
4
5 # Define the inputs that the function add requires
6 var1 = 4.0
7 var2 = 5.0
8 # Call the function, passing the input names
9 # var1 and var2 into the function and getting the
10 # output var3 back
11 var3 = add(var1, var2)
12 print ("The sum of " + str(var1) + " and " +
13         str(var2) + " is ", str(var3))
```

The function `add` is imported to the main program in line 3 and then called in line 11 and the output of the program listed above is:

```
1 The sum of 4 and 5 is 9
```

**⚠ Take Note:**

In this example the main program must be saved in the same location as the `my_functions.py` file.

**⚠ Take Note:**

Functions have a special type of variable called local variables. These variables only exist while the function is running. When a local variable has the same name as another variable (such as a global variable), the local variable hides the other. Sound confusing? Well, these next examples should help clear things up:

```
1 a = 4
2
3 def print_func():
4     a = 17
5     print "in  print_func a = ", a
6
7 print_func()
8 print "a = ", a
```

Which gives the output:

```
1 in print_func a = 17
2 a = 4
```

Variable assignments inside a function do not override global variables, they exist only inside the function. Even though `a` was assigned a new value inside the function, this newly assigned value was only relevant to `print_func`, when the function finishes running, and the `a`'s values is printed again (the originally assigned value 4).

## 11.2 Functions (Continued)

Let's consider another example. Suppose we want to write a function that takes two numbers as inputs, and then performs the simple operations addition, subtraction, multi-

plication and division to provide four outputs. I decide the function name is `arithmetic`, so the syntax of such a function is:

```
1 add, subtract, multiply, divide = arithmetic(num1, num2)
```

where the two scalar inputs are `num1` and `num2` and the four scalar outputs are `add`, `subtract`, `multiply` and `divide`. The function `arithmetic` follows:

```
1 # ----- function arithmetic -----
2 def arithmetic(var1, var2):
3     a = var1 + var2
4     s = var1 - var2
5     m = var1 / var2
6     d = var1 * var2
7     return a, s, m, d
8
9
10 # ----- main program -----
11 add, subtract, multiply, divide = arithmetic(3.0, 4.0)
12 print add
13 print subtract
14 print multiply
15 print divide
```

The output of this program will be:

```
1 7.0
2 -1.0
3 0.75
4 12.0
```



#### Take Note:

You should notice that the names used to call the function have nothing to do with the names used inside the function.

As the programmer, you can choose the data containers for the inputs and outputs of your function, i.e. scalar objects, lists, dictionaries, or `numpy` array's. As an example, let's assume you prefer that the input to your `arithmetic` function is a list of the two number:

```
1 # ----- function arithmetic -----
2 def arithmetic(numbers):
3     a = number[0] + number[1]
4     s = number[0] - number[1]
5     m = number[0] / number[1]
6     d = number[0] * number[1]
7     return a, s, m, d
8
9
10 # ----- main program -----
11 add, subtract, multiply, divide = arithmetic([3.0, 4.0])
12 print add
13 print subtract
14 print multiply
15 print divide
```

Notice that the input is now a single list object `numbers` that contain two components. The function `arithmetic` cannot be called using two inputs any more, it now has to be called using a single list containing two components as an input. The four scalar outputs could also be replaced by a single list object as well, by simply enclosing the returned outputs in square brackets:

```
1 # ----- function arithmetic -----
2 def arithmetic(numbers):
3     a = number[0] + number[1]
4     s = number[0] - number[1]
5     m = number[0] / number[1]
6     d = number[0] * number[1]
7     return [a, s, m, d]
8
9
10 # ----- main program -----
11 results = arithmetic([3.0, 4.0])
12 print results
```

To illustrate this last version of the `arithmetic` function, the output from the program will be:

```
[7.0, -1.0, 0.75, 12.0]
```

As you can see, I called the `arithmetic` function using the [3.0, 4.0] list as an input. I obtained a single list `results` as output, with the four computed values available in the four components.

The final simple function we'll write is a function that computes the components of a vector, if the length and angle of the vector is provided. Let's decide that the angle can be specified in either degrees or radians. I choose the name of the function as `decompose` (meaning a function that decomposes a vector into it's components). The syntax of the function is

```
1 comp_x, comp_y, message = decompose(length, angle, option)
```

where the inputs are the length of the vector (`length`), the vector's angle (`angle`) and the name `option`, which must distinguish between degrees or radians. The outputs are the two components `comp_x` and `comp_y`, and a string name `message` that can contain an error message. As the programmer, you decide how you want to use the `option` name. You can decide that the `option` name can have a value of 1 or 2, where 1 indicates the angle is expressed in degrees and 2 indicates the angle is expressed in radians. Or you can decide that the name `option` is a string object that can either contain a '`d`' to indicate degrees or a '`r`' to indicate radians. I'll use the latter option. If the `option` name doesn't contain either a '`r`' or '`d`', I set the `message` object equal to '`Incorrect Input`'.

Here follows the function `decompose`:

```
1 import math
2 from math import pi
3
4
5 # ----- function arithmetic -----
6 def decompose(mag, angle, option):
7     # Initialize the output
8     message = "Completed Successfully"
9     comp1 = None
10    comp2 = None
11
12    # Check if the option given is valid, if not
13    # change the message and return the outputs
14    if (option != "r") and (option != "d"):
15        message = "Invalid Input Option."
16    return comp1, comp2, message
17
```

```

18 # Check if the angle is expressed in degrees:
19 # if so, convert to radians
20 if option == "d":
21     angle = math.radians(angle)
22
23 # 1st component the cosine of angle times magnitude
24 comp1 = mag * math.cos(angle)
25 # 2nd component the sine of angle times magnitude
26 comp2 = mag * math.sin(angle)
27 # return the outputs
28 return comp1, comp2, message
29
30
31 # ----- main program -----
32 c_x, c_y, msg = decompose(10, pi/4, 'r')
33 print c_x, c_y, msg

```



### Take Note:

The `return` statement exists the function immediately and returns to the calling program, no code (in the function) after the `return` will be executed.

I called the new function from the *IPython Console* to produce the following output. I used both options i.e. expressed the angle as 45 degrees and as  $\frac{\pi}{4}$  radians. I also called the function using incorrect input. As you can see, the first two cases produce the same output (as it should) and the last case produces the error message. In this example the `decompose` function is saved in the `test.py` file.

```

1 In [#]: from test import decompose
2
3 In [#]: from math import pi
4
5 In [#]: decompose(10, pi/4, 'r')
6 Out[#]: (7.071068, 7.071068, 'Completed Successfully')
7
8 In [#]: decompose(10, 45, 'd')
9 Out[#]: (7.071068, 7.071068, 'Completed Successfully')
10
11 In [#]: decompose(10, 45, 1)
12 Out[#]: (None, None, 'Invalid Input Option.')
13
14 In [#]:

```

## 11.3 Function with Optional Inputs

Let us again consider the `range` function. You should recall that the `range` function can be used with additional inputs (i.e. the start integer and the step size):

```
1 my_list = range(start, num_ints, step)
```

These additional inputs are known as optional inputs. We can also create new functions with optional inputs using the following syntax:

```
1 def function_name(input1, input2, input3=value):  
2     function statements  
3     return output1, output2
```

Here the first two inputs (`input1` and `input2`) will be required inputs (and must be specified when calling the functions). The last input (`input3`) is the optional input, its is optional because when we create the function we give `input3` a default value. If `input3` is specified when calling the function then `input3` will equal the specified value, if `input3` is not specified when calling the function then `input3` will equal the default value. Let us consider a few examples to explain this concept further.

```
1 def test_convergence(error, tol=1E-6):  
2     if error < tol:  
3         return True  
4     return False  
5  
6  
7 print test_convergence(1E-8)  
8 print test_convergence(1E-8, tol=1E-10)
```

In this example we give `tol` the default value of 1E-6 when we created the function `test_convergence` (line 1). The first time we call `test_convergence` (line 7) with an error of 1E-8 we do not specify the optional input for `tol`. Thus in the function (line 1) `tol` is given the default value of 1E-6.

The second time we call `test_convergence` (line 8) with an error of 1E-8 we now specify the input for `tol`. Thus in the function (line 1) `tol` is given the value of 1E-10.

Lets consider one last example, you should by now be able to figure out the logic of this example:

```
1 def greet(person, show_title=False, initials_only=True):
2     title = ""
3     name = person["name"]
4     surname = person["surname"]
5
6     if show_title:
7         title = "Mrs "
8         if person["gender"] == "m":
9             title = "Mr "
10
11    if initials_only:
12        name = person["name"][0].upper()
13
14    print ("Hello " + title + name + " " +
15          surname + " and welcome.")
16
17
18 person = {"name": "John",
19            "surname": "Doe",
20            "gender": "m"}
21
22 greet(person)
23 greet(person, initials_only=False)
24 greet(person, show_title=True)
25 greet(person, initials_only=False, show_title=True)
```

which gives the following output:

```
1 Hello J Doe and welcome.
2 Hello John Doe and welcome.
3 Hello Mr J Doe and welcome.
4 Hello Mr John Doe and welcome.
```



#### Take Note:

All keyword arguments (optional inputs) MUST come after the required arguments (required inputs) when creating a function otherwise Python will give an error.

**More Info:**

See [http://en.wikibooks.org/wiki/Non-Programmer%27s\\_Tutorial\\_for\\_Python\\_2.6](http://en.wikibooks.org/wiki/Non-Programmer%27s_Tutorial_for_Python_2.6) for more examples on defining functions.

## 11.4 Gauss elimination using functions

I'll revisit the Gauss elimination program to further illustrate the use of functions. As the programmer, you can decide how to organise your program. In this case, I've decided to make use of two functions. The first, called `reduction`, is a function that performs the forward reduction step of the Gauss elimination process. The second function, called `backsub`, performs the back-substitution step. The inputs that the forward reduction step requires is the `numpy` matrix (`mat_a` in this case) and the `numpy` vector (`vec_a` in this case). The outputs that the forward reduction step produces are the modified matrix `mat_a` and modified vector `vec_a`. Hence the syntax of the `reduction` function is

```
1 mat_a, vec_a = reduction(mat_a, vec_a)
```

The `backsub` function takes the reduced matrix `mat_a` and the vector `vec_a` as input and it produces the solution to the linear system as output (`x_vals` in this case). Hence the syntax of the `backsub` function is

```
1 x_vals = backsub(mat_a, vec_a)
```

The final version of the Gauss elimination program listed in Section 10.10 now reads:

```
1 import time
2 import numpy as np
3
4
5 def reduction(mat, vec):
6     n = len(vec)
7     for i in range(n-1):
```

```
8     for j in range(i+1, n):
9         mult = mat[j, i] / mat[i, i]
10        vec[j] -= mult * vec[i]
11        for k in range(i+1, n):
12            mat[j, k] -= mult * mat[i, k]
13
14 def backsub(mat, vec):
15     n = len(vec)
16     solution = np.zeros((n, 1))
17     solution[n-1] = vec[n-1] / mat[n-1, n-1]
18     for i in range(n-2, -1, -1):
19         solution[i] = vec[i]
20         for j in range(i+1, n):
21             solution[i] -= mat[i, j] * solution[j]
22         solution[i] /= mat[i, i]
23     return solution
24
25
26 # Create matrix A and vector a
27 mat_a = np.random.rand(100, 100)
28 vec_a = np.random.rand(100, 1)
29 # make copies of A and a to use later
30 mat_a_copy = np.copy(mat_a)
31 vec_a_copy = np.copy(vec_a)
32
33 # Start the stopwatch
34 start = time.time()
35
36 # Call the forward reduction function
37 reduction(mat_a, vec_a)
38 # Call the back-substitution function
39 x_vals = backsub(mat_a, vec_a)
40
41 # Stops stopwatch and displays time in Command Window
42 stop = time.time()
43 print stop - start
44 # Starts stopwatch again
45 start = time.time()
46
47 # Solve system using inverse of A_copy*a_copy
48 x_vals = np.dot(np.linalg.inv(mat_a_copy), vec_a_copy)
49
50 # Stops stopwatch and displays time in Command Window
51 stop = time.time()
```

```

52 print stop - start
53 # Starts stopwatch third and final time
54 start = time.time()
55
56 # Solve system using backslash operation
57 x_vals = np.linalg.solve(mat_a_copy, vec_a_copy)
58 # Stops stopwatch and displays time in Command Window
59
60 stop = time.time()
61 print stop - start

```

The performance of the new version of the Gauss elimination program is similar to the original program. This example is useful as an educational example, but is probably not realistic. I can't imagine a scenario where we would like to only perform forward reduction or only back-substitution. Rather, a single function `gauss` makes more sense. It takes the matrix `mat_a` and vector `vec_a` as input and it produces the vector `x_vals` as output i.e.

```
1 x_vals = gauss(mat_a, vec_a)
```

```

1 import time
2 import numpy as np
3
4
5 def gauss(mat, vec):
6     n = len(vec)
7     solution = np.zeros((n, 1))
8     # reduction
9     for i in range(n-1):
10         for j in range(i+1, n):
11             mult = mat[j, i] / mat[i, i]
12             vec[j] -= mult * vec[i]
13             for k in range(i+1, n):
14                 mat[j, k] -= mult * mat[i, k]
15     # backsub
16     solution[n-1] = vec[n-1] / mat[n-1, n-1]
17     for i in range(n-2, -1, -1):
18         solution[i] = vec[i]
19         for j in range(i+1, n):
20             solution[i] -= mat[i, j] * solution[j]
21         solution[i] /= mat[i, i]

```

```
22     return solution
23
24
25 # Create matrix A and vector a
26 mat_a = np.random.rand(100, 100)
27 vec_a = np.random.rand(100, 1)
28 # make copies of A and a to use later
29 mat_a_copy = np.copy(mat_a)
30 vec_a_copy = np.copy(vec_a)
31
32 # Start the stopwatch
33 start = time.time()
34
35 # Call the gauss function
36 x_vals = gauss(mat_a, vec_a)
37
38 # Stops stopwatch and displays time in Command Window
39 stop = time.time()
40 print stop - start
41 # Starts stopwatch again
42 start = time.time()
43
44 # Solve system using inverse of A_copy*a_copy
45 x_vals = np.dot(np.linalg.inv(mat_a_copy), vec_a_copy)
46 # Stops stopwatch and displays time in Command Window
47
48 stop = time.time()
49 print stop - start
50 # Starts stopwatch third and final time
51 start = time.time()
52
53 # Solve system using backslash operation
54 x_vals = np.linalg.solve(mat_a_copy, vec_a_copy)
55 # Stops stopwatch and displays time in Command Window
56
57 stop = time.time()
58 print stop - start
```

## 11.5 Numerical integration using functions

I conclude this section on functions by giving a final example. Consider the numerical integration program in Section 8. I propose that each of the four methods of integration can become a function. So all we have to determine is the inputs and outputs of these functions. As the programmer, I decide that as outputs I want the numerical value of the integral as well as the required number of intervals. Regarding the inputs, here we have no choice. A function that must perform numerical integration needs i) the function to be integrated, ii) the lower and upper bounds of integration iii) the start number of intervals and iv) the required accuracy. All other values can be defined within the function. So the syntax of the integration functions is

```
1 integral, num = left_point(func, lower, upper, acc)
```

where the inputs are the function `func`, the lower bound `lower`, upper bound `upper`, and the required accuracy `acc`. The initial number of intervals `num` is created as an optional input to the functions. The function will then provide the integral `integral` and the required number of intervals `num`. Here's the listing of the modified numerical integration program and only the `left_point` function. The other functions only differ in the details of the area computation.

```
1 from math import *
2
3
4 def left_point(func, lower, upper, acc, num=10):
5     # Define initial Error greater as acc,
6     # to enter while loop
7     error = 1
8     # Set Int to zero to have a value the first time
9     # the while loop is executed
10    integral = 0
11    # Start while loop that checks if consecutive values
12    # converged
13    while error > acc:
14        # Compute the interval size Delta_x
15        delta_x = (upper - lower) / num
16        # Make a copy of the previous integral just before
17        # the program section starts that computes the new value
18        integral_old = integral
19        integral = 0
```

```
20     # For loop to compute area of each rectangle and
21     # add it to integral
22     for i in range(num):
23         x = lower + i * delta_x
24         integral += delta_x * func(x)
25     # Compute Error i.e. difference between consecutive
26     # integrals
27     error = abs(integral - integral_old)
28     # Double the number of intervals
29     num *= 2
30     return integral, num
31
32 def right_point(func, lower, upper, acc, num=10):
33 .
34 .
35 .
36
37 def mid_point(func, lower, upper, acc, num=10):
38 .
39 .
40 .
41
42 def trapezium(func, lower, upper, acc, num=10):
43 .
44 .
45 .
46
47
48 def main():
49     # Get user inputs
50     f_string = input("Please enter function to be integrated: ")
51     func = eval("lambda x: " + f_string)
52     lower = input("Enter the lower bound of integration: ")
53     upper = input("Enter the upper bound of integration: ")
54     acc = input("Enter the required accuracy: ")
55
56     # Test if lower or upper is a string
57     if isinstance(lower, str):
58         upper = eval(upper)
59     if isinstance(upper, str):
60         upper = eval(upper)
61
62     # Display integration options
63     print
```

```
64     print "1: Left-point 2: Right-point 3: Mid-point"
65     print "4: Trapezium 5: Quit program"
66     method = input("Please enter your option: ")
67
68     # My program repeats until the user inputs option 5
69     while method != 5:
70         #Since the number of intervals change during program
71         # execution, reset N to the initial number of intervals
72         if method == 1:      # left-point method
73             integral, num = left_point(func, lower, upper, acc)
74         elif method == 2:    # right-point method
75             integral, num = right_point(func, lower, upper, acc)
76         elif method == 3:    # mid-point method
77             integral, num = mid_point(func, lower, upper, acc)
78         elif method == 4:    # trapezium method
79             integral, num = trapezium(func, lower, upper, acc)
80
81         # Display answer
82         print ("Integrating " + f_string + " between " +
83                 str(lower) + " and " + str(upper) + " = " +
84                 str(integral))
85         print (str(num) +
86                 " intervals required for an accuracy of " +
87                 str(acc))
88         # Display options again
89         print
90         print "1: Left-point 2: Right-point 3: Mid-point"
91         print "4: Trapezium 5: Quit program"
92         method = input("Please enter your option: ")
93
94     # Outside while loop, user entered option 5 = Quit
95     # Displays a friendly message.
96     print "Thanks for using the program. Have a good day."
97
98
99 if __name__ == "__main__":
100     main()
```

## 11.6 Comment statements and code documentation

It is good programming practise to use comment statements liberally throughout your programs. These comments should explain the program logic. Their main purpose is to remind you what you were thinking when you first developed the program. It also help other programmers to use your programs (and helps your lecturer to grade your assignments). Using descriptive names reduces the need for extensive comments.

Comment statements are included in any Python program by typing a hash sign (#), followed by your comment. An entire program line can be a comment, or you can append the comment to the end of a normal program line. One of the previous programs is repeated below, with comment statements added:

```
1 def pi_taylor_series(accuracy=1E-3):
2     """
3         Taylor series approximation to pi using a while loop
4
5     Optional Input:
6         accuracy (default = 1E-3)
7
8     Returns:
9         k: number of terms needed to compute pi
10        taylor_pi: taylor series approximation of pi
11    """
12
13    # Initialize k to zero
14    k = 0
15    # Compute 1st term in Taylor series
16    taylor_term = (4.0 * (-1)**k) / (2*k + 1)
17    taylor_pi = taylor_term
18    # Start of while loop
19    while abs(taylor_term) > accuracy:
20        k += 1
21        taylor_term = (4.0 * (-1)**k) / (2*k + 1)
22        taylor_pi += taylor_term
23    return k, taylor_pi    # return approximation
```

The second type of information you can add to your code is called docstrings (documentation strings), this is shown in the example above in lines 2–11. Docstrings are used to add documentation to your modules and functions which will help you (or anyone else who is using your code) remember how to use that function. The information you have

been seeing in the *Object inspector* window of *Spyder* is exactly this, docstrings added to the code. The information you have been seeing in the *IPython Console*, when you type `object?`, is docstrings added to the code.

You add a docstring to your functions by enclosing the information in 3 double-quotation marks. This docstring must be the first thing appearing in the function, after you define the function (i.e. starts directly after the function definition). Save the above example in a file (in my case it is saved in `test.py`) and import the `pi_taylor_series` function into the *IPython Console*. If you now type `pi_taylor_series?` you should see the following output:

