

Natural Sciences, Part IB, Introduction to Computing

Introduction to Programming, Data Analysis and Manipulation in python

Tutorial and Instructions for Practical Sessions

Hrvoje Jasak
Department of Physics
The Cavendish Laboratory
(based on earlier versions by Christopher Lester,
David MacKay, Roberto Cipolla and Tim Love)

24th January 2022

Summary

This document provides an introduction to computing and the use of the `python` programming language. It will help you teach yourself to write, compile, execute and test simple computer programs. It describes the computing exercises to be completed in the IB course.

Please refer to the Literature for some books which will help you learn `python` syntax and language features. In this guide we shall mention main aspects of the language, what they do and why we need them.

You are advised to attend the hands-on lab sessions. Demonstrators will be on-hand each day between 2pm and 4.30pm to help you and answer your questions. Attendance at one session per week, for six weeks, should be sufficient to complete the course. Details of the Laboratory Sessions are given below.

The Course Homepage is

<https://www-teach.phy.cam.ac.uk/students/courses/introduction-to-computing/83>

Literature

1. The “original” `python` book, written by the author of the language. It is free, easy to read and a good place to start:
Guido van Rossum and the Python development team: “Python Tutorial, Release 3.7.0”, Python Software Foundation, September 02, 2018
https://bugs.python.org/file47781/Tutorial_EDIT.pdf
2. **Anthony Scopatz and Kathryn D. Huff**:
Effective Computation in Physics, O’Reilly Media, Inc, June 2015, ISBN: 9781491901533
<https://www.oreilly.com/library/view/effective-computation-in/9781491901564/>
3. **Allen B. Downey**: Think Python, O’Reilly Media; 2nd edition, January 2016, ISBN-13: 978-1491939369
<https://greenteapress.com/wp/think-python-2e/>

Laboratory Sessions

You are advised to attend the hands-on lab sessions. These sessions run from Thursday 27 January 2022 to Wednesday 9 March 2022 inclusive, and take place on-line using a Teams channel.

Demonstrators will be on-hand each day between 2:00pm and 4.30pm to help you and answer your question. Attendance at one session per week, for six weeks, should be sufficient to complete the course.

Session Schedule

Session schedule and College allocation is given in Table 1.

Day	Colleges
Monday	CAI, CC, CHR, CHU
Tuesday	CL, CTH, DOW, ED, EM, F
Wednesday	G, HH, HO, JE, JN, K
Thursday	M, N, NH, PEM, PET, Q
Friday	R, SE, SID, T, TH, W

Table 1: Teams laboratory sessions.

Can I Change My Day/Session?

It is possible to swap sessions if you and someone else to swap with: we aim to keep all sessions of similar size. Please contact the course organiser (see front of this handout) if you would like to do this as action will be required on their part to activate the swap in Microsoft Teams.

If you have swapped sessions, your session may no longer correspond to the one that most of those in your college go to. If you become unsure which session you are in, it is possible to rediscover the answer by looking in Microsoft Teams to see which channel you have access to in the IB Computing team.

How Should I Use the Help Sessions?

For each session (*e.g.* 'Friday PM') there is a dedicated Microsoft Teams channel of which you have been made a member, within the The IB Physics B Scientific Computing team, of which everyone on the course is a member.

We recommend that if you attend a help sessions that you do so actively collaborating in a group of around two to five other students. Most of you would organise those groups within your college, but they could span multiple colleges if you want them to. Those of you entering that way should create a group chat within Teams for yourselves.

During the help session, you should discuss your progress and problems with your peers within the private chat of the local group session you started. This chat will be visible only to the small number of those of you in it. You can use text messages in the chat, share screenshots, including video calls, screen screens, or collaborate any way you wish.

If you encounter a problem that you cannot overcome after having discussed it with your peers privately, you can then go to the main (global) chat for that channel to ask for a demonstrator to join your private group to chat with you there. Simply type a short message such as “we could use help” in the global channel and wait. As soon as a demonstrator is available, they will tell you who they are (there are more than one demonstrator per session), and then YOU can then add that demonstrator to your private call or chat group. They will join, discuss your problem with you in the privacy of your own group, and then when done will leave your personal group chat and return to the global one to see if others need help. The main (global) chat therefore acts as a queuing system, and the demonstrators should mark the messages they have dealt with, so everyone sees where they are in the queue.

For Demonstrators

Demonstrators are encouraged mark the messages they have dealt with, so that everyone sees where they are in the queue.

Assessment Deadlines

The laboratory sessions attended by demonstrators begin on 27 Jan 2022, and will then run for seven successive weeks (from each Thursday to the Wednesday following). Attendance at the sessions is not compulsory, but is recommended for those of you who feel you would benefit most from the help of people with substantial programming experience. Such people would be expected to attend one of these lab sessions per week and could submit their work for assessment at the end of the session. Regardless of whether a student attends a lab session or not, the deadlines for submission are the same.

The **nominal deadline** for the task in “SESSION 1” (pages 30-56) is 11pm on Wednesday 02 Feb 2022, as this is the last day of the 1st week of lab sessions. Similarly, the work for “SESSION 2” is due at 11pm on the next Wednesday, and so on. This is summarised in Table 2

Name	Pages	Corresponding lab sessions	Nominal Deadline	(extended deadline)
SESSION 1	30-56	27 Jan 2022 – 02 Feb 2022	11pm 02 Feb 2022	11pm 09 Feb 2022
SESSION 2	56-77	03 Feb 2022 – 09 Feb 2022	11pm 09 Feb 2022	11pm 16 Feb 2022
SESSION 3	77-88	10 Feb 2022 – 16 Feb 2022	11pm 16 Feb 2022	11pm 23 Feb 2022
SESSION 4	88-112	17 Feb 2022 – 23 Feb 2022	11pm 23 Feb 2022	11pm 02 Mar 2022
SESSION 5	113-122	24 Feb 2022 – 02 Mar 2022	11pm 02 Mar 2022	11pm 04 May 2022
SESSION 6	122-137	03 Mar 2022 – 09 Mar 2022	11pm 09 Mar 2022	11pm 04 May 2022

Table 2: Submission deadlines, 2022.

You will note the additional column headed “extended deadline” which is always at least one week later than the nominal deadline. Very occasionally people are seriously ill, or affected by an unavoidable and unforeseeable event that makes it not possible to meet a deadline. If such a situation arises, an extension will automatically be granted to the “extended deadline” for no loss of credit.¹

¹It is not necessary to apply for this extension! Don’t mail me to ask for one! Handing work in between the Nominal and Extended deadlines is *itself* treated as an application for extension and is automatically granted. Note that extensions beyond the extended deadline, even by seconds, will *never* be granted.

Contents

Contents	6
1 Course Aims	12
1.1 Before You Start	12
1.2 The Six Pieces Of Submitted Work (“SESSIONS”) . . .	13
1.3 Why Self-Assessment?	14
1.4 Collaboration	15
1.5 Copying	15
1.6 Have I Done Enough?	16
1.7 Feedback	16
2 Installing Software	17
2.1 Installing and Managing <code>python</code>	17
2.1.1 Pip	18
2.1.2 <code>python</code> Versions and Compatibility	19
2.2 Apple Mac (MacOS-X)	21
2.2.1 Terminal	21
2.2.2 Python 3	21
2.2.3 Text Editor	23
2.3 Microsoft Windows	23
2.3.1 Terminal	23
2.3.2 Python 3	24
2.3.3 Text Editor: Installing NotePad++	26
2.4 Linux	27
2.4.1 Terminal	27
2.4.2 Python 3	27
2.4.3 Text Editor	28
3 Programming a Computer	30
3.1 What is a Computer Program?	30
3.2 Software Development Process	31
3.3 Compiled and Interpreted Languages	31
3.4 Programming Paradigms	33
3.5 The C++ Programming Language	34
3.6 Python	35
3.7 In This Course	35

4	Python for Absolute Beginners	36
4.1	Need a Calculator?	36
4.2	Using Variables	37
4.3	Hello, World!	38
4.4	C++ and Hello, World!	39
4.5	Source Code or Direct Interaction?	39
4.6	Indentation is a Part of the Program	39
5	What Has python Ever Done For Me?	41
5.1	Module and Package	41
5.2	Built-In Modules	42
5.3	Extension Modules	43
6	A Simple Adder	44
6.1	Execute the Program	44
6.2	What's in the File?	44
6.3	Development Cycle	45
7	Session 1: Simple Adder	47
7.1	Objectives	47
7.2	Getting Started	47
7.3	Find Yourselves a Computer	47
7.4	Some Terminal-Based Commands	48
7.5	Filing System and Directory Structure	48
7.6	A Text Editor	49
7.7	Computing Exercise	51
7.8	Assessment	54
8	Working with Variables	56
8.1	Basic Data Types in Computing	56
8.2	Variable Type and Declaration	56
8.3	Type Conversion	57
8.4	Duck Typing	58
8.5	Arithmetic Expressions	58
8.6	Working with Floating Point Numbers	60
9	Basic Containers	62
9.1	Lists	62
9.2	Tuples	62
9.3	Sets	63
9.4	Dictionaries	64

9.5 Which Container to Use?	65
10 Basic Programming Structures	66
10.1 Control Statements	66
10.2 Loops	69
10.2.1 Repetition Control	69
10.2.2 Looping on Containers	70
10.2.3 Statistical Analysis	70
11 Session 2: Computing	73
11.1 Objectives	73
11.2 Computing Exercises	73
11.3 Tips	74
11.4 Assessment	75
12 Functions, Modules, and Packages	77
12.1 Functions	77
12.1.1 A Simple Function	77
12.1.2 Return Value	78
12.2 Modules	79
12.3 Packages	81
12.4 Some Examples	82
13 Session 3: Monte Carlo	84
13.1 Objectives	84
13.2 Simple Monte Carlo	84
13.3 Data Visualisation	86
13.4 Computing Exercises	86
13.5 Assessment	87
14 Object-Oriented Programming	88
14.1 Introduction	88
14.2 Working with Classes	89
14.3 Defining a Class	89
14.4 Inheritance	91
14.5 Types of Attributes and Methods	93
14.6 Attributes	94
14.7 Methods	94
14.8 Operator Overloading	95

15 Working With Strings	96
15.1 The Basics	96
15.2 Slicing and Splitting	97
15.3 String Manipulation	98
15.4 Example: Check for Anagrams	99
16 Formatted Output	101
16.1 Print Statements	101
16.2 Escape Characters	102
16.3 Raw strings	103
17 String Formatting	104
17.1 Using the % operator	104
17.2 Using <code>.format()</code>	106
17.3 Using f-strings	107
17.4 The choice of formatting	107
18 Session 4: Method of Bisection	108
18.1 Objectives	108
18.2 Computing Exercises	108
18.3 The Bisection Method	108
18.4 Notes on the Algorithm and Implementation	109
18.5 Assessment	111
19 Notes Concerning the Remaining SESSIONS	112
20 Session 5: Planets	113
20.1 Objectives	113
20.2 Your Task	113
20.3 Ideas for What to Do	115
20.4 Assessment	116
20.5 Worked Solution for the Planets Problem	117
20.5.1 Step-by-Step Guidance	117
20.5.2 Sample Code Listing	117
21 Advanced Containers	122
21.1 More On Lists	122
21.1.1 Lists and Loops	122
21.1.2 Nested Lists and List Comprehension	123
21.1.3 Further Useful Methods for Lists	125
21.2 More On Tuples	126
21.3 More on Sets	126

21.4 More on Dictionaries	128
21.4.1 Dictionaries and Loops	128
21.4.2 Dictionary Comprehension	129
22 Session 6: Collisions	130
22.1 Objectives	130
22.2 Your Task	130
22.3 Ideas for What to Do	131
22.4 Assessment	133
22.5 Worked Solution for the Collisions Problem	134
22.5.1 Sample Code Listing	134
23 Regular Expressions	138
23.1 Metacharacters	138
23.2 Special sequences	140
23.3 Sets	141
24 Working with Text Files	144
24.1 File Access Modes	144
24.2 Opening and Closing Files	145
24.3 Reading	146
24.4 Writing	146
24.5 Some Examples	146
25 Manipulating CSV files with pandas	150
A Submitting Your Work	155
B Some Useful UNIX Commands	158
C Some Useful Windows Commands	165

Preliminaries

1 Course Aims

This course provides a tutorial introduction to computing in general, with aspects of the `python` programming language and its many extension libraries. For those whose future career or programming needs require a more complex programming environment, we will briefly summarise the work-flow of a compiled programming language such as C++. However, the versatility and wide use of `python` in industry and academia has pushed the course in the direction of an interpreted language: `python`.

The objective of the course is not to teach you how to program – this is something best achieved by doing it – but to give you basic concepts and practical tools to manipulate data and develop your programming curiosity. It will help you teach yourself to **write, (compile) execute, test, debug and use simple computer programs** which read data from the keyboard, perform some computations on that data, and print out the results; and how to display the results graphically.

In these sessions you should work through the examples and exercises described in this document. Each session consists of a tutorial (which must be read **before** the start of the laboratory session) and interleaved computing exercises. To benefit most from the laboratory session and from the demonstrators' help you must **read the tutorial sections before the start of each session**.

Please allow two hours preparation time per week.

You are expected to attend the sessions weekly (though it is not a problem if you attend more frequently). If you attend weekly for the full seven weeks, and if you submit the required self-assessed work at the end of each session, you will meet all six submission deadlines, and you will find the seventh week is spare.

You are permitted to do the work for this course in College or on your own laptop, *etc.* but please bear in mind that it is your responsibility (with the help of Demonstrators) to install the necessary software: mainly `python` and its packages.

1.1 Before You Start

Before you start working on course materials, you need to set up your environment. Please review Section 2, (pages 12 – 29) and make sure `python` works for you.

1.2 The Six Pieces Of Submitted Work (“SESSIONS”)

In this course, your progress will be **self-assessed** by submitting **six pieces of work** – one for each laboratory session. Unimaginatively, but hopefully intuitively, the six pieces of work are entitled:

- “SESSION 1” (pages 30 – 56),
- “SESSION 2” (pages 56 – 77),
- “SESSION 3” (pages 77 – 88),
- “SESSION 4” (pages 88 – 112),
- “SESSION 5” (pages 113 – 122) and
- “SESSION 6” (pages 122 – 137)

and contain instructions for what to do for each self assessment.

The all-important submission deadlines are found in Section A of the Appendix of this document.

Most people will submit their self-assessed work at the rate of one submission a week, uploading the work at the end of each of the first six laboratory sessions using the link on the course web-page (URL on the front cover of this document).² Doing is good practice as it will ensure that you submit all work in good time.

Everyone who is switched on should expect to achieve full marks. In 2020, approximately 90% of people taking the course achieved full marks. The majority of those who lost marks lost them by failing to meet one or more submission deadlines (in some cases by mere seconds!) or by failing to hand in any work at all. Please do not leave your work to the last minute.

There are a total of **8 marks**³ available in total for the the work submitted in the course. The submissions for sessions 1 through 4 can acquire 1 mark each. The submissions for the extended projects (sessions 5 and 6) contain 2 marks each (the additional mark reflecting the additional physics/investigative content needed by these sessions). Late work, work that suggests the student did not engage with the spirit of the self assessment, or submissions which fail to justify why the nominated self-assessed task was appropriate for the student in question, scores nothing.⁴

²Note, there is nothing to stop people submitting work at a faster rate if they so desire. In 2010, one student submitted responses to all six assessments by the end of week two.

³These marks do not carry the same weight as marks in a IB Physics B Tripos examination.

⁴All students who enter into the spirit of the course and submit work on time, should

1.3 Why Self-Assessment?

The self-assessment process is based on a desire to treat you as adults, and on the assumption that you are interested in learning, capable of choosing goals for yourself, and capable of self-evaluation.

There are different types of programmers. Some people like to get into the nuts and bolts and write programs from scratch, understanding every detail – like building a house by first learning how to make bricks from clay and straw. Other people are happy to take bricks as a given, and get on with learning how to assemble bricks into different types of building. Other people are happy to start with an existing house and just make modifications, knocking through a wall here, and adding an extension there.

I don't mind what sort of programmer you become. All these different skills are useful. I encourage you to use this course to learn whatever skills interest you. The exercises give opportunities for various styles of activity.

Programming is an essential skill, just like graph-sketching. If you don't know how to sketch graphs, what do you do? – practice, read books, talk to supervisors and colleagues, practice some more. As second-year physicists, graph sketching should be second nature, and you should use it as a tool in all physics problems you study. Similarly, programming is a valuable tool for thinking, exploring, investigating, and understanding almost any scientific or technical topic. You should find that programming can help you with your 2nd year, 3rd year, and 4th year courses.

This course exists to get you up to speed on programming, for the benefit of all your courses.

As the main form of assessment for all this work will be self-assessment. Ask yourself, “Have I mastered this stuff?” If not, take appropriate action. Think, read, tinker, experiment, talk to demonstrators, talk to colleagues. Sort it out. Sort it out well before the final deadline. When you have checked your own work and fully satisfied your self-assessment, you should submit an electronic record of your self-assessment manner described at the end of the SESSION 1 instructions (pages 30 – 56) – *i.e.* by using the assessment upload link on the course web-page. By submitting your work, you are confirming

be able to score 100%, and indeed this has been the modal score in all previous years, even though there is always a sizeable number of students who lose marks through late submissions or through failing to justify why their self-assessed task was appropriate for them. Programming is supposed to be a useful skill that will help you solve problems, not a time consuming chore. If the task you are setting yourself is taking ages and ages, then perhaps you have set your sights too high. Different people have different pre-existing skills and will need to set themselves tasks at different levels. That is why this is a self-assessed course! No one who follows the course instructions carefully, should need this course to consume inordinate amounts of his/her time.

that you have assessed yourself and achieved the session’s objectives. All self-assessed submissions will be made available to the IB examiners.

Note that every year a small number of students spend far too much time on this course, and complain about that time, presumably having failed to note the almost binary nature of the mark scheme, and that it is up to them to set themselves tasks of an appropriate difficulty.

Each session has tasks of two types: first, “write a program to do X”; and second “set your own additional goal (Y), further testing your skill, and write a program to do Y”. This “additional programming goal” doesn’t have to be any harder than X. You should choose the goal yourself, and be imaginative. This goal and its solution are what you submit.

1.4 Collaboration

How you learn to program is up to you, but let me make a recommendation:

I recommend that you do most of your programming work in **pairs**, *with the weaker programmer doing the typing*, and the stronger one looking over his/her shoulder.

Working in pairs greatly reduces programming errors. Working in pairs means that there is always someone there to whom you can explain what you are doing. Explaining is a great way of enhancing understanding. It’s crucial to have the weaker programmer do the typing, so that both people in the pair understand everything that’s going on.

At the end of the day, you must all self-assess *individually*, and you must submit *individual* electronic records of your work via the link on the course web-page.

1.5 Copying

When programming in real life, copying is strongly encouraged.

- Copying saves time;
- Copying avoids typing mistakes;
- Copying allows you to focus on your new programming challenges.

Similarly, in this course, copying may well be useful. For example, copy a working program similar to what you want to do; then modify it. Feel free to copy programs from the internet. The bottom line is: “do you understand how to solve this sort of programming problem?” Obviously, copying

someone else's perfect answer verbatim does *not* achieve the aim of learning to program. Always self-assess. If you don't understand something you've copied, tinker with it until you do.

You should not copy anyone else's "additional programming goal". You should devise this goal yourself, and solve it yourself.

If necessary, set yourself further additional programming goals.

1.6 Have I Done Enough?

If you find yourself asking "Have I done enough for this week's session?", the answer is "Please self-assess!" Do you feel you've got a firm grasp of all the highlighted concepts? If so, then that's enough. One thing to check is "*Have I developed my understanding of this week's physics topic?*" That's one of the aims this term – to get fresh insights into orbits, dynamics, waves, statistical physics, pressure, temperature, and so forth.

1.7 Feedback

Your feedback on all aspects of this new course is welcome. Feedback given *early* is more valuable than feedback delivered through the end-of-term questionnaires. Early feedback allows any problems arising to be fixed immediately rather than next year!

Dr. Hrvoje Jasak

Department of Physics, The Cavendish Laboratory

Maxwell Building 3.98

E-mail: hj348@cam.ac.uk

2 Installing Software

This section provides instructions for installation of software you need to run the course. All programming assignments can be completed using `python`: a modern, expressive, efficient and versatile interpreted programming language.

At the very least, you will need the following:

- a Terminal application, used for command-line input;
- `python`. The course examples use `python` version 3 (and newer), but with minor syntax changes you will be able to achieve the same with `python` version 2.

You are **strongly discouraged** to use `python` version 2 in your work. It is obsolete and being phased out;

- A text editor for source code. Note for Windows users: Microsoft Word won't do as it does not manage unformatted text properly.

The above, with some additional `python` packages that can be added as needed is the basis of your software development environment.

To proceed, please review the section related to the Operating System (OS) you intend to use.

- Students working on an Apple Mac should read Section [2.2](#);
- Students working on a Windows PC should read Section [2.3](#);
- Students Working on a Linux system should read Section [2.4](#).

2.1 Installing and Managing `python`

`python` comes pre-installed on Mac OS X so it is easy to start using it. This is usually `python-2.7`, which is incompatible with the new release series of `python3`. However, to take advantage of the latest versions of Python, you will need to download and install newer versions alongside the system one.

Linux operating system also comes with pre-installed `python`, as parts of the operating system depend on it. This is most likely `python-2.7` again, so we will install the newer version side-by-side.

On Windows, you will typically need to install your own `python` version.

2.1.1 Pip

`pip` is the standard `python` package manager. It manages the installation of `python` packages that are not part of the standard `python` distribution. Each `python` file of source code is called a **Module** and can be imported into another `python` source code file or process. A collection of modules is called a **Package**.

The following instructions are written with assumption that `pip` is already installed. To check that write:

```
pip --version
```

into the terminal. This will return something similar to:

```
pip 21.3 from \path{/home/user/lib/python3.8/site-packages/pip}
(python 3.8)
```

To see a detailed list of commands available with `pip`, type

```
pip help
```

into the terminal.

