

Whirlwind Tour of Python

Mukesh Kumar & Sandeep S Ghugre

UGC DAE Consortium for Scientific Research, Kolkata Centre,
Sector III, LB-8
Bidhan Nagar, Kolkata 700 106

email : ssg@alpha.iuc.res.in

email : ssg.iuc@gmail.com

©UGC DAE CSR,KC 2020

July 5, 2020

Contents

Why Python	2
Python Installation	3
Python on <i>Windows</i>	3
Python on Android	5
Getting Started	7
Introduction	7
Python IDLE	7
Variables & Types	9
Variables	9
Assignment	9
Variable Types	9
Operators	13
Mathematical Operators	13
Conditional & Relational operators	14
Modules & Packages	15
Appendix	17
Floating point numbers	17
Errors	20
Loops & Functions	24
Introduction	24
How to repeat yourself	24
<i>while</i> Loop	25
<i>for / range</i> Loop	27

nested loops	29
Compound Conditions	29
User Function	29
Arrays	31
Introduction	31
List	31
Array	32
Data types	32
Special arrays	34
Creating arrays	35
Plots	38
X-Y plots	38
Random Numbers	42
Histograms	42
Read Data from a File	46
Ascii file	46
.wav file	48
Data Fitting	51
Linear / Quadratic Least Square Fit	53
Least Square Fit to User Function	56

List of Figures

0.1	Downloading the Python software.	3
0.2	Screen shots for installation process	4
0.3	Getting started.	5
0.4	Invoking the Python IDE.	7
0.5	Opening a file in IDLE	8
0.6	Saving the program file.. . . .	8
0.7	Category of the variables.	12
0.8	String Variables.	13
0.9	Mathematical Operations.	14
0.10	Flow chart for a repetitive set of calculations.	26
0.11	<i>while</i> loop.	27
0.12	First Plot.	38
0.13	Simple plot with title, and labels for the axes.	39
0.14	Simple plot with legends for multiple plots in the same figure.	40
0.15	Vertically stacked plots.	41
0.16	Histogram from given data set.	43
0.17	Random numbers generated.	45
0.18	Schematic representations of the canonical form of the <i>.wav</i> file.	48
0.19	Linear Fit to obtain ADC calibration.	55
0.20	Voltage across the capacitor during charging.	58

Disclaimer: These notes have not been subjected to the usual scrutiny expected for formal publications. They may be distributed outside the intended audience only with the permission of the author.

The material has been developed following the discussions with our colleagues, during the workshops that have been conducted in collaboration with

- Department of Physics, Asutosh College, Kolkata : November 7th – 8th , 2017
- Department of Physics & Computer Science, Bethune College, Kolkata, July 19th – 20th 2018
- R. P. Gogate College of Arts & Science, Ratnagiri, December 18th – 19th 2018
- S. H. Kelkar College, Devgad, December 22nd – 23rd 2018
- Department of Physics, Government Holkar Science College, Indore, September 17th – 18th 2019
- Department of Physics, R.D & S.H National College, Mumbai, February 6th – 7th 2020
- Department of Physics, Victoria Institution (College), Kolkata, July 6th – 10th 2020

The help and support received from Ms Kathakali Biswas, Victorial Institution (College), Kolkata and Dr Rajamani Raghunathan, UGC DAE CSR, Indore Centre in preparing this manuscript is gratefully acknowledged.

Why Python

UGC-DAE Consortium for Scientific Research, Kolkata Centre, is in the process of developing a range of innovative, low cost experiments based on the routinely available resources for the undergraduates. These experiments are expected to be illustrative and contribute in their understanding of the basics of the subject, apart from rejuvenating the fun factor in the learning process. And all this with an accompanying rigor on the extracted numbers.

When we started developing innovative experiments using open source resources, the first decision facing us was [which software do we use for the analysis of the data](#). The analysis usually comprises of the following operations

1. Read the data
2. Process the data
3. Visualise the data and / or the results

Therefore there is a need to resort to the use of conventional programming [Compiler](#) based languages such as *C++*, *C*, *Fortran*, to name a few. The major difficulties confronted by non-experts during the use of these languages, are the requirements to adhere strictly to the verbose syntax and definitions of various components, with minimal flexibility.

To circumvent this bottleneck, which at times severely constrains the developments in the domain of transparent and flexible experiments, we decided to use [Scripting languages](#), such as *Octave*, *Python*, in our present endeavour. The major advantages being

1. Free and open software, with a vibrant support from a large global community.
2. Flexible and extremely user friendly syntax and coding format.
3. Structured coding which is easy to read and follow.
4. Extensive collection of libraries with numerical algorithms for several domains such as [numerical methods](#), [quantum mechanics](#), [signal processing](#), [sound-recording](#), to name a few.
5. Besides these are available on all major operating systems.

Hence, we decided to seek recourse to [Python](#) for the presentation and detailed analysis of the results following the innovative experiments. Python besides being simple is powerful enough to be used for [serious numerical work](#), to compliment the conventional class room teaching in numerically intense courses such as [Mathematical Methods & Quantum mechanics](#).

Python Installation

The Python programming language combines remarkable power with very clean, simple, and compact syntax making it the preferable choice for educational numerical computation. Python is easy to learn and very well suited at an introductory level to computer programming especially in the domain of numerical and graphical data analysis, which is of contemporary relevance in present day classroom teaching.

Python on Windows

One of the challenges of getting started with Python is that you might have to install Python and related software on your computer. Besides there are two versions of Python, called *Python 2.x* and *Python 3.x*. The general consensus between novice and non-experts is to go for **version 2.x**. The current stable version being *v2.7*, and can be found at <https://www.python.org/downloads/release/python-2713/>, the screenshot is presented in Fig.0.1.

Files					
Version	Operating System	Description	MD5 Sum	File Size	GPG
Gzipped source tarball	Source release		17add4bf0ad0ec2f08e0cae6d205c700	17076672	SIG
XZ compressed source tarball	Source release		53b43534153bb2a0363f08bae8b9d990	12495628	SIG
Mac OS X 32-bit i386/PPC installer	Mac OS X	for Mac OS X 10.5 and later	4c60d95cb637423b53c59c3064cc2e69	24251431	SIG
Mac OS X 64-bit/32-bit installer	Mac OS X	for Mac OS X 10.6 and later	862d11e2e356966246451388ee9e4b99	22457385	SIG
Windows debug information files	Windows		dc0d9cc0266ec79e434c3d93a094de90	24703142	SIG
Windows debug information files for 64-bit binaries	Windows		7b1da6dc1947031cb362270b0644925e	25505958	SIG
Windows help file	Windows		95040f65a4a6db3d17c40fbd882f7eae	6224783	SIG
Windows x86-64 MSI installer	Windows	for AMD64/EM64T/x64	268fd335aad649df7474adb13b6cf394	20082688	SIG
Windows x86 MSI installer	Windows		0f057ab4490e63e528eaa4a70df711d9	19161088	SIG

Figure 0.1: Downloading the Python software.

Following the successful downloading we can proceed with the installation by double clicking the file we have just downloaded, following the steps pictorially represented in Fig.0.2

The Python interpreter is a program that reads and executes Python code. It can be run (activated) by clicking on an icon (which starts the IDE) , or by typing *python* on a command line. When it starts, you should see output similar to the one presented in Fig.0.3.

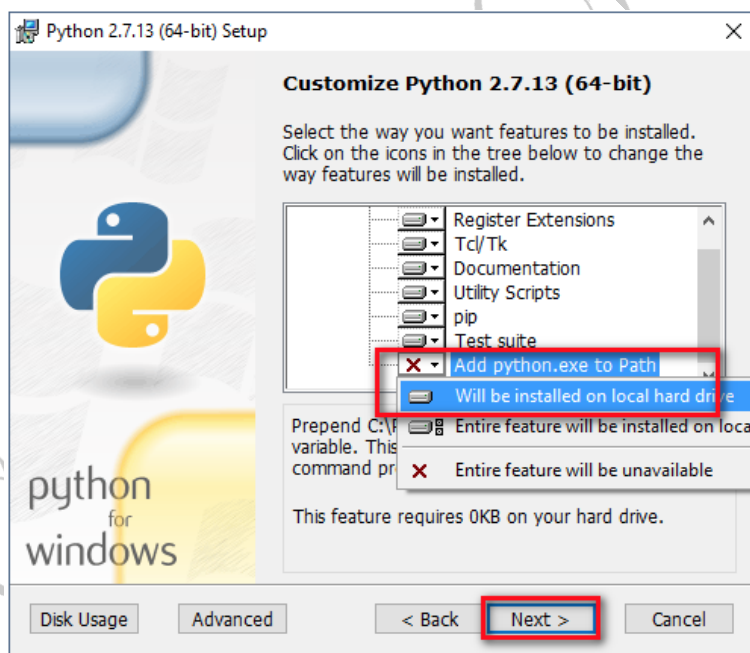
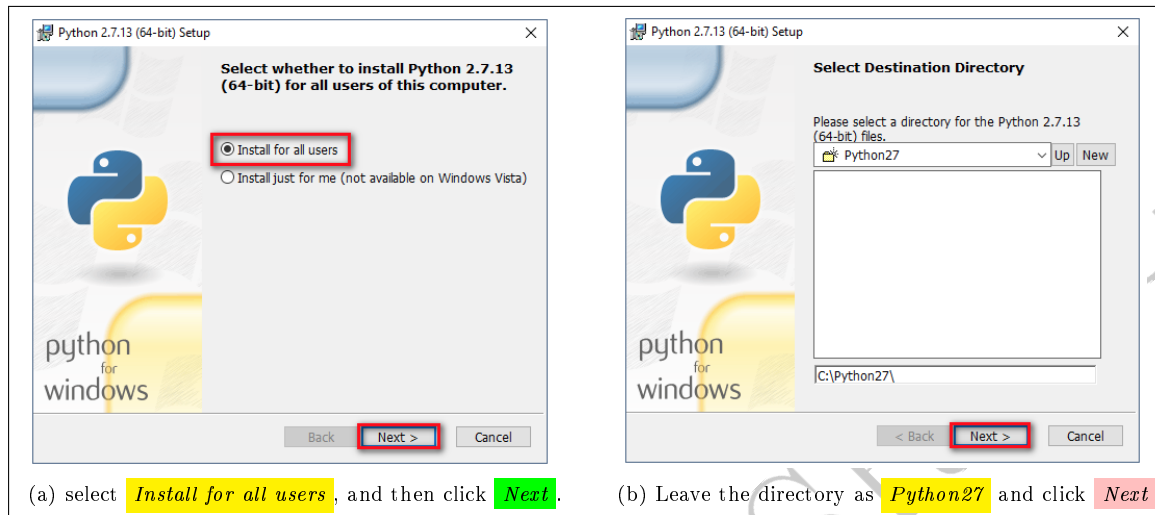
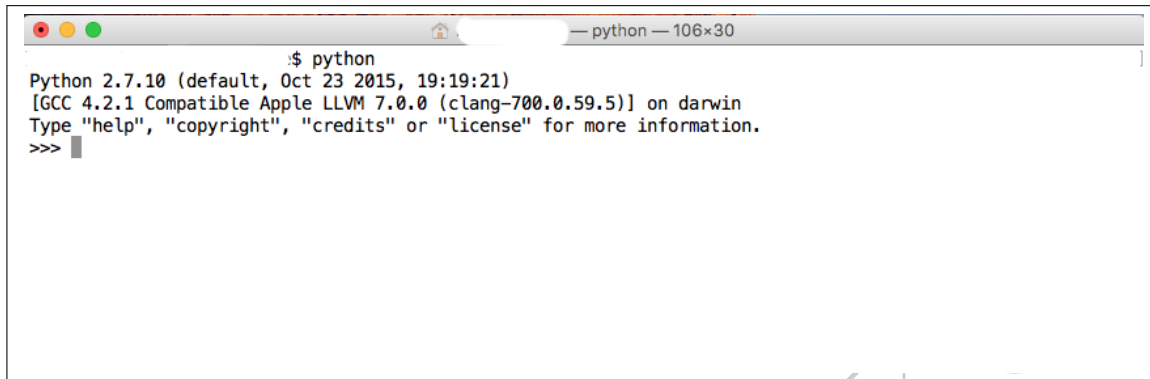


Figure 0.2: Screen shots for installation process



```
$ python
Python 2.7.10 (default, Oct 23 2015, 19:19:21)
[GCC 4.2.1 Compatible Apple LLVM 7.0.0 (clang-700.0.59.5)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 
```

Figure 0.3: Getting started.