```
1 In [#]: from test import pi_taylor_series
2
3 In [#]: pi_taylor_series?
4 .
5 .
6 .
7 Taylor series approximation to pi using a while loop
8
9 Optional Input:
10     accuracy (default = 1E-3)
11
12 Returns:
13     k: number of terms needed to compute pi
14     taylor_pi: taylor series approximation of pi
15
16 In [#]: pi_taylor_series()
17 Out[#]: (2000, 3.1420924036835256)
18
19 In [#]:
```

## 11.7 Exercises

1. Write a function `mysphere` which takes the radius  $r$  of a sphere as input. The function must then calculate the surface area and volume of the sphere. The surface area and volume of the sphere must then be returned as outputs of the function.

The surface area of a sphere is given by:

$$4\pi r^2. \quad (11.1)$$

The volume a sphere is given by:

$$\frac{4}{3}\pi r^3. \quad (11.2)$$

2. Consider a rectangular section with the x-axis aligned with the width  $b$  of the section and the y-axis aligned with the height  $h$  of the section.

Write a function `rectmi` that takes the width  $b$  and height  $h$  of a rectangular section as input. The function must calculate the moment of inertia around the centroidal y-axis  $\bar{I}_y$ , x-axis  $\bar{I}_x$  and polar  $\bar{I}_z$  with

$$\begin{aligned} \bar{I}_x &= \frac{1}{12}bh^3 \\ \bar{I}_y &= \frac{1}{12}hb^3 \\ \bar{I}_z &= \frac{bh}{12}(b^2 + h^2) \end{aligned} \quad (11.3)$$

3. Write a function `twodice` that takes as input the number of double dice throws. For every dubbel throw (two dices that are thrown simultaneously) the function must then generate two random integers `dice1` and `dice2` between 1 and 6. The sum of values of the two dices namely `dice1+dice2` must then be calculated. The function must then keep count of how many times every possible sum i.e. between 2 and 12 was thrown using a vector. The first entry of the vector must contain the number of times a 2 was thrown, the second entry the number of times a 3 was thrown etc. The function must then return the vector containing the number of counts as output.
4. Write a function `my_multiply` that takes as input a matrix  $\mathbf{A}$  and a matrix  $\mathbf{B}$ . The function must then calculate the matrix-matrix product  $\mathbf{A} \times \mathbf{B}$  and return the matrix-matrix product  $\mathbf{A} \times \mathbf{B}$  as output.
5. Write a function `vector_rotate` that accepts a vector  $\mathbf{x}$  of length 2 and an angle  $\theta$  as input. The function must then compute and return the rotated vector  $\mathbf{x}_{rotated}$  as output.

In two dimensions the rotation matrix  $\mathbf{Q}$  is given by:

$$\mathbf{Q} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \quad (11.4)$$

The rotated vector  $\mathbf{x}_{rotated}$  can then be computed by

$$\mathbf{x}_{rotated} = \mathbf{Q}\mathbf{x} \quad (11.5)$$

Use function `my_multiply` from the previous question to compute the matrix-vector product.

Plot the un-rotated vector and then the rotated vector. The tail of the each vector is given by the origin  $[0, 0]$  and the head of each vector by the components given by  $\mathbf{x}$  or  $\mathbf{x}_{rotated}$ .

For interesting sake: A rotation matrix is an orthogonal matrix (i.e. square matrix of which the transponent is equal to the inverse) whose determinant is equal to 1. A rotation matrix does not change the length of a vector but only rotates it.

6. Revisit as many of the previous chapter exercises as possible and change the programs to work as functions. Instead of specifying a object or vector or asking the user to enter a number, string or vector; specify it as inputs of a function and determine the appropriate outputs.
7. In 1972 Marsaglia derived an elegant method for generating random points uniformly on the surface of a unit sphere. The method consists of picking  $x_1$  and  $x_2$  from independent uniform distributions between -1 and 1. The points  $x_1$  and  $x_2$  are accepted if  $x_1^2 + x_2^2 < 1$  otherwise the points  $x_1$  and  $x_2$  are rejected and picked again. The x, y and z coordinates of a point randomly distributed on a unit sphere are then

$$\begin{aligned}x &= 2x_1\sqrt{1 - x_1^2 - x_2^2} \\y &= 2x_2\sqrt{1 - x_1^2 - x_2^2} \\z &= 1 - 2(x_1^2 + x_2^2)\end{aligned}\tag{11.6}$$

when  $x_1$  and  $x_2$  have been accepted.

Write a function `rnd_sphere` that takes the number of points  $N$  as input argument. The function must then return a  $N \times 3$  matrix that contains the x-coordinates in the first column, y-coordinates in the second column and z-coordinates in the third column of the  $N$  randomly distributed points on the unit sphere.

8. Winograd introduced an algorithm for multiplying two  $n \times n$  matrices  $A$  and  $B$

$$C = AB$$

for  $n$  an even number.

The Winograd algorithm works as follows: The product of the  $i$ th row and  $j$ th column is given by:

$$\begin{aligned}a(i) &= \sum_{k=1}^{n/2} A(i, 2k-1) A(i, 2k) \\b(i) &= \sum_{k=1}^{n/2} B(2k-1, j) B(2k, j) \\C(i, j) &= \sum_{k=1}^{n/2} [A(i, 2k-1) + B(2k, j)] [A(i, 2k) + B(2k-1, j)] - a(i) - b(j)\end{aligned}$$

Write a function `winomult` that takes two matrices  $A$  and  $B$  as input. The function must then return  $C = AB$  using the Winograd algorithm. Assume the matrices  $A$  and  $B$  are square and  $n$  even.



# Chapter 12

## File handling

The world of computers will be much simpler if we could do all our work using a single application. But this is not the case, so we have to make sure that whatever programs we are using, we can exchange information with other programs. Python is no exception. Even though we might be very accomplished Python programmers, we at times require ways to access data from other sources, or generate data for other applications.

Also once a computer is switched off all the computed results are lost as everything is stored in temporary memory, except for the programs we have written. Every time you switch your computer on and want to look at the results you would have to run a program. Can you imagine doing that if the calculations require a day or two of computations e.g. computational mechanics (CFD or FEM) or the optimization of a large cities road network.

There are so many ways to read data into Python and write data from Python that I can't even begin to cover them all. In this section I'll briefly cover a few commonly used methods to read data into Python, or to generate files so that other programs (e.g. *Open Office Calc*) can get access to data that we generated in Python. The data would then still be available once a computer is switched off and on.

### 12.1 The `numpy.save` and `numpy.load` commands

The `numpy.save` command is used to save a `numpy.array` to a “.npy” file that can later be retrieved using the `numpy.load` command. This is useful for when your program takes days to execute and you want to make sure the information is not lost if the power fails or your PC shuts off during the execution of your program.

The `numpy.load` command can be used to load a `numpy.array` from a “.npy” file into Python’s memory. The requirements are that the file must have been created using the `numpy.save` function. The syntax of the `numpy.load` command is:

```
1 from numpy import load  
2  
3 my_array = load("filename.npy")
```

You have to specify both the filename and the “.npy” extension. The content of the file is then saved in the variable `my_array`.



#### Take Note:

In the example above, it is assumed that the file "filename.npy" is saved in the same location as your Python (.py) file executing this example. See section ?? on how to load files from a different directory.

Suppose instead you wrote a Python program that produced the answers to some problem. If you want to produce a “.npy” output file that contains the output of your program, you can use the `numpy.save` command. The syntax of the `numpy.save` command is:

```
1 from numpy import array, save  
2  
3 my_array = array([12, 14, 22, 55], dtype=float)  
4 save("filename", my_array)
```



#### Take Note:

In the example above the file "filename.npy" will be saved in the same location as your Python (.py) file executing this example. See section ?? on how to save files to a different directory.

You don’t need to specify the “.npy” extension after the filename when using the `numpy.save` function. Lets look at the following examples

```
1 In [#]: from numpy import array, load, save  
2  
3 In [#]: mat = array([[1, 2, 3],
```

```
4      ....:          [2, 5, 2],  
5      ....:          [9, 2, 8]], dtype=float)  
6  
7 In [#]: mat  
8 Out[#]:  
9 array([[ 1.,  2.,  3.],  
10        [ 2.,  5.,  2.],  
11        [ 9.,  2.,  8.]])  
12  
13 In [#]: save("tempfile", mat)  
14  
15 In [#]: load("tempfile.npy")  
16 Out[#]:  
17 array([[ 1.,  2.,  3.],  
18        [ 2.,  5.,  2.],  
19        [ 9.,  2.,  8.]])  
20  
21 In [#]: new_mat = load("tempfile.npy")  
22  
23 In [#]: new_mat  
24 Out[#]:  
25 array([[ 1.,  2.,  3.],  
26        [ 2.,  5.,  2.],  
27        [ 9.,  2.,  8.]])  
28  
29 In [#]:
```

## 12.2 The `numpy.savetxt` and `numpy.loadtxt` commands

Unfortunately output files generated using the `numpy.save` are not accessible by any other program other than Python + `numpy`. A slightly more general method to read data into Python is the `numpy.loadtxt` statement. The syntax of the `numpy.loadtxt` statement is:

```
1 from numpy import loadtxt  
2  
3 my_array = loadtxt("filename.extension", delimiter=',')
```

**Take Note:**

In the example above, it is assumed that the file "filename.extension" is saved in the same location as your Python (.py) file executing this example. See section ?? on how to load files from a different directory.

where the `delimiter` optional input is the special type of character that is present between the columns of data and any delimiter is allowed. Common delimiters are the space ' ', a comma ',', or the tab '\t'. As an example, if data is available in a file `data.txt` and columns are delimited with commas, this data can be read into a variable `input_data` with the statement:

```
1 from numpy import loadtxt  
2  
3 my_array = loadtxt("data.txt", delimiter=',')
```

Similarly, Python can generate an output file using the `numpy.savetxt` statement. As before, any delimiter is valid, but the usual ones are the space, comma or tab. The syntax of the `numpy.savetxt` statement is

```
1 from numpy import savetxt  
2  
3 savetxt("filename.txt", my_array, delimiter=',')
```

**Take Note:**

In the example above the file "filename.txt" will be saved in the same location as your Python (.py) file executing this example. See section ?? on how to save files to a different directory.

The above statement will save the variable `my_array` into the file `filename.txt` and will use the specified delimiter between the columns of data. Lets look at the following examples

```
1 In [#]: from numpy import array, loadtxt, savetxt  
2  
3 In [#]: mat = array([[1, 2, 3],  
4 ....., [2, 5, 2],  
5 ....., [9, 2, 8]], dtype=float)
```

```
6
7 In [#]: mat
8 Out[#]:
9 array([[ 1.,  2.,  3.],
10        [ 2.,  5.,  2.],
11        [ 9.,  2.,  8.]])
12
13 In [#]: savetxt("test.txt", mat, delimiter=', ')
14
15 In [#]: loadtxt("test.txt", delimiter=', ')
16 Out[#]:
17 array([[ 1.,  2.,  3.],
18        [ 2.,  5.,  2.],
19        [ 9.,  2.,  8.]])
20
21 In [#]: new_mat = loadtxt("test.txt", delimiter=', ')
22
23 In [#]: new_mat
24 Out[#]:
25 array([[ 1.,  2.,  3.],
26        [ 2.,  5.,  2.],
27        [ 9.,  2.,  8.]])
28
29 In [#]:
```

The format of the data (save in the text output file) can also be modified using the optional input `fmt=""`, as shown in the syntax below.

```
1 from numpy import savetxt
2
3 savetxt("filename.txt", my_array, delimiter=', ', fmt=".3e")
```

The text output file (from the example above, using `fmt=".3e"`) contains the following information, which can now easily be read into another program (e.g. *Open Office Calc*):

```
1 1.000e+00, 2.000e+00, 3.000e+00
2 2.000e+00, 5.000e+00, 2.000e+00
3 9.000e+00, 2.000e+00, 8.000e+00
```

## 12.3 The csv module

The problem with the `numpy` methods, discussed above, is that the data needs to be stored in a `numpy` array object. If you have both string and numerical information, stored in either a list or tuple object, then you need to use the `csv` module to read and write the information from and to a file. The following example shows how to write information to a file using the `csv` module:

```
1 import csv
2
3
4 # create the data to write to the file
5 headings = ['x1', 'x2', 'x3', 'f(x)']
6 data = [[0, 1, 1, 10],
7         [9, 2, 3, 21],
8         [8, 2, 0, '-']]
9
10 # open the file [wb -> write binary mode]
11 with open('my_file.csv', 'wb') as csvfile:
12     # initialize the csv.writer object
13     csvwriter = csv.writer(csvfile, delimiter=',')
14     # write the headings for the data
15     csvwriter.writerow(headings)
16     # write each row of the data list to file
17     for row_data in data:
18         csvwriter.writerow(row_data)
```

And this example following will show how to read that same information from the specified file.

```
1 import csv
2
3
4 # initialize an empty data list
5 data = []
6 # open the file [rb -> read binary mode]
7 with open('my_file.csv', 'rb') as csvfile:
8     # initialize the csv.reader object
9     csvreader = csv.reader(csvfile, delimiter=',')
10    # read each line from the file
```

```
11     for cnt, row in enumerate(csvreader):
12         if cnt == 0:
13             # store the headings
14             headings = row
15         else:
16             # store the data
17             data.append(row)
18
19 # display the headings and data
20 print headings
21 print data
```

## 12.4 Additional methods to read and write data

The standard built-in method in Python to read and write data is to use the `open` command in Python. The `open` command has various functions associated to it, like the `readline` and `write` functions. I'll now show one example of how you can write a string template (populated with data) to a output file using this built-in method:

```
1 from numpy import array
2
3
4 TEMPLATE = \
5 """-
6 This log file contains:
7 x1 \t x2 \t x3 \t f(x)
8 -----
9 %s
10 -----"""
11
12 LINE_ENTRY = "%.3f \t %.3f \t %.3f \t %.3f\n"
13
14
15 def write_file(data):
16     # initialize the string that contains the values from
17     # the data array
18     body = ""
19     # loop over each row vector in the data array (matrix)
20     for vector in data:
21         # use the LINE_ENTRY string template to format the
```

```
22     # values of the row vector into a string and join
23     # this to body
24     body += LINE_ENTRY % tuple(vector)
25     # remove the last newline character
26     body = body.rstrip("\n")
27     # populate the log file TEMPLATE with body
28     log_info = TEMPLATE % body
29     # display the log file info to the screen
30     print log_info
31     # open the data.txt file (write mode [w]) to write the
32     # log_info
33     f_h = open("data.txt", "w")
34     # write the log_info into the data.txt file
35     f_h.write(log_info)
36     # close the data.txt file
37     f_h.close()
38
39
40 def main():
41     # create a data array for testing
42     data = array([[0.001, 0.011, 2.001, 10.123],
43                  [0.002, 0.021, 3.001, 21.103],
44                  [0.003, 0.031, 4.001, 32.123],
45                  [0.004, 0.041, 5.001, 43.123],
46                  [0.005, 0.051, 6.001, 54.120],
47                  [0.006, 0.061, 7.001, 65.123],
48                  [0.007, 0.071, 8.001, 76.123],
49                  [0.008, 0.081, 9.001, 87.023]], dtype=float)
50     # call the function to write the data array to a file
51     write_file(data)
52
53
54 if __name__ == "__main__":
55     main()
```

And now this example that follows shows how to read the same information from the specified file:

```
1 from numpy import array
2
3
4 def read_file():
```

```
5     # initialize the data list to store all data
6     data = []
7     # open the data.txt file (read mode [r]) to read from
8     # the file
9     f_h = open("data.txt", "r")
10    # read the first line from the file
11    line = f_h.readline()
12    cnt_line = 1
13    # continue reading from the file until it reaches the
14    # end of the file
15    while line:
16        # read the next line from the file
17        line = f_h.readline()
18        cnt_line += 1
19        # skip the file header
20        if cnt_line > 4:
21            # skip the last line
22            if "--" in line or not line:
23                continue
24            # split the string by any whitespace
25            # character
26            line = line.split()
27            vector_list = []
28            for str_val in line:
29                vector_list.append(float(str_val))
30            data.append(vector_list)
31            # convert the list to an array and return it
32            data = array(data, dtype=float)
33            return data
34
35
36 if __name__ == "__main__":
37     data = read_file()
38     print data
```

## 12.5 Exercises

1. Write a function that writes a matrix to a file. The function inputs are the name of the file and the matrix. Use the `numpy.save` function to save the matrix to a file.
2. Redo the previous exercise using the `numpy.savetxt` function to save the matrix to a file.

3. Write a function that reads a matrix to a file. The function inputs are the name of the file. The function must then return the matrix as output.
4. Write a function that writes the contents of a list of dictionaries to a text file. The text file must contain as headings the common keywords of the dictionaries. The file must be a comma separated file (CSV). Each following row must then contain the corresponding information of each respective keyword.

Hence for a list of dictionaries the output of the file would be:

```
1 Name, Age, Gender
2 Piet, 25, M
3 Thabo, 23, M
4 Tanya, 24, F
5 John, 30, M
6 Thandi, 27, F
```

5. Write a function that reads the contents of the CSV as created above and stores it in a list of dictionaries.
6. Write a function `randomdata` that takes an integer number  $n$  and a string `filename` as input. The function `randomdata` must then generate a  $n \times 1$  vector of normally (uniform) distributed random numbers (type `help(numpy.random)` for additional help on normally distributed random numbers) and save it to an ASCII file. The name of the ASCII file is the user specified `filename` with a `.data` extension i.e. `filename.data`.

Write a second function `processdata` that takes only a string as input which specifies the name of a file. The function `processdata` must then load the data from the user specified file. The output of the function is the following descriptive statistical measures,

- average (type `help(numpy.mean)`)
- standard deviation (type `help(numpy.std)`)
- median (type `help(numpy.median)`)
- variance (type `help(numpy.var)`)

# Chapter 13

## Graphs

### 13.1 2D Graphs

Before you can draw a graph of any data, the data must be available. The data can either be available in some file (maybe it is experimental data you obtained from a colleague) or it is described by some mathematical function. Regardless of where the data comes from, before you can plot it, you must have it.

I'll usually illustrate plotting by using simple analytical functions. As a first example, let's plot the function

$$f(x) = \sin(x) \quad x \in [0; 2\pi] \quad (13.1)$$

For this example, the independent variable is  $x$ . So let's generate data points along the x-axis evenly spaced between 0 and  $2\pi$ . You've already seen the two methods available in Python:

1. If you know the number of data points you wish to generate, use the `numpy.linspace` command. The following example creates  $N$  evenly spaced data points between the specified lower bound LB and upper bound UB:

```
1 from numpy import linspace  
2  
3 x_vals = linspace(x_lower, u_upper, 200)
```

The first entry in the vector `x` will equal `x_lower` and the  $N$ -th entry will equal `x_upper`.

2. If instead you know the required spacing between data points, let's say `delta_x`, rather use the command `numpy.arange`:

```
1 from numpy import arange  
2  
3 x_vals = arange(x_lower, u_upper, delta_x)
```

In this case, the first entry of `x` will equal `x_lower`, the second entry will equal `x_lower + delta_x` and so on. The last entry of `x` will either be less than or equal to `x_upper`.

Instead of writing the program one line at a time, I present the complete program below and then I'll explain the program lines:

```
1 from math import pi  
2 from numpy import linspace, sin  
3 from matplotlib import pyplot as plot  
4  
5  
6 def plot_sin():  
7     x_vals = linspace(0, 2*pi, 200)  
8     y_vals = sin(x_vals)  
9  
10    plot.plot(x_vals, y_vals)  
11  
12  
13 def main():  
14     plot.figure(1)  
15     plot_sin()  
16     plot.show()  
17  
18  
19 if __name__ == "__main__":  
20     main()
```

The module used for plotting in Python is the `matplotlib` module, specifically the `pyplot` module from the `matplotlib` package. In program line 3 I import `pylab` as `plot`. In program line 7 I define the vector `x_vals`. The corresponding `y_vals` vector is generated in program line 8. Since `x_vals` is a vector, `sin(x_vals)` results in a vector of the same length as `x_vals`.

**Take Note:**

When doing more advanced computations of `numpy` arrays, such as `sin`, `cos`, etc. you need to import these functions from the `numpy` module (as shown in line 2 in the above example).

Program line 14 opens a new plotting window. This statement is not absolutely necessary, but I prefer to explicitly decide in which figure window I want my plot to appear. If you want to plot many figures simultaneously, the `figure` command becomes a necessity. Program line 10 completes the program but plotting the `x_val` vector vs. the `y_val` vector. The `plot` command can only be used for 2D plotting. Also, the two vectors that you are plotting against each other must have the same length. In this example this will be the case since program line 8 will always create the `y_val` such that it has the same length as vector `x_val`. In program line 16, the `show` command is used to display all the figure windows with the respective plots. The program listed above produces the graph depicted in Figure 13.1.

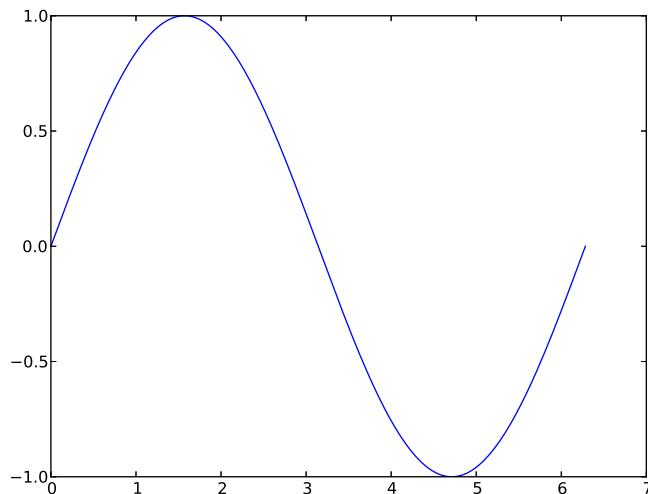


Figure 13.1: 2D graph of  $f(x) = \sin(x)$  for  $x \in [0; 2\pi]$  produced using the `plot` command.

### 13.1.1 Graph annotation

In my opinion, the graph seems incomplete. We need to describe the x and y axes and maybe add a title. This is achieved by the `xlabel`, `ylabel` and `title` statements.