For presentation purposes let us focus on a single `python` package `matplotlib`. This is a `python` package for creating visualisations. If you try to import a missing module or a package into your `python` code, `python` will return an error similar to

```
>>> import matplotlib
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'matplotlib'
```

This error is easily fixable with Pip package installation. Type:

```
pip install matplotlib
```

or (depending on the version)

```
python -m pip install matplotlib
```

or

```
python3 -m pip install matplotlib
```

into the terminal. The installation steps should look similar to Fig. 1. This could vary depending on the previously installed package dependencies on your computer.

To uninstall packages with Pip type

```
pip uninstall matplotlib
```

into the terminal. The uninstall steps should look similar to Fig. 2.

To see the list of `python` packages installed on your PC type

```

~/home$ pip install matplotlib
Defaulting to user installation because normal site-packages is not writeable
Collecting matplotlib
  Using cached matplotlib-3.5.1-cp38-cp38-manylinux_2_5_x86_64.manylinux1_x86_64.whl (11.3 MB)
Requirement already satisfied: python-dateutil<2.7 in ./horvat/.local/lib/python3.8/site-packages (from matplotlib) (2.8.1)
Requirement already satisfied: packaging<20.0 in ./horvat/.local/lib/python3.8/site-packages (from matplotlib) (21.3)
Requirement already satisfied: numpy>=1.17 in ./horvat/.local/lib/python3.8/site-packages (from matplotlib) (1.20.3)
Requirement already satisfied: cycler<0.10 in ./horvat/.local/lib/python3.8/site-packages (from matplotlib) (0.10.0)
Requirement already satisfied: kiwisolver<=1.0.1 in ./horvat/.local/lib/python3.8/site-packages (from matplotlib) (1.3.1)
Requirement already satisfied: pyparsing<=2.2.1 in ./horvat/.local/lib/python3.8/site-packages (from matplotlib) (2.4.7)
Requirement already satisfied: pillow<=6.2.0 in ./horvat/.local/lib/python3.8/site-packages (from matplotlib) (8.2.0)
Requirement already satisfied: fonttools<=4.22.0 in ./horvat/.local/lib/python3.8/site-packages (from matplotlib) (4.28.5)
Requirement already satisfied: six in /usr/lib/python3/dist-packages (from cycler<=0.10->matplotlib) (1.11.0)
Installing collected packages: matplotlib
Successfully installed matplotlib-3.5.1

```

Figure 1: Pip installation of matplotlib package.

```

~/home$ pip uninstall matplotlib
Found existing installation: matplotlib 3.5.1
Uninstalling matplotlib-3.5.1:
  Would remove:
    /home/./local/lib/python3.8/site-packages/matplotlib-3.5.1-py3.8-nspkg.pth
    /home/./local/lib/python3.8/site-packages/matplotlib-3.5.1.dist-info/*
    /home/./local/lib/python3.8/site-packages/matplotlib/*
    /home/./local/lib/python3.8/site-packages/mpl_toolkits/axes_grid/*
    /home/./local/lib/python3.8/site-packages/mpl_toolkits/axes_grid1/*
    /home/./local/lib/python3.8/site-packages/mpl_toolkits/axisartist/*
    /home/./local/lib/python3.8/site-packages/mpl_toolkits/mplot3d/*
    /home/./local/lib/python3.8/site-packages/mpl_toolkits/tests/*
    /home/./local/lib/python3.8/site-packages/pylab.py
Proceed (Y/n)? y
Successfully uninstalled matplotlib-3.5.1

```

Figure 2: Pip uninstall of matplotlib package.

`pip freeze`

into the terminal.

`pip` will allow you to quickly and easily manage other people's work in `python`. As there are many Python users world wide who contribute to Open Source eco-system, in many cases you will find useful `python` software developed by others, which does "just what you need!" Keep looking for it and using it!

2.1.2 python Versions and Compatibility

We have mentioned a preference to `python` version 3 over the older version 2. In the `python` community, this has developed almost into a religious war, although the differences in syntax are quite small. Two of them are mentioned below, in case you need to adapt existing `python3` code for an earlier version.

First, the format of the print statement has changed.

```

prompt> python
Python 2.7.18 (default, Mar  8 2021, 13:02:45)
[GCC 9.3.0] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>> print "Hello World!"
Hello World!
>>> quit()

```

```
prompt> python3
Python 3.8.10 (default, Nov 26 2021, 20:14:08)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>> print("Hello World!")
Hello World!
>>> quit()
```

Secondly, rules for dividing integers have changed: `python3` will perform a floating point division and `python` version 2 will produce an integer answer.

```
prompt> python
Python 2.7.18 (default, Mar  8 2021, 13:02:45)
[GCC 9.3.0] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>> 5/3
1
>>> quit()
```

```
prompt> python3
Python 3.8.10 (default, Nov 26 2021, 20:14:08)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>> 5/3
1.6666666666666667
>>> quit()
```

All other differences have much less impact; with the two changes above, you will be able to make most `python3` programs run in version 2.

2.2 Apple Mac (MacOS-X)

2.2.1 Terminal

A terminal application is available by default, shown in Fig. 3.

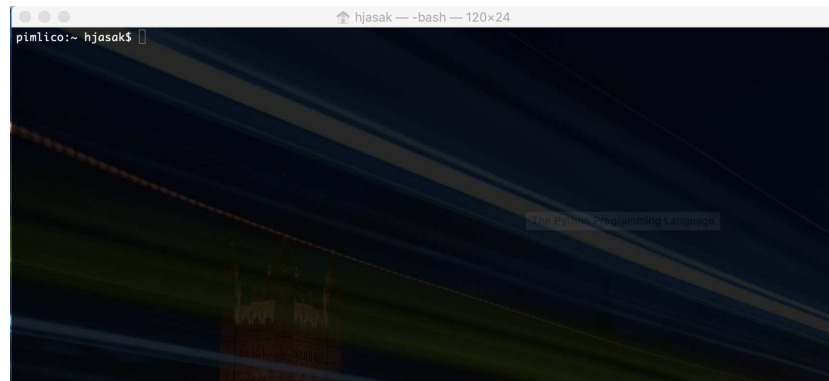


Figure 3: Mac OS-X Terminal.

To open a **Terminal**, do one of the following:

- Click the Launchpad icon in the Dock, type **Terminal** in the search field, then click **Terminal**;
- In the Finder, open the **/Applications/Utilities** folder, then double-click **Terminal**.

To Quit a **Terminal**, type **Ctrl-D** in the command line, or choose **Terminal > Quit Terminal** in the Menu.

2.2.2 Python 3

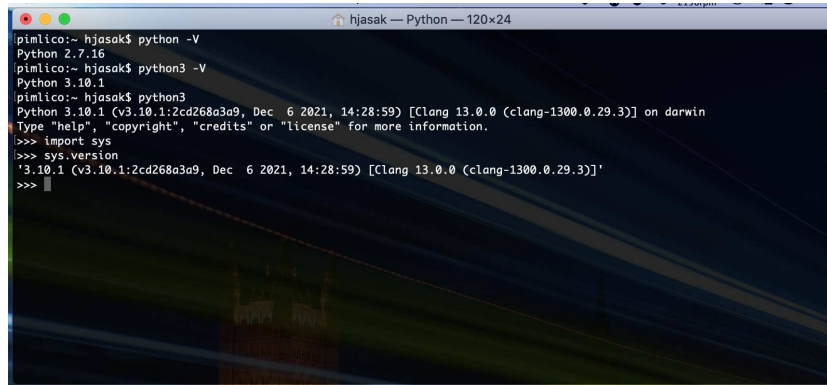
Mac-OSX since version 10.8 comes with **python 2.7** pre-installed by Apple. To check this, please open a terminal and type **python** and **python3**, Fig. 4:

You can check the version, either when starting python in the terminal:

```
prompt> python -V
Python 2.7.18
prompt> python --version
Python 2.7.18
```

or, from the **python** command line:

```
>>> import sys
>>> sys.version
'2.7.18 (default, Mar  8 2021, 13:02:45) \n[GCC 9.3.0]'
```

A terminal window titled 'hjasak — Python — 120x24' showing the following commands and output:

```
pimlico:~ hjasak$ python -V
Python 2.7.16
pimlico:~ hjasak$ python3 -V
Python 3.10.1
pimlico:~ hjasak$ python3
Python 3.10.1 (v3.10.1:2cd268a3a9, Dec 6 2021, 14:28:59) [Clang 13.0.0 (clang-1300.0.29.3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys.version
'3.10.1 (v3.10.1:2cd268a3a9, Dec 6 2021, 14:28:59) [Clang 13.0.0 (clang-1300.0.29.3)]'
>>>
```

Figure 4: Python version check on Mac-OSX.

If the above reports `python` version starting with “3”, no further action is needed. You can also check whether both versions are present in the system, the `python` command you wish to use is

```
python3
```

Thus, from the terminal:

```
prompt> python3 --version
Python 3.8.10
```

or

```
prompt> python3
Python 3.8.10 (default, Nov 26 2021, 20:14:08)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>> import sys
>>> sys.version
'3.8.10 (default, Nov 26 2021, 20:14:08) \n[GCC 9.3.0]'
>>> quit()
```

For ease of use, you should aim to run this course using **python version 3**. You can install the most recent version of `python3 3` from the python website:

<https://www.python.org>

To install it, go to:

<https://www.python.org/downloads/macos/>

download and install the latest installer. Some screen-shots of the process are given in Fig. 5.

Follow the on-screen installation process and on completion, execute:

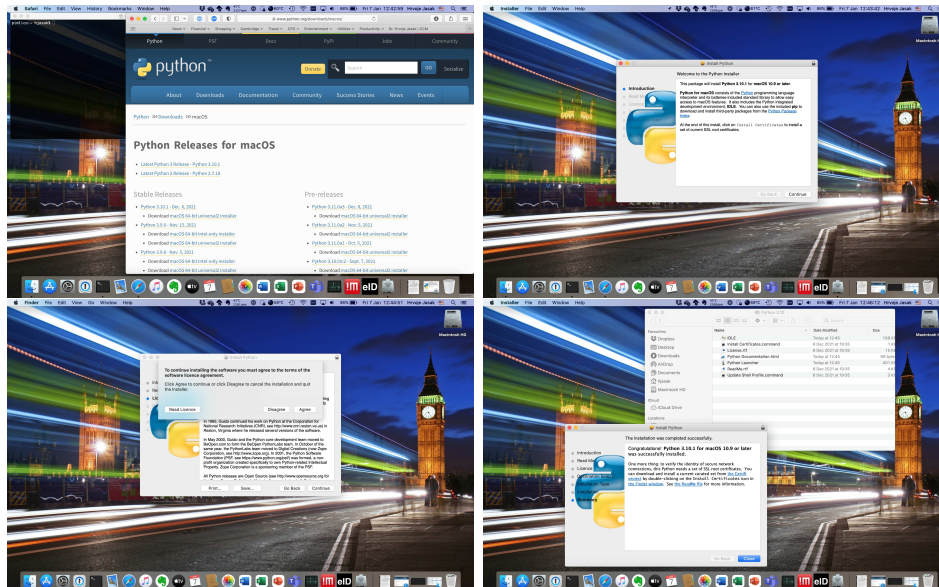


Figure 5: Download and install python3 on Mac-OSX.

```
Install Certificates.command
Update Shell Profile.command
```

as instructed. To test things, open a **Terminal** and type

```
python3 -V
Python 3.10.1
```

2.2.3 Text Editor

For the text editor on a Mac, you can either use **TextEdit** or install a version of **Emacs** or other add-on tools you may like to use. If all else fails, **vi** or **vim** text editor run from the command-line is always available to you.

2.3 Microsoft Windows

2.3.1 Terminal

Microsoft Windows 10 comes with a very useful terminal application: **Windows PowerShell**. **PowerShell** is a modern command shell that includes the best features of other popular shells. Unlike most shells that only accept and return text, **PowerShell** accepts and returns **.NET** objects.

To run it, from the **Start Menu** type **PowerShell**, and then click **Windows PowerShell**. You may wish to create an appropriate shortcut.

If PowerShell is not available, please use the older (and less user-friendly) cmd shell.

2.3.2 Python 3

Unlike most Linux distributions, Windows does not come with the `python` programming language by default. These instructions cover the installation of the latest `python3` on Windows 10. The procedure is similar for other versions. Please note: you can have multiple versions installed in the same time.

First determine if you already have `python` installed. Do this by opening cmd or Powershell and type:

```
python --version  
python3 --version
```

If the result is the `python` version 3 installed on your PC you can stop the process here (see Fig. 6). Otherwise, continue to the next step.

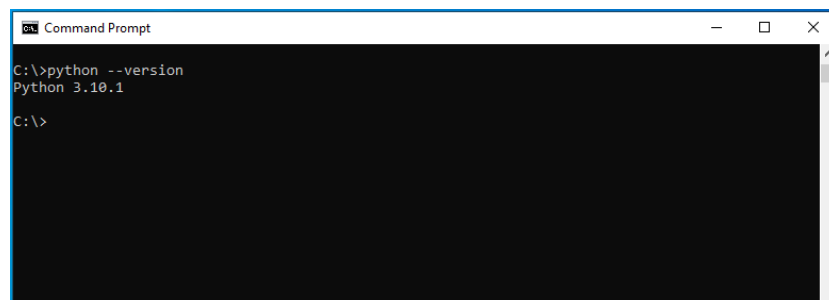


Figure 6: The response of the `python` version check on Windows.

Download the `python3` installer. The installer should be downloaded from the official `python` web-site:

<https://www.python.org/downloads/windows/>

This should present you with the menu shown in Fig. 7. The instruction will be continued assuming `python3.10.1` version.

Once downloaded, run the `python` installer. Make sure you select the following check-boxes:

- Install launcher for all users;
- Add `python3.10` to PATH.



Figure 7: The Python installer.

The latter places the interpreter into the execution path. If you skip it here, this will have to be done manually later.

Before closing the ‘Setup was successful’ window, make sure you click on “Disable path length limit” (see Fig. 8). Turning it on will resolve potential name length issues that may arise with `python` projects developed in Linux you might want to execute in Windows. This option will not affect any other system settings.

For all recent versions of `python`, recommended installation includes `pip`.

Finally, check that the installation was successful. Do this by following the same procedure as in the initial step. Open `cmd` or `Powershell` and type:

```
python --version
```

This should return 3.10.1 as shown in Fig. 6.

Additionally, check if `pip` is installed correctly. Do this by typing

```
pip --version
```

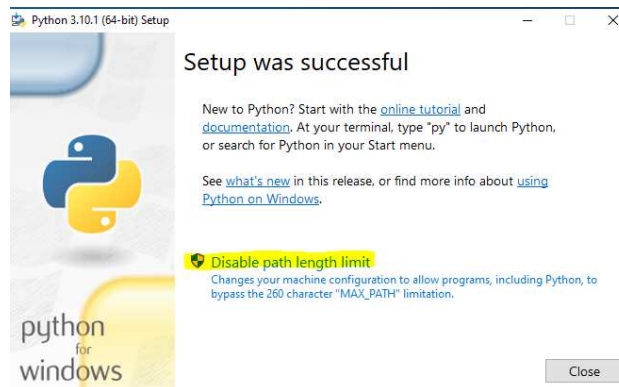


Figure 8: Successful python installation.

which should return something similar to

```
pip 21.2.4 from \path{C:\Users\user\AppData\Local\Programs\
Python\Python310\lib\site-packages\pip} (python 3.10)
```

as shown in Fig. 9.

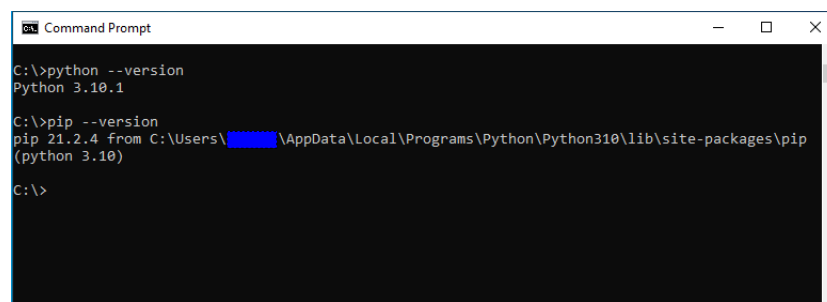


Figure 9: The response of the Pip version check.

You are now ready to use python3 on Windows 10.

2.3.3 Text Editor: Installing NotePad++

One would assume that Microsoft Windows provides a usable text editor that can understand and edit the source code. Regrettably, most Office tools are useless for this purpose: instead, we shall install NotePad++. It can be downloaded from:

<https://notepad-plus-plus.org/downloads/v7.9.2>

Please do not worry if your version is slightly different: for software under active development it is always best to use the latest version. The installation procedure follows the Windows installation pattern, Fig. 10.

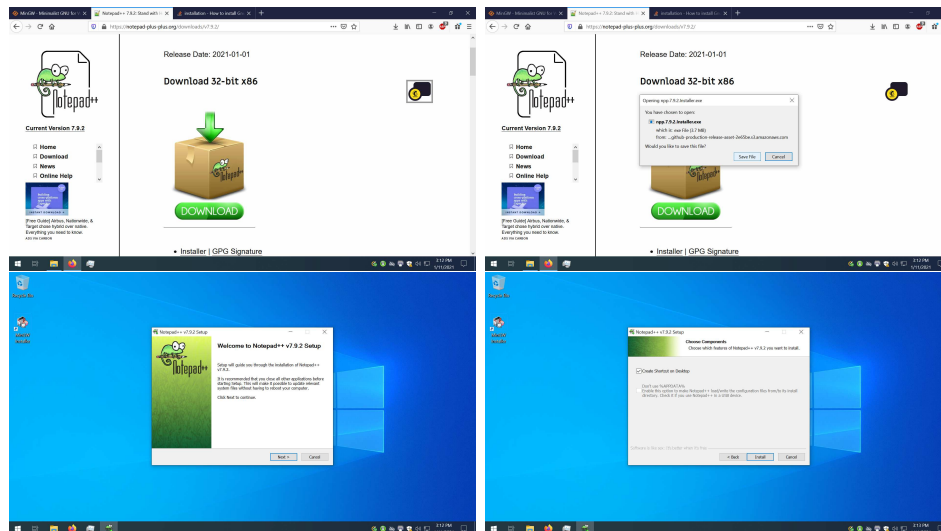


Figure 10: Download and install NodePad++ on Windows.

The easiest way to open files with NodePad++ is to drop them onto the NodePad++ icon; when you are finished with them, please close the files to avoid clutter.

This completes the environment you need – enjoy!

2.4 Linux

2.4.1 Terminal

Opening a terminal on Linux may be slightly different from distribution to distribution. To find a terminal, click the “Show Applications” button at the bottom-left corner of your screen, on the “dash” bar. Type “Terminal” and press Enter to find and launch the Terminal shortcut. You can also locate the Terminal icon in the list of all applications that appears here and click it.

If you are in trouble, most default distributions also include `xterm` application: it is always there!

2.4.2 Python 3

Many linux-based systems already have `python` installed by default, sometimes both `python 2.7` and `python 3`. If you open a terminal, you can check:

```
prompt> python
```

```
Python 2.7.18 (default, Mar  8 2021, 13:02:45)
[GCC 9.3.0] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>> quit()
prompt> python3 --version
Python 3.8.10 (default, Nov 26 2021, 20:14:08)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>> quit()
```

If the shell responds with `python3: Command not found`, or similar, you may need to install it.

The most popular systems (Ubuntu, Debian, Mint, Elementary OS, ...) use the `apt` command for package management, and so if you run one of those you will probably be able to install `python3` by opening a terminal, and entering the following command:

```
sudo apt install python3
```

[Note: in some older systems you may need to replace `apt` with `apt-get`.]

Other linux systems (such as Redhat or CentOS) use either the `yum` command (if they are older) or the `dnf` command (if they are newer). On those systems you will probably be able to install `python3` by opening a terminal, and running either:

```
# On CentOS/RHEL 7/6/5 and systems prior to 2019
sudo yum update
sudo yum install python3
```

or

```
# For Fedora 28/27/26/25/24/23 and systems after 2019
sudo dnf update
sudo dnf install python3
```

depending on the age of your system. If in doubt, try the newer command (`dnf`) before the older command (`yum`).

2.4.3 Text Editor

Text editors of Linux are plentiful: feel free to use any of them.

The Course

3 Programming a Computer

In this section we shall review some basic principles of working with computers.

3.1 What is a Computer Program?

Computers process data, perform computations, make decisions and instigate actions under the control of sets of instructions called **computer programs**. The computer (central processing unit, keyboard, screen, memory, disc) is commonly referred to as the **hardware** while the programs that run on the computer (including operating systems, word processors, and spreadsheets) are referred to as **software**.

Program instructions are stored and processed by the computer as a sequence of binary digits (i.e., 1-s and 0-s) called *machine code*. In the early days programmers wrote their instructions in strings of numbers called **machine language**. Although these machine instructions could be directly read and executed by the computer, they were too cumbersome for humans to read and write.

Later, *assemblers* were developed to map machine instructions to English-like abbreviations called mnemonics (or **assembly language**). In time, **high-level programming languages** (e.g. FORTRAN (1954), COBOL (1959), BASIC (1963) and PASCAL (1971)) were developed, which enabled people to work with something closer to the words and sentences of everyday language. The instructions written in the high-level language are automatically translated by a **compiler** (which is just another program) into machine instructions which can be executed by the computer later. The compiled program may be called a *binary* or an *executable*.

The word “program” is used to describe both the set of written instructions created by the programmer and also to describe the resulting piece of executable software. The former is usually referred to as the **source code**, which is then converted into an **executable**. With consistency, good practice and some luck (“Do operating system developers play nicely?”), source code is *portable* between different computing platforms, while the executable is built for particular hardware and operating system – using automatic tools.