The first three lines contain information about the interpreter and the operating system it's running on, so it might be different for you. But you should check that the version number, which is 2.7.10 in this example, which begins with 2, which indicates that you are running Python 2. If it begins with 3, you are running (you guessed it) Python 3.

The last line is a prompt that indicates that the interpreter is ready for you to enter code. If you type a line of code and hit Enter, the interpreter displays the result

We need to install **packages**, which essentially refer to a set of software, which have been specifically developed for a certain use. For example, we shall require

1. **NumPy**, fundamental package for scientific computing with Python. It can be installed using the command **`python -mpip install numpy`**, from the command terminal.
2. **Scipy** fundamental package for scientific and engineering computing with Python. It can be installed using the command **`python -mpip install scipy`**, from the command terminal.
3. **Matplotlib** the 2D plotting library which produces publication quality figures is also required frequently, It can be installed from the command prompt, using the command **`python -mpip install matplotlib`**.
4. In addition, if we wish to interface a micro-controller, we shall need the following packages **`serial`**, **`pyserial`**, which can be installed by the command described above, except replacing the **`numpy`** in the command in S.No 1, with the corresponding package name.

Python on Android

Now, we shall walk you through the installation of Python on an [Android Smartphone](#).

Besides the basic Python-3 software (Python 3.7.2) we shall also install the associated numpy, scipy (for numerical computations and mathematical tools) and matplotlib (for plotting) libraries. The steps could be summarized as

1. Download and install “pydroid 3” - IDE App in Android O.S. of the mobile.
2. After that open python software app in mobile and locate the quick install option in pip.
3. In quick install option, install
 - Matplotlib - python plotting package.
 - Numpy - Array processing for numbers, strings, records and objects.
 - Scipy - Scientific library for python.

UGC DAE CSR KC

Getting Started

Introduction

In this section we shall introduce the very basic operational commands to run a **Python** code. We hope by end of this chapter, users would be able to involve the Python console, type the commands in the console directly, or run a program **a program is a text file containing Python commands**. In this chapter we shall also introduce the basic program syntax.

Python IDLE

Every Python installation comes with a **Integrated Development and Learning Environment**, which help you **write & execute** your code. Besides the IDLE, we also have the **Python console**, which can serve in a limited manner the same end-purpose. The Python interpreter can be run (activated) by clicking on an icon (Fig. 0.4) (which starts the IDLE) ,

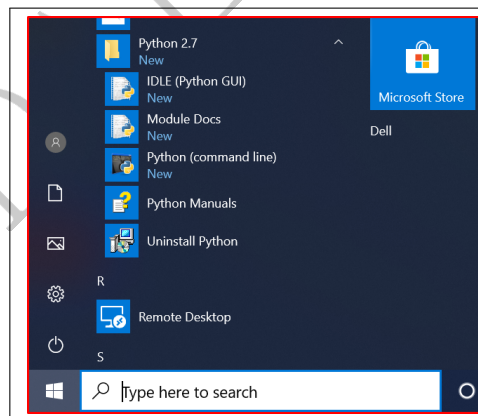


Figure 0.4: Invoking the Python IDE.

The **console** can be activated by typing the command **python** from either a terminal (Linux, or MacOS) or the command prompt cmd (Windows). The output on the terminal (Fig.0.3) indicates version of the Python that has been currently installed in the system.

Using the **Python console** to type in commands has severe limitations, *viz.* it doesn't save the work for the future ; it is a pain in the neck to correct a mistake or make any modifications. Instead, we shall write a **program**.

A program is a **text file** containing Python commands, having an extension **py**. The file has to be written with a plain text editor, Microsoft Word is not a text editor. We shall use the **File** option in the IDLE to create our program files.

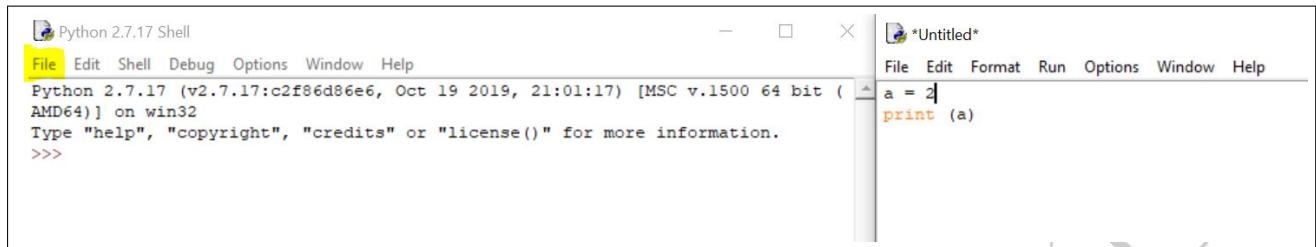


Figure 0.5: Opening a file in IDLE .

When we click the **File** option, a blank file opens up, wherein we can type our program (detailed subsequently) and we can save it using the **save As** option, and we select the extension as **.py** (Fig. 0.6) by using **Save as type Python files**.

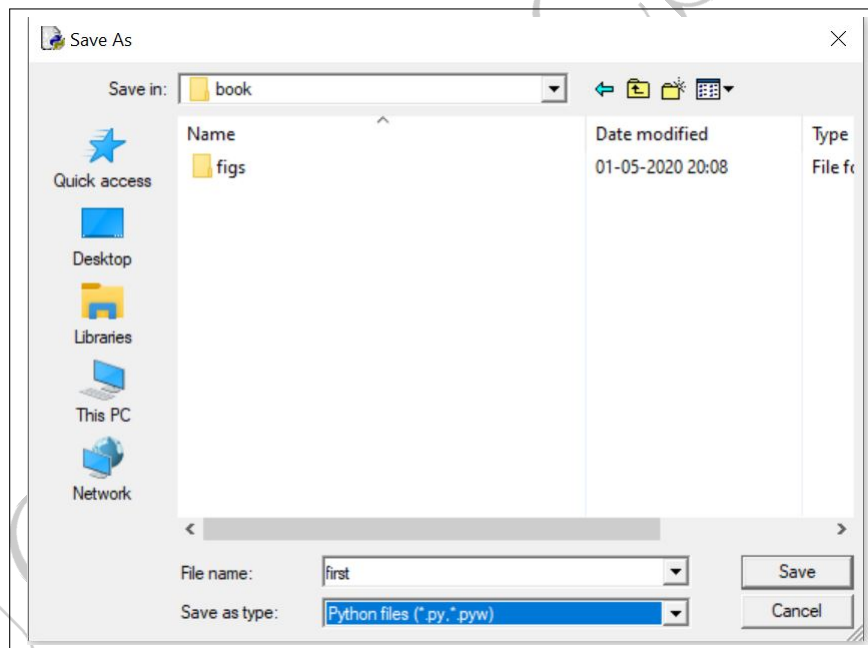


Figure 0.6: Saving the program file..

Every line in a Python program is a Python statement, except the line commencing with **#**, which is assumed to a comment line.

Once we have our code in the file, we execute it using the **Run** option, with the **Run Module**

Variables & Types

Variables

We generally choose names for variables. Variable names in Python can contain **alphanumeric characters** **a-z**, **A-Z**, **0-9** and **some special characters** such as `_`. Variable names must start with a letter. By convention, variable names start with a lower-case letter.

Some of the reserved words (variable names which have a predefined meaning) are

Reserved Words in Python				
and	assert	break	class	continue
def	del	elif	else	except
exec	finally	for	from	global
if	import	in	is	lambda
not	or	pass	print	raise
return	try	while		

In addition to these we should avoid the names of commonly used Python functions *viz.* `pi`, `sqrt`, `sin`, `cos`, ...

Assignment

An **assignment statement** creates a new variable and gives it a value. The assignment operator in Python is `=`. Assigning a value to a variable creates the variable.

```
>>> a = 2
>>> b = 5.0
>>> p = 3.14159
```

If you start a variable name with a numeric or a special character, it will give it a syntax error.

Python is a dynamically typed language, so we do not need to specify the type of a variable when we create one. This is one of the strengths or pitfall of Python.

Variable Types

We normally encounter numerical values. These are classified as

1. **Integer**, An integer number can be stored exactly in a computer provided sufficient number of bits are available. Hence data word 16 bit long can store positive numbers from 0 to 65535 (2^{16}). Integers don't divide quite like you'd expect, though! In Python, $1/2 = 0$, because 2 goes into 1 zero times (this operation does not yield an integer result).

```
>>>1/2
```

```
0
```

2. **Real**, they can not be represented exactly and always have a **decimal point** associated with them, these can be represented as :

- (a) **Float** Hence 3.0 is a floating point type number, and requires more memory for storage than it's integer counterpart. But the division of two real numbers results in a non-zero value.

```
>>>1.0/2.0
```

```
0.5
```

- (b) **Exponential** The number 6.634×10^{-34} , is a valid floating point number.

3. **Complex** A number $3 + 2j$, is a valid complex number, where $j = \sqrt{-1}$.

```
>>>a = 3 + 2j
```

```
>>>a.real
```

```
3.0
```

```
>>>a.imag
```

```
2.0
```

4. **Long** Integers greater than 2,147,483,647 are stored, automatically, as long integers. These are indicated by a trailing **L**. The maximum size of a long integer is limited by the available memory (RAM) of computer. Integers will automatically convert to long integers if required.
5. **String** A string is a sequence of characters. Strings are delimited by either single "or double quotes ""There are also some special characters in strings. To place a <tab> character, use **\t**. For a newline (linefeed), use **\n**.

Now since the *real numbers* are stored as an approximation it would typically result in **Rounding Off error**. During mathematical operation, the finite accuracy associated with the individual storage could lead to a large cumulative error.

For example 0.03125 is stored as 3.1×10^{-2} , resulting in a round-off error of

$$\frac{0.03125 - 0.031}{0.03125} = 0.008$$

Hence, if the above quantity is used for substantial number of computations, the individual round-off errors would build and contributed significantly to the error.

For example if we were to

```
>>> round (2.567,2)
2.57
```

This is because Python stores these as floating point numbers¹, and we can have a look at the exact value stored

```
>>> import decimal as dec
>>> dec.Decimal(2.565)
'2.564999999999999946709294817992486059665679931640625'
```

Now, if a mathematical operation were to include a very large number and another which is substantially smaller then, the mathematical operation of either the addition or subtraction, does not affect the magnitude of the larger number due to **limited accuracy of the floating point numbers**. This is elaborated later.

We can check, the **category** of the variable using the **type** command, as shown in the code below (Chap_1:data_type.py).

Category of the variable

```
1 v0_int =5
2 v0_float = 5.0
3 t = 0.6
4 g = 9.81
5 a = 2+3j
6 print ('type for v0_int is ', type(v0_int))
7 print (' ')
8 print ('type for v0_float is ', type(v0_float))
9 print (' ')
10 print ('type for t ', type(t))
11 print (' ')
12 print ('type for a ', type(a))
13 print (' ')
14 print ('Real part of a is ', a.real)
15 print (' ')
16 print ('Imaginary part of a is ', a.imag)
17 print (' ')
18 print ('Congugate of a is ', a.conjugate())
19 print (' ')
```

On executing the above code, the output is presented in Fig. 0.7.

¹the floating point representation, and the command **import** are subsequently discussed

The screenshot shows a Python IDE with two windows. The left window, titled 'data_type.py', contains the following code:

```

v0_int = 5
v0_float = 5.0
t = 0.6
g = 9.81
a = 2+3j
print ('type for v0_int is', type(v0_int))
print (' ')
print ('type for v0_float is', type(v0_float))
print (' ')
print ('type for t', type(t))
print (' ')
print ('type for a', type(a))
print (' ')
print ('Real part of a is ', a.real)
print (' ')
print ('Imaginary part of a is ', a.imag)
print (' ')
print ('Congugate of a is ', a.conjugate())
print (' ')

```

The right window, titled 'Python 2.7.17 Shell', shows the output of the code:

```

Python 2.7.17 (v2.7.17:c2f86d86e6, Oct 19 2019
D64)] on win32
Type "help", "copyright", "credits" or "licens
>>>
RESTART: C:\ssg\workshops\national_college\le
y
('type for v0_int is', <type 'int'>)
('type for v0_float is', <type 'float'>)
('type for t', <type 'float'>)
('type for a', <type 'complex'>)
('Real part of a is ', 2.0)
('Imaginary part of a is ', 3.0)
('Congugate of a is ', (2-3j))
>>> |

```

Figure 0.7: Category of the variables.