```
10  plot.xlabel("x")
11  plot.ylabel("f(x)")
```

```
12 plot.title("Graph of the function f(x)=sin(x)")
```

As the programmer, you decide on the descriptions you want to add to your graph. Remember to include the descriptions in single inverted quotes. Another feature I don't particularly like in Figure 13.1 is the x and y axes limits i.e.  $x \in [0; 7]$  and  $y \in [-1; 1]$ . The `axis` command is used to explicitly define new limits i.e. the statement

```
1 from matplotlib import pyplot as plot  
2  
3 plot.axis([x_min, x_max, y_min, y_max])
```

will only show the graph for  $x \in [x_{\min}; x_{\max}]$  and  $y \in [y_{\min}; y_{\max}]$ . Program line 8 for the example program now reads

```
13 plot.axis([0, 2*pi, -1.1, 1.1])
```

If you make these changes and view the graph on the computer screen, it is acceptable. However, if you attempt to include this graph in a document (such as these notes), you might find that the text appears too small compared to the graph itself. In such a case, you can change the font size by specifying an additional argument for the `xlabel`, `ylabel` and `title` statements. I'll illustrate below, where I present the complete program:

```
1 from math import pi  
2 from numpy import linspace, sin  
3 from matplotlib import pyplot as plot  
4  
5  
6 def plot_sin():  
7     x_vals = linspace(0, 2*pi, 200)  
8     y_vals = sin(x_vals)  
9  
10    plot.plot(x_vals, y_vals)  
11  
12    plot.xlabel("x", fontsize=14)  
13    plot.ylabel("f(x)", fontsize=14)  
14    plot.title("Graph of the function f(x)=sin(x)", fontsize=14)  
15    plot.axis([0, 2*pi, -1.1, 1.1])
```

```
16  
17  
18 def main():  
19     plot.figure(1)  
20     plot_sin()  
21     plot.show()  
22  
23  
24 if __name__ == "__main__":  
25     main()
```

Do you see how I used the `fontsize` optional input to increase the size of the text in the figure? After these changes, the graph now appears as in Figure 13.2. A significant improvement if you ask me.

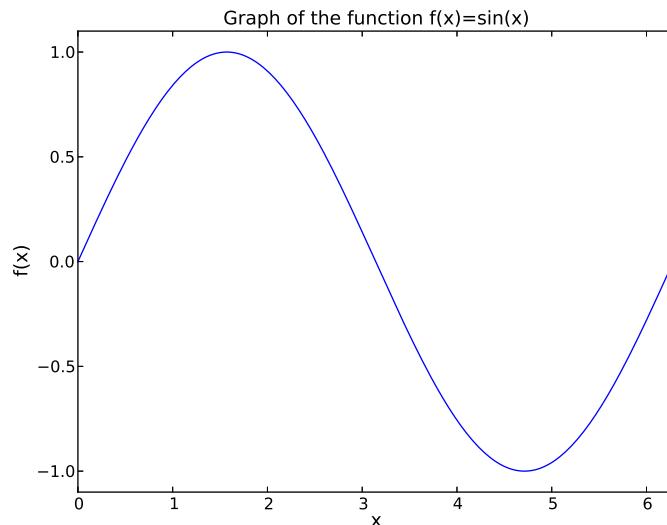


Figure 13.2: Improved 2D graph of  $f(x) = \sin(x)$ .

Other `matplotlib` commands that can be used to annotate your graphs is the `text` command. These commands are used to add any description to your graph. The syntax of the `text` command is

```
15 from matplotlib import pyplot as plot  
16  
17 plot.text(x_coord, y_coord,  
18             "The text string you want to add to your graph.")
```

where `x_coord` and `y_coord` are the  $x$  and  $y$  coordinates of the point where the text string must appear within your graph. As an example, if we add the statements

```
15 plot.text(0.5, -0.8,
16             "I can add text wherever I want.",
17             fontsize=14)
18 plot.text(1.5, 0.9, "Top")
19 plot.text(4.5, -0.9, "Bottom")
```

to the previous program, it produces the graph depicted in Figure 13.3. As before, setting the font size is optional. The first sentence uses a font size of 14pt, while the last two text statements uses the default of 12pt.

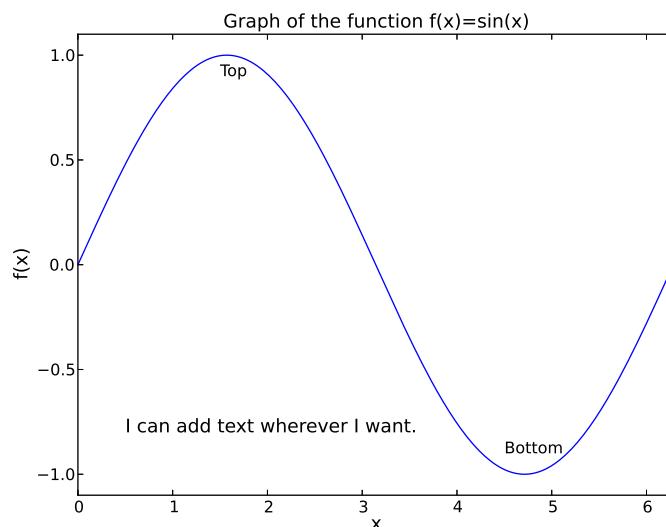


Figure 13.3: Graph of  $f(x) = \sin(x)$  with examples of `text` statements.

### 13.1.2 Multiple plots and plot options

This was just the beginning. `matplotlib` is capable of much more. Import `pylab` from `matplotlib` and type `help(pylab.plot)` to see all the additional arguments for the `plot` command. Here's the help output as displayed in the Command Window:

```
1 .
2 .
3 .
```

```
4 Plot lines and/or markers to the
5 :class:`~matplotlib.axes.Axes`. *args* is a variable length
6 argument, allowing for multiple *x*, *y* pairs with an
7 optional format string. For example, each of the following is
8 legal::
9
10    plot(x, y)          # plot x and y using default line style
11                      # and color
12    plot(x, y, 'bo')    # plot x and y using blue circle markers
13    plot(y)             # plot y using x as index array 0..N-1
14    plot(y, 'r+')      # ditto, but with red plusses
15
16 If *x* and/or *y* is 2-dimensional, then the corresponding
17 columns will be plotted.
18
19 An arbitrary number of *x*, *y*, *fmt* groups can be
20 specified, as in::
21
22     a.plot(x1, y1, 'g^', x2, y2, 'g-')
23
24 Return value is a list of lines that were added.
25
26 The following format string characters are accepted to control
27 the line style or marker:
28
29 ====== ======
30 character      description
31 ====== ======
32 '---'          solid line style
33 '--.'          dashed line style
34 '-.'          dash-dot line style
35 ':'          dotted line style
36 '.'          point marker
37 ','          pixel marker
38 'o'          circle marker
39 'v'          triangle_down marker
40 '^'          triangle_up marker
41 '<'          triangle_left marker
42 '>'          triangle_right marker
43 '1'          tri_down marker
44 '2'          tri_up marker
45 '3'          tri_left marker
46 '4'          tri_right marker
47 's'          square marker
```

```

48   ' ' , p ' '
49   ' ' , * ' '
50   ' ' , h ' '
51   ' ' , H ' '
52   ' ' , + ' '
53   ' ' , x ' '
54   ' ' , D ' '
55   ' ' , d ' '
56   ' ' , | ' '
57   ' ' , _ ' '
58 =====      =====
59
60
61 The following color abbreviations are supported:
62
63 =====  =====
64 character  color
65 =====  =====
66 'b'        blue
67 'g'        green
68 'r'        red
69 'c'        cyan
70 'm'        magenta
71 'y'        yellow
72 'k'        black
73 'w'        white
74 =====  =====
75 .
76 .
77 .

```

All the information in the above help file may not be useful (or understandable), but all you'll need often is the different line colours, line types and marker symbols. To illustrate the use of different line colours, line styles and marker symbols, I'll plot the functions

$$f(x) = 10e^{-0.2x} \sin\left(\frac{\pi}{2}x\right) \quad (13.2)$$

$$g(x) = \tan\left(\frac{\pi}{4}x\right) \quad (13.3)$$

$$h(x) = 5(1 - e^{-0.5x}) \quad (13.4)$$

on the same axes for  $x \in [0; 10]$  and  $y \in [-10; 10]$ . I'll plot the function  $f(x)$  using a blue line with x-marks at every data point. The function  $g(x)$  will be plotted using a red line with points while  $h(x)$  will be drawn with a dash-dot line. The following program produces the graph depicted in Figure 16.6:

```
1  from math import pi
2  from numpy import linspace, exp, sin, tan
3  from matplotlib import rc
4  from matplotlib import pyplot as plot
5
6
7  def plot_functions():
8      x_vals = linspace(0, 10, 100)
9      f_vals = 10 * exp(-0.2 * x_vals) * sin(0.5 * pi * x_vals)
10     g_vals = tan(0.25 * pi * x_vals)
11     h_vals = 5 * (1 - exp(-0.5 * x_vals))
12
13     plot.plot(x_vals, f_vals, "b-x",
14                x_vals, g_vals, "r.-",
15                x_vals, h_vals, "m-.")
16
17
18  def add_plot_info():
19      plot.axis([0, 10, -10, 10])
20      legend_list = [r"$10e^{-0.2x} \sin(0.5 \pi x)$",
21                     r"$\tan(0.25 \pi x)$",
22                     r"$5(1-e^{-0.5x})$"]
23      plot.xlabel("x", fontsize=14)
24      plot.ylabel("Function value", fontsize=14)
25      plot.legend(legend_handle, loc=4, prop={'size': 14})
26
27
28  def main():
29      plot.figure(1)
30      rc('text', usetex=True)
31
32      plot_functions()
33      add_plot_info()
34
35      plot.grid()
36      plot.show()
37
38
39  if __name__ == "__main__":
40      main()
```

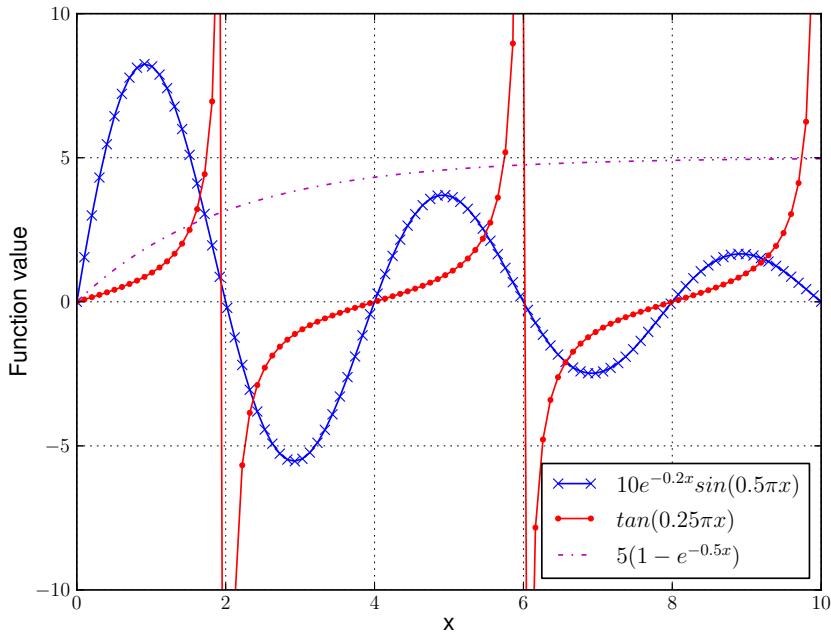


Figure 13.4: Plotting numerous graphs in a single plot window.

Program line 8 defines the `x_vals` vector. Using the `x_vals`- vector, program lines 9–11 define the variables `f_vals`, `g_vals` and `h_vals` that contain the corresponding function values. Again remember that you need to import the the advanced mathematical function from the `numpy` module in order to compute, for example, `sin` of a `numpy` array.

Program line 29 opens a plotting window. The three curves are produced with program line 13. The first pair of vectors are plotted with the option '`b-x`', which results in a solid blue line with `x`-marks at data points. The second pair of vectors are plotted with the option '`r.-`', which uses a red solid line with points to plot the curve. The last pair of vectors are plotted with the option '`m-.`', which plots the curve using a magenta dash-dot line. Program line 19 sets the axes limits to  $x \in [0; 10]$  and  $y \in [-10; 10]$ .

Program line 25 adds a legend to the graph. A legend is only necessary if more than one graph is plotted in a single figure. Use the `legend` statement after the `plot` statement used to generate the curves. The usage is usually

```

1  from matplotlib import pyplot as plot
2
3  plot.legend(["curve1 description", "curve1 description", ...])

```

and it automatically generates a square box with the line types and curve descriptions within. I used the name `legend_list` to store the string descriptions for each of the curves. If you need equations in any of your labels, title or legend, then you need to add a `r` in front of the string description and you need to enclose the equation in dollar signs (\$) as shown in lines 20–20. The explanation for why the `r` is needed in front of the string description is outside the scope of these notes, just remember that it is needed. The dollar (\$) signs are needed to tell `matplotlib` the string (equation) in between the dollar (\$) signs needs to be interpreted by LaTeX and converted to a neat looking equation. Program line 30 is also needed to tell `matplotlib` to turn the LaTeX interpreter on. This unfortunately is just boiler plate code that you need to remember.

Program lines 23 and 24 labels the x and y axes respectively. The `fontsize` is also set to 14pt in these lines. Program line 35 adds grid lines to the graph.

An alternative method to draw multiple graphs in a single plotting window is to replace the program line

```
14 plot.plot(x, f, "b-x",
15           x, g, "r.-",
16           x, h, "m-.")
```

with the commands

```
14 plot.plot(x, f, "b-x")
15 plot.plot(x, g, "r.-")
16 plot.plot(x, h, "m-.")
```

### 13.1.3 Superscripts, subscripts and Greek letters

If you view Figure 16.6, you'll see that the curve descriptions contain superscripts and Greek letters (the letter  $\pi$ ). Superscripts are created by typing the `^` symbol followed by the required superscript enclosed in curly braces i.e.  $e^{-0.2x}$  is created by

```
1 r"$e^{-0.2x}$"
```

Subscripts are created by the underline symbol(`_`), followed by the subscript enclosed in curly braces. As an example,  $x_{k+1}$  is created by

```
1 r"$x_{k+1}$"
```

You can also create the complete Greek alphabet (lowercase and uppercase) by typing the backslash symbol (\) followed by the Greek letter spelt out e.g.  $\pi$  is created by the statement

```
1 r"\pi"
```

Superscripts, subscripts and Greek letters can be used in any of the following commands: xlabel, ylabel, zlabel, title, legend, text and gtext. To illustrate the use of superscripts, subscripts and Greek letters, consider the following Octave program:

```

1 from matplotlib import pyplot as plot
2 from matplotlib import rc
3
4
5 def add_text():
6     plot.text(0.1, 0.90,
7               r"$X_{k+1} = X_k - 2 Y_k^2$",
8               fontsize=14)
9
10    plot.text(0.1, 0.75,
11               r"$Y_{k+1} = Y_k + X_{k+1}^{0.5}$",
12               fontsize=14)
13
14    plot.text(0.1, 0.60,
15               ("Some upper case Greek letters : " +
16                r"\Gamma \Delta \Xi \Pi \Sigma"),
17               fontsize=14)
18
19    plot.text(0.1, 0.45,
20               ("Some lower case Greek letters : " +
21                r"\gamma \delta \rho \sigma \xi \eta \nu"),
22               fontsize=14)
23
24    plot.text(0.1, 0.30,
25               "An example of a formula :",
26               fontsize=14)
27
```

```

28     plot.text(0.3, 0.20,
29                 (r"$A\ x = \lambda\ x \quad where \quad " +
30                  r"\lambda = \omega^2$"),
31                 fontsize=14)
32
33
34 def main():
35     plot.figure(1)
36     rc('text', usetex=True)
37     add_text()
38     plot.show()
39
40
41 if __name__ == "__main__":
42     main()

```

which produces the ‘graph’ in Figure 13.5. Again take note that you need to add a `r` in front of the string description and you need to enclose the equation in dollar signs and you need to add line 36 to your program.

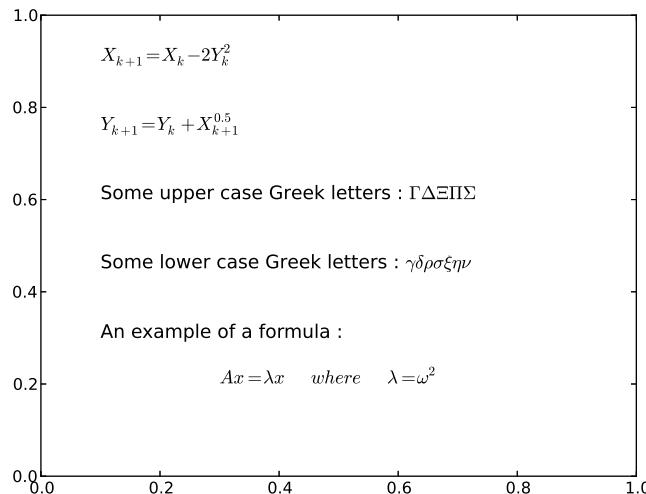


Figure 13.5: Illustrating the use of superscripts, subscripts and Greek letters in Octave.

### 13.1.4 Other 2D graphs

`matplotlib` has many other built-in 2D graphs but I won't discuss these in detail. Rather use the `help` and `?` statements and look at all the various examples presented therein. 2D graphs that I have used include

```
1 from matplotlib import pyplot as plot
```

- `plot.bar`: Plots vectors as vertical bars.
- `plot.loglog`: Logarithmic 2D plot
- `plot.semilogx`: Logarithmic x-axis and linear y-axis
- `plot.semilogy`: Linear x-axis and logarithmic y-axis.
- `plot.hist`: Draws a histogram
- `plot.polar`: Polar coordinate plot
- `plot.step`: Plot a step graph



#### More Info:

More examples can be found at: <http://nbviewer.ipython.org/urls/raw.github.com/jrjohansson/scientific-python-lectures/master/Lecture-4-Matplotlib.ipynb>  
<http://matplotlib.org/users/index.html>

You can also find a gallery of example plots at the following address and clicking on an image will take you to the Python code used to create the image <http://matplotlib.org/gallery.html>

## 13.2 3D Graphs

`matplotlib` has extensive 3D graphical capability. This topic won't be tested in the exam, but I include it for completeness.

### 13.2.1 The plot3D command

The first type of 3D graph I'll discuss is produced by the `plot3D` statement. This statement is used to plot lines and/or points in 3D space. As with any other plotting statement, first make sure you have the data you want to plot. As an example, I'll show you how to plot the parametric curve described by

$$x = r \cos(\theta) \quad (13.5)$$

$$y = r \sin(\theta) \quad (13.6)$$

$$z = t \quad (13.7)$$

where  $r = t$ ,  $\theta = \frac{3}{2}\pi t$  and  $t \in [0; 4]$ .

```
1 from matplotlib import pyplot as plot
2 from mpl_toolkits.mplot3d import Axes3D
3 from numpy import linspace, cos, sin, pi
4
5
6 def plot_3d_func(axis):
7     t_vals = linspace(0, 4, 200)
8     theta = 3 * pi/2 * t_vals
9     x_vals = t_vals * cos(theta)
10    y_vals = t_vals * sin(theta)
11
12    axis.plot3D(x_vals, y_vals, t_vals)
13
14
15 def add_plot_info(axis):
16     axis.set_xlabel("x")
17     axis.set_ylabel("y")
18     axis.set_zlabel("z")
19
20
21 def main():
22     fig = plot.figure(1)
23     axis = Axes3D(fig)
24     plot_3d_func(axis)
25     add_plot_info(axis)
26
27     plot.show()
28
29
```

```

30  if __name__ == "__main__":
31      main()

```

For 3D plotting we have to make use of the `Axes3D` function (line 23), which takes the `figure` object as an input. In program line 12 the `plot3D` function from the `Axes3D` object is used to create the 3D plot. The graph is depicted in Figure 13.6.

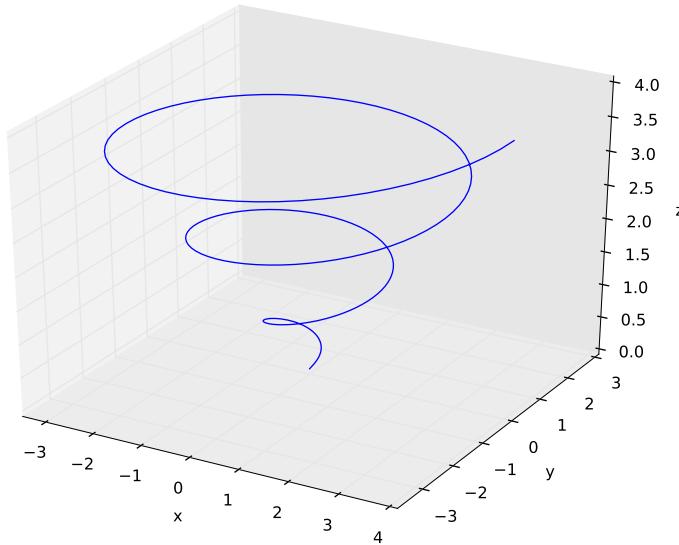


Figure 13.6: Illustration of a graph generated by the `plot3D` command.

### 13.2.2 The `plot_wireframe` and `plot_surface` commands

The `plot3D` command was used to plot points and lines in 3D space. If instead you need to plot surfaces in 3D space, you can use the `plot_wireframe` and `plot_surface` statements. The `plot_wireframe` statement produces a wire-frame representation of the surface while the `plot_surface` statement produces a coloured-in representation of the surface.

Again, I'll illustrate these commands via an example. Consider the surface described by

$$z(x, y) = \sin(x) \cos(y) \quad \text{for } x \in [-2.5; 2.5] \text{ and } y \in [-2.5; 2.5] \quad (13.8)$$

Before we can plot this surface, the data must be generated. In order to generate a rectangular grid of  $(x, y)$  data points where we can evaluate the function  $z(x, y)$ , we need the `meshgrid` statement. The `meshgrid` statement is the 2D equivalent of the 1D

`linspace` command. Instead of generating a single vector containing evenly spaced data points, a regular 2D grid is created by the `meshgrid` command. The syntax is:

```
1 from numpy import meshgrid
2
3
4 x = array([1, 2, 3])
5 y = array([4, 5, 6, 7])
6 X, Y = meshgrid(x, y)
```

The Python program required to create the 3D surfaces follows:

```
1 from matplotlib import pyplot as plot
2 from matplotlib import cm
3 from mpl_toolkits.mplot3d import Axes3D
4 from numpy import meshgrid, linspace, cos
5
6
7 def set_axis_properties(axis):
8     axis.set_xlim3d(-3, 3)
9     axis.set_ylim3d(-3, 3)
10    axis.set_zlim3d(-1, 1)
11
12
13 def plot_3d_func(axis, mesh=True):
14     x_vals = linspace(-2.5, 2.5, 100)
15     y_vals = linspace(-2.5, 2.5, 100)
16     x_mesh, y_mesh = meshgrid(x_vals, y_vals)
17
18     z_mesh = cos(x_mesh) * cos(y_mesh)
19     if mesh:
20         axis.plot_wireframe(x_mesh, y_mesh, z_mesh)
21     else:
22         axis.plot_surface(x_mesh, y_mesh, z_mesh,
23                           rstride=1, cstride=1,
24                           cmap=cm.coolwarm, linewidth=0)
25
26
27 def add_plot_info(axis):
28     axis.set_xlabel("x")
29     axis.set_ylabel("y")
```

```

30     axis.set_zlabel("z")
31     axis.set_title("Graph of z(x,y) = cos(x)cos(y)")
32
33
34 def main():
35     # ----Figure 1----
36     fig = plot.figure(1)
37     axis = Axes3D(fig)
38     plot_3d_func(axis, mesh=True)
39     add_plot_info(axis)
40     set_axis_properties(axis)
41
42     # ----Figure 2----
43     fig = plot.figure(2)
44     axis = Axes3D(fig)
45     plot_3d_func(axis, mesh=False)
46     add_plot_info(axis)
47     set_axis_properties(axis)
48
49     # show both figures
50     plot.show()
51
52
53 if __name__ == "__main__":
54     main()

```

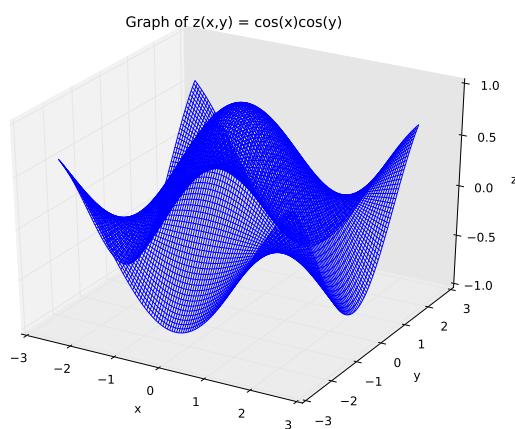


Figure 13.7: Illustration of 3D surface generated by the `mesh` command.

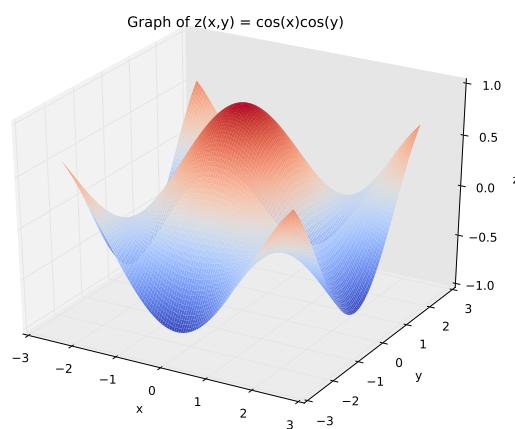


Figure 13.8: Illustration of 3D surface generated by the `surf` command.

The plot depicted in Figure 13.7 is created by the `plot_wireframe` command while the graph in Figure 13.8 is created by the `plot_surface` statement. These graphs are

good examples of when it is necessary to increase the fontsize. The following example shows the changes made to increase the font sizes, not only of the labels and title, but also of the axis tick labels.