Programmers consider the source code as an example of most densely packed information: unlike a novel which can be read and understood even with spelling errors or poor type-setting, every character matters. Note that the source code text is case-sensitive and sometimes (like in **python**) even indentation in the source code matters!

3.2 Software Development Process

In order to make the computer do what you wish it to do, clear instructions need to be provided. In modern programming languages, instructions are issued in a source code **text file**. The text of the source code is then translated into instructions that the hardware understands. This is done either in **compilation** or **interpretation** of the text of the source code. The result is executed on hardware and it produces the desired effect, or not, if you made mistakes in the source code!

The basic software process therefore consists of writing the source code, compiling or interpreting it, executing the instructions on hardware and analysing the results.

Often, things go wrong: the computer is not doing what we expect. In most cases, this involves a human error in writing the source code. Even if you think you have done things right, the computer does not seem to agree with you. As computers “do not have a mind of their own” and are good at following instructions, it is probably the programmer’s error. We refer to this as **a bug**: a programming defect that needs to be resolved in order to make the computer do what the programmer thinks it should be doing.

Unless you are a super-hero, bugs happen; they happen all the time. Therefore, we can expand the software development process into:

1. Writing or modifying source code;
2. Compiling or interpreting the instructions;
3. Executing the instruction set;
4. Analysing the results. Unless the result is what we expect, we will return to stage 1 and repeat until happy.

3.3 Compiled and Interpreted Languages

Several times above, I refer to compilation and interpretation as ways of converting the text of the source code into machine instructions of the executable. What is the difference? Clearly, text analysis and assembly of machine code will be done by the computer itself, using a different computer program. This is a **compiler**.

Compilers have many good properties, such as the ability to lexically and logically analyse the text of the source code and thus eliminate many typing errors, mis-spelled names *etc.* Only once everything is “in order”, *i.e.* the source code can be converted into machine instructions by obeying the rules of the programming language, as understood and enforced by the

compiler, an executable will be assembled. This is good: if the source code is meaningless, the compiler can find this out and warn the programmer without wasting further time. Compilers are also good for optimisation of the software (reshuffling your instructions) and can provide an environment to link a new piece of software with other software/utilities and programming environments. In fact, the Operating System actually only executes the programs called `main()` in the source code, thus providing a unique interface to the execution environment.

Compilers will work by looking at the source code in units (*source code files*), while a single executable may consist of many thousands (millions) of lines of individual source code. To facilitate integration, an intermediate level of compiled code exists: **object code**, usually packed and distributed as **shared object libraries**. For example, a programmer's life would be hard if you had to program a `sin(x)` function in source code every time you wish to use it. Instead, we shall recognise its presence, either at system level or via **library header files** and fetch its machine code from the relevant shared object library. This process is called **linking** of the object files to the `main()` function in one of them, creating a working executable.

Apart from their good sides, compilers also have deficiencies. The worst one is the lack of flexibility and the need to wait between the point a line of source code is written and the point the programmer can see its result. An alternative exists: an **interpreter**.

Instead of fetching the text of the source code in entire files, interpreters will look at them one line, or one source code structure (*i.e.* a loop) at a time, convert them into machine code instructions and executing them immediately. This gives a more interactive feel to the programming, with rapid gratification, but with many more errors. Unlike the compiled code, which cannot be executed until complete (in some way), programming interpreters takes one step at a time. It also allows the current state of the program – for example, value of a variable – to be inspected while the code is being written! Interaction with the software, in terms of providing inputs and examining outputs is also more immediate.

Interpreters, unlike compilers, do not need to interrupt the execution of the software when things go wrong. In case of error, interpreter will preserve the state of the program just before the error and signalise the error state. Compiled programs, on the other hand, have no option but to terminate the execution. Note that in case of interpreted languages, it is the interpreter itself that carries the `main()` function (interfacing with the operating system). It stays alive even if it has been fed “bad input”.

Down-sides of interpreters are many: they are slower, as a single line of the source code may be interpreted millions of times, they do not have scope

for optimisation and they are usually wasteful in memory usage (garbage collecting). Many of those are counter-balanced by the wonderful interactive feel in execution.

When working with interpreters, we can still choose to write source code in files and “feed them” to the interpreter in chunks. This allows the programmer more time to think and write and it feels like we are “compiling” or “executing” source code. This is fine: such source code is usually referred as a **script** and the interpreter still looks at it one line at a time.

Following the same rationale, programming languages are also separated into compiled and interpreted. Here are some examples, Table 3:

Compiled	Interpreted
FORTRAN, COBOL	BASIC, Lisp
Pascal, C	Perl, AWK
JAVA	JavaScript
C++, C#	python

Table 3: Compiled and interpreted programming languages.

There are of course many other programming languages too.

3.4 Programming Paradigms

Once can appreciate that programming is a skilled job and a lot of work has gone into making programmers’ lives easier. An important difference in the approach to programming is how we look at the problem at hand.

At a low level (or, if you are a computer), a program is divided into small parts called functions which operate on data. This is a fine and simple concept, called **procedural programming**. Functions, also known as routines, subroutines or procedures, consist of a series of computational steps to be carried out, with known input and output data. Generally, a function uses and modifies only a small portion of all available data, but this depends on the programmer. Functions are assumed to operate on data one-at-a-time until the desired result is reached.

Procedural programming works well as long as the programmer can keep track of all pieces of data and all functions they work with. This is all that a computer sees anyway: some data and some functions. However, as computer programs get more complex, it is hard for the humans to keep up: for this reason, it is useful to group “some data” and “some functions” together to make a unit that can do its job in isolation. Consider for example

programming a drone: the function I am ultimately interested in is to “make it fly!”, which will in turn consist of interaction between many systems within the drone: power, control, stabilisation, video, communication with operator *etc.* Each of those systems will involve data acquisition, decision making, change of state (thrust, orientation) and interaction with other systems.

It is easier for the programmer to “divide-and-conquer” such a complex system into logical bits and write the control programs for one sub-system at a time. Such an approach groups a set of data items and functions (think about the stability control system of my drone) into items of source code that work together and describe a single entity, called a **class**.

Object-oriented programming expects the code developer to recognise main objects from the system they are dealing with and describe them in software as a combination of data and functions. This data is protected from other parts of the software, making sure it does not get accidentally corrupted.

Let us take a look at the rotor thrust of my drone. Input data includes dozens of parameters, such as aspect, height, wind speed, mission, battery charge level *etc.*, and the output data I am interested in is the thrust setting of the engine. Looking at inputs and outputs is the procedural view of the problem. Conversely, object-oriented view tells me that it is the job of the power management sub-system, which needs to satisfy the requirements of the guidance system, which is in turn using the knowledge of the mission (my hand-held controller) and environmental conditions. This is the object-oriented view.

The computer does not see the difference: it only executes a set of functions on a set of data. Procedural programming or object orientation are simply human aspects that help (or hinder) my understanding of the programming task I am looking to achieve.

3.5 The C++ Programming Language

The C++ programming language (Stroustrup (1988)) evolved from C (Ritchie (1972)) by adding support for object-oriented view of programming. It is emerging as the standard in software development. For example, the *Unix* and *Windows* operating systems and applications are written in C and C++. It is an expressive, powerful and pretty complete example of a compiled object-oriented language.

3.6 Python

Recognising the power of object-orientation and interpreted execution, `python` is the leading interpreted programming language today. To quote its author:

Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.

Google search is mostly written in `python`, with some efficiency-critical parts done in C++.

The two languages seem to complement each other, with my preference being for C++ in large, complex, computationally intensive programming tasks and `python` for smaller and versatile daily use. As most of the work we will attempt in this course consists of several dozens of lines of source code, we shall concentrate on writing `python`.

3.7 In This Course ...

Compiled programming languages are older, more “traditional”, stricter and less flexible than `python`. For this reason, examples in this course and your programming work shall be done in `python` to help you develop transferable skills and save you from the “boring” parts of writing and debugging a compiled source code. Where needed, we shall refer to C++ examples of syntax, limitations, typing and development paradigm to contrast the ways of `python`.

4 Python for Absolute Beginners

`python` is an interpreted programming language. To start playing with it, open a Terminal and type

```
python
```

Please make sure you are running the right version of `python`: version 3. If there are problems, you can try

```
python3
```

If this is the right one, please replace your shell command with the same command throughout the entire course. In case of unexplained/unexplainable syntax errors, check the version of your interpreter.

If things are not working, please refer to Section 2 and come back when things are working.

Throughout this section, examples of little bits of `python` use will be given. As `python` is interpreted, feel free to play; if things go wrong, you can exit the `python` shell (`Ctrl-D`, or `Ctrl-Z` `Enter` on the Windows Power shell) and start it again.

Note: When you close the shell, the state of the program is lost!

It is a good idea to keep track of what you have typed in, for example in a text file. Copy-paste of the `python` prompt text, with some clean-up (such as removing the `python` prompt `>>>`) will allow you to save your work – and re-use it as needed.

Every time you get stuck in `python`, you can interrupt the process with `Ctrl-C`, or leave the program using `quit()`, or `Ctrl-D`.

Do Not Panic!

4.1 Need a Calculator?

Your command line tool (a shell) is not very versatile: it is designed to work with files and issue simple commands to the operating system ...and not much else. For some ideas of what it does, see Section B.

For simple mathematical operations, we can work interactively with `python`:

```

prompt> python
Python 3.8.10 (default, Nov 26 2021, 20:14:08)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>> 2 + 3
5
>>> 3/2
1.5

```

Recognise >>> as the `python` prompt; if you are in a shell and cannot see it, start `python` and continue with the example.

But:

```

>>> 3/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
>>> sin(1.5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'sin' is not defined
>> n
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined

```

We shall address and resolve some of those errors later.

4.2 Using Variables

We can also introduce some **variables**. Thus:

```

>> width = 3.2
>>> length = 1.8
>>> area = width * length
>>> area
5.760000000000001

```

Note that typing a variable's name prints out its value. Continuing the example, we can also make this pretty:

```

>>> print("Area = " + str(area) + " meters^2")
Area = 5.76 meters^2

```

Here, `str(area)` converts the numerical value of `area` into a string of characters, a variable type used, amongst other purposes, to for terminal output using the `print` command.

Also, note that commands are executed in the order they are issued and not “hard-wired” behind the scenes. Thus:

```
>>> area
5.7600000000000001
>>> width = 10
>>> area
5.7600000000000001
>>> area = width * length
>>> area
18.0
```

As you can see, updating `width` did not automatically update `area`: the command relating `area` to `width` needed to be re-executed to get the desired effect.

4.3 Hello, World!

`Hello, World!` is a 1980-s programming joke from the authors of the C Programming Language: every time you learn a new language, you should start by writing a code to produce this message on the screen. Let's do it in `python`.

`python` can either be used **interactively**:

```
>>> print("Hello, World!")
Hello, World!
```

or we can make it look like a program for execution. Thus:

- Open your favourite editor and type into a file:

```
1 print("Hello, World!")
```

Listing 1: `Hello, World!` in `python`.

- Save the file, giving it a name, *e.g.* `HelloWorld.py`;
- Open a shell in the place the file is saved (or find the file you saved in your filing system) and from the shell, execute

```
prompt> python HelloWorld.py
Hello, World!
```

In the programming world, the above command tells that `python` interpreter to read its commands from the file, rather than expect the programmer to type them in one at a time. The execution is still *interpreted*: `python` reads commands one-at-a-time and executes them.

Note that I didn't need a `quit()` at the end: `python` interpreter exited when it reached the end-of-file.

4.4 C++ and Hello, World!

For comparison, the `HelloWorld.cpp` program in C++ reads:

```
1 #include <iostream>
2
3 using namespace std;
4
5 int main()
6 {
7     cout << "Hello, world!" << endl;
8
9     return 0;
10 }
```

Listing 2: Hello, World! in C++

Having typed the text, you would then **compile** and **link** it to create an executable. Once this is completed, you would run the executable to produce the result.

You achieved all this in one line of `python`, without the need for compilation, linking *etc.* Simply speaking, interpreted source code is immediately ready for execution.

4.5 Source Code or Direct Interaction?

Typing into a `python` prompt and receiving immediate feed-back is immensely satisfying, but does not help with preserving your work for the future – or indeed submitting your assignments. It is therefore advised to do your “playing” interactively and then save the results in a file, which can be cleaned up, tested and modified in a more controlled manner.

4.6 Indentation is a Part of the Program

Apart from the simple commands we have seen so far, `python` provides some more complex forms. Consider the following (either typed into a file or directly to a prompt):

```
>>> theWorldIsFlat = True
>>> if theWorldIsFlat:
...     print("Be careful not to fall off!")
...
Be careful not to fall off!
```

The `if <condition>:` command controls the execution of the following line: the `print` statement will appear only if `condition` evaluates to `True`.

Please note the syntax: before `print...` is a Tab character (look at your keyboard; left towards the top).

Indentation is a part of python syntax. C++, in Section 4.4 achieves the same with curly brackets `{...}`, but for interpreted languages this would be impractical. Also, note that **it needs a Tab**): without it, indentation error will be reported:

```
>>> theWorldIsFlat = True
>>> if theWorldIsFlat:
...   print("Be careful not to fall off!")
File "<stdin>", line 2
    print("Be careful not to fall off!")
    ^
IndentationError: expected an indented block
```


5 What Has python Ever Done For Me?

So far, `python` looks like a basic calculator with some programming support: perhaps not enough for an average user to commit his time. This is true: no matter how impressive its set of features may be, it is still quite far from the idea of a “programming Swiss Army Knife” that its reputation implies.

There are three fundamental strengths of the language. The first is the fact that it is simple to use, yet it truly is a programming language with all the structure of variables, execution control, functions, classes and interfaces this implies. `python` also provides native high-level data types (such as `strings` and containers which are versatile and expressive. It provides extensive checking (by necessity – at run-time) and interfaces with other items of computer infrastructure such as a filing system.

The second convincing feature of `python` is that its programs can be split into **modules** (single files of source code) and **packages** (selections of interacting modules). The most compelling feature of modules is that a programmer can use modules written by other people, providing interfaces to file systems, graphing or visualisation.

Finally, `python` is **extensible**: it speaks to other parts of the computing environment. Your favourite program written in C++ can be included as a module and provided with a `python` interface, facilitating interaction with external libraries and helping with data manipulation.

The effort of writing `python` code is so small (compared to some other programming languages) that you can afford to write small snippets of code to solve the “problem at hand”, such as string manipulation or clearing your directory structure and file them for future use. If a program proves useful, it can be easily extended, shared or incorporated into larger tasks.

Have you ever found yourself in a position where you are solving a nasty computing problem and thinking “Someone must have already done this...”. Reusing other people’s effort in form of `python` modules and libraries becomes second nature. A quick internet search or scan for existing `python` extensions is likely to yield amazing results and save ample time via code reuse.

A combination of the above makes a compelling case of using `python` for your daily work. To illustrate some of this, let us review a part of this promise.

5.1 Module and Package

Writing `python` code in a file – which we refer to as **script**, usually with a `.py` extension – naturally creates the smallest ‘unit of source code. When this file is brought into the interpreter, it is referred to as a **module**. A

well-written module should contain a group of variables, functions, classes *etc.* written to perform a function – for example mathematical operations.

A collection of modules that work together create a **package**. Packages are developed as directories containing module files, with an additional file `__init__.py` that declares them as a unit.

Modules and packages can be a part of the `python` release – **built-in modules** – which are shipped with the installation, or **extension modules**, developed independently from core `python` and extending its capability in a user-defined way. Note that `python` allows its modules to be written in languages other than `python`, such as FORTRAN or C++ (if you are so inclined) and used from the interpreter as if they are native commands.

5.2 Built-In Modules

Built-in modules are a part of the `python` release. They provide capability fundamental for `python` itself. Here are some examples:

- `math` module, providing basic mathematical operations;
- `os` module, with the interface to the operating system;
- `random`, containing a pseudo-random number generator;
- `sys`, containing basic system-specific parameters and functions;
- ... and many others.

For a full list, see `python` documentation:

<https://docs.python.org/3/py-modindex.html>

or similar sources of information.

To import a module, use the keyword `import`. Functions available within a module are prefixed with the module name:

```
>>> import math
>>> x = 3.15
>> math.sin(x)
-0.008407247367148618
>>> x = math.pi
>>> x
3.141592653589793
```

This can become tedious, giving rise to different forms of `import` that avoids complications with names

```
>>> from math import sin, cos, pi
>>> x = 2*pi
>>> cos(x)
1.0
```

or indeed

```
>>> from math import *
```

Built-in modules provide functionality which is considered fundamental to the use of `python`.

5.3 Extension Modules

In the sea of problems tackled by `python` there are some true gems of development. One can of course expect a (good) programming language to provide functions such as `sin(x)` or `cos(x)`, but what about, for example Fast Fourier Transforms, or data visualisation libraries? With `python`, it is almost certain that pre-implemented, validated, versatile, efficient, documented, and ready-to-use packages already exist. Here are a few examples:

- Scientific computing in `python` with `numpy`;
- Fast Fourier Transform in `scipy.fft`;
- Visualisation in `python` with `matplotlib`;
- `pandas` for data analysis and manipulation;
- ...and many others.

Some of those may come with your `python` installation; try, for example:

```
>>> import numpy
```

If the `import` command fails, additional packages can be installed using `pip`, see Section [2.1.1](#).

6 A Simple Adder

The following is an example of a simple program that adds two numbers typed by a user at the keyboard, compute their sum and print the result. We shall look at it in detail.

6.1 Execute the Program

The program is written as a `python` script, stored in a file `SimpleAdder.py`.

```
1 # SimpleAdder.py
2 #
3 # A script that adds two numbers
4
5 a = input("Enter the first number: ")
6 b = input("Enter the second number: ")
7
8 total = int(a) + int(b)
9
10 print(f"The sum of {a} and {b} is {total}")
```

To execute it, type:

```
python SimpleAdder.py
```

and follow the prompts.

6.2 What's in the File?

Let us now look at the listing line-by-line:

Line 1-3 Starting with `#`, this line is a comment. It is there for human use and does not get parsed (analysed) by `python`.

Line 5-6 We define a variable `a` (just like in the calculator, Section 4.1, but indicate that the value of the variable should be presented by user input. The text prompting the user to provide input is listed in the brackets.

At this stage, the **variable type** is unknown. A (naughty) user could type *e.g.* `"chair"`, which will have consequences. We shall explore them later.

Line 8 We create a new variable, called `total`. It is a sum of two integers, indicated by type check/conversion of `a` and `b`.

Line 10 Write out the result, in formatted text. Note the `print(f...)` at the start of the string. This indicates a specific way of formatting the output, where the values of `a`, `b` and `total` are referred to in curly brackets.

Blank lines have been introduced to make the program more readable, but they may also play a role in code structure, such as completion of functions or block of execution code

Indentation In `python`, indentation matters. We shall revisit this later.

Other items of source code, such as package import, definition of functions, classes *etc.* also exist in `python` code – but not here! Our example is really simple and very far from everything the language can throw at you. More on this later.

When you are writing your code, try to make it pretty: you are likely to work with it for some time, and all distractions matter. Variable names need to remind you of what you are doing: be creative! If you do not like your lines too long, you can split them using a backslash `\` character, like so:

```
>>> print("This line is longer \textbackslash  
... than I care to tolerate")  
This line is longer than I care to tolerate
```

Usually, we wrap lines so that they don't exceed 79 characters.

When writing scripts, use blank lines to separate parts of functionality. Most decent text editors offer the user the ability to choose what the Tab button on the keyboard does to their code. Ideally, an indentation block (Tab) should be represented by 4 space characters. This makes the code more easily shareable among users with different setups/text editors.

6.3 Development Cycle

Unlike software written in compiled programming languages such as C++, interpreted code in `python` is ready for execution as it is written: literally line-by-line. This, however, does not exclude errors in the software.

One of most useful roles of the compiler is to check the syntax and (parts of) logic of execution of your software. The compiler can do this because it looks at the code in units and follows the names of variables, functions, classes *etc.*, to establish consistency. `python`, in contrast, sees the code one line/statement at a time. If you misspell a function or a variable name, you will simply get another variable and spend time looking for the error.

Having errors in the code is completely normal: you are not a super-person to be able to type for half an hour without a single error.

The process of analysing and fixing the errors (politely: code defects), or in slang *bugs* is known as **debugging**. Since the compiler is not there to help you, you will spend significant time debugging your `python` code.

Taking this in mind, a typical development cycle of software consists of several phases:

1. **Writing the code:** typing the characters of the source code using a text editor or directly into the `python` interpreter. Prefer the former, as it keeps the record of your actions; then copy-paste the code into the interpreter if you wish to execute it from the command line or simply run the script using `python script.py`;
2. **Running the code:** by executing it either line-by-line directly through the terminal or as a script;
3. **Analysing the results:** looking at the code input and output, confirm that the software is doing what is intended. If all is well, your work is done;
4. **Debugging:** if, as is often the case, the code does not work as intended, source code will be modified and re-executed until desired behaviour is achieved. This involves looking at errors and the source code and figuring out what went wrong.

If you are successful, you can take further steps to make code ready to be used again or shared with others: adding documentation (what the code is doing, how and why; what are the limitations *etc.*), packaging it and sharing with others.

7 Session 1: Simple Adder

7.1 Objectives

This is what you can expect from Session 1:

- Familiarisation with the shell-based development environment;
- Write and execute a working program;
- Become familiar with error messages.

7.2 Getting Started

In the first session, we shall familiarise ourselves with a command-line environment and perform some basic operations with `python`. Parts of the session will be given to you verbatim, but it is expected that you deviate from instructions and try some things on your own. Feel free to record some of your “experiments” and submit them together with the rest of your self-assessed work.

We recommend what can at first appear to be an “old-fashioned” style of computer programming, involving a lot of “typing commands in a terminal window”. There are many other ways of learning to program, some of which are particularly handy if you want to design programs with glossy graphical user interface; however we want to expose you the bare bones of lower level programming in this course.