The various operations on **strings** are presented in the code below, ([Chap_1:string_operation.py](#)) and the results are illustrated in Fig. 0.8. In the output statement, we have used **formatted** output, similar in other programming languages (*viz.* C).

Operation on Strings

```

1 s = "Hello world"
2 print (type(s))
3 print (' ')
4 print (s)
5 print (' ')
6 print ('length of string " %s " is %d' %(s, len(s)))
7 print (' ')
8 s2 = s.replace("world", "sandeep")
9 print ('The new string is : %s ' %(s2))
10 print (' ')
11 s4 = "how are you"
12 print ('The concatenation of two strings without space is : %s ' %(s2+s4))
13 print (' ')
14 print ('The concatenation of two strings with space is : %s %s ' %(s2, s4))
15 print (' ')
16 print ('The partial string is : %s ' %(s[:3]))
17 print (' ')
18 print ('The entire string is : %s ' %(s[:len(s)]))

```



```

<type 'str'>

Hello world

length of string " Hello world " is 11

The new string is : Hello sandeep

The concatenation of two strings without space is : Hello sandeephow are you

The concatenation of two strings with space is : Hello sandeep how are you

The partial string is : Hel

The entire string is : Hello world
>>> |

```

Figure 0.8: String Variables.

Operators

Mathematical Operators

As one expects, the mathematical operators (for addition, subtraction, multiplication) can be operated on numbers and numerical variables.

Mathematical Operators	
Addition	+
Subtraction	-
Multiplication	*
Division	/
Modulus	%
Power	**

Division (/) is slightly different. It works perfectly on floats, but on integers result is always the integer portion of the quotient. Hence, we can use the option of type setting, to convert from one type of the variable to another.

```

>>>a = 5
>>>b = float(a)
>>>print(b)

```

5.0

The basic mathematical operators are enumerated in the code below ([Chap_1:numerical_operation.py](#)). The results are presented in Fig. 0.9. In this code we have again used the formatted output (detailed in the table) for the results.

Mathematical operators.

```

1 a = 2
2 b = 3
3 print ('the addition of a and b is ', a+b)
4 print (' ')
5 print ('the addition of a and b is ', float(a+b))
6 print (' ')
7 print ('the product of a and b is ', a*b)
8 print (' ')
9 print ('the division of a and b is ', a/b)
10 print (' ')
11 print ('the division of a and b is ', (float(a)/float(b)))
12 print (' ')
13 print ('the division of a and b is ', round((float(a)/float(b)),4))
14 print (' ')
15 print ('the division of a and b is %0.2f' %((float(a)/float(b))))
16 print (' ')
17 print ('a^ b is ', a**b)
18 print (' ')

```

Formatting indicators	
%xd	Integer) value, with (optional) total width <i>x</i> .
%x:yf	Floating Point value, x wide with y decimal places.
%xs	String of characters, with (optional) total width <i>x</i>

```

('the addition of a and b is ', 5)

('the addition of a and b is ', 5.0)

('the product of a and b is ', 6)

('the division of a and b is ', 0)

('the division of a and b is ', 0.6666666666666666)

('the division of a and b is ', 0.6667)

the division of a and b is 0.67

('a^ b is ', 8)

```

Figure 0.9: Mathematical Operations.

Conditional & Relational operators

A conditional operator evaluates the condition and returns either **true or false**.

Relational Operators		Logical Operators	
Greater than	>	Logical AND	and
Less than	<	Logical OR	or
Equal to	=	Logical NOT	not
Not equal to	!=		
Greater than or equal	>=		
Less than or equal to	<=		

Modules & Packages

One of the major advantage of toolkits such as Python or Octave, is the ready availability of a large number of user friendly [libraries](#), known as [modules](#) or [packages](#).

For example, Python does not have in-built trigonometric functions such as $\sin(x)$, $\cos(x)$, \dots or the value of π or e .

However, there are libraries of modules which have these in-built. For example, we have a module [math](#), which has these functions. So we need to [load](#) the relevant module in our code. This is achieved by the [import](#) command. The basic syntax for loading / activating the module, and to see the various functions available in the module, is presented below

```
>>>import math
>>>dir(math)
['acos', 'asinh', 'sin', 'cos', 'log', 'log10',.....]
```

Having imported the module, we now can use it as shown below, [but remember that all arguments to trigonometric functions are in radians](#).

```
>>>import math
>>>print(math.sin(30))
-0.988031624093
```

The [\(Chap_1:trigo_operation.py\)](#) expands the functionality of the above code, by incorporating an [user input](#), obtained by the [input](#) command.

Import of Modules.

```
1 import math
2 theta=input("Enter Value of the angle")
3 theta=float(theta)
4 radian = (math.pi / 180.0) * theta
5 print (math.sin(radian))
```

You can import everything from a package, each with it's individual name

```
>>>from math import *
>>>print(sin((pi/180.0) * 30))
0.5
```

It is often useful to just import individual elements of a package rather than the entire package, as

```
>>>from math import sin
>>>from math import pi
>>>print(sin((pi/180.0) * 30))
0.5
```

We can also `alias` a module when we import it, as

```
>>>import math as mt
>>>print(mt.sin((mt.pi/180.0) * 30))
0.5
```

The modules required frequently, are for numerical and scientific computations are `numpy`, `scipy` and `pyplot` in `matplotlib` for plotting.

```
>>>import numpy as np
>>>import matplotlib.pyplot as plt
>>>print(np.sin((np.pi/180.0) * 30))
```

Appendix

Floating point numbers

When we deal with very large and / or very small numbers we often resort to using *scientific notation*. For example, Calculators, often represent the results of large calculations using this notation since the display of a calculator has limited digits to present the result to the user. For example

15900000000000000 : could be represented as

$$159 * 10^{14}$$

$$15.9 * 10^{15}$$

$$1.59 * 10^{16}$$

this is referred to as *normalized representation*.

Calculator would display as : **159E14**

159 : is called as **mantissa**

14 : the **exponent**

: the number of places by which we need to move the decimal point

A *real number* is referred to as *floating point number*. A fixed number of bits are designated for the storage of these floating point numbers in the memory. The use of a fixed number of bits to store floating point numbers produces a phenomenon called *finite precision*. This originates due to the fact that real numbers cannot be stored precisely, as we have to **round** off the so as to accommodate it within the limited number of bits available and hence, store an approximation for the number. The loss of precision due to round-off is called *round-off error*.

On a computer we store floating point numbers in binary. This means we have numbers of the form

$$x = r \cdot 2^k$$

where

r : is the **signed mantissa**

: a normalized rational number between 1 and 2

: $1 \leq r < 2$

k : is the **signed exponent**

The bits allocated for a floating point number are

1. Sign bit, represents the sign of the mantissa, 0 signifies a *positive* value, where as 1 is indicative of a negative value.

2. Certain number of bits for an exponent. The sign of the exponent is incorporated using the *excess $2^k - 1$* representation, where k , is the number of bits used for the exponent. Hence, if we were to use a 8-bit representation for the exponent, then we add $2^{8-1} = 127$ to the true exponent and store the result. Using this representation, we would be able to store exponents between -126 to $+127$. To store -126 , we have $-126 + 127 = 1$, and store 00000001 in the exponent, and for 127, we have $127 + 127 = 254$, and hence store 11111110 as the exponent. The notation 00000000 is reserved for 0, and 11111111 for **overflow**.
3. Remaining certain number of bits for the mantisa. Since we use the *normalized* representation the 1 that appears before the point is never stored, because this digit **will always be a one**.

Hence, for a 32 bit real number we have 1 sign bit, 8 bits for the exponential and 23 bits for the mantissa, and a range of $10^{\pm 38}$, while for 64 bit real number we have 1 sign bit, 11 bit exponential and 52 bit mantisa results in a range of $10^{\pm 38}$. The representation for a 32 bit floating point representation is presented in Table 0.1

Table 0.1: Representation of a 32 bit floating point number.

Sign (1 bit)	Exponent (8 bits)								Fraction (23 bits)																						
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
31	30																													23	
Bit Position	Bit Position																														

Fraction (23 bits)																															
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
22																														0	
Bit Position																															

Now to convert a decimal number to a floating point number

1. Generate the binary equivalent of the integer part of the number.
2. Generate the binary equivalent of the fractional part.
3. Then place the two parts together and then *normalize*.

Now to convert 6.75 into a 32 bit floating representation

$6 \Rightarrow 110_2$
 $0.75 \Rightarrow 0.11_2$
 $6.75 \Rightarrow 110.11$
 Normalize :
 $1.1011 * 2^2$
 Sign :
 Positive Number $\Rightarrow S = 0$
 Biased 127 exponent $\Rightarrow 2 + 127 = 129$
 $\Rightarrow 10000001$
 Mantisaa :
 1.1011
 We always have a 1 before the decimal in the mantisa.
 Hence we can drop this 1 and pad the remaining bits.
 $\Rightarrow 1011000000000000000000$

Hence, 6.75 in 32 bit floating point representation is

0	10000001	1011000000000000000000
Sign	Exponent (8)	Mantissa (23)

To convert -0.1173 into a 32 bit floating representation, we have

$0 \Rightarrow 0_2$
 $0.1173 \Rightarrow 0.0001111_2$
 $0.1173 \Rightarrow 0.0001111^*$
 Normalize :
 $1.1111 * 2^{-4}$
 Sign :
 Negative Number $\Rightarrow S = 1$
 Biased 127 exponent $\Rightarrow -4 + 127 = 123$
 $\Rightarrow 01111011$
 Mantisaa :
 1.111
 We always have a 1 before the decimal in the mantisa.
 Hence we can drop this 1 and pad the remaining bits.
 $\Rightarrow 1110000000000000000000$

* We shall have 0.0001111000000111011.....upto total 23 numbers after decimal in order to fill mantisa correctly.

Errors

Now since the *real numbers* are stored as an approximation it would typically result in **Rounding Off error**. During mathematical operation, the finite accuracy associated with the individual storage could lead to a large cumulative error.

For example 0.03125 is stored as 3.1×10^{-2} , resulting in a round-off error of

$$\frac{0.03125 - 0.031}{0.03125} = 0.008$$

Hence, if the above quantity is used for substantial number of computations, the individual round-off errors would build and contributed significantly to the error.

For example if we were to

```
>>> round (2.565,2)  
2.56
```

This is because Python stores these as a floating point number², and we can have a look at the exact value stored

```
>>> import decimal as dec  
>>> dec.Decimal(2.565)  
'2.564999999999999946709294817992486059665679931640625'
```

Now, if a mathematical operation were to include a very large number and another which is substantially smaller then, the mathematical operation of either the addition or subtraction, does not affect the magnitude of the larger number due to **limited accuracy of the floating point numbers**.

²the floating point representation has discussed briefly above

```

1 import numpy as np
2 x= 10000000000000000
3 y = 1
4 sum = x+y
5 print 'the integer sum is {:d}' .format(sum)
6 xe = 1e16
7 ye = 1.0
8 sume = xe+ye
9 print 'the exponential sum is {:2.1E}' .format(sume)
10 xf = 10000000000000000.0
11 yf = 1.0
12 sumf = xf+yf
13 print 'the float sum is {:12.1f}' .format(sumf)

```

We do encounter **Smearing error** in addition or subtraction, wherein the individual terms are greater than the summation itself. For example, if we were to evaluate , $(x + 10^{-20}) - x$ for, $x = 1$ then, we expect the result to be 10^{-20} , but

```

>>> x = 1
>>> (x + 10E - 20) - x
0.0

```

Another category of errors encountered while performing numerical computations is **Truncation error**. Since most mathematical functions are generated either by series expansion or by iteration *eg.*

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \dots$$

$$e^x = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \dots$$

Now for $x = 1$, we have

$$e^x = 1 + x + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \dots$$

$$= 2.718281828459045$$

We obtain the value for e^x for $x = 1$ as follows

```
>>> import math
>>> math.exp(1)-(1+1+(1.0/math.factorial(2))+(1.0/math.factorial(3))+(1.0/math.factorial(4)))
0.009948495125712054
>>> math.exp(1)-(1+1+(1.0/math.factorial(2))+(1.0/math.factorial(3))+(1.0/math.factorial(4))
+ (1.0/math.factorial(5)))
0.0016151617923787498
>>> math.exp(1)-(1+1+(1.0/math.factorial(2))+(1.0/math.factorial(3))+(1.0/math.factorial(4))
+ (1.0/math.factorial(5)) +(1.0/math.factorial(6)))
0.0002262729034896438
```

Hence, after $n = 6$, we have the calculated value within 0.001.