```
 1 from matplotlib import pyplot as plot
 2 from matplotlib import cm
 3 from mpl_toolkits.mplot3d import Axes3D
 4 from numpy import meshgrid, linspace, cos
 5
 6
 7 def set_axis_properties(fig, axis):
 8     axis.set_xlim3d(-3, 3)
 9     axis.set_ylim3d(-3, 3)
10     axis.set_zlim3d(-1, 1)
11
12     gca = fig.gca()
13     gca.set_xticklabels(gca.get_xticks(), fontsize=18)
14     gca.set_yticklabels(gca.get_yticks(), fontsize=18)
15     gca.set_zticklabels(gca.get_zticks(), fontsize=18)
16
17
18 def plot_3d_func(axis, mesh=True):
19     x_vals = linspace(-2.5, 2.5, 100)
20     y_vals = linspace(-2.5, 2.5, 100)
21     x_mesh, y_mesh = meshgrid(x_vals, y_vals)
22
23     z_mesh = cos(x_mesh) * cos(y_mesh)
24     if mesh:
25         axis.plot_wireframe(x_mesh, y_mesh, z_mesh)
26     else:
27         axis.plot_surface(x_mesh, y_mesh, z_mesh,
28                            rstride=1, cstride=1,
29                            cmap=cm.coolwarm, linewidth=0)
30
31
32 def add_plot_info(axis):
33     axis.set_xlabel("x", fontsize=18)
34     axis.set_ylabel("y", fontsize=18)
35     axis.set_zlabel("z", fontsize=18)
36     axis.set_title("Graph of z(x,y) = cos(x)cos(y)",
37                   fontsize=18)
38
39
40 def main():
```

```

41 # ----Figure 1----
42 fig = plot.figure(1)
43 axis = Axes3D(fig)
44 plot_3d_func(axis, mesh=True)
45 add_plot_info(axis)
46 set_axis_properties(fig, axis)
47
48 # ----Figure 2----
49 fig = plot.figure(2)
50 axis = Axes3D(fig)
51 plot_3d_func(axis, mesh=False)
52 add_plot_info(axis)
53 set_axis_properties(fig, axis)
54
55 # show both figures
56 plot.show()
57
58
59 if __name__ == "__main__":
60     main()

```

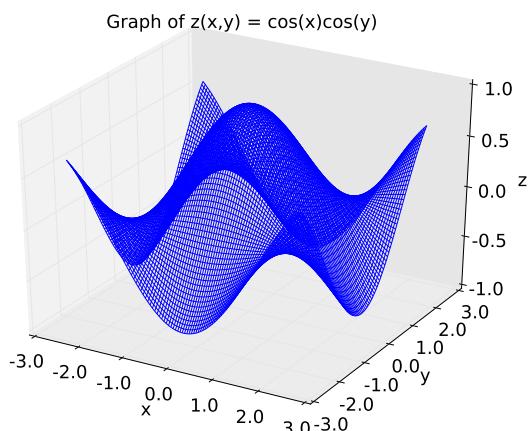


Figure 13.9: 3D surface generated by the `mesh` command with fontsize adjustment.

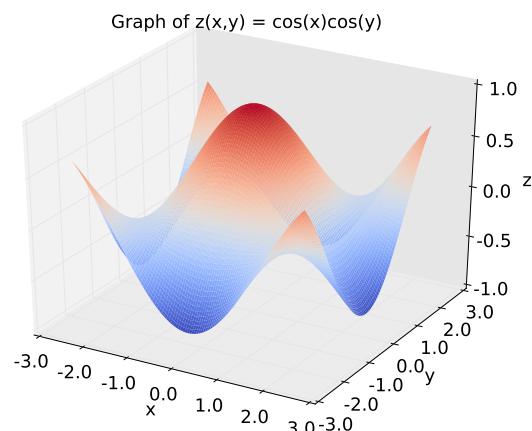


Figure 13.10: 3D surface generated by the `surf` command with fontsize adjustment.

After I make these changes, the resulting graphs are presented in Figures 13.9 and 13.10.

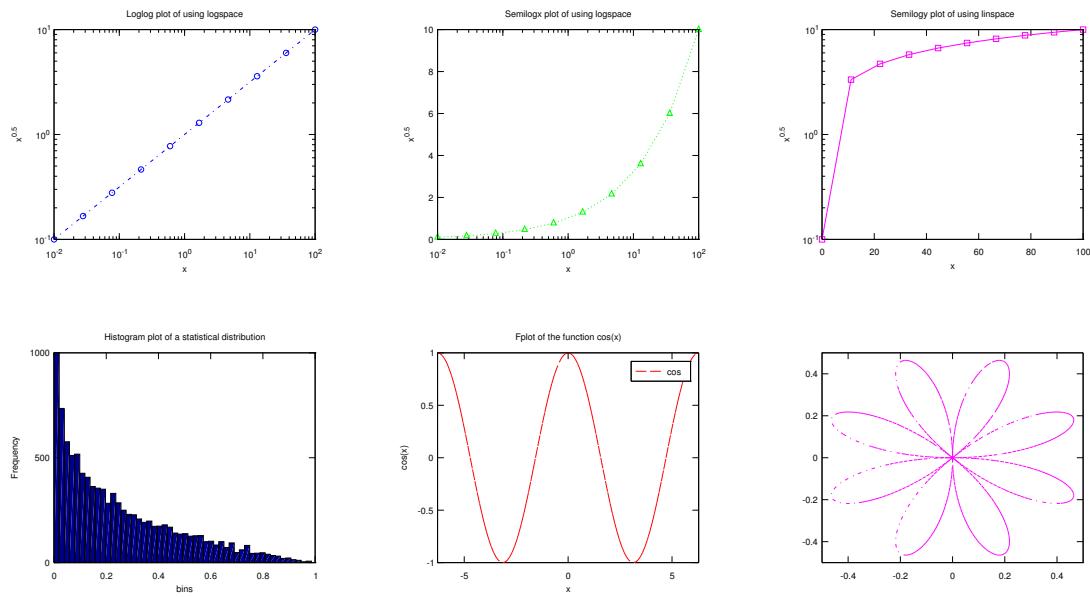


Figure 13.11: Various examples of subplot layouts.

### 13.3 Subplots

So far, whenever we wanted to plot more than one curve, we simply plotted all of them on the same axes. However, this approach doesn't cater for all scenarios. Let's say we want to plot a vector of complex numbers. It is customary to plot such a vector as two separate plots: the first plot shows the magnitude while the second depicts the phase angle. Or we want to plot a 3D curve as three distinct plots: the 2D projection in the  $x$ - $y$  plane, the 2D projection in the  $x$ - $z$  plane and the 2D projection in the  $y$ - $z$  plane. To cater for these scenarios, we use the `subplot` statement.

The `subplot` statement divides a single plotting window into  $m$  by  $n$  plotting areas. We can then plot any graph we like in each of these smaller areas. The syntax of the `subplot` statement is

```
1 from matplotlib import pyplot as plot
2
3 plot.subplot(mnp)
```

which divides the plotting window into  $m$  by  $n$  smaller plotting areas. These smaller plotting areas are counted from left to right, top to bottom. Any subsequent plotting commands will then generate a plot in the  $p$ -th position. Some examples of `subplot` layouts are depicted in Figure 13.11.

I'll give a single example of how to produce 4 figures in one window by using the `subplot` command. I'll draw the graphs of

$$f(x) = \sin(x) \quad g(x) = \cos(x) \quad (13.9)$$

$$h(x) = \tan(x) \quad p(x) = \sin^2(x) \quad (13.10)$$

for  $x \in [0; 4\pi]$ . Here's the Python code that produces the required graphs, as depicted in Figure 13.12:

```
1  from math import pi
2  from numpy import linspace, cos, sin, tan
3  from matplotlib import pyplot as plot
4  from matplotlib import rc
5
6
7  def plot_functions():
8      x_vals = linspace(0, 4*pi, 200)
9      f_vals = sin(x_vals)
10     g_vals = cos(x_vals)
11     h_vals = tan(x_vals)
12     p_vals = sin(x_vals) ** 2
13
14     plot.subplot(221)
15     plot.plot(x_vals, f_vals, "b")
16     plot.axis([0, 4*pi, -1.1, 1.1])
17     add_plot_info("sin(x)")
18
19     plot.subplot(222)
20     plot.plot(x_vals, g_vals, "r")
21     plot.axis([0, 4*pi, -1.1, 1.1])
22     add_plot_info("cos(x)")
23
24     plot.subplot(223)
25     plot.plot(x_vals, h_vals, "g")
26     plot.axis([0, 4*pi, -10, 10])
27     add_plot_info("tan(x)")
28
29     plot.subplot(224)
30     plot.plot(x_vals, p_vals, "m")
31     plot.axis([0, 4*pi, -0.1, 1.1])
32     add_plot_info(r"sin^2(x)")
33
34
35  def add_plot_info(func_str):
```

```
36     plot.xlabel("x")
37     plot.ylabel("f(x)")
38     plot.title(r"$f(x) = %s$" % func_str)
39
40
41 def main():
42     plot.figure(1)
43     rc('text', usetex=True)
44     plot.subplots_adjust(wspace=0.4, hspace=0.4)
45
46     plot_functions()
47
48     plot.show()
49
50
51 if __name__ == "__main__":
52     main()
```

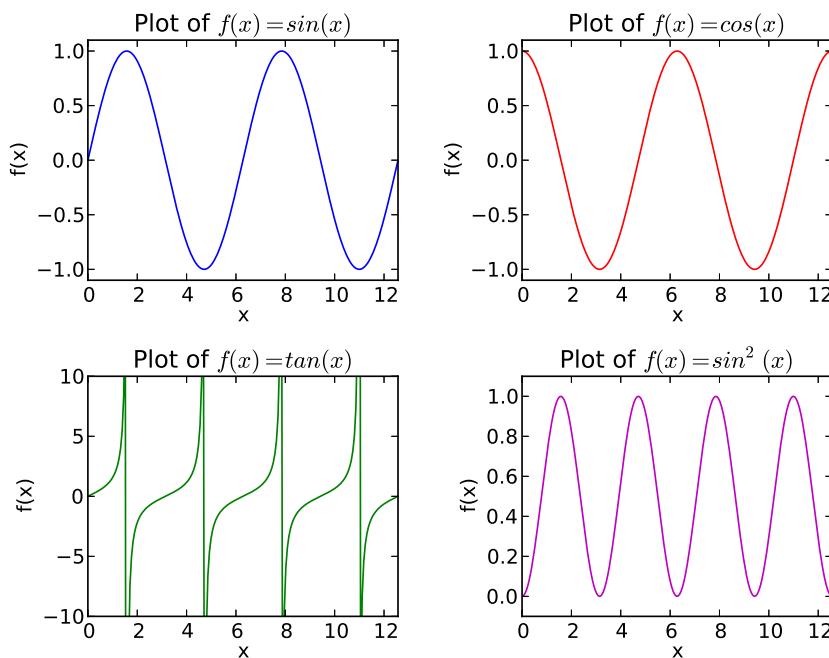


Figure 13.12: Example of  $2 \times 2$  plots generated using the `subplot` command.

## 13.4 Exercises

1. Use Python to plot the following functions, all on one graph. Set the range for the y-axis between -10 and 10. Add appropriate descriptions (use the Legend command).

$$f(x) = 10\sin(x)\cos(x) \quad x \in [-2\pi, 2\pi] \quad (13.11)$$

$$f(x) = x^2 e^{0.1x} - 2x^2 e^{-0.1x} \quad x \in [-2\pi, 2\pi] \quad (13.12)$$

$$f(x) = \omega \tan(\omega x) \quad x \in [-2\pi, 2\pi], \omega = 0.5 \quad (13.13)$$

2. A nice picture can be constructed by plotting the points  $(x_k, y_k)$  generated from the following difference equations

$$x_{k+1} = y_k [1 + \sin(0.7x_k)] - 1.2\sqrt{|x_k|} \quad (13.14)$$

$$y_{k+1} = 0.21 - x_k \quad (13.15)$$

starting with  $x_0 = y_0 = 0$ . Write a program to plot the picture of the individual points.

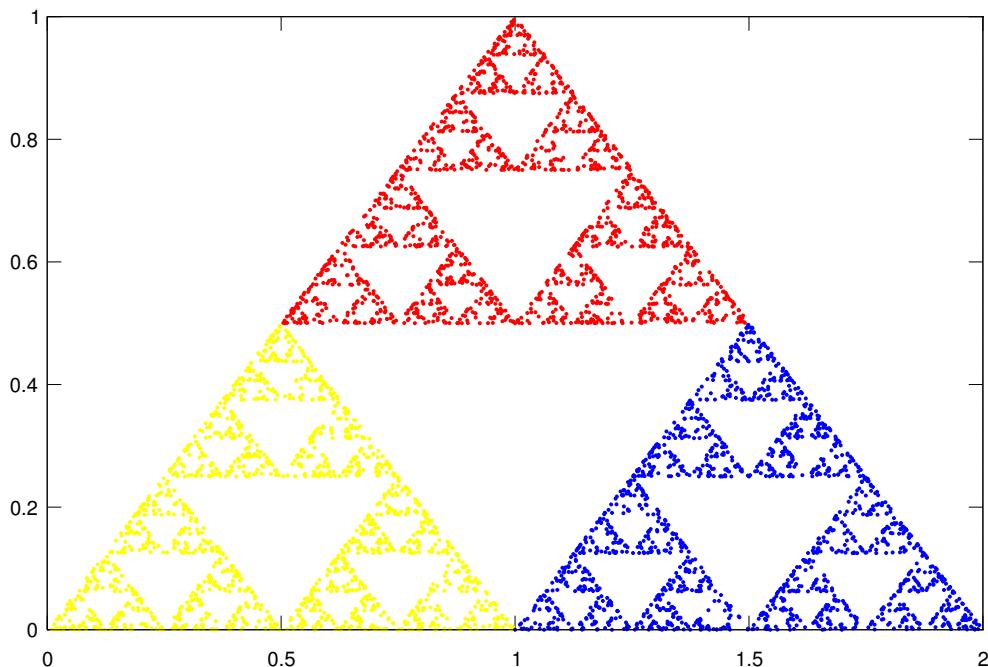


Figure 13.13: Sierpiński triangle.

3. The Sierpiński triangle is a fractal. There are various ways to compute the Sierpiński triangle. The method described below is called an iterated fractal system and starts with a point at the origin  $x_0 = 0$  and  $y_0 = 0$ . The next  $x_{n+1}$  and  $y_{n+1}$  points for  $n = 0, 1, 2, \dots, N$  are then computed by randomly selecting one of the three equations below. Each equation has equal probability to be selected:

- $x_{n+1} = 0.5x_n$  and  $y_{n+1} = 0.5y_n$  with the point plotted in yellow.
- $x_{n+1} = 0.5x_n + 0.5$  and  $y_{n+1} = 0.5y_n + 0.5$  with the point plotted in red.
- $x_{n+1} = 0.5x_n + 1$  and  $y_{n+1} = 0.5y_n$  with the point plotted in blue.

The program must ask the user to enter the number of points  $N$  required. Each point is plotted as a dot without any lines connecting the points.

Figure 13.13 is an example of the Sierpiński triangle using 5000 points. To increase the speed of the plotting only plot once all the points have been generated. To do so you would have to store the points for each colour in separate vectors.

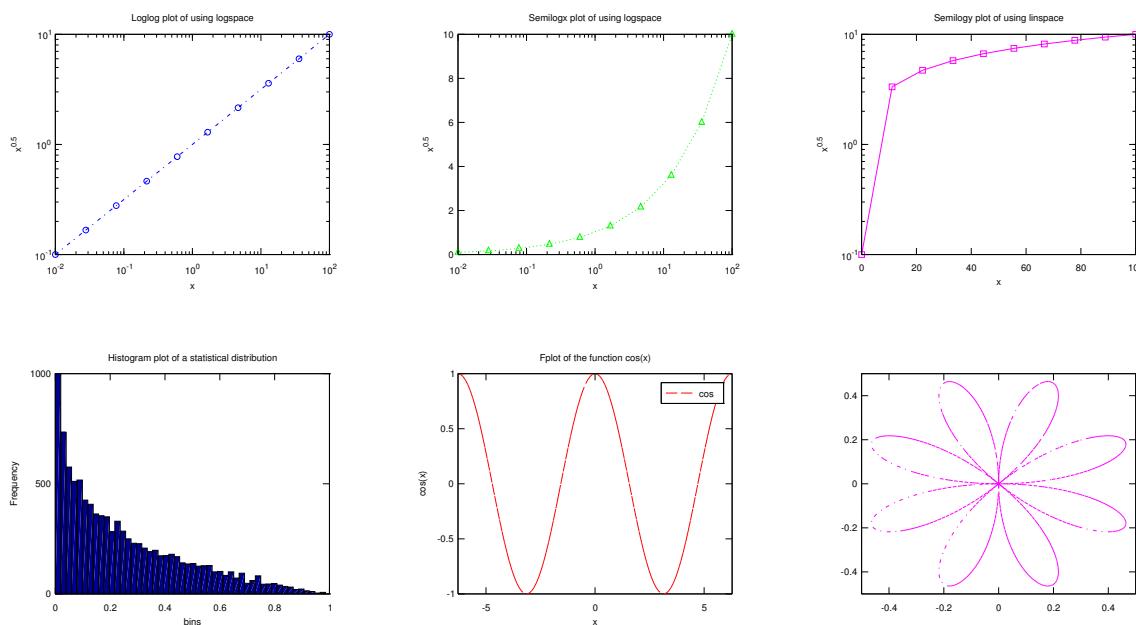


Figure 13.14: Generate this figure

4. Generate Figure 13.14. Here follows the descriptions of each of the six sub plots:

- (a) Loglog plot of  $\sqrt{x}$  with  $x$  between  $10^{-2}$  and  $10^2$  using 10 points. Use `logspace` to generate 10 equally spaced points on a *log* scale. Add appropriate labels and a figure title. Use the line style (dash-dot), line color (**b**lue) and marker type (circle) as displayed.

- (b) Semilogx plot of  $\sqrt{x}$  with  $x$  between  $10^{-2}$  and  $10^2$  using 10 points. Use `logspace` to generate 10 equally spaced points on a *log* scale. Add appropriate labels and a figure title. Use the line style (dot-dot), line color (**green**) and marker type (upwards pointing triangle) as displayed.
- (c) Semilogy plot of  $\sqrt{x}$  with  $x$  between  $10^{-2}$  and  $10^2$  using 10 points. Use `linspace` to generate 10 equally spaced points on a *linear* scale. Add appropriate labels and a figure title. Use the line style (solid line), line color (**magenta**) and marker type (square) as displayed.
- (d) Histogram plot of the product of two uniform random numbers  $r_1$  and  $r_2$  between 0 and 1 using 10000 product pairs  $r_1 \times r_2$ . Use 50 bins for your histogram plot and add appropriate labels and a title.
- (e) Symbolic plot of  $\cos(x)$  between  $-2\pi$  and  $2\pi$ . Scale the axis between  $-2\pi$  and  $2\pi$  using the `axis` command. Use the line style (dash-dash), line color (**red**) and marker type (none) as displayed.
- (f) Polar plot of  $\sin(2x)\cos(2x)$  for  $x$  between 0 and  $2\pi$  in increments of 0.01. Add appropriate labels and a figure title. Use the line style (dash-dot), line color (**magenta**) and marker type (none) as displayed.

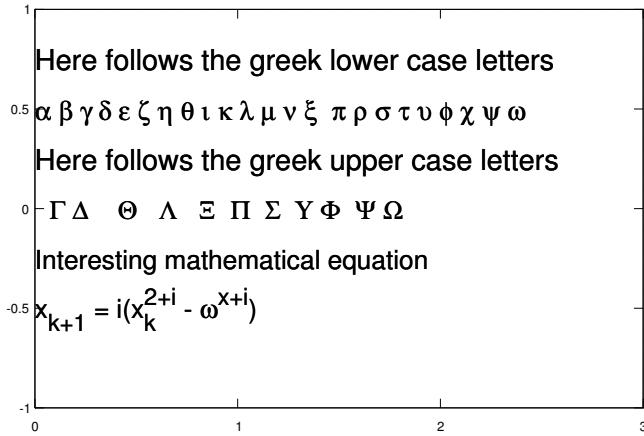


Figure 13.15: Generate this figure

5. The aim of this question is to expose you to some of the features of the `text` command. Generate Figure 13.15. The Greek alphabet is given below:

- *alpha*  $\alpha$  - *beta*  $\beta$  - *gamma*  $\gamma$  - *delta*  $\delta$  - *epsilon*  $\epsilon$  - *zeta*  $\zeta$
- *eta*  $\eta$  - *theta*  $\theta$  - *iota*  $\iota$  - *kappa*  $\kappa$  - *lambda*  $\lambda$  - *mu*  $\mu$
- *nu*  $\nu$  - *xi*  $\xi$  - *omicron* - *pi*  $\pi$  - *rho*  $\rho$  - *sigma*  $\sigma$

- tau  $\tau$  - upsilon  $\upsilon$  - phi  $\phi$  - chi  $\chi$  - psi  $\psi$  - omega  $\omega$

with a font size of 20. The text

Here follows the greek lower case letters  
and

Here follows the greek lower case letters  
has a font size of 23. The text

Interesting mathematical equation  
and the equation

$$x_{k+1} = i(x_k^{2+i} - \omega^{x+i}) \quad (13.16)$$

has a font size of 20.

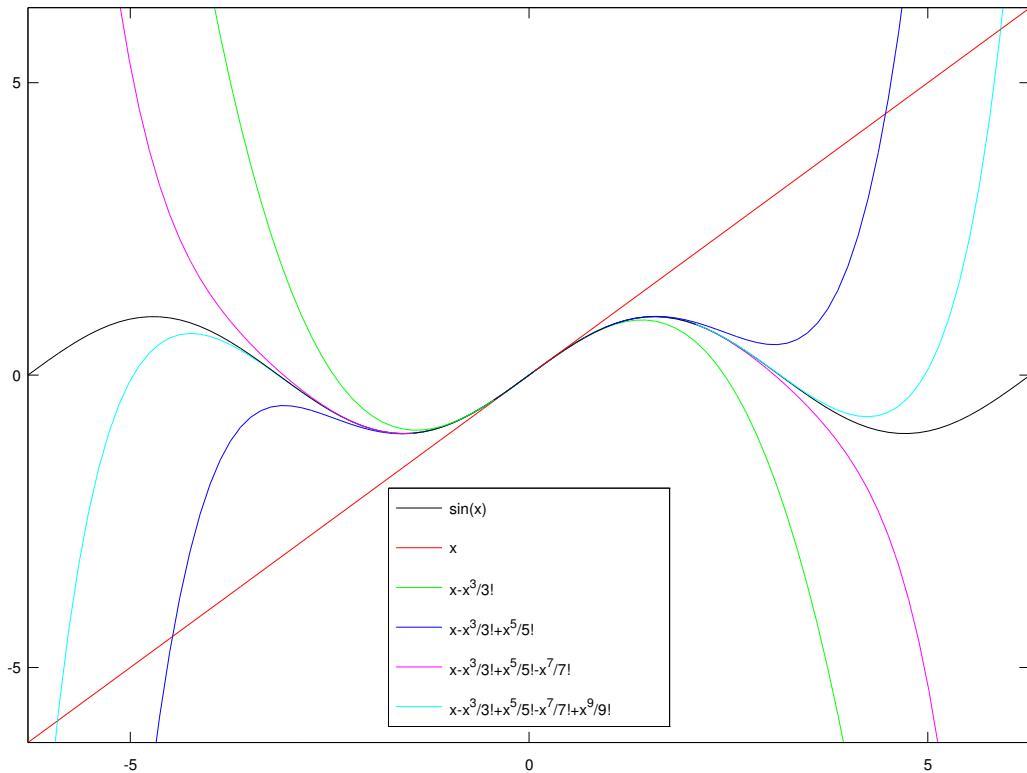


Figure 13.16: Taylor approximation of  $\sin(x)$  around 0.

6. The taylor expansion of the first five terms of  $\sin$  around the point 0 is given below:

$$\sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} \quad (13.17)$$

Generate Figure 13.16 with the same color scheme as indicated and place the legend in the same location. Use `axis` to define the  $x$  and  $y$  boundaries of the figure.

Note how the accuracy of the Taylor approximation improves as the number of terms increase.

7. A potential energy function is given by

$$V_m(x) = V_0 \left[ 1 - \exp\left(-\frac{x}{\delta}\right) \right]^2$$

where we choose  $V_0 = \delta = 1$ .

A quadratic approximation for the above is

$$V_{n2}(x) = a_2 x^2$$

and the eight-order approximation is

$$V_{n8}(x) = \sum_{i=0}^8 a_i x^i = a_0 + a_1 x + a_2 x + \cdots + a_8 x$$

The constants  $a_i$ ,  $i = 1, 2, 3, 4, 5, 6, 7, 8$  are:

$$a_0 = 1.015 \times 10^{-4} \quad a_1 = 0.007 \quad (13.18)$$

$$a_2 = 0.995 \quad a_3 = -1.025 \quad (13.19)$$

$$a_4 = 0.611 \quad a_5 = -0.243 \quad (13.20)$$

$$a_6 = 0.061 \quad a_7 = -0.009 \quad (13.21)$$

$$a_8 = 5.249 \times 10^{-4} \quad (13.22)$$

Write a program plot the quadratic and 8th-order approximations on the same axis. Use appropriate label descriptions, line styles and function descriptions in the legend.

8. Write a program that generates 1000 points  $(x, y)$  of the fractal fern picture (FFP). The FFP consists of two series  $x_1, x_2, \dots, x_n$  and  $y_1, y_2, \dots, y_n$ . The starting values of the two series are  $x_1 = 0$  and  $y_1 = 0$ . During each of the 1000 iterations, one of the following four equations is randomly selected for generation of the next point  $(x_{n+1}, y_{n+1})$

A  $\frac{1}{100}$  chance to choose

$$x_{n+1} = 0 \quad (13.23)$$

$$y_{n+1} = 0.16y_n \quad (13.24)$$

A  $\frac{7}{100}$  chance to choose

$$x_{n+1} = 0.2x_n - 0.26y_n \quad (13.25)$$

$$y_{n+1} = 0.23x_n + 0.22y_n + 1.6 \quad (13.26)$$

A  $\frac{7}{100}$  chance to choose

$$x_{n+1} = -0.15x_n + 0.28y_n \quad (13.27)$$

$$y_{n+1} = 0.26x_n + 0.24y_n + 0.44 \quad (13.28)$$

A  $\frac{85}{100}$  chance to choose

$$x_{n+1} = 0.85x_n + 0.04y_n \quad (13.29)$$

$$y_{n+1} = -0.04x_n + 0.85y_n + 1.6 \quad (13.30)$$

A possible strategy to determine which equation to choose is to generate a random integer number between 1 and 100. Then according to the given probabilities determine which equation to use. For example, if the randomly generated number is 1, we use the first equation since its probability is 1 out of 100. If the randomly generated number is a 2,3,4,5,6,7 or an 8 (i.e. the next seven numbers) we use the second equation since it's probability is 7 out of 100, etc. for the remaining two equations.

Plot the two generated vectors  $x$  and  $y$  against each other, using dots for the data points. You should see a beautiful picture of a fern leaf.

9. In 1807 Joseph Fourier showed that  $2n$ -periodic functions  $f(x)$  can be approximated using a summation of sines and cosines terms. The Fourier expansion of  $f(x) = x$  for  $-\pi < x < \pi$  is given by

$$f(x) = \sum_{n=1}^{\infty} b_n \sin(nx)$$

where the scalar constants  $b_n$  are given by

$$b_n = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) \sin(nx) dx, \quad n = 1, 2, 3, \dots$$

Plot the Fourier approximations for  $n = 1$ ,  $n = 2$  and  $n = 3$  on the same figure with  $x$  between  $-\pi$  and  $\pi$  using 50 equally spaced points. Plot each of the graphs with a different line style or line colour and add an appropriate legend. Appropriately label the x-axis and y-axis and add a title for the figure.

[Hint: You must use `numpy`'s numerical integration function `quad` to compute the constants  $b_n$ ]

10. The equations of motion of earth around the sun is given by

$$\begin{aligned}\frac{dv_x(t)}{dt} &= -\frac{G_m x(t)}{[x(t)^2 + y(t)^2]^{3/2}} \\ \frac{dx(t)}{dt} &= -v_x(t) \\ \frac{dv_y(t)}{dt} &= -\frac{G_m y(t)}{[x(t)^2 + y(t)^2]^{3/2}} \\ \frac{dy(t)}{dt} &= -v_y(t)\end{aligned}\tag{13.31}$$

where  $(t)$  indicates a dependency on time in years ( $yr$ ) and  $G_m = 4\pi^2$  in units  $\frac{AU^3}{yr^2}$ . One AU is the  $yr$  distance from the earth to the sun, also known as the astronomical unit. Compute the position  $(x, y)$  and velocity  $(v_x, v_y)$  of earth for 1 year in increments of 0.01 years. The initial conditions are  $x(0) = 1$  AU,  $y(0) = 0$  AU,  $v_x(0) = 0$  AU/ $yr$  and  $v_y(0) = 2\pi$  AU/ $yr$ .

Plot the  $x$  and  $y$  positions as well as the  $v_x$  and  $v_y$  velocities as functions of time on two separate plots on the same figure. The first plot must depict the  $x$  and  $y$  positions while the second plot must depict the  $v_x$  and  $v_y$  velocities. Give appropriate annotations for the x-axis, y-axis, title and line descriptors of each plot.

[Hint: You can use `scipy.integrate`'s function `odeint` to solve the system of ordinary differential equations]

# Chapter 14

## Exception Handling

### 14.1 Text menus and user input

Before we jump into exception handling lets first consider where we would possibly need to handle exceptions. The one scenario that jumps to my mind is in the case of user input. We often expect the user to input a certain answer (like a number), but what happens when the user enters some unexpected value? In these case we need to handle any exception in our program, should the user have entered the incorrect value. Lets start by developing a generic user menu.

Whenever the user of one of your program needs to choose between a number of options, you can create a little menu that is printed to the screen. So far we have utilised the `print` and `raw_input` statements in such cases. Examples include the numerical differentiation program where the user could choose between the forward, backward and central difference method, or the numerical integration program where the user could choose between the left-point, right-point, mid-point or trapezium methods.

Let us look at how we can program a generic menu function that we can use in any program. Firstly we need to identify the inputs and outputs for this functions. For the inputs I chose the `title` sting of the menu and a list of string `options` the user can chose from. For the outputs I chose to return a numerical representation of the option number selected.