This style of working naturally leads into scripting and automation, and will provide you with skills you can use in many places. Though the terminal-based computing model may seem strange to someone only familiar with “point and click” computing, it is the model almost universally adopted for scientific programming requiring a high degree of automation. Most research computing done in The Cavendish or in similar institutions would be done in the manner described.

7.3 Find Yourselfs a Computer

You are free to work on your own computer or use one of the machines provided in the Lab. Log into the computer and locate a Terminal and an Editor. Make sure that `python` works from the Terminal. Make sure you are running `python` version 3.

The place in which you type commands that the computer will then execute is called a “Terminal”. We will need to create (open) a terminal window so that we have somewhere to type out commands! Let’s do it.

7.4 Some Terminal-Based Commands

It is recommended that you use a UNIX-type terminal. If you are using Microsoft Windows, please try **PowerShell** in preference to **cmd**.

Once your terminal has opened you can run UNIX commands it. For example, you can type:

```
whoami
```

in the terminal, and when you hit the **ENTER** key it should tell you the user-id you used to log in, or you can type

```
cal
```

and after you hit **ENTER** you should see a calendar for the current month. Note that commands take optional “arguments” (*i.e.* extra words after the command) that change what they do. For example, typing:

```
cal 1976
```

will show you a calendar for the year 1976 instead of for the current month. To get help on a command (*i.e.* find out what it can do) some people use google, but it is often faster and easier to use the **man** command (short for “manual” as in “handbook”). For example, to find out what the **cal** command can do, and what options it can be given, type:

```
man cal
```

Press **Page-Up** and **Page-Down** buttons on your keyboard (or perhaps **Shift-Page-Up**, **Shift-Page-Down**) to navigate the manual, and **q** to quit it.

7.5 Filing System and Directory Structure

A computer filing system (for the user) looks like a tree of **files** and **directories**. They are hierarchical, starting from the “root” directory (usually called **/** on UNIX, or by the name of a hard-disk “device” on Windows, such as **C:.**

A command you will use a lot is:

```
ls
```

which should show you the list of the names of files and directories for the place (directory) you are currently “in”.⁵

You might not have any files yet, but you will have some soon.

Also useful is

⁵What windows calls a “folder” UNIX tends to call a “directory”. Two different words, but the concept is identical. Note that slashes separating directories and files are forward slashes in UNIX **/like/this** whereas they are **\backwards\in\windows**.


```
pwd
```

which shows you which place in the directory hierarchy you are currently in. Remember, if none of this is familiar to you, stop this part of the session and go straight to Section **B** of the Appendix to get familiar with some basics UNIX commands. You can try typing all sorts of things in the terminal with little fear of doing serious damage.

Navigating the Directory Structure. From the directory, you can see the **local files** without a prefix, if you need to go down (deeper into) the directory structure, use

```
directoryName/secondDirectoryName/nameOfFile
```

to locate the file. If you need to go up (back towards your home directory) in the structure, use the `../` to identify “one directory up than contains the current directory. To to two levels up, it will be `../../` *etc.*

Deleting Stuff. The worst command you could type is perhaps:

```
rm -rf *
```

which would delete all the files in the current directory and any directories contained within it, or

```
rm *
```

which would delete the files in the current directory only. Losing things in a shell is not the same as in the visual environment: if you delete a file you wrote, it is gone! Not in the recycling bin: it is likely gone forever!

Be very careful with the `rm` command!!! So long as you don’t accidentally type either of those commands (or variants thereof), you should be able to experiment to your heart’s content.

7.6 A Text Editor

Now something more practical. Let’s try to start a plain text editor: the sort of thing we might use to type in and edit a computer program. We don’t want to use a word processor like Microsoft Word, since we are not interested in formatting, fonts and pictures, etc. Programs don’t have those. We would also like to use an editor that knows a little bit about computer programs so that it can help us (e.g. by highlighting/colouring the different kinds of programmings in differently to allow us to spot mistakes).

There are a huge number of different code-editing programs, each with different features, new ones are being created all the time, and no one can

agree which is the best. Here are some examples. Type their names into the terminal window, if they are installed on your system. See which you like best:

geany I had not heard of this one until Oct 2019, but it looks pretty friendly for new users, is present on the MCS computers, and has been recommended by a number of the 2019 demonstrators. It has plenty of useful features for code editing. If you are new, maybe try this one first.

jedit This was a favourite in 2017, and was found to be pretty friendly for new users. It probably doesn't have many high level features for experienced programmers, but it starts up quickly, and does have pretty code formatting.

gedit Easy to use, but seems very "heavy", takes a while to start, not overly specific to programming, but found on many machines. Many of the 2019 demonstrators regard it as a good alternative to **geany**.

kate

sublime Other alternatives worth trying if you don't like those above. Sublime can be heavily extended and customized by plugins. It allows executing **python**, and other scripts directly from the editor.

emacs This is one of the two big grand-daddy programs – beloved by some programmers and hated by others – it has been around for decades. Hugely configurable (but only in an obscure language called "lisp"), it does at least have menus and knows what a mouse is, but the menus are very quirky. Expect to use the control and alt keys a lot.

vi or **vim** Older even than **emacs**, but my personal favourite, this program is so old that it doesn't have menus, doesn't expect you to have a mouse, doesn't expect you to even have arrow keys, expects you to remember to type things like "[ESC]:q![ENTER]" just to quit it, or

```
\quoteText{[ESC]\%:shCathDoghg[ENTER]}
```

to do a search-and-replace from "Cat" to "Dog" everywhere in the document being edited.

vim is hugely powerful, fast, elegant, and does what you want (once you know how). Beginners should not try this program, except in a nuclear winter, when it is the only editor that will still work. Think of it if at any point in your programming career you have a huge (hundreds

of MB or several GB) ascii formatted file which you want to open and take a look at its contents. This will be much faster than with your standard text editor.

Microsoft Windows users should be happy using **NotePad++**, while on a Mac you should try **TextEdit**. To open an editor like **geany**, you would type

```
geany &
```

in a terminal.

Running in the Background. Just typing **geany** (without the **&** symbol at the end) would also start the editor, but then your terminal would remain “locked up” waiting for the editor to quit. In other words, you wouldn’t be able to start new commands in your terminal window until you had quit the editor. You can of course open another terminal and work from there, but this leads to confusion with which files and directories you are working with.

Putting the **&** at the end of any UNIX command makes that command run “in the background”. In effect, the **&** means “run the preceding command (in this case it is “start **geany**”) and then give me back the command prompt so that I can run more commands, or start more programs, while that program is still running”. In practice, the commands you are likely to want to run “in the background” are editors and other terminals.)

7.7 Computing Exercise

Your Assignment

With the aid of the instructions below, use an editor to type in the simple adder program **SimpleAdder.py**, Section 6 listed on page 44 and then and run it.

Then execute **SimpleAdder.py** with different numbers at the input and check that it adds up the numbers correctly.

What happens if you input two numbers with decimal places?

Navigate the Filing System. You’ll be creating many files. To keep things tidy, we suggest that you get into the habit of creating a folder or sub-folder for each topic you tackle. So start as you mean to go on – create a new directory (folder) called, say, **work**.

As an experienced programmer and in expectation of work-to-come, I would create a hierarchy of directories. For example, Fig. 11: where **/home/hjasak** is my home directory.

```
/home/hjasak/
├── work
│   ├── Electromagnetism
│   ├── IntroToComputing
│   │   ├── doc
│   │   ├── Session1-SimpleAdder
│   │   ├── Session3-MonteCarlo
│   │   └── Session4-Bisection
```

Figure 11: Directory structure.

Avoid **putting spaces into the file name**. Note that on UNIX systems your file and directory names will be case-sensitive; I cannot guarantee this for Windows or Mac.

Navigate to the correct place in this hierarchy and open a file.

Create and Edit a New File. Create a new file called `SimpleAdder.py` containing the listing from page 44. There are two ways you can do this. You can either (i) start an editor without any arguments (e.g. `emacs &`), then type in the text, then "Save As" `SimpleAdder.py`, or (ii) you can create an empty file of a given name with `touch SimpleAdder.py`, then open this in an editor (e.g. `emacs SimpleAdder.py &`), then edit it, and finally "Save" your changes as normal. Option (ii) is slightly better, as the editor then knows you are typing a program even before you save it, and so may be able to colour code your program sooner.

`.py` extension informs the editor that the file will be a **python** program (well, script). Avoid putting spaces into the file name as it complicates the way UNIX handles it.

Run Your Code. Try executing your code in two different ways:

1. Interactively: start `python` and copy-paste the code **one line at a time!** Note that the script will ask you to input `a` and `b` when required;
2. As a program: run

```
python SimpleAdder.py
```

When the program prompt appears, input the two integers to the program by typing in two numbers separated by spaces and followed by a return. The program should respond by printing out the sum of the two numbers.

Well done!

Modify Your Code. Modify the program in your editor to change its behaviour in some way. Try make it produce a wrong answer. For example, input `chair` for `a` and follow what happens.

Change the code such that it can add strings, *e.g.*

```
>>> a = "chair"
>>> b = "Table"
>>> total = a + b
>>> print(f"The sum of {a} and {b} is {total}")
The sum of chair and Table is chairTable
```

As you can see, with a small change the code will perform addition operation on strings. What did we have to change? Without knowing the details of string manipulation, could you have guessed what would happen.

Modification With a Condition. Let us have a small modification of the two concepts we have already seen: a conditional (`if`) and a print statement.

```
1 # PrintIfRight.py
2 # Write a polite message if the right number is provided
3
4 a = int(input("Please give me a number between 1 and 10: "))
5
6 if a == 7:
7     print("Well done: 7!")
8 elif 1 <= a <= 10:
9     print("Thank you. Better luck next time...")
10 else:
11     print("This is rude. Good bye.")
```

Listing 3: Guess a number.

Can you describe what this code does? Can you make it fail? Can you make it react to string `chair` instead?

Note how the logical statement controlling the execution reads `a == 7`? What happens if the double-equals is replaced by a single-equals? Refer to `python` documentation to understand logical statements.

Can you write a code to solve a quadratic equation? Does it work for a negative discriminant? What did you have to do to manage complex conjugate roots?

Try, for example `QuadraticEquation.py`.

```
1 # QuadraticEquation.py
2 #
3 # A script that solves a quadratic equation of the form
```

```

4 # a*x**2 + b*x + c = 0
5
6 print(
7     "This program will solve the quadratic equation of the
8     form \
9     a*x**2 + b*x + c = 0"
10 )
11 a = float(input("Enter a: "))
12 b = float(input("Enter b: "))
13 c = float(input("Enter c: "))
14
15 disc = b**2 - 4*a*c
16
17 root1 = (-b - disc**0.5)/(2*a)
18 root2 = (-b + disc**0.5)/(2*a)
19
20 if disc >= 0:
21     print("The equation has real roots")
22 else:
23     print("The equation has complex roots")
24
25 print(
26     f"Roots of the quadratic equation \
27     {a}*x**2 + {b}*x + {c} = 0 \
28     are {root1} and {root2}."
29 )

```

Listing 4: Quadratic equation.

7.8 Assessment

This course proceeds by self-assessment. Here is the part of SESSION 1 that you must submit for assessment. You should do this bit by yourself.

Assessment Assignment

Set yourself an **additional programming task** along the lines of the preceding tasks, **and solve it!** Be imaginative! Use variables, user input, the `if`-statement syntax and formatted output.

If you need something more complex, refer to `python` documentation resources.

See Section [A](#) in the Appendix for submission instructions.

Project Ideas. These are meant to inspire you or to save you if you are despairing. You are expected to pick and **modify** one.

- Guess a Disney Princess, given hair colour and favourite animal (input)
- Calculate number of days in your lifetime
- Simple polynomial solver: find zeros of a polynomial
- Convert height and weight into different units
- Mental maths test

End of Session 1

8 Working with Variables

In this section we will look at working with variables. In this, `python` is significantly different from some other languages, so perhaps it is best to start with some basic information.

For those who need to know more, refer to the concept of *strongly and weakly-typed languages* in computing literature.

8.1 Basic Data Types in Computing

In reality, computers only work with vary basic information: 1 and 0. This is useful when designing electric circuits, but if you wish to make your Tamagotchi pet smile, it is rather limited.

Early (compiled) programming languages develop some **basic (built-in) data types** that underline all computing and present a sufficiently usable basis. In reality, they are all represented by ones and zeros, but in consistent (standardised) ways.

Basic data types in computing are given in Table 4.

Type	Purpose
int	to store a positive or negative integer (whole) number
float	to store a real (floating point) number
bool	to store the logical values True , False
char/string	to store a set character (text) values (string)

Table 4: Primitive data types.

Some of the above are easy to understand and model in ones and zeros: for example, **True** = 1, **False** = 0 ; integers are simply described in base 2, character contain a version of UNICODE representation (a number for a letter/symbol). Others, such as **float** are harder and require their own standard for representation and manipulation (IEEE Standard for Floating-Point Arithmetic (IEEE 754)).

All in all, we know how do to this.

8.2 Variable Type and Declaration

Compiled (strongly typed) language use a concept of **variable declaration**: when a variable is created, its type is specified and cannot be changed. For example:


```
int a = 3;
float b = 7.5;
char c = "H";
```

Trying to violate this results in an error: `a = c;` simply does not work because `a` is an **int** and `c` is a **char**.

High abstraction languages allow the user to define their own types, such as vectors, matrices or *e.g.* telephone directories, as needed for their work. We refer to those as **user-defined types**, which are at the basis of object-oriented programming.

Some compiled languages such as (historic) C Programming Language require the programmer to declare all variables at the start of a logical block. This is bad practice: it is better to declare (= create) a variable just as we need them.

How long do variables live? It depends on the language: in C++, they will disappear when their *scope* is closed: the closed-curly-bracket of a function, class, if-statement or a code. The same is the case with **python**, with some modifications. There are four major types of variable scope and is the basis for the LEGB rule. LEGB stands for Local – Enclosing – Global – Built-in variables. More on this later.

8.3 Type Conversion

However, in some cases (even in strongly typed languages), this is cumbersome: for example, we can make a **float** from an **int** without loss of precision (data), while a conversion in the opposite direction results in data loss.

Weakly typed languages such as **python** recognise this as cumbersome. Instead of strong typing, where the variable type is known at its creation and remains unchanged, each variable in **python** carries its own type which can change during its lifetime, depending on what it does. For example (note the lack of type on creation):

```
# Let's do some integers
>>> a = 3
>>> b = 5
>>> c = a + b
>>> print(f"C = {c}")
C = 8
# Now, some string operations
>>> a = "chair"
>>> link = "And"
>>> b = "table"
>>> c = a + link + b
>>> print(f"C = {c}")
C = chairAndtable
```

As you can see, the type of `c` changes from **int** to **string**; yet the addition operation and printing works just fine.

However:

```
# Does not work: types do not match
>>> a = 3
>>> b = "table"
>>> c = a + b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

produces an error: it is impossible to add **int** and **string** types. You know why.

Type conversion can convert one type into another when needed. For example, when a string interpretation of an integer is needed, we simply convert it to python **str** (string) type:

```
>>> a = 3
>>> b = "-Week Period"
# Better: convert a to a string
>>> c = str(a) + b
>>> print(c)
3-Week Period
```

8.4 Duck Typing

In order to avoid the tedious work related to keeping track of data types and type conversion, python adopts a programming style called **duck-typing**:

If it looks like a duck and quacks like a duck, it must be a duck.

Or, at least, let's treat it as a duck!

This emphasises interfaces of types rather than enforcing type matching; for this reason `a + b` works for integers, strings or (user-defined) matrices, albeit with some execution overhead.

8.5 Arithmetic Expressions

Arithmetic expressions, such as `a + b` have already been encountered and are intuitively reasonable. Perhaps the only comment required here is that `c = a + b` **assigns** the value of `a + b` to `c` rather than checking them for equality.

For the choice of arithmetic operators and their precedence for various types, such as integers, floating point numbers, booleans, and strings,

please consult the `python` documentation. Precedence can be controlled using brackets, in the usual way.

Table 5 lists some basic operators:

Operator	Purpose
Arithmetic Operators	
+	addition
-	subtraction
*	multiplication
/	division
%	modulo division
**	power
//	floor division
Assignment Operators	
=	simple assignment
+=	increment assignment: $a += b \rightarrow a = a + b$
-=	decrement assignment: $a -= b \rightarrow a = a - b$
*=	multiplication assignment: $a *= b \rightarrow a = a * b$
/=	division assignment: $a /= b \rightarrow a = a / b$ <i>etc.</i>
Relational Operators	
==	equal
!=	not equal
>	greater than
<	less than
>=	greater than or equal
<=	less than or equal
Boolean Operators	
and	and
or	or
not	not

Table 5: Some primitive operators.

`python` also allows some interesting syntax, such as *multiple assignment*

```
a, b = b, a+b
```

It is used *e.g.* in calculation of a Fibonacci series, `Fibonacci.py`.

```
1 # Fibonacci.py
2 #
3 # Fibonacci series:
```

```

4 # the sum of two elements defines the next
5 #
6 a, b = 0, 1
7 while a < 100:
8     print(a)
9     a, b = b, a+b

```

Listing 5: Fibonacci sequence.

Can you figure out what it does? Why is multiple assignment useful here? Can you write the code without it?

8.6 Working with Floating Point Numbers

As mentioned above, working with integers on a computer is easy: we simply converted the base-10 natural numbers into base-2 without loss of accuracy. Adding another bit for the sign extends this to negative numbers. The only limit in representation is the number of binary digits we use: this defines the smallest and largest number we have available.

Floating point numbers are more complicated: according to the IEEE standard, they are described as a mantissa–exponent pair, where the “width” of the mantissa and the exponent use a given set of binary (or decimal) *integer* digits. In other words, we expect to describe a floating point number with a pair of integers (one for mantissa and another for exponent).

To put things simply, there isn’t enough integers for ALL real numbers: this is referred to as “loss of accuracy” in floating point number representation.

Why do you care? Because there are consequences to practical work. The most important is the fact that you should not compare floating point numbers (other than zeros) for equality. Here is an example (for now, ignore the first line):

```

>>> from math import sqrt
>>> if (sqrt(2)**2 == 2):
...     print("Hello")

```

returns nothing: $\sqrt{2}^2 \neq 2!!!$.

Instead, you should check if the difference is “small enough”:

```

>>> from math import sqrt, fabs
>>> a = sqrt(2)
>>> if fabs(a**2 - 2) < 1e-10:
...     print("Works this time")
...
Works this time

```

With integers, there are no such problems. Keep this in mind when debugging your code.

9 Basic Containers

Programs usually need to store and process a substantial number of items of data. The types of variable considered so far have been simple data types capable of holding a single value. The programming concept that holds lots of data together is usually referred as a **container**.

Containers are critical for the usability of programming languages and `python3` excels in versatility of its containers. Four built-in types exist in `python`: List, Tuple, Set, and Dictionary. Let us review their basic properties and usage.

9.1 Lists

A `List` is the most versatile container in `python`. It can be simply expressed as a list of comma-separated values contained between square brackets. For example:

```
>>> fibonacci = [1, 1, 2, 3, 5, 8, 13, 21]
>>> fibonacci
[1, 1, 2, 3, 5, 8, 13, 21]
```

Lists can be indexed, with index 0 corresponding to the first element. Their elements can be changed using direct access and they can be “sliced”. Lists can be appended and shortened at will. Their elements do not have to be of the same type. Moreover, lists allow duplicate elements. Some examples:

```
>>> fibonacci_list[3]
3
>>> fibonacci_list[3] = "chair"
>>> fibonacci_list
[1, 1, 2, "chair", 5, 8, 13, 21]
>>> fibonacci_list[2:4]
[2, "chair"]
>>> long_fibonacci_list = fibonacci_list + [34, 55, 89, 144]
>>> len(long_fibonacci_list)
12
>> long_fibonacci_list.remove("chair")
>>> long_fibonacci_list
[1, 1, 2, 5, 8, 13, 21, 34, 55, 89, 144]
```

More functions can be found in the documentation.

9.2 Tuples

A `Tuple` is a very similar container to lists with a difference of tuple being unchangeable. This means that it is impossible to change, add, or remove

elements after a tuple has been created. Accessing elements by indices, calculation of length, addition operator, and many other properties of tuples are exactly the same as of lists.

Examples of similarities to lists:

```
>>> fibonacci_tuple = (1, 1, 2, 3, 5, 8, 13)
>>> fibonacci_tuple[3]
3
fibonacci_tuple[2:4]
(2, 3)
>>> long_fibonacci_tuple = fibonacci_tuple + (34, 55, 89, 144, "chair")
>>> len(long_fibonacci_tuple)
12
>>> long_fibonacci_tuple
(1, 1, 2, 5, 8, 13, 21, 34, 55, 89, 144, "chair")
```

There are also some differences:

```
>>> fibonacci_tuple = (1, 1, 2, 3, 5, 8, 13)
>>> fibonacci_tuple[3] = 100
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> fibonacci_tuple.remove("13")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'remove'
>>> del fibonacci_tuple[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object doesn't support item deletion
```

More functions can be found in documentation.

9.3 Sets

A Set is an unordered collection without duplicate elements which is defined with comma-separated elements between curly brackets. The order of elements in a set is undefined as can be seen from the examples below. Also, if the same item is entered multiple times, only a single instance exists in the final set. The elements of a set cannot be accessed by passing an index since they are unordered and elements do not have indices.

```
>>> set1 = {1, 30, 20}
>>> set1
{1, 20, 30}
>>> set1 = {1, 20, 30, "chair"}
>>> set1
```

```

{"chair", 1, 20, 30}
>>> set1 = {1, 20, 30, "chair", "chair"}
>>> set1
{"chair", 1, 20, 30}

```

The major advantage of using sets, as opposed to lists or tuples, is that they have highly optimized method for checking whether a specific element is contained within them.

```

>>> "chair" in set1
True
>>> 123 in set1
False

```

Sets support mathematical operations like union, intersection, difference, and symmetric difference.