Now, if we were to evaluate the value for e^x for $x = -2.345$, such that the error between the computed and expected value is minimized for a given set of terms used in the expansion.

We know that for $x = -2.345$ e^x is ~ 0.095847 , and can be computed as

```
>>> import math
>>> x = -0.2345E01
>>> 1+x+((x**2)/math.factorial(2))+((x**3)/math.factorial(3))+((x**4)/math.factorial(4))
+ ((x**5)/math.factorial(5)) +((x**6)/math.factorial(6)))
0.15530772962200246
```

The large error is due to the cancellation of intermediate terms with opposite sign. Hence, if we use the following

$$e^{-x} = \frac{1}{e^x}$$

```
>>> import math
>>> x = 0.2345E01
>>> 1.0/(1+x+((x**2)/math.factorial(2))+((x**3)/math.factorial(3))+((x**4)/math.factorial(4))
+ ((x**5)/math.factorial(5)) +((x**6)/math.factorial(6))))
0.09684701025652355
```

Hence, following this rearrangement and increasing the number of terms, we shall approach a value closer to the expected value.

We know that

$$\sinh(x) = \frac{1}{2}(e^x - e^{-x})$$

Evaluate $\sinh(x)$ using three different operations *viz.* using $\sinh()$ from the *math* module ; computing the right-hand side of the above equation, using $\exp()$ from the same module & using power expression e^x in the above expression.

```
1 from math import sinh, exp, e, pi
2 x = 2*pi
3 r1 = sinh(x)
4 r2 = 0.5*(exp(x) - exp(-x))
5 r3 = 0.5*(e**x - e**(-x))
6 print '%.16f %.16f %.16f' %(r1, r2, r3)
7 print( '{:17.10f}, {:17.10f}, {:17.10f}' .format(r1, r2, r3))
```

Loops & Functions

Introduction

As a part of a program, we may wish to perform a series of repetitive operations under some specified conditions. Upon successful completion of the conditions we may wish to stop the execution or perform some other tasks. In this chapter we introduce the user to **Control statements** in Python, which allow a program to do different sequences depending on what a certain criteria. We shall also touch upon the **A function** in Python, which is a bit of code that is given its own name so that it may be used repeatedly by various parts of a program

How to repeat yourself

The conversion from Centigrade to Fahrenheit temperature scales is given by

$$F = \frac{9}{5} C + 32$$

Hence, the Python code to achieve the conversion for $C = 21$ which should yield $F = 69.8$ is

Temperature Conversion

```
1 import math
2 C = 21
3 C = float(C)
4 F = (9.0/5.0)*C+32
5 print F
```

Now we may wish to prompt the user for the temperature in Centigrades and then convert it. This is achieved using (**Chap_2:C_to_F.py**)

User input for temperature conversion

```
1 import math
2 C = input("Please enter the temperature in Celcius ..... \n")
3 C = float(C)
4 F = (9.0/5.0)*C+32
5 print F
```

We then would wish to extend the above calculations for a series of temperature values, say for example starting with $C = 20$ upto $C = 80$ in interval of 5 degrees, then we shall have to repeat a set of statement, till a particular condition ($C \leq 80$ in this case) is satisfied. Hence, the methodology to be pursued would be :

1. Initialize $C = 20$.
2. **While** $C \leq 80$, repeat the following steps
 - (a) Calculate $F = (\frac{9}{5} \cdot C) + 32$
 - (b) Print the values
 - (c) Set $C = C + 5$
3. Stop

The statements (lines) following the **While** line are to be repeated as long the condition $C \leq 80$ is **TRUE**. Hence, we wish to repeat a **block** of statements, while a particular condition is **TRUE**. Therefore we have to

1. Setup a logical condition and perform the subsequent steps as long as this condition is valid.
 - (a) Step 1.
 - (b) Step 2.
 - (c)
2. Stop the calculations on a successful validation of the logical condition.

while Loop

We shall now exploit, the *while* loop feature of Python. This feature allows us to process (execute) a certain pre-defined set of steps (computations), the syntax of the *while* loop is

```
while expression :
    statements
else :
    statements
```

When Python encounters a *while* loop, it firsts tests the expression provided; if it is not true, and an else clause is present, the statements following the else are executed. With no else clause, if the expression is not true, then control transfers to the first statement after the while loop. This is represented in Fig. 0.10

The Python code to achieve the above is

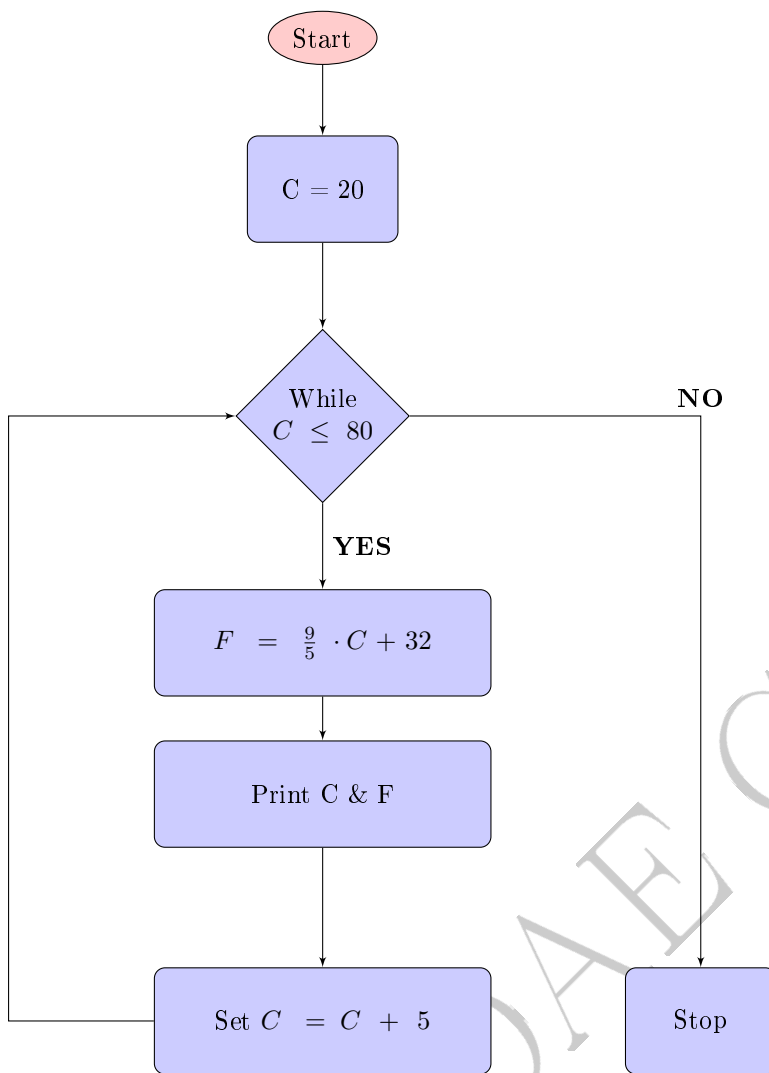


Figure 0.10: Flow chart for a repetitive set of calculations.

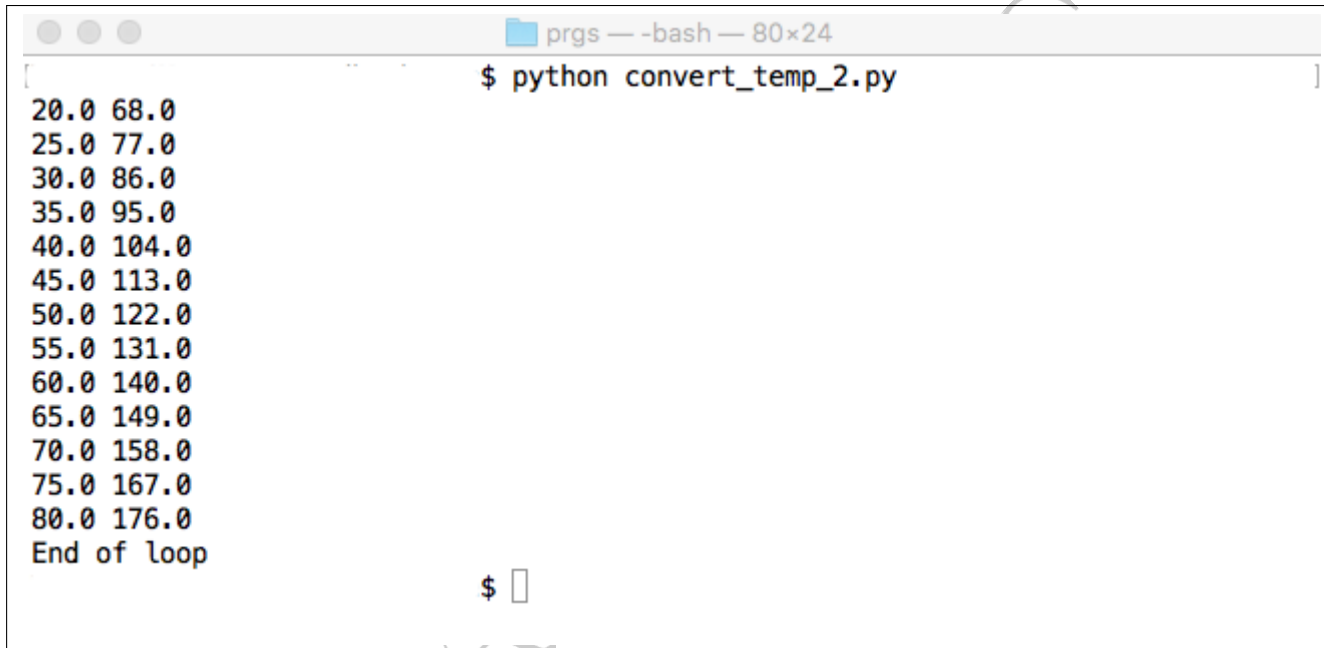
while loop

<pre> 1 import math 2 C = 20.0 3 dC = 5.0 4 while C <= 80 : 5 F = (9.0/5.0)*C + 32 6 print C, F 7 C = C + dC 8 print 'End of loop' </pre>	<pre> # Initialize the value of C # Increment value of C by this amount # Define the loop condition # Compute the temp. in Farenheit # Print out the values # Increment the value of C # End the computation </pre>
--	---

The second line, initializes the value of $C = 20$, the starting point for our calculations, and assigns a value of 5.0 to a variable `dC`, which is utilized to increment the current value of the variable `C`.

The next line (4) defines the loop. The keyword **while** indicates that the subsequent and **indented** statements constitute the **block**. We then setup the logical condition that will be either True or False, which governs the execution of the loop. If the condition is **True**, the statements in the loop will be executed. The colon **:** at the end of the line ends the logical condition and signals that what follows will be the statements inside the loop.

The code block (statements inside the *loop*) to be executed in each pass of the while loop must be grouped or indented. In the example above the block consists of three lines, and all these lines must have exactly the same indentation. The choice of indentation is four spaces.



```
prgs - -bash - 80x24
$ python convert_temp_2.py
20.0 68.0
25.0 77.0
30.0 86.0
35.0 95.0
40.0 104.0
45.0 113.0
50.0 122.0
55.0 131.0
60.0 140.0
65.0 149.0
70.0 158.0
75.0 167.0
80.0 176.0
End of loop
$
```

Figure 0.11: *while* loop.

for / range Loop

The for loop iterates over items in a sequence, repeating the loop block once per item. The most basic syntax is as follows:

```
for Item in Sequence :
    statements
    statements
```

Each time through the loop, the value of **Item** will be the value of the next element in the **Sequence**. There is nothing special about the names **Item** and **Sequence**, they can be any associated variable names. The

code (Chap_2:C_to_F.py) below demonstrates the for loop, for the temperature conversion presented in the earlier section.

for loop.

```
1 import math
2 for theta in range (20,81,5):           # range (start , stop[, step])
3     F = (9.0/5.0)*float(theta)+32
4     print theta , F
5 print 'End'
```

The `range` function generates the integer numbers between the given `start` integer to the `stop` integer, but does not include the stop integer. The default value of `step` is 1. Since the `range()` function returns a `list`, the for command can be run with the `range()` function.