```
1 def menu(title, options):
2     """
3         General menu function for displaying a menu with various
4         options to the screen.
5     """
```

```

6   Inputs:
7     title: title of the menu - string
8     options: list of menu options - [str, str, str, ...]
9
10  Returns:
11    user_input: the users selections as an integer
12      1 = first option
13      2 = second option
14      etc.
15      """
16
17  line = "-" * len(title)
18  n_opts = len(options)
19
20  print title
21  print line
22  for cnt, option in enumerate(options):
23    print "[%2d] " % (cnt+1) + option
24
25  str_template = "\nPick a number from 1 to %d: "
26  user_input = int(raw_input(str_template % (n_opts)))
27  return user_input

```

The syntax of the `menu` command is:

```

1 choice = menu("Title of menu", ["option1", "option2", ...])

```

This statement automatically generates a text menu with the specified title at the top and all the options listed below. The user then selects a number, if the first option is selected, the output name(`user_input` in this example) is assigned a value of 1. Similarly, the second choice assign the value 2 to the output name and so on.

I'll now illustrate the use of the `menu` command by plotting the function  $f(x) = \sin(x)$  for  $x \in [0; 2\pi]$ . The line colour will be chosen by the user, using the `menu` command. Just to make things interesting, I'll let the program repeat until the user selects the `Exit` option. This is achieved by using a `while` loop. The Python program follows:

```

1 from math import pi
2 from numpy import linspace, sin
3 from matplotlib.pyplot import plot, axis, show, figure
4

```

```
5 x = linspace(0, 2*pi, 200)
6 y = sin(x)
7 while True:
8     choice = menu("Choose a colour:",
9                     ["Red", "Green", "Blue", "Exit"])
10
11    if choice == 4:
12        break
13
14    line_col = "k"
15    if choice == 1:
16        line_col = "r"
17    if choice == 2:
18        line_col = "g"
19    if choice == 3:
20        line_col = "b"
21
22 figure(1)
23 plot(x, y, line_col)
24 axis([0, 2*pi, -1.1, 1.1])
25 show()
```

The text menu that is generated by the `menu` command in program line 9 is given by:

```
1 Choose a colour:
2 -----
3 [ 1] Red
4 [ 2] Green
5 [ 3] Blue
6 [ 4] Exit
7
8 Pick a number from 1 to 4:
```

## 14.2 Error handling and user input

In the example above, if the user had entered any integer value the program would have worked perfectly, however if the user entered a floating point number or a string then the program would have crashed giving a trace back similar to the following:

```
1 Traceback (most recent call last):
2   File ".../test2.py", line 38, in <module>
3     ["Red", "Green", "Blue", "Exit"])
4   File ".../test2.py", line 30, in menu
5     user_input = int(raw_input(str_template % (n_opts)))
6 ValueError: invalid literal for int() with base 10: 'a'
```

This error occurs in line 25 of the `menu` function when Python tries to convert the user's input into an integer object. Secondly if the user had entered any integer greater than 4 the program also would have run, but any number greater than 4 is not a valid menu option. The following revision of the `menu` function, we created earlier, uses exception handling and additional logic to account for these unexpected inputs from the user

```
1 def menu(title, options):
2     """
3         General menu function for displaying a menu with various
4         options to the screen.
5
6     Inputs:
7         title: title of the menu - string
8         options: list of menu options - [str, str, str, ...]
9
10    Returns:
11        user_input: the users selections as an integer
12            1 = first option
13            2 = second option
14            etc.
15
16    while True:
17        line = "-" * len(title)
18        n_opts = len(options)
19
20        print title
21        print line
22        for cnt, option in enumerate(options):
23            print "[%2d] " % (cnt+1) + option
24
25        str_template = "\nPick a number from 1 to %d: "
26        user_input = raw_input(str_template % (n_opts))
27        try:
28            user_input = int(user_input)
```

```
29         if not (0 < user_input < (n_opts+1)):  
30             print "Invalid Selection. Please try again."  
31         else:  
32             return user_input  
33     except (ValueError, ):  
34         print "Invalid Selection. Please try again."
```

Program exceptions are handled using the `try:` and then `except:` commands as shown in lines 27–34. Python first attempts to execute the code in the `try:` statement (lines 28–32), if any error is found in this code (lines 28–32) then Python immediately jumps to line 33 and executes the `except:` statement. If the error encountered in lines 28–32 is in the tuple (given in the `except` statement then the code in the `except` statement is executed (line 34) otherwise Python will give a trace back with the error type found.

This logic is coupled with a `while` loop statement to ensure the program continues to execute until the user has made a valid selection.



# Chapter 15

## More ways to loop

### 15.1 Recursion: using functions to loop

We can also loop using functions instead of `for` loop or `while` loop statements. The answer lies in creating a function where the output of the function requires the function itself to be evaluated again. How do we then stop? To stop we just have to make the output equal to something (e.g. some constant or value) which does not involve the function itself to be evaluated. To achieve this the function will have to make a decision on whether the output calls the function itself (when we want to loop) or whether the output is equal to some constant (when we want to stop).

How would we go about computing the factorial of a number  $n$ . We could write it out as  $n! = n \times (n - 1)! = n \times (n - 1) \times (n - 2)!$ , keeping in mind that  $0! = 1$ . Let us see what happens when we write a function `my_fact` that takes a number  $n$  as input and returns as output the number  $n$  multiplied with `my_fact(n-1)` unless  $n = 0$  then the output must equal 1. The `my_fact(n-1)` with which we multiply  $n$  becomes  $n - 1$  multiplied with `my_fact(n-2)` similarly to the mathematical expansion given above. Given below is the function `my_fact` which computes the factorial of a number recursively,

```
1 def my_fact(n):
2     if n == 0:
3         return 1
4     return n * my_fact(n-1)
5
6 print my_fact(4)
```

Let us see if we can add the  $n$  integers using recursive functions given by

$$\sum_{i=1}^n = n + \sum_{i=1}^{n-1} = n + (n - 1) + \sum_{i=1}^{n-2} \quad (15.1)$$

We require a function of which the output is to equal the sum of the input value of the function and the function itself called with 1 less the input value, unless the input value is zero then the output must also be zero.

```

1 def my_sum(num):
2     if num == 0:
3         return 0
4     return num + my_sum(num-1)
5
6 print my_sum(4)

```

We could also make use of lists for instance if we want to generate a list storing the sum of the integers, as shown below

```

1 def my_sum(num, store_list):
2     if num == 0:
3         store_list.append(0)
4         return 0
5     total_sum = num + my_sum(num-1, store_list)
6     store_list.append(total_sum)
7     print store_list
8     return total_sum
9
10 my_list = []
11 sum_int = my_sum(4, my_list)
12 print
13 print sum_int

```

Look how the list `store_list` grows inside the function. Run it and see how `store_list` grows from an empty list to [0, 1, 3, 6, 10]. What will happen if we change line 6 from `store_list.append(total_sum)` to `store_list.insert(0, total_sum)`? Do it and see how the output changes when you type `myvec(5)` from

```
1 [0, 1]
2 [0, 1, 3]
3 [0, 1, 3, 6]
4 [0, 1, 3, 6, 10]
5 [0, 1, 3, 6, 10, 15]
6
7 15
```

to

```
1 [1, 0]
2 [3, 1, 0]
3 [6, 3, 1, 0]
4 [10, 6, 3, 1, 0]
5 [15, 10, 6, 3, 1, 0]
6
7 15
```

Let us consider another recursive example. Remember the bisection method which we used to find the root of a function. I list the method below to refresh our memory.

Recall that we have to start with a function  $f(x)$  as well as an interval  $x \in [x_l, x_u]$  such that the function is either negative at  $x_l$  and positive at  $x_u$  or positive at  $x_l$  and negative at  $x_u$  which can be written as  $f(x_l) < 0$  and  $f(x_u) > 0$ , or  $f(x_l) > 0$  and  $f(x_u) < 0$ .  $x_l$  refers to the lower bound of the interval and  $x_u$  refers to the upper bound of the interval. If the function is continuous and it changes sign somewhere between  $x_l$  and  $x_u$  there must be a point  $x$  between  $x_l$  and  $x_u$  where  $f(x) = 0$ . The aim is to find  $f(x) = 0$ , we do so by dividing the interval  $[x_l, x_u]$  into two equal parts by inserting a midpoint  $x_m = \frac{1}{2}(x_l + x_u)$ . The function is also evaluated at this midpoint  $x_m = \frac{1}{2}(x_l + x_u)$ . All that we have to do now is to decide in which of the two new intervals the solution lies. Either the solution lies between the lower bound  $x_l$  and the midpoint  $x_m$ , or the solution lies between the midpoint  $x_m$  and the upper bound  $x_u$ . We repeat this operation until the absolute difference between  $x_l$  and  $x_u$  is less than a specified tolerance and then return the midpoint of  $x_l$  and  $x_u$ .

The inputs required for our bisection function are therefore  $x_l, x_u$ , the function  $f$  and a tolerance. To write a recursive function that will use the bisection method to find a zero of  $f$  we need to identify when to stop. As stated above we stop when our interval is very small otherwise we continue to bisect our interval. This is exactly what the recursive example computes,

```

1  from math import pi, cos
2
3
4  def bisection(x_lower, x_upper, func, tol=1E-6):
5      x_mid = (x_lower + x_upper) / 2
6      if abs(x_lower - x_upper) < tol:
7          return x_mid
8      elif func(x_lower) * func(x_mid) > 0:
9          return bisection(x_mid, x_upper, func, tol)
10     else:
11         return bisection(x_lower, x_mid, func, tol)
12
13
14 x_mid = bisection(0, pi, cos)
15 print "x = ", x_mid

```

The function `bisection` takes has the four required inputs. We then compute the midpoint. We then check whether our interval distance is less than the user specified tolerance. If it is we return our computed midpoint which is just a scalar. If it is still bigger than the user specified tolerance, we determine in which interval the sign change occurs and bisect that interval using our function `bisection`.

Let us compute the root of  $\cos(x)$  using our function `bisection` with an initial interval between 0 and  $\pi$ ,

```
1  x = 1.5707967013
```

Note how I used an anonymous function to send  $\cos(x)$  as an input argument to the function `bisection`.

## 15.2 Exercises

1. Write a function `fib` that takes an integer  $N$  as input. The function must then compute the  $N^{\text{th}}$  term of the Fibonacci sequence using recursion. The Fibonacci sequence is given by

$$0, 1, 1, 2, 3, 5, 8, 13, 21, \dots$$

[Hint: The definition of the Fibonacci sequence is as follows, the first term  $N = 1$  is defined as 0 and the second term  $N = 2$  is defined as 1. The  $N^{\text{th}}$  Fibonacci term

`fib(N)` is defined as the sum of the  $(N - 1)^{\text{th}}$  term `fib(N-1)` plus the  $(N - 2)^{\text{th}}$  term `fib(N-2)`.]

Why is the recursive version of computing the Fibonacci sequence so slow. Consider as an example  $N = 25$ ? Can you explain why?



# Chapter 16

## numpy and scipy capabilities

### 16.1 Solving systems of equations

#### 16.1.1 Solving linear systems of equations

Recall that we have already solved a linear system of equations  $\mathbf{Ax} = \mathbf{b}$  using the Gauss elimination. We were able to solve an arbitrary  $N \times N$  system with our Gauss elimination function. Before we were able to solve a linear system of equations we had to write Gauss elimination from scratch. We started by figuring out the logic of the forward reduction step and then by coding it in Python. Thereafter we figured out the logic of the back substitution step, coding it and only then we were able to solve a linear system of equations.

Well, `numpy.linalg` has the built in function `solve` that enables us to solve a linear system of equations in a very fast and efficient way. Recall the system

$$\begin{bmatrix} 2 & 3 & 4 & 1 \\ 1 & 1 & 2 & 1 \\ 2 & 4 & 5 & 2 \\ 1 & 2 & 3 & 4 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{Bmatrix} = \begin{Bmatrix} 10 \\ 5 \\ 13 \\ 10 \end{Bmatrix} \quad (16.1)$$

of which the solution is

$$\mathbf{x} = \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{Bmatrix} = \begin{Bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{Bmatrix} \quad (16.2)$$

Let us define matrix  $\mathbf{A}$  and vector  $\mathbf{b}$  in Python and use `numpy.linalg.solve` to solve

the linear system of equations  $\mathbf{Ax} = \mathbf{b}$ :