```

>>> set1 = set("abcd")
>>> set2 = set("cdef")
>>> set2
{"f", "c", "d", "e"}

# UNION: letters in set1 or set2
>>> set1 | set2
{"d", "e", "b", "a", "c", "f"}
# INTERSECTION: letters that are in both set1 and set2
>>> set1 & set2
{"c", "d"}
# DIFFERENCE: letters in set1, but not in set2
>>> set1 - set2
{"a", "b"}
# SYMMETRIC DIFFERENCE: letters in set1 or set2, but not both
>>> set1 ^ set2
{"e", "b", "a", "f"}

```

More functions can be found in documentation.

9.4 Dictionaries

A ‘Dictionary’ is a container addressed by keys for easy lookup. The data dictionaries hold can be considered as key-value pairs with the requirements that keys are unique. They are very useful for associative data, such as phone numbers. Keys can be any immutable type such as strings and numbers and they are separated from the corresponding values by a colon.

```

>>> phone_numbers = {"John": 123, "Sarah": 1530}
>>> phone_numbers
{"John": 123, "Sarah": 1530}

```


The main operations on a dictionary are storing values with some keys and extracting value given the key. It is also possible to delete a key:value pair with `del`. If a value is tried to be stored using a key that is already in use, the old value associated with that key is forgotten. It is an error to extract a value using a non-existent key.

```
>>> phone_numbers["the Queen"] = 1000
>>> phone_numbers
{"John": 123, "Sarah": 1530, "the Queen": 1000}
>>> phone_numbers["the Queen"]
1000
>>> phone_numbers["the Queen"] = "secret"
>>> phone_numbers
{"John": 123, "Sarah": 1530, "the Queen": "secret"}
>>> del phone_numbers["the Queen"]
>>> phone_numbers
{"John": 123, "Sarah": 1530}
>>> "the Queen" in phone_numbers
False
>>> "Sarah" in phone_numbers
True
>>> phone_numbers["Mark"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: "Mark"
```

More functions can be found in documentation.

9.5 Which Container to Use?

The “correct choice” of a container depends on what the programmer intends to do with the data, its origin and characteristics. For example, do you know how many items you need to store? Will they be analysed in order (one-at-a-time), or will they be used for look-up?

The guidance here is your experience: you need to know what characteristics various container types provide and choose the one matching your requirements. This comes with experience.

10 Basic Programming Structures

Computers are designed to perform large number of operations on a large amount of data. Depending on the desired outcome, the work-flow (sequence of operations) may change depending on internal parameters of the program.

In Section 9 we have seen how to store (large amounts) of data: using containers such as lists or dictionaries. Let us now look at how `python` controls the order of operations in execution. You will notice that in examples shown so far, such structures were already used and you were expected to understand them intuitively.

10.1 Control Statements

The basic flow control statement is an if-statement: if a condition is satisfied, execute the **body of the control statement**. Here is an example of the syntax:

```
1 # PrintIfRight.py
2 # Write a polite message if the right number is provided
3
4 a = int(input("Please give me a number between 1 and 10: "))
5
6 if a == 7:
7     print("Well done: 7!")
8 elif 1 <= a <= 10:
9     print("Thank you. Better luck next time...")
10 else:
11     print("This is rude. Good bye.")
```

Listing 6: Guess a number.

Keyword `if` is followed by a Boolean expression `a == 7` (see relational operators in Table 5), a colon character `:` and an **indented block of statements** to be executed if the Boolean expression evaluates to `True`. When you wish to end the block, remove the indentation.

You can also have the if-then-else-if-else types of structures, like in the example. If your Boolean expression is more complex, you can use multiple conditions and parentheses. For example, instead of

```
1 <= a <= 10
```

you can write

```
a > 0 and a <= 10
```

Let us work with some examples.

A College Donor. Assess the user's suitability as a major College donor. We will use the if-then-else control structure:

```
1 # CollegeDonor.py
2 #
3 # Program to assess the user's for suitability as a major college donor
4
5 # Find out how many houses are owned:
6 howManyHousesSheOwns = int(input("How many houses do you "
7                                "own? "))
8
9 # Give feedback
10 if howManyHousesSheOwns <= 0:
11     print("And have you come far today?")
12 elif howManyHousesSheOwns < 10:
13     print("Student hardship funds make a real difference to "
14           "many students lives!")
15 elif howManyHousesSheOwns < 100:
16     print("Have I told you about the business park the "
17           "college would like to build in the Cotswolds?")
18 else:
19     # I'm speechless!
20     print("Gasp!")
```

Listing 7: A College donor.

A Simple Calculator. Let us write a simple calculator using a switch-statement:

```
1 # Calculator.py
2 #
3 # A simple calculator
4
5 # input an operation
6 readOp = input("Type an operation with 3 elements, "
7               "eg. '2 + 3'. Please use spaces: ")
8
9 # split the input via spaces
10 parsed = readOp.split(" ")
11
12 if len(parsed) != 3:
13     print("Syntax Error. Terminating")
14     quit()
15
16 # Strip spaces
17 readOp.replace(" ", "")
18
19 # Let's try the syntax
20 a = float(parsed[0])
21 operation = str(parsed[1])
22 b = float(parsed[2])
23
24 # Analyse the syntax
25 if operation == "+":
26     result = a + b
27 elif operation == "-":
28     result = a - b
29 elif operation == "*":
30     result = a*b
31 elif operation == "/":
32     result = a/b
33 else:
34     print(f"Invalid operation: {operation}. Terminating.")
35
36 # Output the result
37 print(f"Result: {a} {operation} {b} = {result}.")
```

Listing 8: A simple calculator.

Can you make this more robust? Check for division by zero? Can you make it add until the new input is positive? (make negative number terminate the execution).

Can you make this work with complex numbers?

10.2 Loops

Apart from logically controlling code execution, we sometimes like to repeat the same block of code multiple times. Here is an example you have already seen in `Fibonacci.py`:

```
1 # Fibonacci.py
2 #
3 # Fibonacci series:
4 # the sum of two elements defines the next
5 #
6 a, b = 0, 1
7 while a < 100:
8     print(a)
9     a, b = b, a+b
```

Listing 9: Fibonacci sequence.

The `while` statement is again controlled by a Boolean expression: `a < 100` evaluates to `True` or `False`, which decides if the indented bloc of statement will be executed. As `a` increases in the loop, the code will eventually terminate.

You can also “escape” the loop under additional control conditions, using `break`. The next example terminates when `a` equals 21, `FibonacciWithBreak.py`

```
1 # FibonacciWithBreak.py
2 #
3 # Fibonacci series:
4 # the sum of two elements defines the next
5 # If you hit, 21, terminate
6 #
7 a, b = 0, 1
8 while a < 100:
9     print(a)
10    a, b = b, a+b
11    if a == 21:
12        print("Reached 21: break")
13        break
```

Listing 10: Fibonacci sequence with a break statement.

10.2.1 Repetition Control

We can use other form for repetition control. A good way to do this is to use a built-in function `range`, which returns a series of numbers:

```
1 # Range.py
2 #
3 # Loop over a range
```

```
4 for i in range(5):
5     print(i)
```

Listing 11: Range.

`range` generates sequences of numbers, starting from zero of the lower limit, but excluding the upper limit. The numbers are valid indices for items in *e.g.* containers. You can also provide a stride.

Let's take a look at some examples:

```
>>> range(5)
0, 1, 2, 3, 4
>>> range(5, 10)
5, 6, 7, 8, 9
>>> range(0, 15, 3)
0, 3, 6, 9, 12
```

10.2.2 Looping on Containers

When dealing with a large amount of data, it is typically stored in containers, Section 9. Looping on a container should therefore look pretty. Here are some examples:

```
1 # LoopOnContainer.py
2 #
3 # a Physics song
4
5 words = ["Newton", "had", "a", "little", "apple"]
6
7 # index looping: i will be an integer
8 for i in range(len(words)):
9     print(words[i])
10
11 # but also: i is a string
12 for i in words:
13     print(i)
```

Listing 12: Loop on a container.

There are of course many other ways of performing operations on containers. The best way to learn this is to do some examples yourselves.

10.2.3 Statistical Analysis

A combination of the tools shown so far can be used to demonstrate how powerful the language really is. Let us make a little example of statistical analysis.

Let us perform a calculation of mean and variance on a set of data (floating point numbers) provided by the user. To make life easier (for me), we will input data in a comma-delimited format:

```
1, 2, 4, 7, 11, 22, 37, 45
```

When typing numbers, delimit the entries with a comma (,).

Clearly, any other “user-defined” format can also be handled, but it will usually be more effort to massage the format than to calculate the statistics. Here is some code:

```
1 # Statistics.py
2 # Perform a statistical analysis on data in python-compatible format.
3
4 # Read some data as a string
5 inputData = input("Please provide data in python format, eg
6               1, 2, 3 : ")
7
8 # Split it at commas
9 inputData = inputData.split(",")
10
11 # Convert it to numbers
12 numberData = []
13
14 for s in inputData:
15     numberData.append(float(s))
16
17 # Calculate number of entries
18 n = len(numberData)
19
20 # Calculate the sum
21 sumData = sum(numberData)
22
23 # Calculate sum of squares
24 sumSquares = 0
25
26 for f in numberData:
27     sumSquares += f**2
28
29 # Calculate mean and variance
30 mean = sumData/n
31 variance = sumSquares/n - mean**2
32
33 # Report
34 print(f"Mean = {mean}, Variance = {variance}")
35 print("Good Bye")
```

Listing 13: Statistics: mean and variance.

As you can see, there was more trouble preparing a list of numbers than

actually calculating the mean and variance. Intuitively, operations like `sum` did the right thing on a list of numbers without me needing to look at documentation. In expressive languages such as `python` you can always try and **guess** the right syntax first and then, if it does not work, try to fix it.

11 Session 2: Computing

11.1 Objectives

By now you should feel able to:

- Create variables of various types;
- Assign values to variables and manipulate them in arithmetic expressions;
- Write the value of variables to the screen with some formatting;
- Read in the value of variables from the keyboard;
- Write simple programs that use loops and if-statements.

You will now use these skills to write and execute the program described in the following computing exercise, Section 11.2.

Some tips to help you with the exercise are given in Section 11.3

11.2 Computing Exercises

Here are some computing exercises for you.

Exercise 1: Simple Adder

Modify the `SimpleAdder.py` program from earlier so that it converts a physical quantity in one set of units to another set of units and displays the result as a floating point number. For example, read in a time in years, and display that time in seconds; or read in a power in kWh per day and display it in W.

Exercise 2: Sum of Odd Numbers

Write a program that uses a for-loop to sum the first N odd numbers, where N is a number supplied by the user. When it has computed the sum, your program should check that

$$\sum_{i=1}^N (2i - 1) = N^2$$

Exercise 3: A Nested Fraction

Write a program that uses a loop to print out the sequence of floating-point numbers:

$$1, 1 + \frac{1}{1}, 1 + \frac{1}{1 + \frac{1}{1}}, 1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1}}}, 1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1}}}}, \dots$$

Print out the first 20 terms. Try to adjust output formatting and the number of digits shown.

Exercise 4: Square Root Sequence

Write a program that uses a loop to print out the sequence of floating-point numbers:

$$\sqrt{1}, \sqrt{1 + \sqrt{1}}, \sqrt{1 + \sqrt{1 + \sqrt{1}}}, \sqrt{1 + \sqrt{1 + \sqrt{1 + \sqrt{1}}}}, \dots$$

What does this sequence tend to?

For the square root, you can use `x**-0.5`. There is also the square root function in `python`, living in the `math` package and called `math.sqrt()`. To tell the `python` that you want to use maths functions, at the start of your code add

```
import math
```

or, to lose the annoying prefix:

```
from math import sqrt
```

If you need more `math` functions, try importing everything

```
from math import *
```

More on packages later in the course.

11.3 Tips

1. **Start with a working program.** The easiest way to write programs is to modify the existing programs. Make small modifications and try

to run the code whenever possible. You can always resort to typing commands into the `python` prompt one at a time;

2. **Making modifications.** Mentally check that the program is doing what it is required to do. Check that your program is laid out correctly with indentation and structure. For example, check your semi-colons in if-commands and in loops. If you don't know what the variable holds, print it out.

Explain to yourself or your partner what you are trying to achieve; proceed strictly step-by-step, testing as often as possible. Save working versions of the code so that you can always revert to a valid check-point in your work if things go badly.

3. **Debugging tips.** Watch for syntax and think about types. When an error occurs **read the error message!**

If you are feeling stuck, a really good way to track down the cause of an error message is to explain to a friend, step by step, how you modified the code from the last reliable check-point and what happened. Try to separate the process one-change-at-a-time;

4. **Finding logical errors.** Run your program and check that it gives the correct answer. Remember: the computer is right, you are wrong. The poor computer is only doing what you told it to do.

Good Luck!

11.4 Assessment

Assessment Assignment

Set yourself an **additional programming task** along the lines of the preceding tasks, **and solve it!** Be imaginative! Submit your task statement and your solution in the usual manner.

See Section [A](#) in the Appendix for submission instructions.

Project Ideas. These are meant to inspire you or to save you if you are despairing. You are expected to pick and **modify** one.

- Convert decimal into binary
- Calculate prime numbers up to a limit
- Translate an English word into Morse code

End of Session 2

12 Functions, Modules, and Packages

Computer programs that solve real-world problems are usually much larger than the simple programs discussed so far.

In this section we will explore some tools that will allow you to organise your programming into manageable chunks. This will allow you to consider the programming task one item at a time and attack complex tasks by combining individual elements. A good practice would require you to develop, test, document and package such smaller chunks before deploying them to work together.

Until now you have encountered programs where all the code (statements) has been written inside a single “function”, usually referred to as the `main()` code. Every executable in a programming environment (UNIX, Windows) has at least this function. In `python`, the `main()` code is the `python` interpreter itself (managing the interface with the operating system), which allows you to simply start typing your program into the `python` command line.

12.1 Functions

Let us revisit the Fibonacci sequence from Section 10.2. Calculating a Fibonacci sequence is useful: how much of it would you like? This allows us to define a **function** that writes the Fibonacci sequence up to an arbitrary limit n . The n , provided to the function is called a **function argument** (or parameter). To start the work, we need to describe to the World what our function does and which arguments it takes.

12.1.1 A Simple Function

```
1 # FibonacciPrintFunction.py
2 #
3 # A function writes n elements of the Fibonacci series:
4 # the sum of two elements defines the next
5 #
6 def fib(n):    # write n elements of the Fibonacci series
7     """Write n elements of the Fibonacci series."""
8     a, b = 0, 1
9
10    for i in range(n):
11        print(a, end=" ")
12        a, b = b, a+b
13    print()
14
```

```

15
16 # Call the fib() function
17 fib(20)

```

Listing 14: Fibonacci print function.

Let us analyse this line-by-line:

- The **keyword** `def` indicates **function definition**: we are introducing a new “word” in the language of our program, which will do something new. This “word” is `fib`. The function takes one argument: `n`. When we define what the function does, we will refer to this `n`; with different `n`-s we can make the function do different things.
- Line 7 in the function defines the documentation string, or **docstring**. As other people may wish to use the function, it is polite to tell them what it is intended to do.

Once the function is defined, you will access its documentation with

```
help(fib)
```

If you just type `fib`, you will get some info on the name: it is a function.

```

>>> fib
<function fib at 0x154d9369bc10>

```

- Lines 8–13 are the **function body**: this is the work the function does, already familiar to you. When you type all this into **python** **nothing happens!** So far, we have **defined the function**, but did not use it.
- Line 17, actually calls the function, providing the argument: $n = 20$. Suddenly, things start happening: the function is executed.

The real use of the function is that you can execute it repeatedly, with different arguments, thus re-using the code written once.

12.1.2 Return Value

Printing things out out of a function is nice, but even for simple functions such as `sin` or `cos`, I would prefer to “receive” the value instead. Let’s try this instead:

```

1 # FibonacciFunction.py
2 #
3 # A function calculates and returns n-th elements of the Fibonacci series:
4 # the sum of two elements defines the next
5 #
6 def fib2(n):

```

```

7     """Return n-th elements of the Fibonacci series."""
8     a, b = 0, 1
9
10    if n < 0:
11        print(f"Error, negative index: {n}")
12        quit()
13    elif n == 0:
14        return a
15    else:
16        for i in range(n - 2):
17            a, b = b, a+b
18
19        return b
20
21
22 # Call the fib() function
23 result = fib2(20)
24 print(f"Fib(20) = {result}")

```

Listing 15: Fibonacci function.

This time, function `fib2` returns the value for further re-used, as was the case with *e.g.* `sqrt` function we have seen in the `math` package.

Functions can have multiple arguments, return one or many items, they can have named arguments, default argument values and many other options. For the moment, let's work with simple functions; if you need something more complex, please consult the documentation.

12.2 Modules

When `python` quits, its state will be lost: in practice, the functions you have so carefully defined will disappear. For practical work it is therefore better to prepare the input for the `python` interpreter as a text file and then “import it” in some way.

If your script gets longer, it may itself be split in several files. Soon, you will be using other people's code too, and some of your work may depend on it. To facilitate this, we can include multiple functions in a file and have them ready for import. But, there is a problem: what if another person, in another script, happened to use the same function name as I insist on using in my code? Are we now incompatible? Which function is used in preference?

To help with this, `python` introduces a concept of a **module**. A module is a python file containing definitions of one or many functions that may come useful. Let us build one with the Fibonacci functions above:

- Open a new file, `fibonacci.py`. The name of the file will also be the name of the module;
- Paste both functions we developed into it (and save);
- Start the interpreter and **import** the file `fibonacci.py`. This is similar to copying and pasting the source code into the interpreter; this time, the code “remembers” where it came from.

```

1 # A function writes n elements of the Fibonacci series:
2 # the sum of two elements defines the next
3 #
4 def fib(n): # write n elements of the Fibonacci series
5     """Write n elements of the Fibonacci series."""
6     a, b = 0, 1
7
8     for i in range(n):
9         print(a, end=" ")
10        a, b = b, a+b
11    print()
12
13
14 # A function calculates and returns n-th elements of the Fibonacci series:
15 # the sum of two elements defines the next
16 #
17 def fib2(n):
18     """Return n-th elements of the Fibonacci series."""
19     a, b = 0, 1
20
21     if n < 0:
22         print(f"Error, negative index: {n}.")
23         quit()
24     elif n == 0:
25         return a
26     elif n >= 1:
27         for i in range(n - 2):
28             a, b = b, a+b
29
30     return b

```

Listing 16: Fibonacci module.

Note that we are likely to have a problem if there are two functions with the same name in the file, as the second one would hide (over-write) the first.

With the package system, our functions have their names prefixed by the module name, thus avoiding clashes. One can also import single functions from modules or “drop” all module functions into the interpreter. Let’s look at all the options:


```

>>> import fibo
>>> fibo.fib(10)
0 1 1 2 3 5 8 13 21 34
>>> fibo.fib2(10)
34
>>> # Import a single function; prefix no longer needed
>>> from fibo import fib, fib2
>>> fib2(15)
377
>>> # Import all functions; prefix no longer needed
>>> from fibo import *
>>> fib2(25)
46368

```

You can find the contents of an imported module by using

```

>>> import fibo
>>> dir(fibo)

```

Apart from our functions, you will see some others, starting with a double underscore. This is referred to as *dunder*, or magic methods and they relate with how `python` does its job. Here is some dunder for the `fibo` module:

```

>>> import fibo
>>> dir(fibo)
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__',
 '__name__', '__package__', '__spec__', 'fib', 'fib2']

```

Additional problems may occur when working with lots of modules: where should `python` look for them? For the moment, assume it is enough to look in the current directory; for details, consult documentation.

You have already seen some built-in modules, such as `math`. Below is the list of some of the standard functions it provides, Table 6.

For further details, refer to the documentation.

12.3 Packages

As Life becomes more complicated, one may start working with many modules, some of which require other modules in turn. To help with organisation of software – and sanity of programmers, multiple modules are organised in **packages**.

One may think of packages as a hierarchical representation of groups of modules that can in turn use their initialisation dunder to make sure all the necessary support packages are also imported.

Function	Description
<code>sqrt(x)</code>	square root
<code>sin(x)</code>	trigonometric sine of x (in radians)
<code>cos(x)</code>	trigonometric cosine of x (in radians)
<code>tan(x)</code>	trigonometric tangent of x (in radians)
<code>exp(x)</code>	exponential function
<code>log(x)</code>	natural logarithm of x (base e)
<code>log10(x)</code>	logarithm of x to base 10
<code>fabs(x)</code>	absolute value (unsigned)
<code>ceil(x)</code>	rounds x up to nearest integer
<code>floor(x)</code>	rounds x down to nearest integer
<code>pow(x,y)</code>	x raised to power y

Table 6: Mathematical functions.

12.4 Some Examples

Factorial Function. Here is a factorial function. Be careful with indentation when typing the function!

```

1 # Factorial.py
2 #
3 # Factorial function
4
5 def Factorial(n):
6     """ Function to calculate factorial of a number """
7     if n == 0:
8         return 0
9     elif n > 20:
10        print(f"Large n, danger of overflow.  n = {n}")
11        return
12
13    result = 1
14
15    for i in range(1, n + 1):
16        result *= i
17
18    return result

```

Listing 17: Factorial Function.

and

```

>>> from Factorial import *
>>> Factorial(10)
362880

```

Binomial Coefficient. Using the above, calculate the binomial coefficient:

$$\frac{n!}{k!(n-k)!}$$

Note: this is not the best way to compute a binomial coefficient.