The examples below, demonstrate the use of `range()` function to create (i) a list of 100 numbers (ii) create a list of even numbers from 6 to 17, however this is valid for version 2.7 only.

```
>>> axis = range(100)
>>> print(axis)
[0,1,...,99]
>>> evens = range(6,17,2)
>>> print(evens)
[6,8,10,12,14,16]
```

For version for version 3.6 only, the syntax is

```
>>> axis = range(10)
>>> for i in axis :
>>>     print(axis[i])
0,1,...,9]
>>> evens = range(6,17,2)
>>> for i in range(len(evens)) :
>>>     print(evens[i])
6,8,10,12,14,16
```


nested loops

We would like to convert $\theta = 0$ to 100 degree to Fahrenheit, but only for those temperatures which are exactly divisible by 10.

nested loops

```

1 import math
2 for theta in range (0,100):
3     if theta % 10 == 0:          # check if remainder is 0
4         F = (9.0/5.0)*float(theta)+32
5         print theta , F
6 print 'End'
```

In this example we have used the `%` operator, which provides us the remainder when *theta* is divided by 10. If the remainder is zero then the temperature is divisible by 10, and we convert it to Fahrenheit.

Please note that if the program ends with a loop ensure that you have either

1. A print statement like *print 'End of loop'*
2. Empty blank line

else the program would never terminate.

Compound Conditions

We would like to convert $\theta = 0$ to 100 degree to Fahrenheit, but only for those temperatures which are exactly divisible by 10 and less than 50. (Chap_2:C_to_F_for_range.py)

Compound Condition

```

1 import math
2 for theta in range (0,100):
3     if ((theta % 10 == 0) and (theta < 50)):
4         F = (9.0/5.0)*float(theta)+32
5         print theta , F
6 print 'End'
```

User Function

A User function (referred to as function subsequently) is a collection of statements that can be executed at any location in the main program. We can send variables to the function which may be used by the function to perform calculations and the function can return back to the main program the computed results.

Thus the anatomy of the User function would be

```
def name(arg1,arg2.....,argn) :
    implementation of the function
    return result
```

The first word after the **def**, is the name of the function, and it is used subsequently to call the function.

After the function name, list of input parameters (information passed as input to the function) is within parentheses. If there are no input parameters the brackets still need to be there. If there is more than one parameter, they should be separated by commas.

After the bracket there is a **colon** `:`. The use of colons to denote special blocks of code identified by their indentation.

The **def** line with the function **name** and **arguments** is often referred to as the **def** function header, while the indented statements constitute the **function body**.

The code (Chap_2:func_example.py) below demonstrates the use of **user function** in Python.

Example of User Function : I

```
1 import numpy as np
2
3 def sq(x):
4     x = x**2
5     return x
6
7 x=input("Input a number")
8 print("Square of x %4.2f is %4.2f" %(x, sq(x)))
```

Example of User Function : II

```
1 #{\textcolor{magenta}{(Chap\_2:C\_to\_F\_func.py)}}
2 import math
3 def deg_to_faren(theta):          # define function , argument is temp in degree
4     F = (9.0/5.0)*float(theta)+32
5     return F                     # return the result
6
7 for theta in range(0,100):
8     if (theta % 10 == 0) and (theta < 50):
9         print theta, deg_to_faren(theta)          # call the function
10 print 'End'
```

Arrays

Introduction

Python provides `list` and `arrays` to store a collection of objects. The major difference between the two being that

1. A list is a data structure that is built into Python and holds a collection of items.
2. To use an array in Python, we need to import this data structure from the `numpy` package or the `array` module.

The common feature being that both these are sequentially structured, and hold items in an organized manner. From computational point of view `arrays` are more efficient to handle, and hence are used frequently. In this chapter we shall focus on the various operational aspects on an `array`.

List

A list is an ordered collection of objects. Lists are defined by square brackets, `[]`. Objects in the list are separated by commas.

We can access a member of a list by giving its `name`, `square brackets`, and the `index` of the member (as in *C* programming, the index starts from 0): A list can be empty [list2](#)

```

>>> list1 = [0,1,2,3,4]
>>> list2 = []
>>> (print(type(list1))
<type, 'list '>
>>> print(len(list1))
[5]
>>> print(len(list2))
[0]
>>> print(list1[1])
[1]

```

Now, if we have an empty list, then we need to append the objects to the list, via the `list_name.append` command, as shown below, where we have obtained three readings for the variable `y` for a particular setting of the variable `x`, and we wish to tabulate the values of x and y_{avg} (Chap_3:empty_list.py)

Empty list

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 x=[0,1,2,3,4,5,6]
4 y1=[4.4,1.81,0.51,0.19,0.07,0.03,0.01]
5 y2=[4.4,1.68,0.52,0.22,0.10,0.04,0.02]
6 y3=[4.4,1.72,0.50,0.22,0.11,0.03,0.02]
7 yav=[] # empty list
8 for i in range(len(x)):
9     temp = (y1[i]+y2[i]+y3[i])/3.0
10    yav.append(temp) #append the value into the yav array
11    print x[i], yav[i]

```

Array

An **array** is a data structure which can hold more than one value at a time. It is a collection or ordered series of elements of the **same type**.

Data types

Besides being advantageous from storage viewpoint, **array** provides greater flexibility in numerical operations. However, we need to declare **array** before we can use them, similar to other programming languages. Creating an array, requires a specific function from either the array module (`array.array()`) or NumPy package (`numpy.array()`)

```
>>> import numpy as np
>>> m=np.array([1,2,3,4,5])
>>> print(type(m))
<type 'numpy.ndarray'>
>>> print(len(m))
[5]
```

A 2-D array, is termed as **matrix**

```
>>> import numpy as np
>>> mat=np.array([[1,1],[1,2]])
>>> print(type(m))
<type 'numpy.ndarray'>
>>> mat
array([[1,1],
       [1,1]])
```

Common data types are *int*, *float*, *complex*, *bool*. However, the **default data type of an array is float**. We can check the data type via the **m.dtype** command, where *m* is the array name.

```
>>> import numpy as np
>>> mat=np.array([1,2,3,4], dtype=float)
>>> print(type(m))
<type 'numpy.ndarray'>
>>> mat
array([1.0,2.0,3.0,4.0])
>>> mat.dtype
dtype('float64')
```

Very often we read a **2 Dimensional array**, and we wish to break it down into the equivalent **1 Dimensional arrays**. This is demonstrated below

```
>>> import numpy as np
>>> mat=np.array([[1,1],[1,2]])
>>> mat
array([[1,1],
       [1,2]])
>>>
>>> mat[:,0]
array([1,1])
>>>
>>> mat[:,1]
array([1,2])
```

Special arrays

We can create arrays containing either `Zeros(0)` or `Ones(1)` , using a command as

```
>>> import numpy as np
>>> mat=np.zeros(5)
>>> print((mat))
<type 'numpy.ndarray'>
>>> mat
array([0.0,0.0,0.0,0.0,0.0])
>>> mat.dtype
dtype('float64 ')
>>> mat1=np.ones(5)
array([1.0,1.0,1.0,1.0,1.0])
```

We can also create a `identity` and a `diagonal` matrix as

```
>>> import numpy as np
>>> mat=np.eye(2)
>>> mat
array([[1.0,0.0],
       [0.0,1.0]])
```

```
>>> import numpy as np
>>> mat=np.diag(np.full(2,1))
>>> mat
array([[1,0],
       [0,1]])
```

where `np.full` creates a constant array, with the argument supplied.

Creating arrays

Using `np.arange(start,stop, step, dtype)`

The `np.arange()` command returns evenly spaced values, within the interval, and excluding the **stop** interval. The default value of **step** is 1.

```
>>> import numpy as np
>>> mat=np.arange(0,10,1,dtype=int)
>>> mat
array([0,1,2,3,4,5,6,7,8,9])
>>>
>>> mat=np.arange(0,10,2,dtype=int)
>>> mat
array([0,2,4,6,8])
>>>
>>> mat=np.arange(10,dtype=float)
>>> mat
array([0.,1.,2.,3.,4.,5.,6.,7.,8.,9.]
```

Using **np.linspace(start,stop, num, dtype)**

The **np.linspace()** generates **num** evenly spaced samples within the interval.

```
>>> import numpy as np
>>> mat=np.linspace(1,10,1,dtype=int)
>>> mat
array([1,2,3,4,5,6,7,8,9,10])
>>>
>>> mat=np.linspace(1,10,20,dtype=int)
>>> mat
array([1,1,1,2,2,2,3,3,3,4,4,4,5,5,5,6,6,7,7,8,8,9,9,10])
>>>
>>> mat=np.linspace(1,10,20,dtype=float)
>>> mat
array([1.,1.47,1.94,2.42,2.89, ..... ,10.]
```


Create arrays

```

1  #{\textcolor{magenta}}{(Chap\_3: arrays\_linspace.py)}}
2  import numpy as np
3
4  theta = np.arange(0,2*np.pi,np.pi/6.0)
5  num1=len(theta)
6
7  for i in range(len(theta)):
8      print "theta %.2f,\t cos(theta)%.4f"%((theta[i]*(180.0/np.pi)),
9          np.cos(theta[i]))

```

Plots

There are many Python plotting libraries depending on your purpose. However, the standard general-purpose library is `matplotlib`, which is often through its `pyplot` interface. Hence, prior to its use we need to *import* it first.

X-Y plots

The code below (`Chap_4:plot_1.py`) demonstrates a simple first plot.

Basic Plot

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 X =[1, 2.2, 3.3, 4.7]
4 Y = [4.75, 4.88, 4.99, 4.99]
5 plt.plot(X,Y)
6 plt.show()
```

The result of the above code is presented in Fig. 0.12

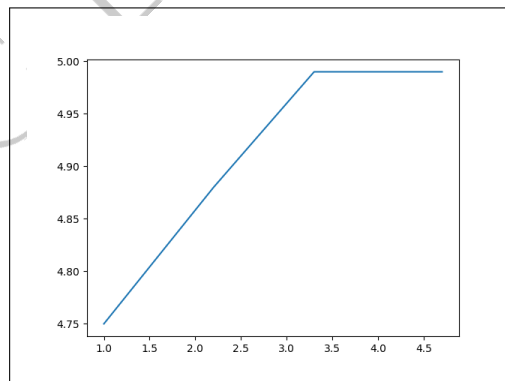


Figure 0.12: First Plot.

The most important command is the `plt.show()`, in absence of this command, the plot would **not** be displayed.