```

1 from numpy import array
2 from numpy.linalg import solve
3
4
5 mat_A = array([[2, 3, 4, 1],
6                 [1, 1, 2, 1],
7                 [2, 4, 5, 2],
8                 [1, 2, 3, 4]])
9
10 vec_B = array([[10], [5], [13], [10]])
11
12
13 x_sol = solve(mat_A, vec_B)
14 print "x = \n", x_sol

```

The output of this example is:

```

1 x =
2 [[ 1.]
3  [ 1.]
4  [ 1.]
5  [ 1.]]

```

### 16.1.2 Solving overdetermined linear systems of equations

What happens when  $\mathbf{A}$  is not an  $N \times N$  system? When  $\mathbf{A}$  is a  $M \times N$  system with  $M > N$  the linear system of equations is referred to as *overdetermined* as there are too many equations for the number of unknowns. For example if we add the additional equation

$$4x_1 + 3x_2 + x_3 + 5x_4 = 7$$

to the above system given in (16.1) we obtain the following *overdetermined* system of linear equations

$$\begin{bmatrix} 2 & 3 & 4 & 1 \\ 1 & 1 & 2 & 1 \\ 2 & 4 & 5 & 2 \\ 1 & 2 & 3 & 4 \\ 4 & 3 & 1 & 5 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{Bmatrix} = \begin{Bmatrix} 10 \\ 5 \\ 13 \\ 10 \\ 7 \end{Bmatrix} \quad (16.3)$$

that has five equations and only four unknowns. Because there are too many equations and too few variables to satisfy each equation exactly, we only satisfy the equations approximately  $\mathbf{Ax} \approx \mathbf{b}$ , resulting in an error. To minimize the error a least squares method is used, which is given by

$$\mathbf{x} = (\mathbf{A}^T \mathbf{A})^{-1} \mathbf{A}^T \mathbf{b}$$

This overdetermined system can be solved using the `numpy.linalg.lstsq` function:

```

1 from numpy import array, dot
2 from numpy.linalg import lstsq
3
4
5 mat_A = array([[2, 3, 4, 1],
6                 [1, 1, 2, 1],
7                 [2, 4, 5, 2],
8                 [1, 2, 3, 4],
9                 [4, 3, 1, 5]])
10
11 vec_B = array([[10], [5], [13], [10], [7]])
12
13
14 results = lstsq(mat_A, vec_B)
15 x_sol = results[0]
16 print "x = \n", x_sol
17 print "\n", dot(mat_A, x_sol)

```

for which we obtain

```

1 x =
2 [[-0.09529277]
3 [ 0.46268657]
4 [ 1.96440873]
5 [ 0.80711825]]
6
7 [[ 9.86222732]
8 [ 5.10332951]
9 [ 13.09644087]
10 [ 9.95177956]
11 [ 7.00688863]]

```

We can see there is an error when we multiply  $\mathbf{A}\mathbf{x}$  we do not obtain the right hand side vector  $\mathbf{b}$  exactly but only approximately as shown above,

### 16.1.3 Solving underdetermined linear systems of equations

Let us see what happens when is a  $\mathbf{A} M \times N$  system, with  $M < N$ . The linear system of equations is then referred to as *underdetermined* as there are too few equations for the number of unknowns. For example if we remove the equation

$$x_1 + 2x_2 + 3x_3 + 4x_4 = 10$$

of the system given in (16.1) we obtain the following *underdetermined* system of linear equations

$$\begin{bmatrix} 2 & 3 & 4 & 1 \\ 1 & 1 & 2 & 1 \\ 2 & 4 & 5 & 2 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{Bmatrix} = \begin{Bmatrix} 10 \\ 5 \\ 13 \end{Bmatrix} \quad (16.4)$$

that has only three equations and four unknowns. Because there are too few equations and too many variables, we have an infinite number of solutions each of which satisfy the equations exactly. How do we distinguish between the infinite number of solutions to obtain an unique solution? When we apply the following least squares method

$$\mathbf{x} = \mathbf{A}^T(\mathbf{A}\mathbf{A}^T)^{-1}\mathbf{b}$$

to an *underdetermined* system of equation to obtain the solution of  $\mathbf{x}$  that minimises the length (norm) of  $\mathbf{x}$ .

`numpy.linalg.lstsq` computes the minimum norm solution of the underdetermined system

```

1  from numpy import array, dot
2  from numpy.linalg import lstsq
3
4
5  mat_A = array([[2, 3, 4, 1],
6                  [1, 1, 2, 1],
7                  [2, 4, 5, 2]])
8
9  vec_B = array([[10], [5], [13]])
10

```

```

11
12 results = lstsq(mat_A, vec_B)
13 x_sol = results[0]
14 print "x = \n", x_sol
15 print "\n", dot(mat_A, x_sol)

```

for which we obtain

```

1 x =
2 [[ 0.6]
3 [ 0.8]
4 [ 1.4]
5 [ 0.8]]
6
7 [[ 10.]
8 [ 5.]
9 [ 13.]]

```

We can see that  $\mathbf{Ax}$  satisfies the right hand side vector  $\mathbf{b}$  exactly as shown above. To show you that other solution exist, consider one of the infinite solutions

$$\mathbf{x} = \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{Bmatrix} = \begin{Bmatrix} -1 \\ 0 \\ 3 \\ 0 \end{Bmatrix} \quad (16.5)$$

which also solves the system exactly, as shown below

```

1 from numpy import array, dot
2
3 mat_A = array([[2, 3, 4, 1],
4 [1, 1, 2, 1],
5 [2, 4, 5, 2]])
6
7 print dot(mat_A, array([-1, 0, 3, 0]))

```

for which we obtain

```
1 [[10]
2 [ 5]
3 [13]]
```

### 16.1.4 Solving Non-Linear systems of equation

To solve non-linear systems of equations we are required to iterate on the solution until our solution is within a specified accuracy. For example consider the equation below,

$$\sin(x) + \cos(x) = 1.25 \quad (16.6)$$

Which values of  $x$  would satisfy (16.6)? Are there more than one  $x$  that would satisfy (16.6)? How could we go about finding an  $x$  that would satisfy (16.6)? We could rewrite (16.6) by subtracting 1.25 on both sides,

$$\sin(x) + \cos(x) - 1.25 = 0 \quad (16.7)$$

and find a root of (16.7). Recall the bisection and Newton's method which we used to find roots. But instead of writing our own program to find a root of (16.7), we are going to use the function `scipy.optimize.fsolve`. Type `help(fsolve)` (after importing it) and the following is displayed,

```
1 .
2 .
3 .
4 Find the roots of a function.
5
6 Return the roots of the (non-linear) equations defined by
7 ``func(x) = 0`` given a starting estimate.
8
9 Parameters
10 -----
11 func : callable f(x, *args)
12     A function that takes at least one (possibly vector) argument.
13 x0 : ndarray
14     The starting estimate for the roots of ``func(x) = 0``.
15 args : tuple
16     Any extra arguments to 'func'.
17 fprime : callable(x)
```

```
18     A function to compute the Jacobian of 'func' with derivatives
19     across the rows. By default, the Jacobian will be estimated.
20 full_output : bool
21     If True, return optional outputs.
22 col_deriv : bool
23     Specify whether the Jacobian function computes derivatives down
24     the columns (faster, because there is no transpose operation).
25
26 Returns
27 -----
28 x : ndarray
29     The solution (or the result of the last iteration for
30     an unsuccessful call).
31 infodict : dict
32     A dictionary of optional outputs with the keys::
33
34     * 'nfev': number of function calls
35     * 'njev': number of Jacobian calls
36     * 'fvec': function evaluated at the output
37     * 'fjac': the orthogonal matrix, q, produced by the QR
38             factorization of the final approximate Jacobian
39             matrix, stored column wise
40     * 'r': upper triangular matrix produced by QR factorization of same
41             matrix
42     * 'qtf': the vector (transpose(q) * fvec)
43
44 ier : int
45     An integer flag. Set to 1 if a solution was found, otherwise refer
46     to 'mesg' for more information.
47 mesg : str
48     If no solution is found, 'mesg' details the cause of failure.
49 .
50 .
51 .
```

```
1 from math import sin, cos
2 from scipy.optimize import fsolve
3
4
5 def my_func(x):
6     return sin(x) + cos(x) - 1.25
7
8
```

```

9  def main():
10     x_start = 0
11     solution = fsolve(my_func, x_start)
12     print "root = ", solution[0]
13     print "f(x) = ", my_func(solution[0])
14
15
16 if __name__ == "__main__":
17     main()

```

To use `fsolve` we need to define a function  $f(x)$ . The function `fsolve`, solves  $f(x)$  such that  $f(x) = 0$  i.e. find a root of  $f(x)$ . I create a function `myfunc` of (16.7) in program lines 5–6. Before we can find a root of (16.7), we need to guess a starting point, let us guess  $x_0 = 0$  (line 10) and use `fsolve` to find  $x$  (line 11). The following output is displayed

```

1  root =  0.298703208323
2  f(x) = -3.5527136788e-15

```

which shows that  $f(x)$  with  $x = 0.29870$  satisfies the equation  $f(x) = 0$ . Let us re-run this example with an initial guess  $x_0 = 1$ , program line 10 changes to `x_start = 1` and we then obtain

```

1  root =  1.27209311847
2  f(x) = -6.66133814775e-16

```

which is another root of (16.7) and as shown  $x = 1.2721$  satisfies (16.7). In the examples above we only specified a function  $f(x)$  and starting point  $x_0$  when using `fsolve`. The function `fsolve` uses the derivative of  $f(x)$  to help it locate a root of the equation. When the derivative of  $f(x)$  is not specified it is computed numerically. However, we can specify the derivative  $\frac{df(x)}{dx}$  together with the function  $f(x)$ . Consider the example below

```

1  from math import sin, cos
2  from scipy.optimize import fsolve
3
4

```

```

5  def my_func(x):
6      return sin(x) + cos(x) - 1.25
7
8  def deriv_func(x):
9      return [cos(x) -sin(x)]
10
11
12 def main():
13     x_start = 1
14     solution = fsolve(my_func, x_start, fprime=deriv_func)
15     print "root = ", solution[0]
16     print "f(x) = ", my_func(solution[0])
17
18
19 if __name__ == "__main__":
20     main()

```

which gives

```

1 root = 1.27209311847
2 f(x) = -8.881784197e-16

```

Up to now, when dealing with non-linear systems, we only solved a single equation that depends on a single variable. But can we solve a system of equations i.e. many non-linear equations where each non-linear equation may depend on many variables. The function `fsolve` also allows us to solve systems of non-linear equations  $F(\mathbf{x}) = \mathbf{0}$ ,

$$f(\mathbf{x}) = \begin{bmatrix} F_1(x_1, x_2, \dots, x_n) \\ F_2(x_1, x_2, \dots, x_n) \\ \vdots \\ F_n(x_1, x_2, \dots, x_n) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \mathbf{0} \quad (16.8)$$

where each  $F_i$ ,  $i = 1, 2, \dots, n$  corresponds to an equation. For example consider the following non-linear system of  $n = 2$  equations

$$\begin{aligned} x_1^2 + \cos(x_2) &= 3 \\ \sin(x_1) - \sqrt{x_2} &= -2 \end{aligned}$$

The system is non-linear since  $x_1$  and  $x_2$  is a function of  $\sin, \cos$ , square root and squares. We can't write the system as a constant matrix  $\mathbf{A}$  multiplied with a vector of  $\mathbf{x}$

as we did for solving linear systems. To use `fsolve` we have to rewrite the above system

$$\begin{aligned}x_1^2 + \cos(x_2) - 3 &= 0 \\ \sin(x_1) - \sqrt{x_2} + 2 &= 0\end{aligned}$$

such that each equation is equal to zero. We can then find a root of the  $n = 2$  system of equations. The function `fsolve` requires the system to be written in a vector form where each component of the vector corresponds to an equation. We have to construct a function that takes a vector  $\mathbf{x}$  as input and returns a vector  $F(\mathbf{x})$  as output.

```

1  from math import sin, cos
2  from scipy.optimize import fsolve
3
4
5  def my_func(x):
6      return [x[0]**2 + cos(x[1]) - 3,
7              sin(x[0]) - x[1]**0.5 + 2]
8
9
10 def main():
11     x_start = [1, 2]
12     solution = fsolve(my_func, x_start)
13     print "root = ", solution
14     print "f(x) = ", my_func(solution)
15
16
17 if __name__ == "__main__":
18     main()
```

Our initial guess therefore also needs to be a vector as we have to guess an initial value for both  $x_1$  and  $x_2$ . We can then use `fsolve` to solve for the vector valued function using our initial guess vector. The function `fsolve` will then return a vector containing the solution  $\mathbf{x}^*$ .

The initial guess will determine the time `fsolve` takes to solve the non-linear system of equations and if multiple solutions are present it will determine to which solution you converge. Let us start with an initial guess  $x = [1; 2]$  and see what solution we obtain when we use `fsolve`

```

1  root =  [ 1.92420943  8.63300101]
2  f(x) =  [-8.0447204453548693e-11, -8.2178708282754087e-12]
```

Let us change our starting position to  $x = [0.5; 1]$  and see if we can find another solution to the system of equations. Program line 11 changes to `x_start = [0.5, 1]` and the output is:

```

1 .../test2.py:7: RuntimeWarning: invalid value encountered
2 in double_scalars
3     sin(x[0]) - x[1]**0.5 + 2]
4 .../minpack.py:152: RuntimeWarning: The iteration is not
5 making good progress, as measured by the improvement from
6 the last ten iterations.
7     warnings.warn(msg, RuntimeWarning)

```

On the way to the solution `fsolve` encountered had to evaluate the square root of negative values as stated in the warning message and could not solve out system of equations starting from this given starting point.

As with the solution of single non-linear equation we can also specify the derivative (Jacobian) of the system w.r.t. each of the variables  $x_i, i = 1, 2, \dots, n$  i.e.  $n \times n$  matrix of equations. Type `help fsolve` for additional help on computing the Jacobian.

### 16.1.5 Equilibrium of non-linear springs in series

Let us consider a realistic example of a non-linear system of equations. Consider the configuration of springs as shown in Figure 16.1,

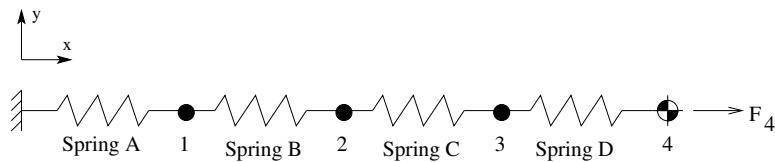


Figure 16.1: Four spring connected in series with a force  $F_4$  applied at the right hand side.

Our aim is to find the positions of points 1, 2, 3 and 4 as shown in Figure 16.1 when the system is in equilibrium after applying  $F_4$ . Normally springs are modelled as linear springs i.e. the force  $F$  displacement  $x$  relationship is given by

$$F = kx$$

where  $k$  is a constant. However, in our case the springs are non-linear and the force  $F$  displacement  $x$  relationship is given by

$$F = k(\text{sign}(x)x^2)$$

where  $k$  is a constant and  $\text{sign}(x)$  is  $+1$  when  $x > 0$ ,  $-1$  when  $x < 0$  and  $0$  when  $x = 0$ . The sign is required to distinguish between tension and compression since  $x^2$  is always bigger or equal to  $0$  for  $x$  a real number. Therefore  $x^2$  does not allow us to distinguish between tension and compression of the spring.

As the spring stretches you need to pull much harder than is the case with the linear spring. For the linear spring you need to pull twice as hard  $2F$  when you want to double the stretch distance  $2x$ . For the non-linear spring you need to pull 4 times as hard  $4F$  when you want to double the stretch distance  $2x$ . Let us write down the equilibrium equations of each of the four points of which we want to know the displacement after the force  $F_4$  is applied.

When  $F_4$  is equal to zero and the system is in equilibrium we consider the displacements  $u_i$ ,  $i = 1, 2, 3, 4$  of the four points 1,2,3 and 4 to be zero i.e. our reference configuration.

When we move point 1 to the right (that is  $u_1$  is positive) then spring A pulls it to the left  $F_A = -k_A(\text{sign}(u_1)(u_1)^2)$ . The change in length of spring B is  $u_2 - u_1$  which pushes point 1 to the left when  $u_1 > u_2$  and pulls point 1 to the right when  $u_1 < u_2$ . The force of spring B on point 1 is given by  $F_B = k_B((u_2 - u_1) + \text{sign}(u_2 - u_1)(u_2 - u_1)^2)$ . Since we want to determine the displacements of the positions when the system is in equilibrium the sum of the forces at point 1 must be equal to zero

$$F_A + F_B = 0$$

to give

$$-k_A(\text{sign}(u_1)(u_1)^2) + k_B(\text{sign}(u_2 - u_1)(u_2 - u_1)^2) = 0$$

Let us consider point 2. When  $u_2 > u_1$  spring B pulls point 2 to the left. The length of spring B is given by  $u_2 - u_1$  to give  $F_B = -k_B(\text{sign}(u_2 - u_1)(u_2 - u_1)^2)$ . Similarly, the length of spring C is given by  $u_3 - u_2$ . When  $u_3 > u_2$  then spring C pulls point 2 to the right. The force is therefore given by  $F_C = k_C(\text{sign}(u_3 - u_2)(u_3 - u_2)^2)$ . Again, the sum of the forces at point 2 must be equal to zero

$$F_B + F_C = 0$$

to give

$$-k_B(\text{sign}(u_2 - u_1)(u_2 - u_1)^2) + k_C(\text{sign}(u_3 - u_2)(u_3 - u_2)^2) = 0$$

Similarly, the forces at point 3 is given by

$$F_C + F_D = 0$$

to give

$$-k_C(\text{sign}(u_3 - u_2)(u_3 - u_2)^2) + k_D(\text{sign}(u_4 - u_3)(u_4 - u_3)^2) = 0$$

and lastly the sum of the forces at point 4 are the force in spring D and the external force  $F$  applied to the system,

$$F_D + F = 0$$

to give

$$-k_D(\text{sign}(u_4 - u_3)(u_4 - u_3)^2) + F_4 = 0$$

Let us consider a system where  $k_A = k_B = k_C = k_D = 1$  with an external force  $F_4 = 1$  which is represented in the function `spring_system` below

```

1  from scipy.optimize import fsolve
2
3
4  def sign_x(disp):
5      """
6          returns the sign of disp multiplied by disp^2
7      """
8      if disp > 0:
9          return disp**2
10     elif disp < 0:
11         return -(disp**2)
12     else:
13         return 0.0
14
15
16 def spring_system(disp):
17     system = []
18     force = 1.0
19     k_vals = (1.0, 1.5, 0.5, 3.0)
20
21     system.append(-k_vals[0] * sign_x(disp[0]) + \
22                   k_vals[1] * sign_x(disp[1] - disp[0]))
23
24     system.append(-k_vals[1] * sign_x(disp[1] - disp[0]) + \
25                   k_vals[2] * sign_x(disp[2] - disp[1]))
26
27     system.append(-k_vals[2] * sign_x(disp[2] - disp[1]) + \
28                   k_vals[3] * sign_x(disp[3] - disp[2]))
```

```

29     system.append(-k_vals[3] * sign_x(disp[3] - disp[2]) + force)
30     return system
31
32
33
34 def main():
35     x_start = [0., 1., 0., 1.]
36     solution = fsolve(spring_system, x_start)
37     print "root = ", solution
38     print "f(x) = ", spring_system(solution)
39
40
41 if __name__ == "__main__":
42     main()

```

note the use of the function `sign_x` which returns  $+x^2$  when the input is positive,  $-x**2$  for negative input and 0 when the input to `sign_x` is zero.

Let us solve the system using an initial guess of  $\mathbf{u} = [0 \ 1 \ 0 \ 1]$

```

1 root = [ 1.          1.81649658  3.23071014  3.80806041]
2 f(x) = [1.1025402812947505e-11, -1.2604928212311961e-10,
3           2.8097080218003612e-11,  8.1849083066742878e-11]

```

and when we set the external force  $F_4 = 4$  in `spring_system` we double the displacement as expected,

```

1 root = [ 2.          3.63299317  6.46142029  7.61612083]
2 f(x) = [1.447645381347229e-08, -3.1230538155568865e-08,
3           4.3613482603177545e-08, -2.9053676797730077e-08]

```

using an initial guess of  $\mathbf{u} = [0 \ 1 \ 0 \ 1]$ . From the above results the solution to the system seems trivial, since all the springs see the same force  $F_4$  we applied at the tip of the system. Due to the simple load case superposition holds and we could have computed the displacement of one spring and the rest follows from super position. For example, the first spring extends by 1, then the second spring also extends by 1 plus the extension of the previous springs etc.

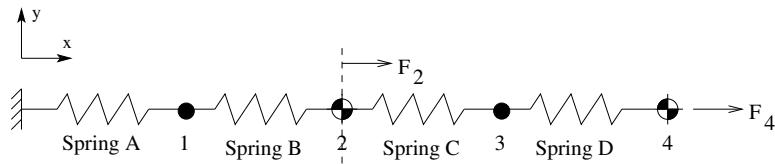


Figure 16.2: Four spring connected in series with two applied forces  $F_2$  and  $F_4$ .

However, if we consider a system an additional force  $F_2$  to the system as shown in Figure 16.2.

The system is given below,

```

1  from scipy.optimize import fsolve
2
3
4  def sign_x(disp):
5      """
6          returns the sign of disp multiplied by disp^2
7      """
8
9      if disp > 0:
10         return disp**2
11     elif disp < 0:
12         return -(disp**2)
13     else:
14         return 0.0
15
16
17  def spring_system(disp):
18      system = []
19      force2 = 1.0
20      force4 = 1.0
21      k_vals = (1.0, 1.0, 1.0, 1.0)
22
23      system.append(-k_vals[0] * sign_x(disp[0]) + \
24                      k_vals[1] * sign_x(disp[1] - disp[0]))
25
26      system.append(-k_vals[1] * sign_x(disp[1] - disp[0]) + \
27                      k_vals[2] * sign_x(disp[2] - disp[1]) + force2)
28
29      system.append(-k_vals[2] * sign_x(disp[2] - disp[1]) + \
30                      k_vals[3] * sign_x(disp[3] - disp[2]))
31
32      system.append(-k_vals[3] * sign_x(disp[3] - disp[2]) + force4)

```