```
1 # Binomial.py
2 #
3 # Calculate the binomial coefficient
4 # (n over k) = n!/(k! (n - k)!)
5
6 from Factorial import Factorial
7
8
9 def Binomial(n, k):
10     """ Calculate the binomial coefficient B(n, k) = n!/(k! (
11     n - k)!) """
12     if k == 1 or n == k:
13         return 1
14     elif k > n:
15         return 0
16     else:
17         return Factorial(n)\
18             // Factorial(k)\
19             // Factorial(n - k)
```

Listing 18: Binomial coefficient.

Can you write a function to calculate the area of triangle given base and height? How about the area of a hexagon?

13 Session 3: Monte Carlo

13.1 Objectives

By now you should feel able to:

- Create variables of various types and perform arithmetic operations;
- Assign values to variables and manipulate them in arithmetic expressions;
- Write the value of variables to the screen with some formatting;
- Read in the value of variables from the keyboard;
- Write simple programs that use loops and if-statements;
- Understand functions, modules and packages;
- Use the `math` library.

We will now do some computing exercises and introduce a graphics package to look at our results. This, of course, involves using libraries written by others, in our case `pyplot`, a part of the `matplotlib` package.

To start, locate the documentation for `matplotlib` on the internet:

<https://matplotlib.org/>

Let us do some coding!

13.2 Simple Monte Carlo

Monte Carlo methods use random numbers to generate approximate answers to mathematical questions. A good Monte Carlo method has the property that the more random numbers are used, the more accurate the answer becomes.

The program `MonteCarlo.py` shows how to generate N pairs of (x, y) coordinates randomly distributed in the unit square, and test whether each pair is in the unit circle or not. For each point, we shall check whether it falls within a circle of unit diameter and count the number of points that fall within the circle. The ratio of points within the circle, compared to the total number of points will be used to calculate π .

Here is some code:

```

1 # Monte Carlo method to calculate pi
2 import math
3 import random
4 import matplotlib.pyplot as plt
5
6 N = 10000
7 print("Parameter: N =", str(N))
8 # Array of iterations.
9 iterations = []
10 # Array of results.
11 results = []
12
13 count_in = 0
14 for i in range(N):
15     # random.random returns a random double in range 0-1.
16     x = random.random()
17     y = random.random()
18     # Condition checks if the random point falls in the inscribed circle.
19     cond = x**2 + y**2
20     outcome = 1 if cond <= 1 else 0
21     count_in += outcome
22     # Percentage of how many times it did fall into the circle.
23     # Analytically, this equals  $A_c/A_s = \pi/4$ .
24     fraction_in = count_in/(i+1)
25
26     # Store the results into the array.
27     results.append(4.0 * fraction_in)
28     # Store iteration into the array.
29     iterations.append(i+1)
30
31     # Print the results - the last printed number should converge to pi.
32     print("Location: " + str(outcome) + "\t" + str(x)
33           + "\t" + str(y) + "\t" + str(count_in) + "\t"
34           + str(i) + "\t" + str(4.0 * fraction_in))
35
36 # Plot the results using pyplot
37 fig = plt.figure()
38 plt.plot(iterations, results, "k-", label="numerical pi")
39 plt.plot([0, iterations[-1]], [math.pi, math.pi],
40          "r-", label="pi")
41
42 plt.grid(True)
43 plt.legend()
44 plt.ylabel("Result [-]")
45 plt.xlabel("Iteration [-]")
46 # plt.savefig("piConvergence.pdf")
47 plt.show()

```

Listing 19: Random number Monte Carlo.

13.3 Data Visualisation

Execution of the script Listing 19 will produce two outcomes. In the terminal, you will see a number of “locations” lines, with the test outcome, (x, y) coordinates of the test point, count of points inside the circle, total count of points and estimated value of π .

You will also see a **graph**, showing the estimated value of π as a function of the number of attempts. The graph was produced **directly from python**, using the **pyplot** module, which is a part of **matplotlib**.

Currently, only some basic graphing options are used: look through the documentation to see how to make this graph prettier.

Can you write a visualisation routine to draw the square, the circle and locations of the points as they appear during evaluation?

Look at line 43 of Listing 19: a statement producing a pdf file of the graph is commented out. Can you try to use it? Have a look at the options in documentation to see how to change the size, contents and formatting.

13.4 Computing Exercises

Does the Monte Carlo method work as expected? How many points are required to get an answer accurate to 1%?

Perform a set of runs, putting the outputs in separate files and compare them. Zoom in the graph on the interesting range in y , such as 3.1 – 3.2.

Exercise: $\ln(2)$

Write a program that estimates $\ln(2)$ to 2 decimal places by a Monte Carlo method. Use the function $y = 1/x$ to work out the areas.

13.5 Assessment

Assessment Assignment

If you solved the previous $\ln(2)$ exercise **by yourself**, then please submit your solution to it as you piece of assessed work. It is important that you briefly describe your approach, and include a statement that you made the solution yourself. Submit this in the usual manner.

If on the other hand, you didn't solve the $\ln(2)$ exercise **by yourself**, please set yourself an **additional programming task** along the lines of the preceding tasks, **and solve it by yourself**. Submit your task statement and your solution in the usual manner.

See Section [A](#) in the Appendix for submission instructions.

End of Session 3

14 Object-Oriented Programming

One of the most important differences between `python` or C++ and some other programming languages is the emphasis on the representation of data using programmer or user-defined data types. In `python`, an extension to the definition of structures allows the user to include both attributes (data members) and methods (member functions) which are allowed to process the data. The encapsulation of data and functions into packages called objects of user-defined types called classes is a key part of Object-Oriented Programming (OOP).

14.1 Introduction

Object-orientation acts in the same way that scientists have always approached complex systems: by breaking them into their fundamental parts. In this way, object-orientation reduces the scientist's/programmer's cognitive load. As programmers increasingly write large and complex source code repositories, the need for object-orientation increases. In the same way that the human brain is not effective at comprehending more than approximately a paragraph of text at a time, it also is not effective at comprehending enormous code blocks or endless variable lists. To solve this, from a programming perspective, object-orientation provides a framework for classifying distinct concepts into comprehensible sizes. These smaller conceptual units facilitate cleaner and more scalable code development.

The website Software Carpentry breaks down object-orientation into five main ideas:

- Classes and objects combine **functions** with **data** to make both easier to manage;
- A class defines the behaviours of a new **kind** of thing, while an object is a **particular** thing.
For example, the concept of a car can serve as an example of a class, in which case a particular car model would be an example of an object.
- Classes have **constructors** that describe how to **create** a new object of a particular kind.
In the car example, a constructor would be a function that defines and processes all of the input data that describes a particular model of a car: type and size of engine, number of doors, colour, etc.
- An **interface** describes what an object can do.
The interface for a car would describe how objects of the car class in-

interact with objects based on other classes, e.g. the class "human". It would define that an object of the human class can perhaps get in the car, drive it, maintain it, etc.

- One class can **inherit** from another and override just those things that it wants to change.

We could derive a new class "car brand" instead of the more general car class. That class would inherit most or all features of the car class, but could then add new details to its description.

An alternative description of object-orientation highlights the following three features:

- *Encapsulation*, the property of owning data and controlling how much data from a certain class can be seen or controlled by other parts of the program;
- *Inheritance*, which establishes a relationship hierarchy between classes;
- *Polymorphism*, which allows for classes to customise their own behaviour even when they are based on other classes.

For more reading on background and theory of object-oriented programming, plenty of information can be found in literature or online. We shall now dive into how object-orientation lives in the world of `python`.

14.2 Working with Classes

We have so far used many of `python`'s built-in types. Actually, everything in `python` is in fact an object! Integers, for example, are classes/types themselves; they have attributes (which number is represented?), methods, interfaces (what happens when you try to add an integer to a float?) *etc.*

Let us define a new class/type. As a practical example, we will continue with the vehicle class theme.

14.3 Defining a Class

Let us define a class called `Vehicle`:

```
1 class Vehicle:
2     """Represents a generic vehicle."""
```

Listing 20: Minimal vehicle class.

Let us now initialise an object of the `Vehicle` type:

```

>>> myCar = Vehicle()
>>> myCar
<__main__.Vehicle object at 0x7f483aeb1400>
>>> Vehicle
<class '__main__.Vehicle'>

```

The code is telling you that `myCar` is an instance (object) of the `Vehicle` class. You are also being presented with its memory location (`0x7f483aeb1400`), which is really not of great importance at this moment.

We can now define some properties of our specific vehicle:

```

>>> myCar.manufacturer = "McMorris"
>>> myCar.topSpeed = 150
>>> myCar.manufacturer
'McMorris'
>>> myCar.topSpeed
150

```

This does not sound too useful so far, but, also, we really have not yet shown what classes have to offer us. Let us define the `Vehicle` class in a more reasonable way.

```

1 class Vehicle:
2     """Represents a generic vehicle."""
3
4     description = "a thing used for transporting people\"
5                   " or goods"
6
7     def __init__(self, manufacturer, topSpeed):
8         self.manufacturer = manufacturer
9         self.topSpeed = topSpeed
10
11     def printBasicInfo(self):
12         print("\nThe vehicle is manufactured by "
13               + self.manufacturer + ".")
14         print("It has a top speed of " + str(self.topSpeed)
15               + ".")

```

Listing 21: Vehicle class.

The `__init__` function here is what is known as a constructor. It is a function which helps us with instantiating specific objects of our class. The `self` keyword/argument helps us refer to the object that is being constructed. `manufacturer` and `topSpeed` are simply the minimum of information about our new object that the class will allow to be used for defining a new object of the `Vehicle` class. Let us test this by trying to define an object without any input arguments for the `__init__` constructor, which is called simply by using the class name and brackets.

```
>>> myCar = Vehicle()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __init__() missing 2 required positional arguments:
'manufacturer' and 'topSpeed'
```

As we can see, this time we were not allowed to initialise an object without the required information. Note that in some other languages (C++), classes can have multiple constructors, depending on how the user wants to initialise the object. In python, a class can officially have only one constructor, although this multi-constructor behaviour can be simulated through some tricks.

`printBasicInfo` is a method (a function belonging to a class) which helps us print out the basic information about our object after it has been initialised. So, let us try to be a bit more careful now and not forget about providing the required data to the constructor function.

```
>>> myCar = Vehicle("Land-Royce", 200)
>>> myCar.printBasicInfo()

The vehicle is manufactured by Land-Royce.
It has a top speed of 200.
```

14.4 Inheritance

Now to demonstrate another concept which is important in the context of object-oriented programming: inheritance. We are now confidently deciding that we are in the future going to be interested specifically in cars, not just any vehicle. Object-orientation allows us to reuse the existing code and structure, and then add or modify anything we want.

```
1 class Car(Vehicle):
2     """Represents a generic car, which is a type of
3     vehicle.
4     """
5
6     description = "a four-wheeled road vehicle"
7
8     def __init__(self, manufacturer, topSpeed, noPassengers,
9                 boot):
10         self.manufacturer = manufacturer
11         self.topSpeed = topSpeed
12         self.noPassengers = noPassengers
13         self.boot = boot
14
15     def printCarInfo(self):
```

```

16         self.printBasicInfo()
17
18         print("The number of passengers that the car can "
19               "transport is "
20               + str(self.noPassengers) + ".")
21
22         if self.boot:
23             print("This car has a boot.")
24         else:
25             print("This car does not have a boot.")
26
27     @classmethod
28     def overwriteDescription(cls, newDescription):
29         cls.description = newDescription
30
31     @staticmethod
32     def turnOnRadio(station):
33         print("\nThe radio is turned on and playing the "
34               + station + " station.")
35
36     def __add__(self, other):
37         newManufacturer = self.manufacturer + "-" \
38                             + other.manufacturer + "-Tron"
39         newTopSpeed = max(self.topSpeed, other.topSpeed) \
40                         * 1.2
41         newNoPassengers = self.noPassengers \
42                             + other.noPassengers
43         newBoot = (self.boot or other.boot)
44         return Car(newManufacturer, newTopSpeed,
45                    newNoPassengers, newBoot)

```

Listing 22: Vehicle class.

Here we see some methods decorated with `@classmethod` and `@staticmethod`, but let us deal with that a bit later. For now, we are using the new class to initialise two new objects of the `Car` class. After we have initialised them, let us use the new method `printCarInfo` (which, as you might notice, is also reusing the old `Vehicle` method `printBasicInfo`) to find out some information about our cars.

```

>>> normalCar = Car("McMorris", 150, 4, True)
>>> bootlessCar = Car("Land-Royce", 200, 2, False)
>>>
>>> normalCar.printBasicInfo()

```

The vehicle is manufactured by McMorris.
It has a top speed of 150.

```

>>> normalCar.printCarInfo()

```

```

The vehicle is manufactured by McMorris.
It has a top speed of 150.
The number of passengers that the car can transport is 4.
This car has a boot.
>>> bootlessCar.printCarInfo()

The vehicle is manufactured by Land-Royce.
It has a top speed of 200.
The number of passengers that the car can transport is 2.
This car does not have a boot.

```

We see not only that the function `printBasicInfo` is still completely usable from the main code, but also, it was reused in defining the new car-specific function `printCarInfo`. At this point it would be interesting to support a previous claim that we have made: everything in `python` is an object. Even functions! We can demonstrate that defining a new function and then using it to overwrite a function that already exists.

```

1 >>> saveFunction = Car.printCarInfo
2 >>> def overwritingFunction(self):
3 ...     print("\nI am a mean function that likes to overwrite
4 ...     other ones!")
5 >>> Car.printCarInfo = overwritingFunction
6 >>>
7 >>> bootlessCar.printCarInfo()
8
9 I am a mean function that likes to overwrite other ones!
10 >>> Car.printCarInfo = saveFunction

```

The first and last lines here were required so that we do not lose our valuable `printCarInfo` function forever.

14.5 Types of Attributes and Methods

Two types of class attributes (data belonging to a class) and methods (functions belonging to a class) are:

- Class attributes;
- Object/instance attributes.

The most important types of methods (functions belonging to a class) are:

- Instance methods;
- Class methods;
- Static methods.

14.6 Attributes

Class attributes are the ones which belong to the class itself and are shared amongst different objects/instances of the class. In the previous example, the `description` variable in the `Car` class is an example of that. If we modify it through one object, the change is going to affect all other objects of the same class, and even the class itself:

```
>>> normalCar.overwriteDescription("A new description")
>>>
>>> normalCar.description
'A new description'
>>> bootlessCar.description
'A new description'
>>> Car.description
'A new description'
```

Object/instance attributes are the variables that are object-specific. You have seen them already. They can only be accessed through the objects, because they do not even exist in the classes themselves.

```
>>> normalCar.manufacturer
'McMorris'
>>> Car.manufacturer
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: type object 'Car' has no attribute 'manufacturer'
```

14.7 Methods

Instance methods are ones that operate on objects and their data. In the `Car` class example, `printCarInfo` is an example of an instance method. As opposed to `overwriteDescription` and `turnOnRadio`, it has no decorator (keywords preceded by `@`). The `self` keyword is used as the first function argument while defining the method and it represents the specific object/instance whose data is being manipulated and used. The `self` keyword is present while defining the method, however, it is not necessary to include it (actually, its inclusion results with an error) while calling the method. Instance methods are used to access object/instance attributes.

Class methods operate on data belonging to the class. They are preceded by the `@classmethod` decorator. The `cls` keyword is similar to `self` with instance methods, but now it represents the class that it belongs to, instead of the specific object. Class methods are used to access class attributes.

Note that the choice of keywords `self` and `cls` is in theory arbitrary, but in practice, using `self` and `cls` is recommended, as it is a general consensus

to use those for the sake of clarity when sharing code.

Static methods are something of an auxiliary item when dealing with classes. They have no access to class or object attributes, but we might want to still keep them close. For example, with a real car, the radio does not need to have anything to do with the rest of the functions of the car, but we still like to have it. That is seen with the `turnOnRadio` method.

```
>>> normalCar.turnOnRadio("CambsFM")
```

```
The radio is turned on and playing the CambsFM station.
```

14.8 Operator Overloading

One neat thing about classes is that they allow for defining what different operators (+, -, etc.) do when used on objects of the class. Let us use this property to merge our existing two cars into a cool robot. This was already defined in our `Car` class using the `__add__` method. `__add__` in this case is a special keyword in `python`, used to overload the "+" operator behaviour. The `__add__` function returns a new object of the `Car` class, with properties which are a combination of the two cars used to create the new one. Looking at the `__add__` method, you should be able to figure out how the properties of the resulting car are derived.

```
>>> robotCar = normalCar + bootlessCar
```

```
>>>
```

```
>>> robotCar.printCarInfo()
```

```
The vehicle is manufactured by McMorris-Land-Royce-Tron.
```

```
It has a top speed of 240.0.
```

```
The number of passengers that the car can transport is 6.
```

```
This car has a boot.
```

Assessment assignment idea: Design a similar class structure for some other topic. Examples: Date/Time class, people in the Cavendish (Students, Lecturers, Researchers, Staff, etc.), geometry (Quadrilaterals, which are defined by Lines, which are defined by Points, etc.)

15 Working With Strings

A string is one of the basic variable types in `python` and is defined as a list of characters in order. A character in the context of string is anything you can type on a keyboard; a letter, a number, a space, backslash, bracket etc. Strings in `python` are recognizable by quotation marks. Double or single quotation marks are the same.

```
>>> print("Hello World")
Hello World
```

This section will, in what follows, cover basic string manipulation in `python3` using built-in methods.

15.1 The Basics

A string variable is created by simply assigning a list of characters to a variable name.

```
>>> word = "Programming in Python!"
>>> word
"Programming in Python!"
```

Use `[]` to access a character in a string based on its index. For example, typing `word[0]` will access the first character of a string, `word[1]` the second, and `word[-1]` the last. Indexing wraps around from the start of the string to the end and negative indices mean you can move in both directions. Passing `-1` in `python` as an index return the last element in general and is not specific to strings.

```
>>> word[0]
"P"
>>> word[1]
"r"
>>> word[-1]
"!"
```

To evaluate length of a variable use the `len` method. Please note, the `len` function with a string in its argument returns an integer that tells you how many characters there are in the string. This includes spaces, punctuation marks, *etc.*

```
>>> len(word)
22
```

To get a number of occurrences of a particular character in a string use the `count` method. In the string ‘Programming in Python!’ the character “m” is repeated twice, while the character “x” appears zero times. We can check this using the following code.


```
>>> word.count("m")
2
>>> word.count("x")
0
```

To find at which index does a particular character exist in a string use the `index` method. If only a character is passed in the argument of the method the method will return the index of the first occurrence of the character. If an integer is passed as well, the integer that is passed will be treated as a starting point of the index method. Passing the character "m" without the integer return 6 since this is the index of the first "m", while additionally passing the integer 7 returns the index of its second occurrence.

```
>>> word.index("m")
6
>>> word.index("m", 7)
7
```

15.2 Slicing and Splitting

Slicing a string involves taking a part of the string out of an existing string. This is done by passing two indices in the square brackets to a string. The indices are separated by a colon. Using -1 as the last index is a valid choice as well. However, 0 as the first index, and -1 as the last index can be completely omitted.

```
>>> word[0:5]
"Progr"
>>> word[5:8]
"amm"
>>> word[5:-1]
"amming in Python"
>>> word[:5]
"Progr"
>>> word[5:]
"amming in Python"
```

Splitting a string is a commonly used method in `python`. Any string can be split at a position of a specified character. To split a string at the position of spaces use the `split` method with " " as an argument. This will return a list of individual words of a string and the spaces will be removed. This method is, however, not restricted to spaces: any character can be used.

```
>>> word.split(" ")
["Programming", "in", "Python!"]
>>> word.split("i")
["Progr", "ng ", "n Python!"]
```

python provides methods which test if a string begins or ends with a certain character. The methods to do that are **startswith** and **endswith**. The methods return **True** if the condition is satisfied and **False** otherwise.

```
>>> word.startswith("P")
True
>>> word.endswith("!")
True
>>> word.endswith("a")
False
```

15.3 String Manipulation

It is possible to perform certain operations on strings that are not intuitive at first. For example, addition of two strings returns a combined string. This is called *string concatenation*.

```
>>> word1 = "Python"
>>> word2 = "3"
>>> word1 + word2
"Python3"
```

Similarly, multiplication with an integer returns a string with repeated characters.

```
>>> word = ".,,"
>>> word * 5
".,.,.,.,,"
```

To replace a character with another character use the **replace** method. If the second argument of the method is an empty string, the character in the first argument is removed from the string.

```
>>> word = "P.y.t.h.o.n.3"
>>> word.replace(".", ",")
"P,y,t,h,o,n,3"
>>> word.replace(".", "")
"Python3"
```

To change lowercase letters into capital and vice versa, you can use the **upper** and **lower** methods. To make only first letter of words capital use the method **title**.

swapcase can be used to swap all the capital letter into lowercase and vice versa. Reversing a string can be done by passing `"::-1"` into square brackets after a string variable.

```
>>> word = "python"
>>> word.upper()
"PYTHON"
```

```

>>> word = "IS GREAT"
>>> word.lower()
"is great"
>>> word = "python is great"
>>> word.title()
"Python Is Great"
>>> word = "pROgRammIng"
>>> word.swapcase()
"ProGrAMMiNG"
>>> word = "Python3"
>>> word[::-1]
"3nohtyP"

```

The join method can be used to put a character between two neighboring characters in a string.

```

>>> word = "Python"
>>> ",".join(word)
"P,y,t,h,o,n"

```

Finally, it is possible to check if characters in a string are of certain type.

```

word.isalnum() #check if all char are alphanumeric
word.isalpha() #check if all char in the string are alphabetic
word.isdigit() #test if string contains digits
word.istitle() #test if string contains title words
word.isupper() #test if string contains upper case
word.islower() #test if string contains lower case
word.isspace() #test if string contains spaces

```

There are of course many more things one can do with strings in python: this will give you a flavour of what to expect.

15.4 Example: Check for Anagrams

Do you like anagrams? Here is a little functions which checks if two strings are anagrams. Do you understand how it works?