Now let us update the code to achieve a scatter plot & change the color of the line . This is achieved using the code (Chap_4:plot_2.py)

Simple Plot

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 X =[1, 2.2, 3.3, 4.7]
4 Y = [4.75, 4.88, 4.99, 4.99]
5 plt.plot(X,Y,"ro")
6 plt.plot(X,Y,"b—")
7 plt.xlabel('Load Resistance')
8 plt.ylabel('Output Voltage')
9 plt.xlim(0,5.5)
10 plt.ylim(0,5.5)
11 plt.grid()
12 plt.title('Voltage Regulation using 7805')
13 plt.show()

```

The result of the above code is presented in Fig. 0.13

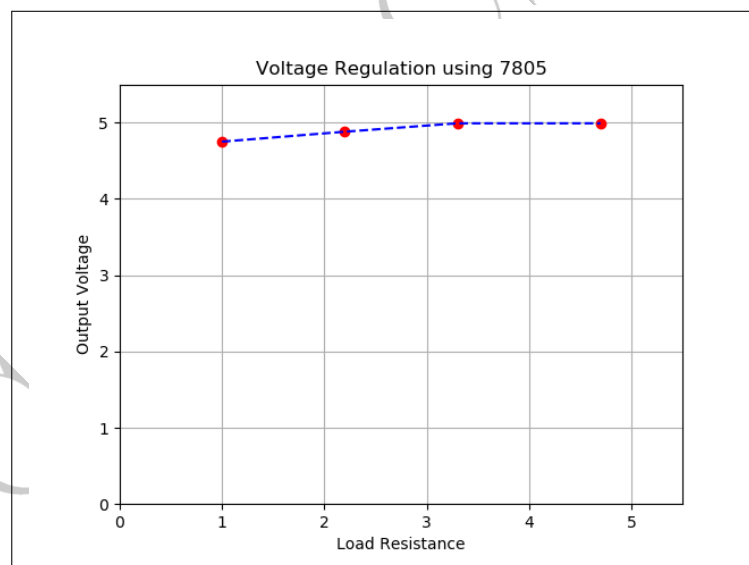


Figure 0.13: Simple plot with title, and labels for the axes.

Lines 5 allows us to obtain a scatter plot, with red coloured circles for the data points, whereas line 6, plots a blue line through the data points. Line 7 & 8 establish the axes labels whereas the limits of the axes range is set in lines 9 & 10.

The code below (Chap_4:plot_3.py) demonstrates the ability to plot two graphs on the same plot, and the results are presented in Fig. 0.14, with the appropriate legends.

Multiple plots and legends

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 X = np.linspace(0, 2*np.pi, 360)
4 print len(X)
5 Y = np.sin(X)
6 Z = np.cos(X)
7
8 plt.plot(X, Y, "b*", markersize=1)
9 plt.plot(X, Z, "go", markersize=2)
10 plt.grid()
11 plt.gca().legend(('sin(x)', 'cos(x)'), loc='best')
12 plt.show()

```

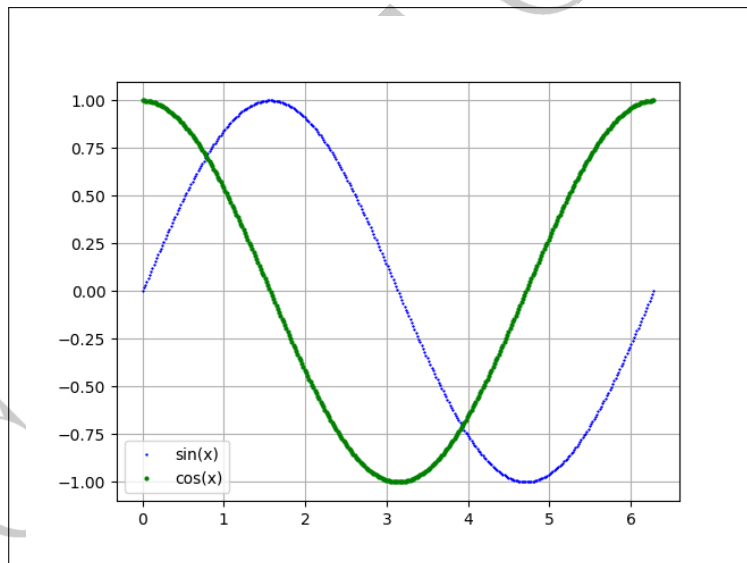


Figure 0.14: Simple plot with legends for multiple plots in the same figure.

The code below ([Chap_4:plot_4.py](#)) demonstrates the possibility to **vertically stack** two figures with individual parameters such as **axis name, title**.

Vertically stacked plots

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 x = np.arange(0,2*np.pi, 0.01)
4 y = np.sin(x)
5 z = np.cos(x)
6
7 print len(x)
8 fig, axs = plt.subplots(3, gridspec_kw={'hspace': 0.6, 'wspace': 0.9})
9 fig.suptitle('Vertically stacked subplots')
10
11 axs[0].plot(x,y,"go", markersize=1, label='sin(x)')
12 axs[0].set_xlabel('x')
13 axs[0].set_ylabel('sin(x)')
14 axs[0].legend(loc='best')
15 axs[0].set_title('Plot of sin(x)')
16
17 axs[1].plot(x,z,"b-", label='cos(x)')
18 axs[1].set_xlabel('x')
19 axs[1].set_ylabel('cos(x)')
20 axs[1].legend(loc='best')
21 axs[1].set_title('Plot of cos(x)')
22
23 axs[2].plot(y,z,"r-", markersize=1)
24 axs[2].set_xlabel('sin(x)')
25 axs[2].set_ylabel('cos(x)')
26 axs[2].set_title('Plot of sin(x) v/s cos(x)')
27 plt.show()

```

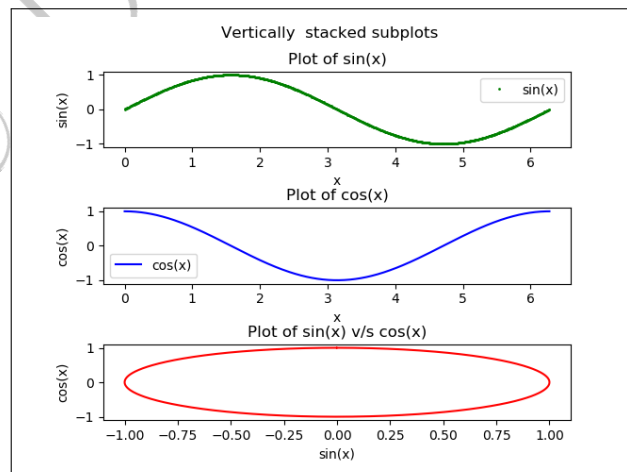


Figure 0.15: Vertically stacked plots.

Random Numbers

A random number generator is a system that generates random numbers from a true source of randomness.

Traditionally random number generator in programming language such as C, Fortran provided us with a number **between 0 and 1**.

Using the `random` library, Python has options to generate either a float random number `random()` between 0 and 1, or an integer, between the user defined limits `randint(start,stop)`.

The code below (`Chap_4:rand_int.py`) prints 10 integer random numbers, between 0 and 10 , both included.

```
Integer Random Numbers
1 # generate random integer values
2 from random import seed
3 from random import randint
4
5 for _ in range(10):
6     value = randint(0,10)
7     print(value)
```

The code below prints 10 float random numbers (which are between 0 and 1).

```
Random Numbers
1 # generate random values between 0 and 1
2 from random import seed
3 from random import random
4
5 for _ in range(10):
6     value = random()
7     print(value)
```

Histograms

Let us say we have collected the data regarding the marks obtained by **100** students, and we would like to look the *frequency distribution* of this data *ie* we would like to have it displayed as a **histogram**.

This is achieved by the `hist(x,bins)`, where `x` is the array containing the data (which need not be sorted), and the default value of `bins` is 10.

The code below (`Chap_4:hist_data.py`) demonstrates the histograms formation from the given data and the results are plotted in Fig. 0.16. In this code we have also calculated some quantities such as **summation of the data** **maximum from the given data set** , to name a few.

Histograms

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 x = [1,1,2,3,3,5,7,8,9,10,
5      10,11,11,13,13,15,16,17,18,18,
6      18,19,20,21,21,23,24,24,25,25,
7      25,25,26,26,26,27,27,27,27,27,
8      29,30,30,31,33,34,34,34,35,36,
9      36,37,37,38,38,39,40,41,41,42,
10     43,44,45,45,46,47,48,48,49,50,
11     51,52,53,54,55,55,56,57,58,60,
12     61,63,64,65,66,68,70,71,72,74,
13     75,77,81,83,84,87,89,90,90,91
14     ]
15
16 n =(len(x))
17 nsum = sum(x)
18 maximum = max(x)
19 minimum = min(x)
20 nrange = maximum-minimum
21 nbins = int(np.sqrt(n))
22
23 plt.hist(x,nbins, histtype='step')
24 # options are 'bar', 'barstacked', 'step', 'stepfilled'
25 plt.xlabel('Bins')
26 plt.ylabel('Frequency')
27 plt.show()

```

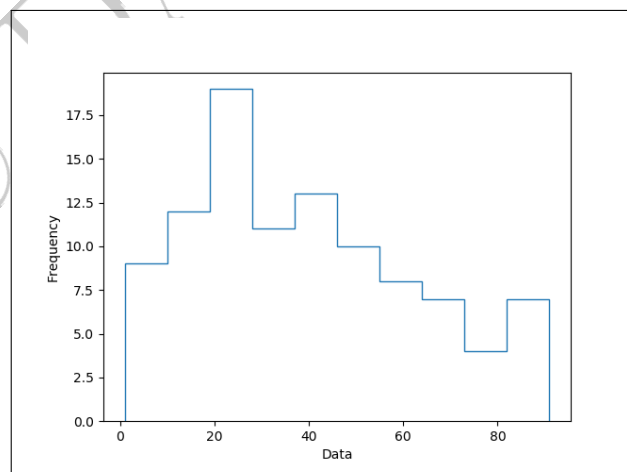


Figure 0.16: Histogram from given data set.

In the earlier section we had discussed the code to generate **random numbers** . We expect that if we generate

greater numbers, then the distribution is expected to be **truly random** *ie* there would be no preferential distribution of the generated numbers, all numbers would be equally probable.

The code below ([Chap_4:hist_rand_int.py](#)) attempts to demonstrate the above, and the results are presented in Fig. 0.17.

```
How random are the generated random numbers
1 import matplotlib.pyplot as plt
2 import random
3 data = [random.randint(1, 100) for _ in range(100)]
4 data1 = [random.randint(1, 100) for _ in range(1000)]
5 data2 = [random.randint(1, 100) for _ in range(10000)]
6
7 fig, axs = plt.subplots(3, gridspec_kw={'hspace': 1.2, 'wspace': 0.9})
8 fig.suptitle('How random are the generated random numbers')
9
10 axs[0].hist(data, histtype='step')
11 axs[0].set_xlabel('Random Numbers')
12 axs[0].set_ylabel('Probability')
13 axs[0].set_title('100 Random numbers generated')
14
15 axs[1].hist(data1, histtype='step')
16 axs[1].set_xlabel('Random Numbers')
17 axs[1].set_ylabel('Probability')
18 axs[1].set_title('1000 Random numbers generated')
19
20 axs[2].hist(data2, histtype='step')
21 axs[2].set_xlabel('Random Numbers')
22 axs[2].set_ylabel('Probability')
23 axs[2].set_title('10000 Random numbers generated')
24
25 plt.show()
```

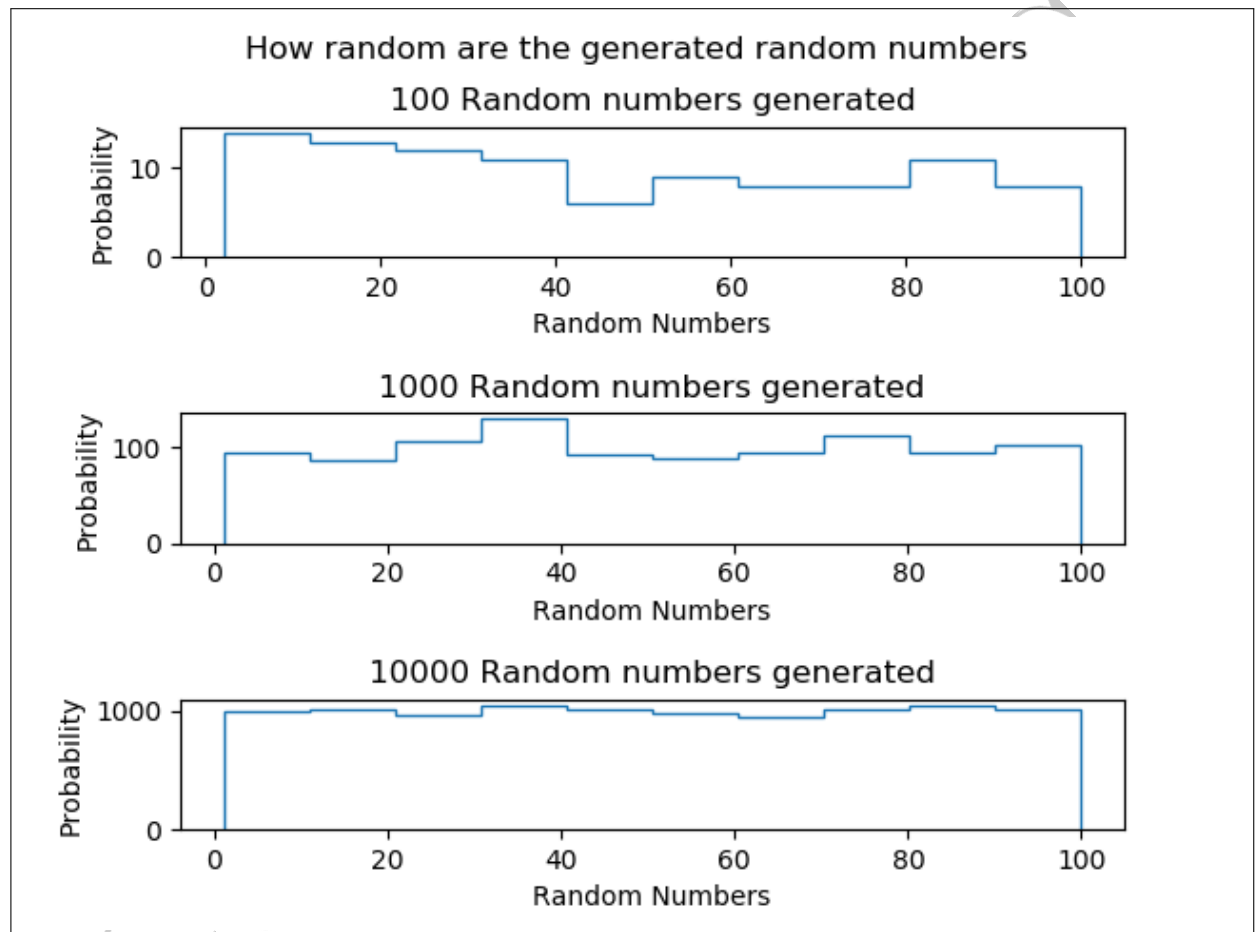


Figure 0.17: Random numbers generated.