```

32     return system
33
34
35 def main():
36     x_start = [0., 1., 0, 1.]
37     solution = fsolve(spring_system, x_start)
38     print "root = ", solution
39     print "f(x) = ", spring_system(solution)
40
41
42 if __name__ == "__main__":
43     main()

```

with  $F_2$  added to the second equation of our system `system(2)`. Let us solve the system with initial guess  $\mathbf{u} = [0 \ 1 \ 0 \ 1]$

```

1 root =  [ 1.41421356  2.82842712  3.82842712  4.82842712]
2 f(x) =  [-1.0368217395750889e-08,  4.9386512657179082e-09,
           4.9667274737430489e-09, -3.558334960018783e-09]

```

## 16.2 Polynomials in Python

To represent polynomials in Python we already have various ways at our disposal. For example, to define the polynomial  $x^4 + 3x^2 + 1$  in Python we can create a function

```

1 def my_poly(x):
2     return x**4 + 3 * x**2 + 1

```

which allows us to easily evaluate the polynomial at various values of  $x$ . However, it does not allow us to compute the roots, derivatives or integrals of polynomials.

Python has a number of functions that does operations on polynomials. However to use these functions we have to define a polynomial in a specific way. These functions require the coefficients of the polynomial to be stored in a vector. The following was obtained from the `numpy` documentation by typing:

```
1 from numpy.polynomial import polynomial as poly  
2  
3 help(poly)
```

and the following is displayed,

```
1 .  
2 .  
3 .  
4 Objects for dealing with polynomials.  
5  
6 This module provides a number of objects (mostly functions) useful for  
7 dealing with polynomials, including a 'Polynomial' class that  
8 encapsulates the usual arithmetic operations. (General information  
9 on how this module represents and works with polynomial objects is in  
10 the docstring for its "parent" sub-package, 'numpy.polynomial').  
11  
12 Constants  
13 -----  
14 - 'polydomain' -- Polynomial default domain, [-1,1].  
15 - 'polyzero' -- (Coefficients of the) "zero polynomial."  
16 - 'polyone' -- (Coefficients of the) constant polynomial 1.  
17 - 'polyx' -- (Coefficients of the) identity map polynomial, ``f(x) = x''.  
18  
19 Arithmetic  
20 -----  
21 - 'polyadd' -- add two polynomials.  
22 - 'polysub' -- subtract one polynomial from another.  
23 - 'polymul' -- multiply two polynomials.  
24 - 'polydiv' -- divide one polynomial by another.  
25 - 'polypow' -- raise a polynomial to an positive integer power  
26 - 'polyval' -- evaluate a polynomial at given points.  
27  
28 Calculus  
29 -----  
30 - 'polyder' -- differentiate a polynomial.  
31 - 'polyint' -- integrate a polynomial.  
32  
33 Misc Functions  
34 -----  
35 - 'polyfromroots' -- create a polynomial with specified roots.
```

```
36     - 'polyroots' -- find the roots of a polynomial.  
37     - 'polyvander' -- Vandermonde-like matrix for powers.  
38     - 'polyfit' -- least-squares fit returning a polynomial.  
39     - 'polytrim' -- trim leading coefficients from a polynomial.  
40     - 'polyline' -- polynomial representing given straight line.  
41     .  
42     .  
43     .
```

and typing `help(poly.polyval)` then displays the following output:

```
1 .  
2 .  
3 .  
4 Evaluate a polynomial.  
5  
6 If 'cs' is of length 'n', this function returns :  
7  
8   "p(x) = cs[0] + cs[1]*x + ... + cs[n-1]*x**(n-1)"  
9  
10 If x is a sequence or array then p(x) will have the same shape as x.  
11 If r is a ring_like object that supports multiplication and addition  
12 by the values in 'cs', then an object of the same type is returned.  
13  
14 Parameters  
15 -----  
16 x : array_like, ring_like  
17     If x is a list or tuple, it is converted to an ndarray. Otherwise  
18     it must support addition and multiplication with itself and the  
19     elements of 'cs'.  
20 cs : array_like  
21     1-d array of Chebyshev coefficients ordered from low to high.  
22  
23 Returns  
24 -----  
25 values : ndarray  
26     The return array has the same shape as 'x'.  
27 .  
28 .  
29 .
```

Therefore our polynomial  $x^4 + 3x^2 + 1$  would be expressed as coefficients ordered from low to high ( $1 + 0x^1 + 3x^2 + 0x^3 + 1x^4$ )

```
1 poly_coeffs = [1, 0, 3, 0, 1]
```

We need to remember to include the coefficients lower than the non-zero coefficient of the highest order of  $x$  that are zero. For example  $0x^3$  and  $0x$  are included since the highest order polynomial that is non-zero is  $x^4$ .

If we had forgotten to include the zeros of the third and first order polynomials and we defined `poly_coeffs` as `[1, 3, 1]` we would have obtained the polynomial  $1 + 3x + x^2$  which is only a second order polynomial. Therefore the length of our coefficient vector minus one is the highest order of the polynomial.

The function `polyval` is then handy, it takes two input arguments. Firstly, a vector containing the  $x$  values where to evaluate the polynomial and secondly a polynomial coefficient vector:

```
1 from numpy.polynomial import polynomial as poly
2
3 print poly.polyval(2, [1, 0, 3, 0, 1])
```

which displays 29 as the output.

To obtain the roots of  $x^4 + 3x^2 + 1$  we can use the `roots` function which takes a polynomial coefficient vector as input argument and returns the roots of the respective polynomial. The roots of  $x^4 + 3x^2 + 1$  is given by

```
1 from numpy.polynomial import polynomial as poly
2
3 print poly.polyroots([1, 0, 3, 0, 1])
```

which gives the four complex roots of  $x^4 + 3x^2 + 1$

```
1 [ 0.-1.61803399j
2   0.-0.61803399j]
```

```

3  0.+0.61803399j
4  0.+1.61803399j]
```

We can compute the product of two polynomials by using `polymul` which stands for *polynomial multiple*. The function `polymul` takes two polynomial coefficient vectors as input and computes the product of the polynomials which it returns as output. Therefore, if we want to multiply  $x - 1$  with  $x + 1$  to obtain  $x^2 - 1$  we can simply type

```

1 from numpy.polynomial import polynomial as poly
2
3
4 print poly.polymul([-1, 1], [1, 1])
```

to get the resulting polynomial of:

```

1 [ -1.,  0.,  1.]
```

If we factor  $x - 1$  out of  $x^2 - 1$  then  $x + 1$  is left with no remainder as shown below

$$\frac{x^2 - 1}{x - 1} = \frac{(x - 1)(x + 1)}{x - 1} = x + 1$$

To factor  $x - 1$  out of  $x^2 - 1$  we can use the `polydiv` function which computes the quotient and also the remainder. Therefore

```

1 from numpy.polynomial import polynomial as poly
2
3
4 print poly.polydiv([-1, 0, 1], [-1, 1])
```

gives

```

1 (array([ 1.,  1.]), array([ 0.]))
```

The remainder tells us that  $x - 1$  factored exactly into  $x^2 - 1$  to leave  $x + 1$ . If we however factor an arbitrary polynomial out of  $x^2 - 1$  the remainder may not be zero. For example, let us factor  $x + 4$  out of  $x^2 - 1$  as follows

```
1 from numpy.polynomial import polynomial as poly  
2  
3 print poly.polydiv([-1, 0, 1], [4, 1])
```

we then obtain

```
1 (array([-4.,  1.]), array([ 15.]))
```

Consequently, when we factor  $x + 4$  out of  $x^2 - 1$  we obtain  $x - 4$  with a remainder of 15. Therefore  $(x + 4) * (x - 4)$  plus remainder of 15 equals our original polynomial  $x^2 - 1$ .

In Python we can also fit polynomial curves through data points e.g. when you have experimental data for which you need to fit a curve to obtain an empirical (based on data) model. The function `polyfit` allows you to do just that. When you type `help polyfit` you will see the following output and more

```
1 .  
2 .  
3 .  
4 Least-squares fit of a polynomial to data.  
5  
6 Fit a polynomial ``c0 + c1*x + c2*x**2 + ... + c[deg]*x**deg`` to  
7 points ('x', 'y'). Returns a 1-d (if 'y' is 1-d) or 2-d (if 'y' is 2-d)  
8 array of coefficients representing, from lowest order term to highest,  
9 the polynomial(s) which minimize the total square error.  
10  
11 Parameters  
12 -----  
13 x : array_like, shape ('M',)  
14     x-coordinates of the 'M' sample (data) points ``(x[i], y[i])``.  
15 y : array_like, shape ('M',) or ('M', 'K')  
16     y-coordinates of the sample points. Several sets of sample points  
17     sharing the same x-coordinates can be (independently) fit with one  
18     call to 'polyfit' by passing in for 'y' a 2-d array that contains  
19     one data set per column.  
20 deg : int  
21     Degree of the polynomial(s) to be fit.  
22 .
```

23     .  
24     .

The function `polyfit` allows you to specify the order (degree) of the polynomial that you would like to fit through the data. The function `polyfit` then returns the coefficients of the polynomial in a vector e.g. when `polyfit` returns the vector [1 0 2 4 0 2] then it represents the following 5<sup>th</sup> order polynomial

$$y = 2x^5 + 0x^4 + 4x^3 + 2x^2 + 0x + 1 \quad (16.9)$$

We can then use `polyval` to evaluate a polynomial at specified values of  $x$ . Let us pretend that we obtained the following data from an experiment we conducted,

$$\mathbf{x} = \begin{Bmatrix} 0.5 \\ 1.4 \\ 2.0 \\ 2.7 \\ 3.2 \end{Bmatrix} \quad \mathbf{y} = \begin{Bmatrix} 0.4 \\ 2.1 \\ 4.2 \\ 7.1 \\ 10.4 \end{Bmatrix} \quad (16.10)$$

```
1  from numpy import linspace
2  from numpy.polynomial import polynomial as poly
3  from matplotlib import pyplot as plot
4
5
6  def main():
7      x_vals = [0.5, 1.4, 2.0, 2.7, 3.2]
8      y_vals = [0.4, 2.1, 4.2, 7.1, 10.4]
9
10     linecoef = poly.polyfit(x_vals, y_vals, 1)
11     quadcoef = poly.polyfit(x_vals, y_vals, 2)
12
13     xpoly = linspace(0, 5, 50)
14     yline = poly.polyval(xpoly, linecoef)
15     yquad = poly.polyval(xpoly, quadcoef)
16
17     plot.plot(x_vals, y_vals, 'ko',
18               xpoly, yline, 'r-',
19               xpoly, yquad, 'b-')
20     plot.legend(['Data', 'Linear', 'Quadratic'])
21
22     plot.show()
```

```

23
24
25 if __name__ == "__main__":
26     main()

```

I start off by define the vectors in lines 7 and 8. Then I fit a linear line (order 1) through the data (program line 10) as well as a quadratic polynomial (order 2)(program line 11). I then use `polyval` to evaluate the each of the polynomials at various values of  $x$ . I used 50 linearly spaced  $x$  values between 0 and 5 to evaluate each of the polynomials. I then plot the data as well as our linear and quadratic polynomials.

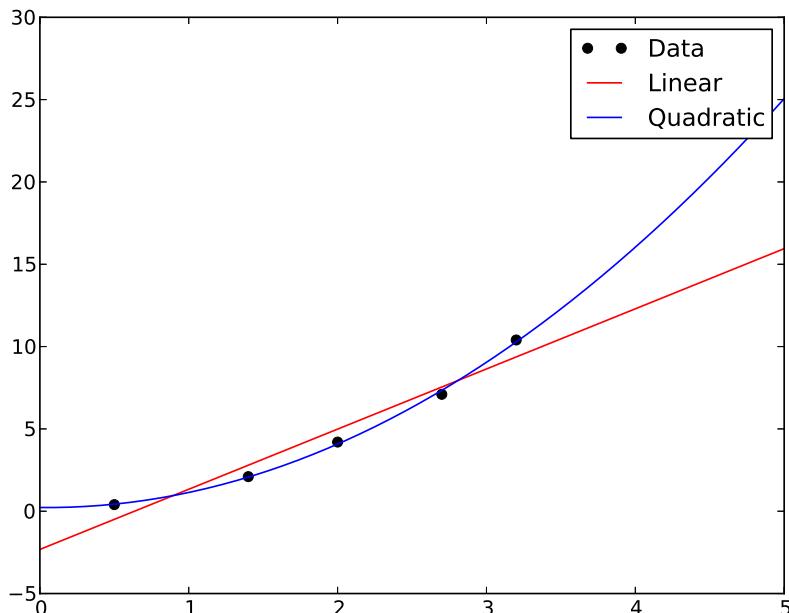


Figure 16.3: Experimental data fitted with a linear and a quadratic polynomial.

To compute the derivative of a polynomial we simply use `polyder` which takes a polynomial as input and return the derivative of a polynomial. For example, the derivative of  $2x^4 + 0.5x^2 - x + 3$  is  $8x^3 + x - 1$  which we can compute as follows

```

1 from numpy.polynomial import polynomial as poly
2
3
4 print poly.polyder([3, -1, 0.5, 0, 2])

```

to give

```
1 [-1.  1.  0.  8.]
```

In turn, we can also integrate a polynomial by simply using `polyint`. The integral of  $2x^4 + 0.5x^2 - x + 3$  is  $\frac{2}{5}x^5 + 16x^3 - \frac{1}{2}x^2 + 3x + c$  where  $c$  is the integration constant. The function `polyint` has two inputs. The first input is the polynomial we want to integrate and the second input specifies the integration constant  $c$  (optional). For example we compute the integral of  $2x^4 + 0.5x^2 - x + 3$  with an integration constant of  $c = 4$  as follows

```
1 from numpy.polynomial import polynomial as poly
2
3
4 print poly.polyint([3, -1, 0.5, 0, 2], k=4)
```

to give

```
1 [ 4.         3.         -0.5        0.16666667  0.          0.4 ]
```

### 16.3 Numerical integration

Recall that we have written our own function to numerically compute the integral of a function, remember the left-point, right-point and midpoint methods. We divided the area under a curve into rectangular section which we summed together. Well, Python has it's own built in function to integrate an arbitrary function of one variable, namely `quad`. For instance how would we go about integrating  $\cos(x)$  between 0 and  $\pi$  with `quad`. Let us start by typing `from scipy.integrate import quad` and then `help(quad)` to see the following

```
1 .
2 .
3 .
4 Compute a definite integral.
5
```

```
6   Integrate func from a to b (possibly infinite interval) using a technique
7   from the Fortran library QUADPACK.
8
9   If func takes many arguments, it is integrated along the axis corresponding
10  to the first argument. Use the keyword argument 'args' to pass the other
11  arguments.
12
13  Run scipy.integrate.quad_explain() for more information on the
14  more esoteric inputs and outputs.
15
16 Parameters
17 -----
18
19 func : function
20     A Python function or method to integrate.
21 a : float
22     Lower limit of integration (use -scipy.integrate.Inf for -infinity).
23 b : float
24     Upper limit of integration (use scipy.integrate.Inf for +infinity).
25 args : tuple, optional
26     extra arguments to pass to func
27 full_output : int
28     Non-zero to return a dictionary of integration information.
29     If non-zero, warning messages are also suppressed and the
30     message is appended to the output tuple.
31
32 Returns
33 -----
34
35 y : float
36     The integral of func from a to b.
37 abserr : float
38     an estimate of the absolute error in the result.
39
40 infodict : dict
41     a dictionary containing additional information.
42     Run scipy.integrate.quad_explain() for more information.
43 message :
44     a convergence message.
45 explain :
46     appended only with 'cos' or 'sin' weighting and infinite
47     integration limits, it contains an explanation of the codes in
48     infodict['ierlst']
49 .
```

```
50 .  
51 .
```

Therefore we have to specify  $\cos(x)$  as a function object, as well as the lower and upper bound over which we want to compute the numerical integral. The following example integrates  $\cos(x)$  between 0 and  $\pi$ ,

```
1 from math import cos, pi  
2 from scipy.integrate import quad  
3  
4  
5 print quad(cos, 0, pi)
```

to give the solution

```
1 4.9225526349740854e-17
```

which is correct from inspection. Now that we understand how `quad` works we can compute a more challenging integral. Consider for example the integral of

$$\int_2^5 \frac{e^{x-3}}{\ln(x)} dx \quad (16.11)$$

which we can compute using `quad`. In the example below I define the function that we want to compute (16.11)

```
1 from math import exp, log  
2 from scipy.integrate import quad  
3  
4  
5 def func(x):  
6     return exp(x-3) / log(x)  
7  
8  
9 def main():  
10    print quad(func, 2, 5)  
11
```

```
12  
13 if __name__ == "__main__":  
14     main()
```

to give

```
1 5.114501864734196
```

Let us verify the answer to the above integral by plotting  $\frac{e^{x-3}}{\ln(x)}$  between 2 and 5 using

```
1 from numpy import exp, log  
2 from numpy import linspace  
3 from matplotlib import pyplot as plot  
4 from scipy.integrate import quad  
5  
6  
7 def func(x):  
8     return exp(x-3) / log(x)  
9  
10  
11 def main():  
12     print quad(func, 2, 5)  
13     x = linspace(2, 5, 100)  
14  
15     plot.plot(x, func(x))  
16     plot.show()  
17  
18  
19 if __name__ == "__main__":  
20     main()
```

From Figure 16.4 we can approximate the area under the curve by the area of a triangle with base of 3 and height of 4. From Figure 16.4 it is clear that the area under the actual curve is less than the area of the triangle. The area of the triangle works out to be  $\frac{1}{2} \times \text{base} \times \text{height} = \frac{1}{2} \times 3 \times 4 = 6$ .

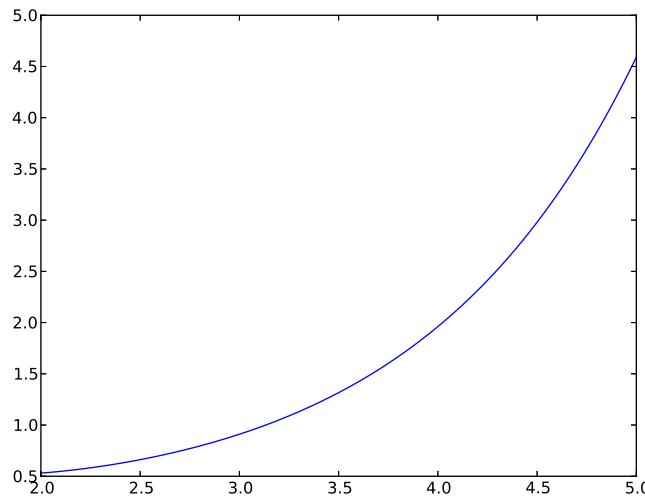


Figure 16.4: Plot of  $\frac{e^{x-3}}{\ln(x)}$  between 2 and 5.

## 16.4 Solving a system of linear differential equations

Consider the following system

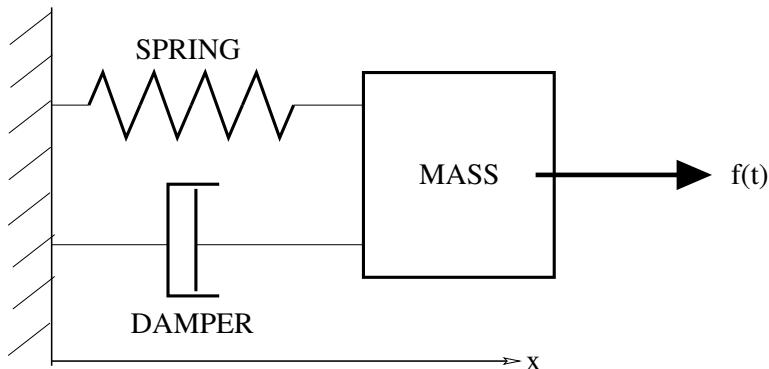


Figure 16.5: Spring-mass-damper system of a mass that move frictionless only in the  $x$ -direction.

which contains a mass  $m$ . Imagine that the mass  $m$  can only move in the  $x$ -direction, frictionless. The position of the mass  $m$  is given by  $x$ . Connected to the mass  $m$  is a spring and a damper which in turn is connected to a wall. There is some applied force  $f(t)$  applied to the mass that can vary with time. If the mass is moving around then the position  $x$  is a function of time  $t$  therefore  $x(t)$ . Our aim is to determine the position  $x(t)$  of the mass  $m$  for various values of  $t$ . But before we solve the system let us look at the mathematical model that describes the position of the mass  $m$ .

Let us start by considering the system in equilibrium with no external force  $f(t) = 0$  applied and let our coordinate axis start at that point i.e.  $x$  is zero at that point. If we move the mass to the right (in the positive x-direction) then the spring wants to pull the mass back (exerts a force to the left) to equilibrium. When we move the mass to the left (in the negative x-direction) then the spring wants to push the mass (exerts a force to the right) to equilibrium. We see that displacing the mass  $m$  in one direction results in a force on the mass in the opposite direction. We can therefore see that the force  $F_s$  the spring exerts on the mass is given by

$$F_s(t) = -kx(t) \quad (16.12)$$

where  $k$  is the spring stiffness given in Newton per meter. Damping is a force that is proportional to the velocity of a mass  $m$  and works in the opposite direction of the velocity  $v(t)$  at which a mass  $m$  is travelling. Think of closing the boot of your car or pushing and pulling on a door with a damper at the top. Let us use a linear damper where the damping force  $F_d$  is linearly proportional to the velocity by some damping coefficient  $c$ . The damping force  $F_d$  can then be written as

$$F_d(t) = -cv(t) = -c\dot{x}(t) \quad (16.13)$$

where  $\dot{x}(t)$  is the time derivative of position which is simply the velocity  $v(t)$  of the mass.

The external forces on the mass are the force exerted by the spring  $F_s$ , the force exerted by the damper  $F_d$  and also the time varying force  $f(t)$  applied to the mass which we made equal to zero in the discussion above. If we now apply Newton's second law,  $\sum F = \frac{d}{dt}(mv)$  which states that the sum of the external forces  $F$  is equal to the change of momentum  $\frac{d}{dt}(mv)$  of the system. But if the mass remains constant then the equation becomes the well known  $\sum F = ma = m\ddot{x}$ . Applying Newton's second law to our mass and summing the forces in the x-direction we obtain

$$F_s(t) + F_d(t) + f(t) = -kx(t) - cv(t) + f(t) = ma(t) \quad (16.14)$$

which we can rewrite by taking the spring force and damping force to the right hand side to obtain

$$f(t) = ma(t) + cv(t) + kx(t) = m\ddot{x}(t) + c\dot{x}(t) + kx(t) \quad (16.15)$$

Let us consider a spring mass damper system given by the following second order differential equation (second order since the second time derivative of  $x(t)$  occurs in the differential equation)

$$m\ddot{x}(t) + c\dot{x}(t) + kx(t) = f(t) \quad (16.16)$$

where  $\ddot{x}(t)$  is the acceleration of mass  $m$ , and  $\dot{x}(t)$  is the velocity with damping coefficient  $c$  and  $x(t)$  is the position of mass  $m$  relative to the free length of the spring with a spring constant of  $k$ . A time varying force  $f(t)$  is applied to mass  $m$ .

We can rewrite the second order differential equation as two first order differential equations which we can solve simultaneously. We do so by considering the position  $x(t)$  and velocity  $v(t)$  as independent variables. The first linear differential equation,  $v(t) = \dot{x}(t)$  relates  $x(t)$  and  $v(t)$ . The second linear differential equation,  $m\ddot{v}(t) + cv(t) + kx(t) = f(t)$  is obtained by appropriately substituting  $v(t)$  and  $x(t)$  into (16.16). The system of differential equations is then given by

$$\begin{aligned}\dot{x}(t) &= v(t) \\ m\ddot{v}(t) + cv(t) + kx(t) &= f(t)\end{aligned}\tag{16.17}$$

`scipy.integrate` has the function `odeint` which solves systems of linear ordinary differential equations. We can use the function `odeint` to solve for the system of linear ordinary differential equations given in (16.17). The function requires the system to be written such that only the first time derivatives appear on the left hand side of the equal sign and the rest appears on the right hand side as shown below

$$\begin{aligned}\dot{x}(t) &= v(t) \\ \dot{v}(t) &= \frac{f(t)}{m} - \frac{cv(t)}{m} - \frac{kx(t)}{m}\end{aligned}\tag{16.18}$$

Whatever appears on the right hand side of the equal sign is the required input for `odeint` as you will shortly see. In addition the function `odeint` requires the equation to be written in a vector form. We do so by substituting  $y_1(t) = x(t)$  and  $y_2(t) = v(t)$  in the system below

$$\begin{aligned}\dot{y}_1(t) &= y_2(t) \\ \dot{y}_2(t) &= \frac{f(t)}{m} - \frac{cy_2(t)}{m} - \frac{ky_1(t)}{m}\end{aligned}\tag{16.19}$$

where the subscript indicates the component of the vector  $\mathbf{y}(t)$ . To solve the system we need an initial conditions for  $x(t) = y_1(t)$  and  $v(t) = y_2(t)$ , the times at which we want to solve the system and the time varying force  $f(t)$  that acts on the system. We also need to specify the spring stiffness  $k$ , the linear damping coefficient  $c$  as well as the mass of the system. In the examples that follow I chose the spring stiffness 1 N/m, the linear damping coefficient 0.2 Ns/m and the mass 1 kg.

Let us consider the scenario where the initial velocity  $v(0) = y_2(0)$  is 0 and the initial position  $x(0) = y_1(0)$  is 10 with no time varying force acting on the system i.e.  $f(t) = 0$ . Let us solve the system for the time interval from 0 to 100 seconds in intervals of 0.5 seconds.

The function `odeint` takes the system as described by only the right hand side of the equal sign (see (16.19)) as the first input argument function object. The second

argument is a vector containing the initial conditions and the third argument is a vector containing the times at which to solve the system. The function then return a matrix with the dimensions of the rows equal to the number of time steps and the dimensions of the columns equal to the number of variables. The following code computes and plots the position and velocity of the mass