```

1 # Anagram.py
2
3 # Check if two strings are anagrams
4
5 def checkAnagram(s1, s2):
6     """ Check if s1 and s2 are anagrams """
7
8     alph1 = sorted(s1.replace(" ", "").lower())
9     alph2 = sorted(s2.replace(" ", "").lower())
10
11     return alph1 == alph2
12

```

```
13
14 s1 = "real Fun"
15 s2 = "funeral"
16
17 if checkAnagram(s1, s2):
18     print(f"The strings {s1} and {s2} are anagrams.")
19 else:
20     print("The string are not anagrams.")
```

Listing 23: Check for anagrams.

16 Formatted Output

It is often necessary to produce computer output in a specific format. In this section we will review some of the `python` tools to make this easy.

16.1 Print Statements

The `print()` function in Python can be used to *print out* a value or expression given in parenthesis. For example:

```
>>> name = "Tessa"
>>> print(name)
Tessa
>>> print("name")
name
```

Strings **must** be enclosed in quotation marks. If a string is not enclosed in quotation marks, the command will produce an error.

```
>>> print(Tessa)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'Tessa' is not defined
>>> print("Tessa")
Tessa
```

The `print()` function will evaluate expressions in parenthesis before printing them out. For example, the sum of two numbers $2 + 3$ can be evaluated and printed out or can be defined as a string by being enclosed in quotation marks, which will print out the string "2+3":

```
>>> print(2+3)
5
>>> print("2+3")
"2+3"
```

The `print()` function will print out individual expressions one after the other if placed inside parenthesis and separated by a comma:

```
>>> print("Name:" , "Tessa" , ", age: " , 2+3)
Name: Tessa , age: 5
```

Strings can also be combined (concatenated) inside the parenthesis of the `print()` command using the addition operator `+`:

```
>>> name = "Tessa"
>>> print("Her name is " + name + "!")
Her name is Tessa!
```

16.2 Escape Characters

The python programming language has a series of **special or escape characters**. **Escape characters** have some unique (special) functionality and help format the output of the `print()` command. These include:

- `\n` - newline,
- `\t` - horizontal tab,
- `\v` - vertical tab,
- `\r` - carriage return (shift to the beginning of line),
- `\b` - backspace,
- `\f` - form feed (delimiter for a page break),
- `\'` - single quote,
- `\"` - double quote,
- `\\` - backslash,
- `\ooo` - octal value of character,
- `\xhh` - hex value of character,
- `\N{character name}` - display character using character name,
- `\uxxxx` - character using 16-bit hex value,
- `\Uxxxxxx` - character using 32-bit hex value.

Examples of using escape characters are given below:

```
>>> print("Hello World!")
Hello World!
>>> print("Hello\nWorld!")
Hello
World!
>>> print("Hello\tWorld!")
Hello   World!
>>> print("Hello\v World!")
Hello
      World!
>>> print("123456\rabc")
abc456
>>> print("Hello7\b World!")
```

```

Hello World!
>>> print("Hello World!\f")
Hello World!
>>> print("\"Hello World!\")
"Hello World!"
>>> print("Hello \\World!\\'")
Hello 'World!'
>>> print("This is a backslash \\")
This is a backslash \
>>> print("\110\145\154\154\157")
Hello
>>> print("\x48\x65\x6c\x6c\x6f")
Hello

>>> print("\N{GREEK CAPITAL LETTER DELTA}")
Δ
>>> print("\u0394")
Δ
>>> print("\U00000394")
Δ

```

16.3 Raw strings

It is important to note that some characters (special or escape characters) carry special meaning in `python` (e.g. `\n`, `\t`, `\b`, etc.). If we want to print such a character, without using its special meaning, we need to escape it with another backslash `\` or we can use **raw strings**. To print a string as a **raw string**, we put the character `r` in front of our string, e.g. `r'a raw string'`.

The examples below show how to escape special characters with and without the use of raw strings:

```

>>> print("Backslash: \ ")
File "<stdin>", line 1
    print(\
          \^
SyntaxError: unexpected character after line continuation
character
>>> print("Backslash: \\")
Backslash: \\
>>> print("New Line char: \n")

>>> print("New Line char: \\n")
\n
>>> print(r"This is a backslash: \\ and a new line: \n")
This is a backslash: \\ and a new line: \n

```

The last of the examples above shows how we can print special characters using a raw string. Raw strings treat a backslash as a literal character, thus

characters such as "n" are treated as two separate literal characters and not as a special character.

17 String Formatting

There are three distinct ways to format strings in python's core language:

1. % operator,
2. .format(),
3. using f-strings.

17.1 Using the % operator

Formatting strings using the % operator is consider the "old style" of formatting and is similar to the printf style in C.

```
>>> n_birds = 7
>>> print("There are %d birds in the tree." % n_birds)
There are 7 birds in the tree.
>>> name = "Tessa"
>>> print("It seems %s is not here today." % name)
It seems Tessa is not here today.
```

These examples show how the output formatting may be controlled in a similar manner as when using the C function printf (which means print, formatted).

- To print out the string "his age is 84 years" in the C programming language:

```
int age = 84; printf( "his age is %d years" , age ) ;
```

- To do the same in python we would use:

```
>>> age = 84
>>> print("his age is %d years" % age)
his age is 84 years
```

We can use the same operator to print out more than one variable:

- Using a tuple:


```
>>> age_1 = 84; age_2 = 63
>>> print("his age is %d, but her age is %d" % (age_1, age_2))
his age is 84, but her age is 63
```

- Using a dictionary:

```
>>> print("Tom is %(tom)d, Jenny is %(jenny)d and Sarah is %(sarah)d" % {'tom': 28, 'sarah': 29, 'jenny': 26})
Tom is 28, Jenny is 26 and Sarah is 29
```

Each format string that we printed in the examples above used a **format specifier** written as "%something". The most widely used **format specifiers** are:

- %c - for a single character,
- %s - for a string of characters,
- %d - for integers,
- %f - for floating point numbers,
- %g - for floating number in exponential format,
- %.<number of digits>f - for floating point numbers with a fixed number of digits,
- %.<number of digits>g - for floating number in exponential format with a fixed number of digits.

```
>>> x = 123.456
>>> print("x = %d" % x)
x = 123
>>> print("x = %f" % x)
x = 123.456000
>>> print("x = %.2f" % x)
x = 123.46
>>> print("x = %.3f" % x)
x = 123.456
>>> print("x = %.2g" % x)
x = 1.2e+02
```

A full list of **format specifiers** may be found at the following [link](#).

17.2 Using `.format()`

With `python3` a new style of formatting strings was introduced. This new style gets rid of the format specifier syntax and the `%` operator. The new syntax relies on calling the `.format()` method on a string object. The functionality of the `.format()` formatting style was backported to `python 2.7`.

The `.format()` method relies on simple positional formatting, the same as when using the `%` operator, but also enables the use of named variables. This means we can order the variables any way we want and refer to them by name. Examples of both uses are shown next:

```
>>> age = 84
>>> print("his age is {} years".format(age))
his age is 84 years

>>> age_1 = 84; age_2 = 63
>>> print("his age is {}, her age is {}".format(age_1, age_2))
his age is 84, her age is 63

>>> age_tom = 28; age_jenny = 26; age_sarah = 29
>>> print("Tom is {tom}, Jenny is {jenny} and Sarah is {sarah}".
format(sarah=29, tom=28, jenny=26))
Tom is 28, Jenny is 26 and Sarah is 29
```

We can use the same format specifiers as was the case when using the `%` operator. The specifiers shown in the previous section may also be used with the `.format()` method:

```
>>> x = 123.456
>>> print("x = {:.f}".format(x))
x = 123.456000
>>> print("x = {:.2f}".format(x))
x = 123.46
>>> print("x = {:.3f}".format(x))
x = 123.456
>>> print("x = {:.2g}".format(x))
x = 1.2e+02
```

The format specifiers form a “mini-language” that governs how individual values are represented. The most important examples are shown above, but more can be read about this topic online ([link](#)). For example, we can change the grouping option using “_” and “,”:

```
>>> X = 123456789; Y = 987654321
>>> print("Y = {Y:_.3f}, X = {X:,.2f}".format(X = X, Y = Y))
Y = 987_654_321.000, X = 123,456,789.00
```

17.3 Using f-strings

Using **f-strings** (or **string interpolation**) is the latest formatting approach and was added with **python** 3.6. The f-string format lets us embed **python** expressions inside string constants and allows us to do inline arithmetic:

```
>>> fruit = "apples"
>>> print(f"I would like some {fruit}")
I would like some apples.

>>> x = 23; y = 6
>>> print(f"x + y = {x+y}")
x + y = 29
```

We can use the same **format specifier** logic as with the `.format()` method to change the output format of our variables to make them more legible: -

```
>>> x = 123.456
>>> print(f"x = {x:f}")
x = 123.456000
>>> print(f"x = {x:.2f}")
x = 123.46
>>> print(f"x = {x:.3f}")
x = 123.456
>>> print(f"x = {x:.2g}")
x = 1.2e+02

>>> X = 123456789; Y = 987654321
>>> print(f"Y = {Y:_.3f}, X = {X:,.2f}")
Y = 987_654_321.000, X = 123,456,789.00
```

17.4 The choice of formatting

The “correct choice” of a formatting depends on the type of the data you have and how you want to present it. Different formatting specifiers will help you manage how the data is displayed and you should choose the most appropriate option for your needs. When talking about the different options available for string formatting in **python**, a simple rule of thumb may be used:

- Always use string interpolation (f-strings) if using **python** 3.6+;
- If your **python** is not **python** 3.6+, use the `.format()` method;
- Using the `%` operator is to be avoided in the latest standard.

18 Session 4: Method of Bisection

18.1 Objectives

After reading through Sections 8, 12.1, 12.2 and 12.3, you should now be familiar with:

- Definition, declaration and calling of functions;
- Passing values to and returning values from functions;
- `math` and system library functions.

Let us now use these skills in a practical computing exercise.

18.2 Computing Exercises

Finding a solution to $f(x) = 0$ by iteration.

Exercise: Bisection

Write a function that computes the square root of a number in the range $1 < x \leq 100$ with an accuracy of 10^{-4} using the bisection method. The `math` library function *must not* be used.

Test your function by calling it from a program that prompts the user for a single number and displays the result.

Modify this program so that it computes the square roots of numbers from 1 to 10. Compare your results with the answers given by the `sqrt()` mathematical library function.

18.3 The Bisection Method

The problem of finding the square root of a number, c , is a special case of finding the root of a non-linear equation of the form $f(x) = 0$ where $f(x) = c - x^2$. We would like to find values of x such that $f(x) = 0$.

A simple method consists of trying to find values of x where the function changes sign. We would then know that one solution lies somewhere between these values. For example: If $f(a) \times f(b) < 0$ and $a < b$ then the solution x must lie between these values: $a < x < b$. We could then try to narrow the range and hopefully converge on the true value of the root. This is the basis of the so-called bisection method.

The bisection method is an iterative scheme (repetition of a simple pattern) in which the interval is halved after each iteration to give the approximate location of the root. After i iterations the root (let's call it x_i , *i.e.* x after i iterations) must lie between

$$a_i < x_i \leq b_i$$

and an approximation for the root is given by:

$$p_i = \frac{(a_i + b_i)}{2},$$

where the error between the approximation and the true root, ϵ_i , is bounded by:

$$\epsilon_i = \pm \frac{(b_i - a_i)}{2} = \pm \frac{(b_1 - a_1)}{2^i}.$$

At each iteration the sign of the functions $f(a_i)$ and $f(p_i)$ are tested and **if** $f(a_i) \times f(p_i) < 0$ the root must lie in the half-range $a_i < x < p_i$. Alternatively, the root lies in the other half (see figure).

We can thus update the new lower and upper bound for the root:

if $f(a_i) \times f(p_i) < 0$ then $a_{i+1} = a_i$ and $b_{i+1} = p_i$ else $a_{i+1} = p_i$ and $b_{i+1} = b_i$
--

The bisection method is guaranteed to converge to the true solution but is slow to converge since it uses only the sign of the function.

An alternative is the Newton-Raphson method, which takes into account the gradient of the function, $f'(x)$, and only needs one starting guess for the root. In the Newton-Raphson method the next approximation to the root is given by:

$$p_{i+1} = p_i - \frac{f(p_i)}{f'(p_i)}.$$

18.4 Notes on the Algorithm and Implementation

The square root of a number will be found by calling a user-defined function to implement one of the iterative algorithms (*i.e.*, repetition of a pattern of actions) described above.

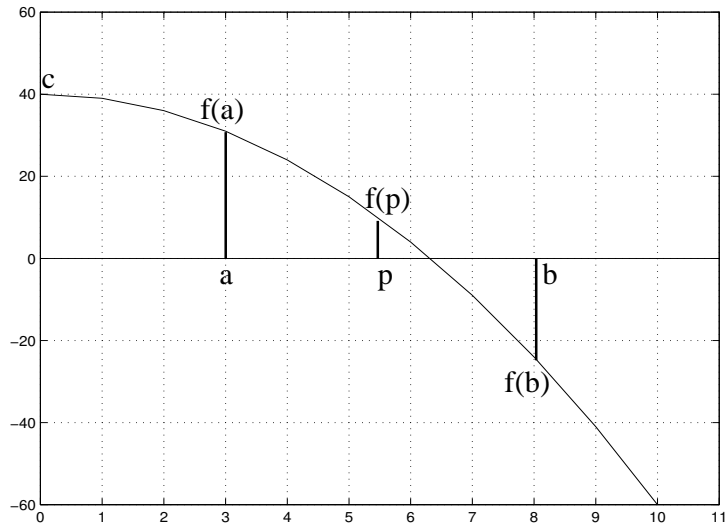


Figure 12: Bisection root-finding algorithm.

Please review Section 12.1 which describes how to define a function and how to pass parameters to the function and return values.

You are required to define a function to find the square root of a number c , i.e., find the zeroes of $f(x) = c - x^2$. Your solution is to be accurate to 5 decimal places. Since the number input by the user is between 1 and 100 the root will satisfy $0 < x \leq 10$ and a valid initial guess for the lower and upper bound for the solution will always be $a_1 = 0.1$ and $b_1 = 10.1$.

The error after i iterations will be $\pm \frac{10}{2^i}$. To produce a solution that is accurate to 5 decimal places we will need more than 20 iterations.

18.5 Assessment

Assessment Assignment

Set yourself an **additional programming task** along the lines of the preceding tasks, and solve it by yourself. Submit your task statement and your solution in the usual manner.

See Section [A](#) in the Appendix for submission instructions.

End of Session 4

19 Notes Concerning the Remaining SESSIONS

The aim of the remaining SESSIONS will be not only to learn more about programming, but also to see how programming can help you understand physics better.

This final two assessed tasks are harder than the ones you have seen so far.

It is essential to prepare before each week's programming exercises. Read this manual and sketch out a plan of what you are going to do **before** sitting down at the computer.

The exercises involve physics, so you will need to prepare both physics thoughts and computing thoughts.

Please allow two hours preparation time per week.

20 Session 5: Planets

20.1 Objectives

Physics Objectives. To better understand collisions, conservation laws, and statistical physics, especially equipartition, properties of ideal gases, the concept of temperature, and the Boltzmann distribution; also the way in which microscopic physics leads to macroscopic phenomena; what happens when a piston compresses an ideal gas; adiabatic expansion; fluctuations and dissipation, equilibration of systems with different temperatures.

Computing objectives. Classes and arrays; visualisation.

20.2 Your Task

Simulate Newton's laws for a small planet moving near a massive sun. For ease of plotting, assume the planet and its velocity both lie in a two-dimensional plane. Put the sun at the origin $(0,0)$ and let the planet have mass m and initial location $\mathbf{x}^{(0)} = (x_1, x_2)$ and initial velocity $\mathbf{v}^{(0)} = (v_1, v_2)$.

The equations of motion are:

$$\frac{d\mathbf{x}}{dt} = \mathbf{v}(t) \tag{1}$$

$$m \frac{d\mathbf{v}}{dt} = \mathbf{f}(\mathbf{x}, t), \tag{2}$$

where, for a standard inverse-square law (gravity or electrodynamics) the force \mathbf{f} is:

$$\mathbf{f}(\mathbf{x}, t) = A \frac{\mathbf{x}}{\left(\sqrt{\sum_i x_i^2}\right)^3}, \tag{3}$$

with $A = -G M m$ for gravitation, and $A = Q q / (4\pi\epsilon_0)$ for electrodynamics with two charges Q and q .

Try to write your programs in such a way that it will be easy to switch the force law from inverse-square to other force laws. For example, Hooke's law states:

$$\mathbf{f}(\mathbf{x}, t) = k \mathbf{x}. \tag{4}$$

How should we simulate Newton's laws, Eqs. (1 and 2) on a computer? One simple method called Euler's method makes *small, simultaneous* steps in \mathbf{x} and \mathbf{v} .

We repeat lots of times the following block:

EULER'S METHOD

- 1: Find $\mathbf{f} = \mathbf{f}(\mathbf{x}, t)$
- 2: Increment \mathbf{x} by $\delta t \times \mathbf{v}$
- 3: Increment \mathbf{v} by $\delta t \times \frac{1}{m}\mathbf{f}$
- 4: Increment t by δt

We might hope that, for sufficiently small δt , the resulting state sequence (\mathbf{x}, \mathbf{v}) in the computer would be close to the true solution of the differential equations.

Euler's method is not the only way to approximate the equations of motion. and indeed, as you'll see, it is very inaccurate compared with the alternatives.

An equally simple algorithm can be obtained by reordering the updates. In the following block, the updates of \mathbf{f} and \mathbf{x} have been exchanged, so the force is evaluated at the new location, rather than the old.

- A: Increment \mathbf{x} by $\delta t \times \mathbf{v}$
- B: Find $\mathbf{f} = \mathbf{f}(\mathbf{x}, t)$
- 3: Increment \mathbf{v} by $\delta t \times \frac{1}{m}\mathbf{f}$
- 4: Increment t by δt

It is not at all obvious that this minor change should make much difference, but often it does. This second method, in which position and velocity are updated alternately, is called the Leapfrog method. Here is a precise statement of the Leapfrog method:

LEAPFROG METHOD

```
Repeat
{
    Increment  $\mathbf{x}$  by  $\frac{1}{2}\delta t \times \mathbf{v}$ 
    Increment  $t$  by  $\frac{1}{2}\delta t$ 
    Find  $\mathbf{f} = \mathbf{f}(\mathbf{x}, t)$ 
    Increment  $\mathbf{v}$  by  $\delta t \times \frac{1}{m}\mathbf{f}$ 
    Increment  $\mathbf{x}$  by  $\frac{1}{2}\delta t \times \mathbf{v}$ 
    Increment  $t$  by  $\frac{1}{2}\delta t$ 
}
```

In this version, we make a half-step of \mathbf{x} , a full-step of \mathbf{v} , and a half-step of \mathbf{x} . Since the end of every iteration except the last is followed by the beginning of the next iteration, this algorithm with its half-steps is identical to the previous version, except at the first and last iterations.

When simulating a system like this, there are some obvious quantities you should look at: the angular momentum, the kinetic energy, the potential energy, and the total energy.

20.3 Ideas for What to Do

This is a self-directed and self-assessed course, and I'd like you to choose a planet-simulating activity that interests you. Here are some suggestions.

You don't have to do all of these. You can also invent your own.

The more you do, the more educational it will be. But do take your pick, and feel free to steal working code (*e.g.* from the course website) if you'd prefer to focus your energy on experimenting with working code, rather than writing and debugging your own.

1. Write code that implements Euler's method and the Leapfrog method. Get it going for one initial condition, such as $\mathbf{x}^{(0)} = (1.5, 2)$, $\mathbf{v}^{(0)} = (0.4, 0)$.

Before running your program, predict the correct motion, roughly. Compare the two methods by looking at the resulting trajectories, the energies, and the angular momenta.

If you'd like more detailed guidance through a possible approach to this task, A worked solution for this first part is also available.

2. Once you have a trustworthy simulation: take a collection of similar initial conditions, all having the same initial position and initial energy but differing in the direction of the initial velocity. What happens? Show all the evolving trajectories in a single movie.
3. Or, take initial conditions that differ slightly in their initial position or velocity, such as "the space-shuttle and the spaceman who is 100 yards from the space-shuttle, both orbiting the earth" (in what direction does he need to nudge himself in order to get home?).

Or "the spaceman and the hammer" – if he throws his hammer 'forwards', where does it end up going? If he throws it 'back' or 'sideways', where does it end up going?

(Here the idea is to simulate two planets orbiting a single sun; to get the spaceman analogy, think of the spaceman and his shuttle being like the two planets, and the earth playing the role of the sun.)

4. Get an initial condition that leads to a perfectly circular orbit. Now perturb that initial condition by giving a little kick of momentum in 8 different directions.

Show all nine resulting trajectories in a single movie.

- Which kicks lead to the period of the orbit changing?

- Which kicks lead to the period of the orbit staying the same?
- Which kicks lead to orbits that come back to the place where the kick occurred?

If an object is in an elliptical orbit, what kicks do you give it in order to make the orbit larger and circular? What kicks do you give it to make the orbit smaller and circular? What's the best time, in an elliptical orbit, to give the particle a kick so as to get the particle onto an orbit that never comes back, assuming we want the momentum in the kick to be as small as possible?

5. Take initial conditions coming in from a long way away, particles traveling along equally-spaced parallel lines. What happens? (The distance of the initial line from the sun is called the impact parameter of the initial condition.)
6. People who criticise Euler's method often recommend the *Runge–Kutta method*. Find out what Runge–Kutta is (Google knows), implement it, and compare with the Leapfrog method. Choose a big enough step-size δt so that you can see a difference between the methods.

Given equal numbers of computational operations, does Runge–Kutta or Leapfrog do a better job of making an accurate trajectory? For very long runs, and again assuming equal numbers of computational operations, does Runge–Kutta or Leapfrog do a better job of energy conservation? Of angular momentum conservation?

20.4 Assessment

This SESSION has a slightly different problem than SESSIONS 1 to 4 as the emphasis is now on using computing to help you learn about physics, rather than just learning to program.

There is greater importance attached to the “Brief Description” part of the submission than in the earlier sessions, and consequently it will be longer than in the earlier sessions, and will contain a greater discussion of physics directed goals than of the computing ones.

You will be expected to use the “brief description” part of the web-based submission form to explain not only the technical/computational challenges which you faced, but also to devote a greater than normal time to explaining the *physics goals* which you set yourself – perhaps based on the suggestions of Section 20.3 – and what use of your program helped you learn about them.

This additional part of the “Brief Description” is the part you need to ‘engage with’ if you are to gain the extra credit for this section.

End of Session 5

20.5 Worked Solution for the Planets Problem

Approach: **Chop the problem into small pieces.**

20.5.1 Step-by-Step Guidance

Here are some steps to help you walk through the problem.

1. Create a **SolarSystem** class that manages the entire system. You can think of the SolarSystem as a canvas, where you can insert as many planets as you want, plot the planets, check for collisions, gravity interaction etc.
2. Define a class that contains the state of the **Planet**: position, mass, velocity, to which solar system does it belong etc. As you continue, you will probably think of other sensible things to add to the structure. You might find it elegant to give a name to the dimensionality of the position space.
3. Write a simple script that defines a planet and change its parameters. Add it a **draw** function to plot its position. Add a gravity function to evaluate the force.
4. Develop a **move** function, which updates its position and velocity. Move has to be defined at the level of the Planet class, but has to be called for all planets from the SolarSystem.
5. Create the Sun planet, whose position is fixed. Sun class is inherited from Planet class.

Develop and test items one-at-a-time.

20.5.2 Sample Code Listing