Read Data from a File

Most of the time we are interested to store the acquired data set for a detailed offline analysis. This is usually achieved by archiving the data in files. Conventionally, we store (write) the data into files, which are classified as **ASCII** and **Binary** files. The major difference between the two is that ASCII files are human readable (like your Python code), whereas the Binary file is not. Speaking purely in a layman term we can read the contents of an Ascii file directly on the screen using *system* command such as `edit, cat` whereas Binary files need their own software to read this data.

Ascii file

At times data is stored in a **ASCII** file, which is a **text** file, that is readable to human eye. It contains only the text entered by the user and does not include formatting. There are no fonts, italics or boldface.

Table 0.2: Data Stored in an ASCII file

Voltage	Current (micro-amps)
0	0
2	10
2.4	79
2.6	208
2.8	380
3.0	535
3.2	742
3.4	915
3.6	1121
3.8	1285
4.0	1513
4.2	1688
4.4	1860
4.6	2029
4.8	2180

Read data from ASCII file :I

```

1  #{\textcolor{magenta}}{(Chap\_4:read\_data\_ascii\_1.py)}}
2  import numpy as np
3  import matplotlib.pyplot as plt
4  filename="test.txt"
5  xdata=[]
6  ydata=[]
7  f = open(filename)
8  k = 0
9  for line in f:                                # read all the lines in the file
10     data = line.split()                       # split up into columns : data[0], data[1],...
11     print data[0], "\t", data[1]
12     xdata.append(float(data[0]))
13     ydata.append(float(data[1]))
14
15  plt.plot(xdata,ydata,"*")
16  plt.xlabel("Voltage")
17  plt.ylabel("Current")
18  plt.title("Current Voltage Characteristics for RED LED")
19  plt.show()
20  f.close()

```

Incase the data are separated by commas, the command is `data = line.split(",")`. The disadvantage of the above method is that we need a apriori knowledge of the number of data columns, which we then update into the individual 1-d array.

An alternate method to read in the contents of the file is, where

Read data from ASCII file :II

```

1  ##{\textcolor{magenta}}{(Chap\_4:read\_data\_ascii\_2.py)}}
2  import numpy as np
3  import matplotlib.pyplot as plt
4  filename="data_ascii.txt"
5
6  f = open(filename, 'r')
7  data = np.genfromtxt(f)                       # info from the file -> data[:,0], data[:,1]
8  print (np.shape(data))                       # details about number of columns and rows in data
9
10  plt.plot(data[:,0],data[:,1],"r*")
11  plt.xlabel("Voltage")
12  plt.ylabel("Current")
13  plt.title("Current Voltage Characteristics for RED LED")
14  plt.show()
15  f.close()

```

The mandatory requirement for the function `genfromtxt` in the package `numpy` is the source of the data *viz.* `genfromtxt(datasource)`. It would load the data from the textfile into an appropriately shaped array. We

can retrieve the information about the data size, *number of rows and columns* from the `shape` function in the package `numpy`.

`.wav` file

The `.wav` (Windows Audio Format) format is a subset of Microsoft's RIFF specification for the storage of multimedia files.

A RIFF consists entirely of chunks, starts out with a file header followed by a sequence of data chunks.

A WAVE file consists of two sub-chunks

1. `fmt` chunk specifying the data format
2. `data` chunk containing the actual sample data.

An almost complete description which seems totally useless unless you want to spend a week looking over it and is schematically presented in Fig.0.18, (the figure has been taken from public domain)

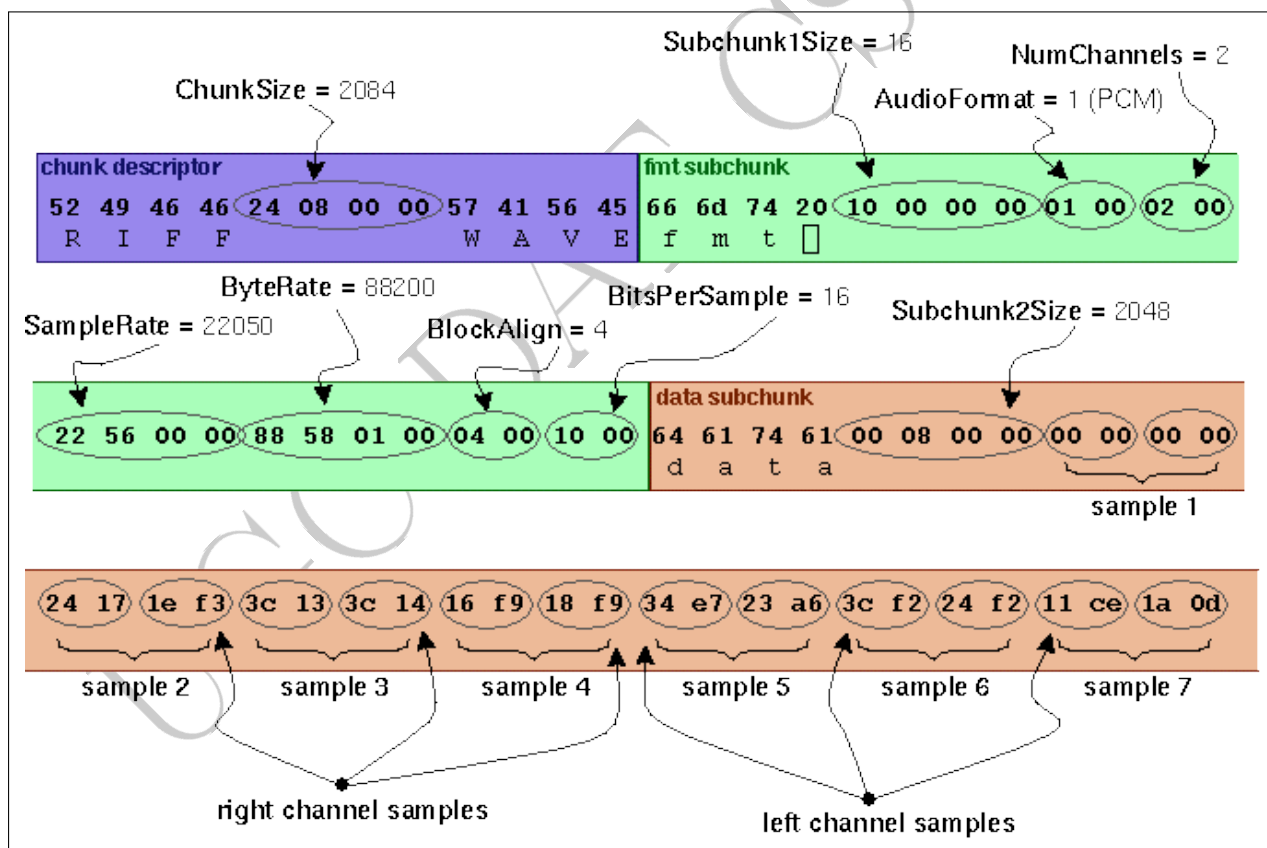


Figure 0.18: Schematic representations of the canonical form of the `.wav` file.

The structure (user created data format) for the *.wav* file is

```

1 struct wavfile
2 {
3     char    id[4];           // should always contain "RIFF"
4     int32_t totallength;     // total file length minus 8
5     char    wavefmt[8];     // should be "WAVEfmt "
6     int32_t format;         // 16 for PCM format
7     int16_t pcm;            // 1 for PCM format
8     int16_t channels;       // channels
9     int32_t frequency;      // sampling frequency
10    int32_t bytes_per_second;
11    int16_t bytes_by_capture;
12    int16_t bits_per_sample;
13    char    data[4];         // should always contain "data"
14    int32_t bytes_in_data;
15 }

```

Libsndfile is a C library for reading and writing Windows WAV files through one standard library interface.

It is available under the GNU Public License, textcolorblue<http://mega-nerd.com/libsndfile>

Once installed, you can

```

    compile the source code
    gcc -o sndfile-to-text -lsndfile sndfile-to-text.c
    run the executable
    sndfile-to-text inp.wav output.txt

```

Hence, the *.wav* file can be read using either a **C** program or a **Python or Octave** code.

Now the data in the *.wav* file is said to be **time stamped**, *ie.* we know exactly the time at which the data was recorded, ofcourse the time information is relative, *ie.*, the time interval between two successive data points is **constant**, and is governed by the **sample rate**, actually it is the inverse of **sample rate**. The value of this parameter is stored in the file, in the first chunk. Besides this information, the other relevant information is the number of **channels** we have in the data set.

Now the product of the **length** of the data set and the **sample rate** (it's inverse) would give us the total time of acquisition.

The package **scipy.io** has function **wavfile.read** to read the data stored in a *.wav* file. This command returns the **sample rate** and the **data**, where the stored data is retrieved into an appropriately shaped array.

Thus the use of the command is **samplerate, data = read('liss_pat.wav')** then, parameters **samplerate** holds the value of the sampling rate at which the data was acquired, and the array **data** now contains the stored data. If we have a **stereo** recording then **data** would have two columns, **data[:,0]** and **data[:,1]**, whereas if the recording was **mono** we shall have only one column data.

The command **times = np.arange(len(data))/float(samplerate)**, generates an array whose length equals the

length of the data set (total number of data points) and have a time difference which equals the *inverse* of the **samplerate**.

Read data from *.wav* file

```

1 from scipy.io.wavfile import read          # load module to read the .wav
2 import matplotlib.pyplot as plt
3 import numpy as np
4 samplerate, data = read('liss_pat.wav')     # samplerate holds the sampling rate
5                                             # data would c
6 times = np.arange(len(data))/float(samplerate) # generate the time info
7 left = data[:,0]                          # extract the single channel info
8 right = data[:,1]
9 (samples, channels) = data.shape
10 duration = len(data)/samplerate
11 print ("the recording duration is {0:3.2f} seconds".format(duration))
12 print ("the number of channels are {0:d}".format(channels))
13
14 fig, ax = plt.subplots(3)
15 fig.suptitle('Lisszous pattern')
16 ax[0].plot(times, left)
17 ax[1].plot(times, right)
18 ax[2].plot(left, right)
19
20 ax[0].set_title('Across R')
21 ax[1].set_title('Across C')
22 ax[0].set_xlim(0,0.2)
23 ax[1].set_xlim(0,0.4)
24 ax[0].set_xlabel('time')
25 ax[0].set_ylabel('voltage')
26 ax[1].set_xlabel('time')
27 ax[1].set_ylabel('voltage')
28 ax[2].set_xlabel('Vc')
29 ax[2].set_ylabel('Vr')
30
31 plt.subplots_adjust(hspace = 0.7)
32 plt.show()
33 #plt.savefig('fig2_16.jpg')

```

Data Fitting

In experiments, we usually acquire a convenient discrete set of data, and then we wish to establish a global relationship between the acquired data, so that a global behavior of the same could be arrived at. For example, we have acquired a 2 parameter data set (x_i, y_i) , and now we wish to deduce, based on this set a function which maps or establishes the relationship between the two variables, i.e

$$\begin{aligned} y &= f(x) \\ &= a_2 + [a_1 \cdot x] \\ &= c + m \cdot x \end{aligned}$$

assuming a linear dependence on the two variables.