```
1  from numpy import linspace
2  from matplotlib import pyplot as plot
3  from scipy.integrate import odeint
4
5
6  def func(y, *args):
7      return [y[1],
8              -0.2 * y[1] - y[0]]
9
10
11 def main():
12     time = linspace(1, 100, 200)
13     results = odeint(func, [10, 0], time)
14
15     plot.plot(time, results[:, 0], 'r')
16     plot.plot(time, results[:, 1], 'b')
17     plot.legend(['position', 'velocity'])
18     plot.xlabel('time')
19
20     plot.show()
21
22
23 if __name__ == "__main__":
24     main()
```

## 16.5 Optimization

In most cases in Engineering many solutions exist to a problem where some of the solutions are considered better than others. The formal process of finding the best solution is referred to as optimization. For example, if we want to design a cooldrink can that holds 330 ml of cooldrink. What should the radius  $r$  and height  $h$  of the can be? Before we can start we need a criteria to judge whether a design is good or bad. In optimization this is referred to as a cost (or design) function.

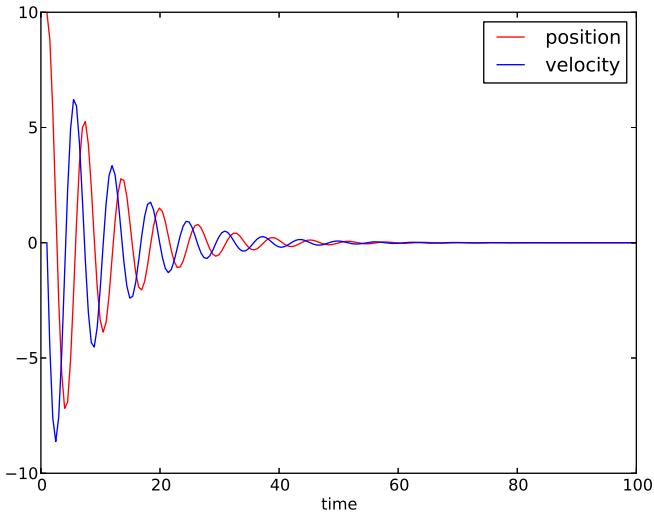


Figure 16.6: Response of the spring-mass-damper after displacing it 10 units from the equilibrium position.

For our can example we could say that we want to minimize the material used to make a can. If we assume the required thickness of the can to be independent of the dimension of the can, we can calculate the amount of the material used to make a can by calculating the surface area times the thickness of the can plate. So let us start by minimizing the surface area of a cylindrical can that holds 330 ml of cooldrink. The surface area of a cylindrical can is given by two times  $\pi r^2$  for the top and bottom parts and  $2\pi rh$  for the cylindrical section,

$$f(r, h) = 2\pi r^2 + 2\pi rh \quad (16.20)$$

The volume of the can has to be equal to 330 ml. For simplicity let us work in units of cm for which 330 ml is 330 cm<sup>3</sup>. The volume  $V(r, h)$  of the can is therefore

$$V = \pi r^2 h = 330 \text{cm}^3 \quad (16.21)$$

for  $r$  and  $h$  given in cm. We can write both (16.20) and (16.21) in a vector form which is required for the optimization function `fmin_slsqp` in `scipy.optimize`. Type `help(fmin_slsqp)` to see

```

1   .
2   .
3   .
4   Minimize a function using Sequential Least Squares Programming
5

```

```
6 Python interface function for the SLSQP Optimization subroutine
7 originally implemented by Dieter Kraft.
8
9 Parameters
10 -----
11 func : callable f(x,*args)
12     Objective function.
13 x0 : 1-D ndarray of float
14     Initial guess for the independent variable(s).
15 eqcons : list
16     A list of functions of length n such that
17     eqcons[j](x0,*args) == 0.0 in a successfully optimized
18     problem.
19 f_eqcons : callable f(x,*args)
20     Returns a 1-D array in which each element must equal 0.0 in a
21     successfully optimized problem. If f_eqcons is specified,
22     eqcons is ignored.
23 ieqcons : list
24     A list of functions of length n such that
25     ieqcons[j](x0,*args) >= 0.0 in a successfully optimized
26     problem.
27 f_ieqcons : callable f(x0,*args)
28     Returns a 1-D ndarray in which each element must be greater or
29     equal to 0.0 in a successfully optimized problem. If
30     f_ieqcons is specified, ieqcons is ignored.
31 bounds : list
32     A list of tuples specifying the lower and upper bound
33     for each independent variable [(xlo, xu0),(x1, xu1),...]
34 fprime : callable 'f(x,*args)'
35     A function that evaluates the partial derivatives of func.
36 fprime_eqcons : callable 'f(x,*args)'
37     A function of the form 'f(x, *args)' that returns the m by n
38     array of equality constraint normals. If not provided,
39     the normals will be approximated. The array returned by
40     fprime_eqcons should be sized as ( len(eqcons), len(x0) ).
41 fprime_ieqcons : callable 'f(x,*args)'
42     A function of the form 'f(x, *args)' that returns the m by n
43     array of inequality constraint normals. If not provided,
44     the normals will be approximated. The array returned by
45     fprime_ieqcons should be sized as ( len(ieqcons), len(x0) ).
46 args : sequence
47     Additional arguments passed to func and fprime.
48 iter : int
49     The maximum number of iterations.
```

```
50 acc : float  
51     Requested accuracy.  
52 .  
53 .  
54 .
```

```
1 from math import pi  
2 from numpy import array  
3 from scipy.optimize import fmin_slsqp  
4  
5  
6 def costfunc(x, *args):  
7     return 2 * pi * x[0]**2 + 2 * pi * x[0] * x[1]  
8  
9  
10 def equalcon(x, *args):  
11     return array([pi * x[0]**2 * x[1] - 330], dtype=float)  
12  
13  
14 def inequalcon(x, *args):  
15     return array([x[0], x[1]], dtype=float)  
16  
17  
18 def main():  
19     x_start = array([3, 12], dtype=float)  
20     results = fmin_slsqp(costfunc,  
21                           x_start,  
22                           f_eqcons=equalcon,  
23                           f_ieqcons=inequalcon)  
24  
25     print "\n", "x_opt = ", results  
26     print "f(x) = ", costfunc(results)  
27     print "g(x) = ", equalcon(results)  
28     print "h(x) = ", inequalcon(results)  
29  
30  
31 if __name__ == "__main__":  
32     main()
```

The function `fmin_slsqp` requires the cost function given by (16.20) to take a vector as input. We do so by relating the first entry of the vector  $x(1)$  to  $r$  and the second entry  $x(2)$  to  $h$  as shown in program lines 6–7. The volume of the can must equal 330cm

<sup>3</sup> and must be written in the form  $h(\mathbf{x}) = \mathbf{0}$  which is shown in program lines 10–11. In addition the radius  $x(1)$  and height  $x(2)$  has to be greater or equal to zero, as negative distances does not make sense. It is important to include these considerations as optimization algorithms may otherwise exploit non physical solutions in order to minimize the value of the cost function. The inequality constraint has to be written in the form  $g(\mathbf{x}) \geq \mathbf{0}$ , which is shown in program lines 14–15. We then require an initial guess for  $\mathbf{x}$ , let's guess a radius  $x(1)$  of 3 cm for which the height  $x(2)$  is then  $h = x(2) = \frac{330}{\pi x(1)^2} \approx 12$  cm. We can then use `fmin_slsqp` to compute the optimal dimensions for the 330 cm<sup>3</sup> as shown in program lines 19–23, which gives the following solution:

```

1 Optimization terminated successfully.      (Exit mode 0)
2         Current function value: 264.356810925
3             Iterations: 58
4             Function evaluations: 276
5             Gradient evaluations: 56
6
7 x_opt = [ 3.74491154  7.48998486]
8 f(x) = 264.356810925
9 g(x) = [ -5.82161874e-09]
10 h(x) = [ 3.74491154  7.48998486]
```

When we plug our solution back into our equality constraint function `equalcon` we see that we get an answer of very close to zero, thus the optimal dimensions of the can are feasible and clearly satisfies all the constraints. The optimal beverage can therefore has a radius that is half of the height of the can, or a diameter which is equal to the height of the can.

However the dimensions above do not correspond to the physical dimensions that we see in cans we buy in the shops, this tells that the amount of material used is not the only consideration when designing beverage cans, e.g. ergonomics and different materials used in cans as top parts are made from Aluminium whereas the rest is tin plated steel. For example, let us assume aluminium is twice as expensive as steel. If we consider designing a can of which the top part is made of aluminium and the rest of the can is steel we have the following

```

1 from math import pi
2 from numpy import array
3 from scipy.optimize import fmin_slsqp
4
5 def costfunc(x, *args):
```

```
7     cost_alu = 2.0
8     cost_steel = 1.0
9     return (cost_alu * pi * x[0]**2 +
10            cost_steel * (2 * pi * x[0] * x[1] + pi * x[0]**2))
11
12
13 def equalcon(x, *args):
14     return array([pi * x[0]**2 * x[1] - 330], dtype=float)
15
16
17 def inequalcon(x, *args):
18     return array([x[0], x[1]], dtype=float)
19
20
21 def main():
22     x_start = array([3, 12], dtype=float)
23     results = fmin_slsqp(costfunc,
24                           x_start,
25                           f_eqcons=equalcon,
26                           f_ieqcons=inequalcon)
27
28     print "\n", "x_opt = ", results
29     print "f(x) = ", costfunc(results)
30     print "g(x) = ", equalcon(results)
31     print "h(x) = ", inequalcon(results)
32
33
34 if __name__ == "__main__":
35     main()
```

when we conduct another optimization run with our new cost function and we obtain

```
1 Optimization terminated successfully.      (Exit mode 0)
2     Current function value: 302.61300657
3     Iterations: 29
4     Function evaluations: 134
5     Gradient evaluations: 28
6
7 x_opt = [ 3.27150524  9.81451461]
8 f(x) = 302.61300657
9 g(x) = [-1.14312115e-10]
10 h(x) = [ 3.27150524  9.81451461]
```

a more realistic beverage can design. Just by considering the different materials a beverage can is made from significantly influences the optimal design, now the diameter our beverage can is two third of the height of our can.

Careful consideration should be taken when constructing your cost and constraint functions in optimization.

## 16.6 Exercises

- Determine the roots of the following polynomial

$$-15x^5 - 64x^4 + 173x^3 + 82x^2 - 116x + 24 \quad (16.22)$$

- Find the roots of the following polynomial

$$2x^4 - 7x^2 + x - 7 \quad (16.23)$$

- Divide

$$x^3 - 1 \quad (16.24)$$

by  $x + 2$ . What is the remainder?

- Which polynomial has the factors  $x - 3$ ,  $x + 4$  and  $x - 1$ ?
- Write a function `polyfromroots` where `polyfromroots` takes a list of numbers as the input argument. This list contains the roots of the polynomial, i.e.  $x^3 - x = x(x - 1)(x + 1)$  has the roots  $[-1, 1, 0]$  from  $x + 1 = 0$ ,  $x - 1 = 0$  and  $x = 0$  respectively. The function must return the polynomial coefficients created from the list of roots (without using `scipy`'s `polyfromroots` function). You may check your answer by using `scipy`'s `polyfromroots` function.

[Hint: use `scipy`'s `polymul` function]

- Solve the following differential equation using `scipy`'s `odeint` function,

$$f(t, y) = -3y(t) + 6t + 5$$

for  $y(0) = 3$  and  $t(0) = 0$ .

- What is the solution  $\mathbf{x}$  of the following system of linear equations?

$$\begin{aligned} x_1 - 2x_3 &= 1 \\ x_1 + x_2 &= 2 \\ x_1 - x_2 + x_3 &= 5 \end{aligned} \quad (16.25)$$

with  $\mathbf{x} = [x_1 \ x_2 \ x_3]^T$ .

8. Background: Buoyancy driven flow driven by natural convection along a heated vertical plate is described by the following two coupled non-linear differential equations

$$\begin{aligned} \frac{d^3 f}{d\eta^3} + 3f \frac{d^2 f}{d\eta^2} - 2\left(\frac{df}{d\eta}\right)^2 + T = 0 \\ \frac{d^2 T}{d\eta^2} + 3\text{Pr} f \frac{dT}{d\eta} = 0 \end{aligned} \quad (16.26)$$

where  $\text{Pr}$  is the Prandtl number,  $f$  is a stream function,  $\frac{df}{d\eta}$  is the velocity,  $\frac{d^2 f}{d\eta^2}$  relates to the shear stress in the fluid stream,  $T$  is the temperature and  $\frac{dT}{d\eta}$  is the heat flux.

We can decompose the above system into five first-order differential equations by using the following set of dependent variables

$$\begin{aligned} y_1(\eta) &= f \\ y_2(\eta) &= \frac{df}{d\eta} \\ y_3(\eta) &= \frac{d^2 f}{d\eta^2} \\ y_4(\eta) &= T \\ y_5(\eta) &= \frac{dT}{d\eta} \end{aligned} \quad (16.27)$$

The five first order differential equations given in terms of the above variables are

$$\begin{aligned} \frac{dy_1}{d\eta} &= y_2 \\ \frac{dy_2}{d\eta} &= y_3 \\ \frac{dy_3}{d\eta} &= 2y_2^2 - 3y_1y_3 - y_4 \\ \frac{dy_4}{d\eta} &= y_5 \\ \frac{dy_5}{d\eta} &= -3\text{Pr}y_1y_5 \end{aligned} \quad (16.28)$$

Solve the above system of first order differential equations for  $0\eta \leq 10$  in increments of 0.1 using  $\text{Pr} = 0.7$  and the following initial conditions

$$\begin{aligned} y_1(0) &= 0 \\ y_2(0) &= 0 \\ y_3(0) &= 0.68 \\ y_4(0) &= 1 \\ y_5(0) &= -0.5 \end{aligned} \quad (16.29)$$

Plot then  $y_1, y_2, y_3$  and  $y_4$  as function of  $\eta$  on the same figure but each as an individual plot using subplot.

9. Depicted in Figure 16.7 is a four-bar linkage with  $L_1, L_2, L_3$  and  $L_4$  the length of the links. Link 1 is fixed in a horizontal position. For a given  $\theta_1$ , we can solve for  $\theta_2$  and  $\theta_3$  using the following two non-linear equations

$$\begin{aligned} L_2 \cos(\theta_1) + L_3 \cos(\theta_2) - L_4 \cos(\theta_3) - L_1 &= 0 \\ L_2 \sin(\theta_1) + L_3 \sin(\theta_2) - L_4 \sin(\theta_3) &= 0 \end{aligned} \quad (16.30)$$

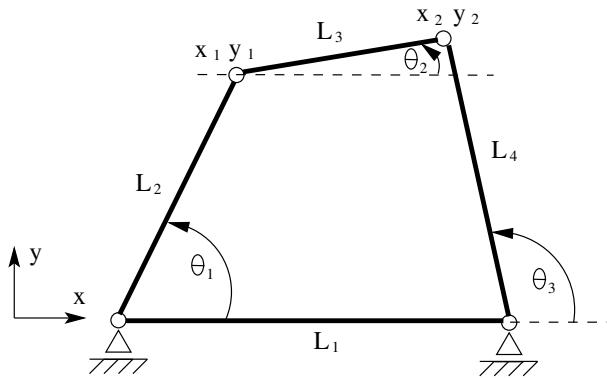


Figure 16.7: Four-bar linkage.

The link lengths are  $L_1 = 3$ ,  $L_2 = 0.8$ ,  $L_3 = 2$  and  $L_4 = 2$ . Solve for  $\theta_2$  and  $\theta_3$  for  $\theta_1$  between 0 and  $2\pi$  using 100 linearly spaced intervals. As a guess to the solution of the non-linear system use  $[5; 5]$  for  $\theta_1 = 0$ . For  $\theta_1 > 0$  the guess must be the solution the previously computed non-linear system.

Plot then link three for the various values of  $\theta_1$  and  $\theta_2$ . The coordinates of link three are given by

$$\begin{aligned} x_1 &= L_2 \cos(\theta_1) \\ y_1 &= L_2 \sin(\theta_1) \\ x_2 &= L_2 \cos(\theta_1) + L_3 \cos(\theta_2) \\ y_2 &= L_2 \sin(\theta_1) + L_3 \sin(\theta_2) \end{aligned} \tag{16.31}$$



# Chapter 17

## What's Next

So far, what has been covered in these notes is just the tip of the iceberg. As mentioned Python was created to be lightweight and easily extendible by other modules and packages, and it has been. Python has hundreds (if not thousands) of available packages, each tailored for a specific task or set of tasks. It is because of this that Python has become quite a popular programming language and it is used in all industries. To list a just a few applications:

1. Internet and Web Development  
(modules: Django, Flask, many more)
2. Database access and management  
(modules: SQLAlchemy,
3. GUI's and Graphics  
(modules: TKinter, wxWidgets, pyQT, many more)
4. Numerical and Scientific Computing  
(modules: Numpy, Scipy, Sympy, PIL, many more)
5. Education
6. Network Programming
7. Software Development
8. Gaming

Python is thus tool that you can continuously invest time in and learn to use it as an effective engineering tool. A list of available packages can be found at [http:](http://)

//[pypi.python.org/pypi/](http://pypi.python.org/pypi/). A list of applications of Python in industry can be found at <http://www.python.org/about/apps/> and lastly a list a success stories can be found at <http://www.python.org/about/success/>

## 17.1 Python(x,y) Packages

As mentioned Python(x,y) comes with many pre-installed Python packages. You can get additional information about these packages from <http://code.google.com/p/pythonxy/wiki/StandardPlugins>. Clicking on the package link will take you to that packages official website where you can get more information.

## 17.2 Installing New Packages

If you need to install new packages, that don't come with Python(x,y), you can follow these easy steps:

- Make sure your PC is connected to the internet
- Open the windows command prompt: push the “Windows” button, and while holding it in push the **r** button.
- Type **cmd** in the “run menu” and hit enter
- in the windows command prompt type **pip install <package name>** and hit enter.
- Wait for the installation to finish and you are done.