Now, let us guess, a relationship between the two variables, such that

$$y_{modelled} = a_0 + [a_1 \cdot x_{measured}]$$

Now, if our guess function, is correct, then the **difference**, between the obtained (modeled) value of y_i , and the actual value, where the measurement is available should be minimum. Since, the sign of the difference is numerically irrelevant, we usually compute the square of the difference (ignore the nomenclature),

$$\chi^2 = \left[y_i - [a_0 + [a_1 \cdot x_i]] \right]^2$$

and the good-ness of our model, fit, would be provided if these differences are minimal. Hence, the name **least-square fit**, where in we attempt to deduce the coefficients a_0 , & a_1 , so that the difference (or rather the square of the difference) between the calculated value of y_i^{calc} , and the y_i^{actual} , is minimized by some measure.

Let us guess a mathematical relationship between the variables

$$y_{model} = a_0 + a_1 \cdot (x)$$

From the observations

$$[(x_1, y_1), (x_2, y_2), \dots, (x_N, y_N)]$$

we look at

$$[y_1 - (a_0 + a \cdot x_1), y_2 - (a_0 + a \cdot x_2), \dots, (y_N - (a_0 + a_1 \cdot x_n))]$$

Now, we wish to evaluate the goodness of the model we have arrived upon

$$\text{Perfect model} \implies y_i - (a_0 + a_1 \cdot x_1) = 0$$

Practical model $\implies y_i - (a_0 + a_1 \cdot x_1) \longrightarrow 0$

Best model

- mean should be small
- variance measure of goodness

$$\begin{aligned} a_0 &= b \\ a_1 &= a \\ \sigma_{y-(a \cdot x + b)}^2 &= \frac{1}{N} \sum_{n=1}^N (y_n - (a \cdot x_n + b))^2 \end{aligned}$$

Ofcourse the errors would have to be folded in to ensure appropriate weightage

Ignoring this for the moment, our goal is to find the values of a and b , that minimizes the error

$$\begin{aligned} E(a, b) &= \sum_{n=1}^N (y_n - (a \cdot x_n + b))^2 \\ \frac{\delta E}{\delta a} &= 0 \\ \frac{\delta E}{\delta b} &= 0 \\ \text{Differentiating } E(a, b), &\implies \\ \frac{\delta E}{\delta a} &= \sum_{n=1}^N -2[y_n - (a \cdot x_n + b)] \cdot (x_n) \\ \frac{\delta E}{\delta b} &= \sum_{n=1}^N -2[y_n - (a \cdot x_n + b)] \cdot 1 \end{aligned}$$

setting the derivative to zero \implies

$$\sum_{n=1}^N [y_n - (a \cdot x_n + b)] \cdot (x_n) = 0$$

$$\sum_{n=1}^N [y_n - (a \cdot x_n + b)] \cdot 1 = 0$$

$$\sum_{n=1}^N x_n \cdot y_n = \left[\sum_{n=1}^N x_n^2 \right] a + \left[\sum_{n=1}^N x_n \right] b$$

$$\sum_{n=1}^N y_n = \left[\sum_{n=1}^N x_n \right] a + \left[\sum_{n=1}^N 1 \right] b$$

Where a & b can be determined from corresponding algebra

Now if we were to incorporate errors, then we wish to give lower weightage to points with greater errors and vice-versa.

Hence, the quantity to minimize, known as Chi-square, which gives us the *goodness* of the *fit* is

$$\chi^2 = \sum_{n=1}^N \left[\frac{(y_n - (a \cdot x_n + b))}{\sigma_i} \right]^2$$

Linear / Quadratic Least Square Fit

Suppose, we have an ADC, which gives us the following channel numbers 0, 670, 1023 corresponding to the following voltages 0, 3.3, 5. Then we wish to derive a relation between the observed channel number and the corresponding input voltage.

Thus, we wish to fit the given data and obtain the values of the coefficients a_0 & a_1 , which in turn help us determine the unknown voltage from the given ADC channel number. The package `numpy` contains a function `polyfit(x,y,deg)`, which fits a polynomial $p(x) = p[0] * x^{deg} + \dots + p[deg]$. If $deg = 1$, we have a `linear` fit to the data-set, and a $deg = 2$ results in a `quadratic` fit to the data set. This is achieved by the following code

In the code (`prg:chap5:fit_adc_calib.py`) *Lines* : 15 – 18, help us calculate the *modelled* curve, using the coefficients a_0 & a_1 . Please crosscheck with the version you are using for the order of the polynomials that are returned. Unfortunately, `np.polynomial.polynomial.polyfit` returns the coefficients in the opposite order of that for `np.polyfit` and `np.polyval`.

Linear Fit To the Data

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 y = [0.0, 3.3, 5.0]
5 x = [0, 670, 1023]
6 z = []
7 # store in the values in an array / list
8
9 fit = np.polyfit(x,y,1)
10 # perform the 1d least square fit
11 # p=polyfit(x,y,n)
12 # returns a polynomial p(x), of degree n, such that the variable y,
13 # can now be approximated as
14
15 for n in range(0, len(x)):
16     temp = fit[1] + fit[0]*x[n]
17     z.append(temp)
18     temp = 0
19
20 plt.plot(x,y,"ro",x,z,'—k')
21 # defining the axis ranges
22 plt.ylim(0, 6)
23 plt.xlim(0, 1100)
24 # setting the x & y axis labels
25 plt.xlabel("Channel")
26 plt.ylabel("Voltage")
27 plt.title("ADC Calibration")
28 plt.show()
29 #plt.savefig("fig3_6.jpg")
```

The same can be achieved using the following code, and the results are presented in Fig. 0.19.

Linear Fit To the Data

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 y = [0.0, 3.3, 5.0]
5 x = [0, 670, 1023]
6 # store in the values in an array / list
7
8 fit = np.polyfit(x,y,1)
9 # perform the 1d least square fit
10 # p=polyfit(x,y,n)
11 # returns a polynomial p(x), of degree n, such that the variable y,
12 # can now be approximated as
13
14 fit_fn = np.poly1d(fit)
15 # fit_fn is now a function which takes in x and returns an estimate for y
16 print fit_fn
17
18 plt.plot(x,y, 'ro', x, fit_fn(x), '—k')
19 # defining the axis ranges
20 plt.ylim(0, 6)
21 plt.xlim(0, 1100)
22 # setting the x & y axis labels
23 plt.xlabel("Channel")
24 plt.ylabel("Voltage")
25 plt.title("ADC Calibration")
26 plt.show()
27 #plt.savefig("fig3_6.jpg")

```

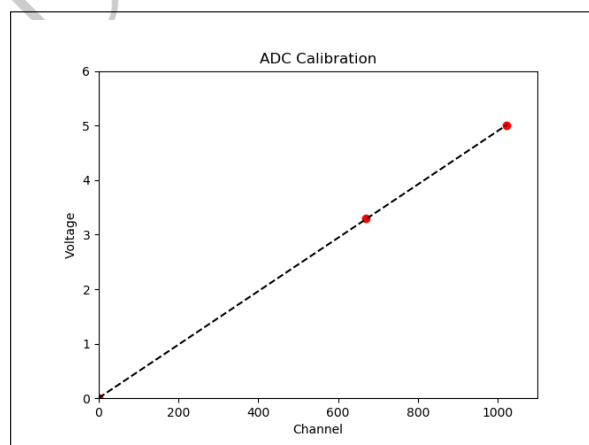


Figure 0.19: Linear Fit to obtain ADC calibration.

Least Square Fit to User Function

We know that **curve fitting** is the process of constructing a mathematical function, which best represents the data set, so that we can extract information at those points (values) for which we do not have the experimental data. However, we may have a situation wherein the measured data points (x_i, y_i) have a specific relation, for example, we know that given a RC circuit, during charging, the voltage across the capacitor $V(t)$ at time t is given by

$$V(t) = V_0 \left(1 - e^{\left[\frac{t}{RC} \right]} \right)$$

A **fit** to the charging or discharging data, can help us extract the **time constant τ** of the circuit.

Suppose we have recorded the data for the voltage across a capacitor ($R = 100k\Omega, C = 220\mu F$) during the charging process. It is stored in a two column *ascii* file, in the format **time,voltage**. We read this data, and subject it to **non linear least square fit to the expected function**, using the function **curve_fit** from the **scipy.optimize** package.

The arguments for the function are **curve_fit(func,xdata,ydata)**, where **func** is a the model function, and it takes the **independent** variable **xdata** with **ydata** being the dependent variable. It returns an array **popt** which is an array, such that the quantity **func(xdata, *popt)-ydata** is *minimized* to obtain the *best fit*.

There is a provision to pass to function the initial guess value, as **curve_fit(func,xdata,ydata,p0)**, where **p0** is an array which contains the guess value, and the number of parameters should equal the parameters which would be returned by the function on a successful fit.

The code (**prg:chap5:fit_rc_user_function.py**) for the same is

The Lines5-6, define the function to be used to fit the *time, voltage* data. The form is

$$\begin{aligned} V(t) &= V_0 \left(1 - e^{\left[\frac{t}{RC} \right]} \right) \\ &= V_0 - V_0 e^{\left[\frac{t}{RC} \right]} \\ &= a - b \exp\left[\frac{time}{c} \right] \end{aligned}$$

Lines8-12 open the *ascii* file, which has a header which is just read, then the data is extracted into a 2D array called **data**, where **data[:,0]**, **data[:,1]** correspond to the two 1D arrays, which now contain the **time** and **voltage** information. .

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from scipy.optimize import curve_fit
4
5 def func(sdata,a,b,c) : nc
6     return a - b*np.exp(-c*sdata[:,0])
7
8 f = open('charging_data.txt', 'r')
9 header1 = f.readline()
10 data = np.genfromtxt(f)
11 f.close()
12 print len(data[:,0])
13
14 guess = np.array([5.0,5.0,0.07])
15 popt, pcov = curve_fit(func, data[:,0], data[:,1], guess)
16 print popt.size
17 print pcov.size
18 print popt
19 # to use the fitted parameters pass them to the func
20 yfit = func(data[:,0], *popt)
21 plt.plot(data[:,0], data[:,1], 'ro', data[:,0], yfit, 'b—')
22 plt.xlabel("Time")
23 plt.ylabel("Voltage across capacitor")
24 plt.ylim(0,5.2)
25 plt.show()
```

Line 14 creates the *guess* values and the curve fitting is obtained in Line 15. We then use the fitted parameters in Line 20, to construct the curve obtained via the fitting procedure.

When the above code is executed (results presented in Fig. 0.20), we obtain the following values

$$\begin{aligned}a &= 4.92 \\b &= 4.84 \\c &= 0.047 \\&= \frac{1}{RC} \\&= \frac{1}{100^3 \times 220^{-6}} \\&= 0.045\end{aligned}$$

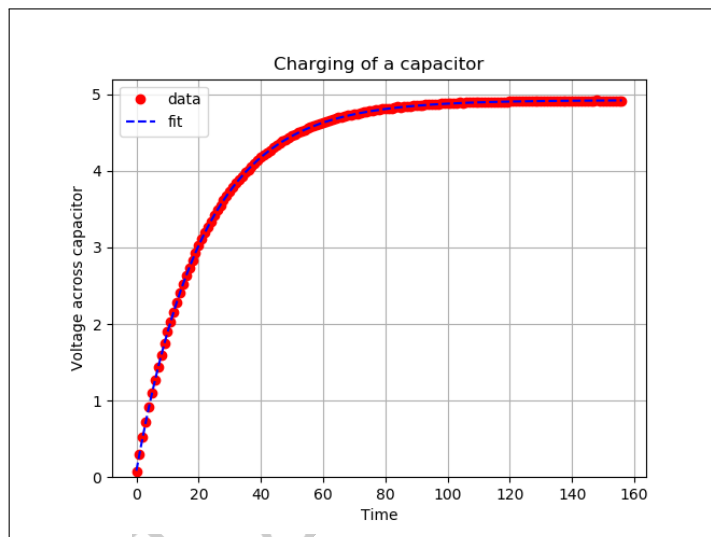


Figure 0.20: Voltage across the capacitor during charging.