```

1 import matplotlib.pyplot as plt
2 import numpy as np
3 import matplotlib.animation as animation
4 from mpl_toolkits.mplot3d import Axes3D
5
6
7 class SolarSystem():
8     """This class creates the SolarSystem object."""
9     def __init__(self):
10         """With self you can access private attributes of the
11         object.
12         """
13         self.size = 1000
14         self.planets = []
15         # This initializes the 3D figure
16         self.fig = plt.figure()
17         self.ax = Axes3D(self.fig, auto_add_to_figure=False)
18         self.fig.add_axes(self.ax)
19         self.dT = 1
20
21     def add_planet(self, planet):
22         """Every time a planet is created it gets put into
23         the array.
24         """
25         self.planets.append(planet)
26
27     def update_planets(self):
28         """This method moves and draws all of the planets."""
29         self.ax.clear()
30         for planet in self.planets:
31             planet.move()
32             planet.draw()
33
34     def fix_axes(self):
35         """The axes would change with each iteration
36         otherwise.
37         """
38         self.ax.set_xlim((-self.size/2, self.size/2))
39         self.ax.set_ylim((-self.size/2, self.size/2))
40         self.ax.set_zlim((-self.size/2, self.size/2))
41
42     def gravity_planets(self):
43         """This method calculated gravity interaction for
44         every planet.
45         """
46         for i, first in enumerate(self.planets):
47             for second in self.planets[i+1:]:
48                 first.gravity(second)
49

```

```

50
51 class Planet():
52     """This class creates the Planet object."""
53     def __init__(
54         self,
55         SolarSys,
56         mass,
57         position=(0, 0, 0),
58         velocity=(0, 0, 0)
59     ):
60         self.SolarSys = SolarSys
61         self.mass = mass
62         self.position = position
63         self.velocity = velocity
64         # The planet is automatically added to the SolarSys.
65         self.SolarSys.add_planet(self)
66         self.color = "black"
67
68     def move(self):
69         """The planet is moved based on the velocity."""
70         self.position = (
71             self.position[0]+self.velocity[0]*SolarSys.dT,
72             self.position[1]+self.velocity[1]*SolarSys.dT,
73             self.position[2]+self.velocity[2]*SolarSys.dT
74         )
75
76     def draw(self):
77         """The method to draw the planet."""
78         self.SolarSys.ax.plot(
79             *self.position,
80             marker="o",
81             markersize=10,
82             color=self.color
83         )
84
85     def gravity(self, other):
86         """The method to compute gravitational force for two
87         planets. numpy module is used to handle vectors.
88         """
89         distance = np.subtract(other.position, self.position)
90         distanceMag = np.linalg.norm(distance)
91         distanceUnit = np.divide(distance, distanceMag)
92         forceMag = self.mass*other.mass / (distanceMag**2)
93         force = np.multiply(distanceUnit, forceMag)
94         # Switch makes force on self opposite to other
95         switch = 1
96         for body in self, other:
97             acceleration = np.divide(force, body.mass)
98             acceleration = np.multiply(force, SolarSys \

```

```

99         .dT*switch)
100         body.velocity = np.add(body.velocity,
101                                acceleration)
102         switch *= -1
103
104
105 class Sun(Planet):
106     """This class is inherited from Planet. Everything is
107     the same as in planet, except that the position of the
108     sun is fixed. Also, the color is yellow.
109     """
110     def __init__(
111         self,
112         SolarSys,
113         mass=1000,
114         position=(0, 0, 0),
115         velocity=(0, 0, 0)
116     ):
117         super(Sun, self).__init__(SolarSys, mass,
118                                   position, velocity)
119         self.color = "yellow"
120
121     def move(self):
122         self.position = self.position
123
124
125 # Instantiating of the solar system.
126 SolarSys = SolarSystem()
127
128 # Instantiating of planets.
129 planet1 = Planet(SolarSys,
130                  mass=10,
131                  position=(150, 50, 0),
132                  velocity=(0, 5, 5))
133 planet2 = Planet(SolarSys,
134                  mass=10,
135                  position=(100, -50, 150),
136                  velocity=(5, 0, 0))
137 planet3 = Planet(SolarSys,
138                  mass=10,
139                  position=(-100, -50, 150),
140                  velocity=(-4, 0, 0))
141
142 # Instantiating of the sun.
143 sun = Sun(SolarSys)
144
145
146 def animate(i):
147     """This controls the animation."""

```



```

148     print("The frame is:", i)
149     SolarSys.gravity_planets()
150     SolarSys.update_planets()
151     SolarSys.fix_axes()
152
153
154 # This calls the animate function and creates animation.
155 anim = animation.FuncAnimation(SolarSys.fig, animate,
156                                frames=100, interval=100)
157 # This prepares the writer for the animation.
158 writervideo = animation.FFMpegWriter(fps=60)
159 # This saves the animation.
160 anim.save("planets_animation.mp4",
161           writer=writervideo, dpi=200)

```

Listing 24: Example of a code for gravitational interaction between 3 Planets and a Sun.

21 Advanced Containers

The basics of containers have been discussed in Section 9. The possibilities with these data structures are countless.

21.1 More On Lists

This section presents a couple of interesting cases and how to use lists to your advantage. Hopefully these examples show how versatile lists are and why every `python` programmer should master them. Of course, this is not an exhaustive list of examples. More functions can be found in documentation.

21.1.1 Lists and Loops

Very common operation on lists is creating them with loops or performing loops on them. The example below shows a short script that creates an array with a for loop and later uses a for loop to calculate its sum.

```
1 arr = []
2
3 for i in range(10):
4     arr.append(i**2)
5
6 print(arr)
7
8 square_sum = 0
9 for el in arr:
10     square_sum += el
11
12 print(f"compare the sums: {square_sum}, {sum(arr)}")
```

Listing 25: Lists and loops.

The print statement at line 6 results in `[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]`. The print statement at line 12 checks if the sum from the for loop is correct.

A very useful tool for looping is the range method. It returns a sequence of numbers that starts from a starting integer, increments by a specified step, and stops before a specified number. The general syntax is:

```
range(start, stop, step)

start - optional,
stop - required,
step - optional
```

Start and step arguments are not required. If only the required stop argument is passed to the function, the returned sequence starts from 0,

increments by 1, and stops when the length of sequence equals the passed integer.

Lists can be nested. This means that inside a list it is possible to have another list as an element.

```
>>> arr = [1, 2, 3]
>>> arr_nested = [arr, arr]
>>> arr_nested
[[1, 2, 3], [1, 2, 3]]
```

Nested lists are common in `python` since they allow us to create more complex data structures. For example, nested lists resemble matrices. The next example is a piece of code that creates an array resembling a 3x3 matrix.

```
1 arr = []
2
3 for i in range(3):
4     arr_temp = []
5     for j in range(3):
6         arr_temp.append(i*3+j)
7     arr.append(arr_temp)
8
9 print(arr)
```

Listing 26: Nested for loop.

The structure is very similar to the first example. The difference here is the nested for loop. At every iteration of the outer loop a new temporary array 'arr_temp' gets initialised. The inner loop populates the temporary array which is finally appended into the resulting array.

21.1.2 Nested Lists and List Comprehension

For the next example, try to figure out how would you code a method that returns a transpose of a matrix given a 3x3 array.

```
1 matrix = [[0, 1, 2],
2           [3, 4, 5],
3           [6, 7, 8]]
4 transpose = []
5
6 for i in range(len(matrix)):
7     transpose.append([row[i] for row in matrix])
8
9 print(transpose)
```

Listing 27: Matrix transpose.

To better understand the example above, let us introduce a concept of list comprehension. The concept allows writing simple for loops in a single

line. In example below the argument of the append method replaces the for loop in the first example.

```
>>> arr = [i**2 for i in range(10)]
>>> arr
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

To take list comprehension to the next level, let us define a single line that replaces nested for loops.

```
>>> matrix = [[0, 1, 2], [3, 4, 5], [6, 7, 8]]
>>> transpose = [[row[i] for row in matrix] for i in range(len(
matrix))]
>>> transpose
[[0, 3, 6], [1, 4, 7], [2, 5, 8]]
```

However, even though this is possible and syntactically correct, a programmer should think about readability of the code.

python comes with another very useful command for handling arrays. The zip method returns an iterator of tuples based on the iterable objects. First let us get familiar with the '*' operator for unpacking of the arrays.

```
>>> arr = [1, 2, 3, 4, 5]
>>> print(*arr)
1 2 3 4 5
>>> matrix = [[0, 1, 2], [3, 4, 5], [6, 7, 8]]
>>> print(matrix)
[[0, 1, 2], [3, 4, 5], [6, 7, 8]]
>>> print(*matrix)
[0, 1, 2] [3, 4, 5] [6, 7, 8]
```

Putting '*' in front of the iterable, such as a list, unpacks the elements. The next example shows how to use the zip method and what the method does:

```
>>> arr1 = [1, 2, 3, 4, 5]
>>> arr2 = [10, 20, 30, 40, 50]
>>> zip(arr1, arr2)
<zip object at 0x7f4f016911c0>
>>> list(zip(arr1, arr2))
[(1, 10), (2, 20), (3, 30), (4, 40), (5, 50)]
```

The method takes two iterables, creates a new iterable zip object that can be transformed into a list by the list method. It can be seen that the newly created list is a list of tuples that each take elements with the same index from the given arrays. Combining zip method, '*' operator, and nested arrays gives you tools to simply create transpose of matrices:

```
>>> matrix = [[0, 1, 2], [3, 4, 5], [6, 7, 8]]
>>> transpose = list(zip(*matrix))
```

```
>>> transpose
[(0, 3, 6), (1, 4, 7), (2, 5, 8)]
```

21.1.3 Further Useful Methods for Lists

python built-in functions allow further functionality for handling lists. For example:

```
# Sorting
>>> arr = [1, 2, 5, 6, 7, 3]
>>> arr.sort()
>>> arr
[1, 2, 3, 5, 6, 7]
# Backward sorting
>>> arr = [1, 2, 5, 6, 7, 3]
>>> arr.sort(reverse=True)
>>> arr
[7, 6, 5, 3, 2, 1]
# Clearing
>>> arr
[1, 2, 3, 5, 6, 7]
>>> arr.clear()
>>> arr
[]
# Counting of repeating instances
>>> arr = [1, 1, 1, 1, 2, 3, 4]
>>> arr.count(1)
4
# Removing elements by index with pop
# use pop if the removed value is needed
>>> arr = [1, 2, 5, 6, 7, 3]
>>> arr.pop(2)
5
>>> arr
[1, 2, 6, 7, 3]
# Removing elements by index with del
# use del if the removed value is not needed
>>> arr = [1, 2, 5, 6, 7, 3]
>>> del arr[2]
>>> arr
[1, 2, 6, 7, 3]
# Removing by value
>>> arr = [1, 2, 5, 6, 7, 3]
>>> arr.remove(5)
>>> arr
[1, 2, 6, 7, 3]
```

Please note, these methods are exclusive to lists. They do not exist for tuples.

21.2 More On Tuples

Working with tuples is very similar to working with lists and are used in similar situations. Since the elements of a tuple are unchangeable, they are preferred option when the data has to remain constant. Tuples guarantee write protection.

A slight difference between generating tuples and lists is when we need a sequence of a single element. Putting a single element between square brackets is sufficient to create a list. Conversely, this is not the case for a tuple. To create a tuple with a single element, a trailing comma is needed.

```
>>> arr = [1]
>>> arr
[1]
>>> tup = (1)
>>> tup
1
>>> tup = (1,)
>>> tup
(1,)
```

To create a tuple the brackets can be omitted. However, it is a good practice to use brackets when creating a tuple.

```
>>> tup = 1, 2, 3, 4
>>> tup
(1, 2, 3, 4)
```

Individual elements of a tuple cannot be deleted. The `del` function can be used to delete the container completely.

```
>>> tup = (1, 2, 3, 4)
>>> del tup
>>> tup
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'tup' is not defined
```

21.3 More on Sets

Sets are useful when working with data and the order or frequency of the elements do not matter.

Sets can be created from other types of data and containers with `set` method. Note how `set1` and `set2` generated from different lists are the same. This is because sets do not accept duplicate items.

```
>>> set1 = set([1, 2, 3])
>>> set1
```

```
{1, 2, 3}
>>> set2 = set([1, 2, 3, 3])
>>> set2
{1, 2, 3}
```

Sets can be created from tuples and strings. Note how there are two pairs of brackets when set1 is created from a tuple. The first pair are the brackets of the set method, while the second pair are the brackets of the passed tuple.

```
>>> set1 = set((1, 2, 3))
>>> set1
{1, 2, 3}
>>> set2 = set("abcde")
>>> set2
{"e", "b", "a", "c", "d"}
```

If you try to generate a set from an integer the following error appears.

```
>>> set1 = set(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not iterable
```

This indicates a very important property of the set method. In order to use it we need to pass an iterable in the argument. To do this with integers, the range method can be used.

```
>>> set2 = set(range(10))
>>> set2
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Let's try to use sets to get more insight into lists and tuples. First, we will create an empty list and tuple. Second, to get the list of all the possible methods that can be invoked with the type we will use dir method.

```
>>> example1 = list()
>>> dir(example1)
[list of list-methods]
>>> example2 = tuple()
>>> dir(example2)
[list of tuple-methods]
```

We can use set operations to get more information on these two sets. For example, intersection of these sets return which methods exist for both tuples and lists.

```
>>> set1 = set(dir(example1))
>>> set2 = set(dir(example2))
>>> set1.intersection(set2)
{set where set1 and set2 are the same}
```

Please note, this will return names of the methods and some methods will be surrounded by double leading and double trailing underscores. These methods are called ‘python Magic or Dunder Methods’ and have been already introduced in Section 14. Briefly, the Magic methods are not meant to be invoked directly by a user, but the invocation happens internally from the class on a certain action. For example, when you add two numbers using the + operator, internally, the `__add__()` method will be called.

21.4 More on Dictionaries

A dictionary in python can be created in different ways. For example from a list of keys and the method `fromkeys`. This, however, assigns to every key the same value.

```
>>>dict1 = {}.fromkeys(["one", "two", "three"], 0)
>>>dict1
{"one": 0, "two": 0, "three": 0}
```

21.4.1 Dictionaries and Loops

It is possible to create iterable from a dictionary, what means it is possible to loop over its elements. Unpacking of keys and values is made possible by the `.items()` method that is called on a dictionary.

```
1 dict1 = {"k1": 1, "k2": 2, "k3": 3}
2
3 for key, value in dict1.items():
4     print(f"For this iteration key is {key} and the value is {value}")
```

Listing 28: For loop with dictionary.

The following example shows how to add elements to a dictionary in a loop. The elements of the resulting dictionary have the string integers for keys with their squares as values. Please note, `str(i)` is not strictly required, since keys can be integers as well.

```
1 dict1 = {}
2
3 for i in range(10):
4     dict1[str(i)] = i**2
5
6 print(dict1)
```

Listing 29: Creating dictionary in a for loop.

21.4.2 Dictionary Comprehension

It is possible to change the inputs of a dict with dictionary comprehension.

```
>>>dict1 = {"k1": 1, "k2": 2, "k3": 3}
>>>dict1 = {key: value**2 for key, value in dict1.items()}
>>>dict1
{"k1": 1, "k2": 4, "k3": 9}
```

Comprehension in general works with conditionals. In this example the third element was not added since the value of the third element in dict1 is not less than 3.

```
>>>dict1 = {"k1": 1, "k2": 2, "k3": 3}
>>>dict1 = {key:value**2 for key, value in dict1.items() if value
< 3}
>>>dict1
{"k1": 1, "k2": 4}
```

22 Session 6: Collisions

22.1 Objectives

Physics Objectives. To better understand Newton’s laws, circular motion, angular momentum, planets’ orbits, Rutherford scattering, and questions about spacemen throwing tools while floating near orbiting objects.

Computing objectives. Classes and containers; using visualisation libraries; simulation methods for ordinary differential equations (Euler, Leapfrog).

You are encouraged to work in pairs, with the weaker programmer doing all the typing.

22.2 Your Task

We are going to simulate hard spheres colliding with each other elastically in a box. An incredible range of interesting physics phenomena can be studied in this way, Fig. 13.

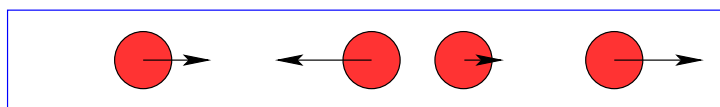


Figure 13: Computing collisions.

The heart of this computing exercise is going to be a single function – let’s call it `collide` – which receives two particles with masses m_1 and m_2 and velocities u_1 and u_2 , and returns the new velocities v_1 and v_2 of the two particles after an elastic collision between the two.

You could start by writing a function that implements this elastic collision. Check your function with special cases that you’ve solved by hand, and by putting in a range of initial conditions and evaluating whether total momentum and total energy are indeed conserved as they should be.

An elegant approach to this programming task uses a structure to represent each particle – something like this, for a particle moving in one dimension.

```
class particle
    position
    momentum
    inverse mass
```

velocity
kinetic energy
radius of particle

At this stage it's not crucial, but at some point I recommend making sure all references to the particle's mass use the *inverse mass*, rather than the mass – this allows you to treat the walls of the box as standard particles that just happen to have infinite mass (that is, inverse-mass zero).

Now, define a list (or some other container) of particles and let them interact with each other.

22.3 Ideas for What to Do

There is a lot of choice. This is a self-directed and self-assessed course, and I'd like you to choose a bonking-simulating activity that interests you. Here are some suggestions.

You don't have to do all of these. You can also invent your own. The more you do, the more educational it'll be. But do take your pick, and feel free to steal working code (*e.g.* from the course website) if you'd prefer to focus your energy on experimenting with working code, rather than writing and debugging your own.

1. Write code that uses a one-dimensional `collide` function to simulate the motion of N particles in a one-dimensional box. Each particle can collide only with its immediate neighbours; each of the two end particles can collide with one wall too. To simulate the dynamics, you must identify the times at which collisions occur, and (assuming you want to make a realistic movie) spit out the state of the simulation *at equally spaced times*.

A suggested strategy is to take the current state and figure out which pair of adjacent particles will collide *next*. Then advance the simulation exactly to the moment of that next collision (stopping if appropriate at intermediate times along the way, so as to spit out the required states for the movie, equally spaced in time). Collisions should be handled by passing the pair of particles to the `collide` function. Motion in between collisions is simple (since there are no forces) and it should be handled by another function, `leapForward`, say. Printing out of the state at certain times of interest should be handled by another function, `showState`, say.

Note. A possible difficulty with this approach of computing all collisions that occur is that it is conceivable that the true number of collisions in a finite time might be very large, a phenomenon known as chattering. You can get the idea of chattering by imagining quickly squashing a moving ping-pong ball between a table-tennis bat and a table.

2. Testing: Put just two particles in a box, with equal masses. Check that the dynamics are right. Make the two masses unequal. Make a scatter-plot of the positions of the two particles. Make a scatter-plot of the velocities of the two particles. Use more masses. Check that kinetic energy is conserved.
3. Put quite a few *unequal* masses in the box (say, 10 masses, with a variety of masses spanning a range of 4 orders of magnitude), run the simulation for a long time, and make histograms of the velocities of two of the particles whose masses are in the ratio 4:1. What do you find? Make histograms of the *positions* of the particles. If you make some of the particles *really heavy* compared to their neighbours, what happens to the histogram of the positions of the neighbours? For example, make all the particles except for the two end particles be much larger; or make half the particles (those on the left hand side) heavy, and the other half light. (In all simulations make sure no two adjacent particles have identical masses.)
4. What happens if the right-hand wall (with infinite mass) is moved at constant speed towards or away from the other wall?

You have heard it said that, under some circumstances, $pV^\gamma = \text{const.}$ What is γ for a *one-dimensional* ideal gas? How should the total energy vary with V under these conditions?

5. Set up N_1 light masses to the left of a single big heavy mass, and N_2 more light masses to the right of the heavy mass. Call the heavy mass a piston, if you like, and think of it as separating two ideal gases from each other. The light masses don't need to be identical to each other. An example set of masses for $N_1 = N_2 = 5$ could be (1.1, 1.2, 1.1, 1.3, 1.1, 100.0, 4.1, 4.2, 4.1, 4.8, 4.4) where the 100-mass is the piston. Give randomly chosen velocities to the particles. What should happen? How long does it take for 'equilibrium' to be reached? Give an enormous velocity to the piston and small velocities to the other particles. What should happen? How long does it take for 'equilibrium' to be reached?

Can you get the piston to oscillate roughly sinusoidally (before ‘equilibrium’ is reached)? What is the frequency of such oscillations? Predict the frequency using the theory of adiabatic expansion/compression of gases.

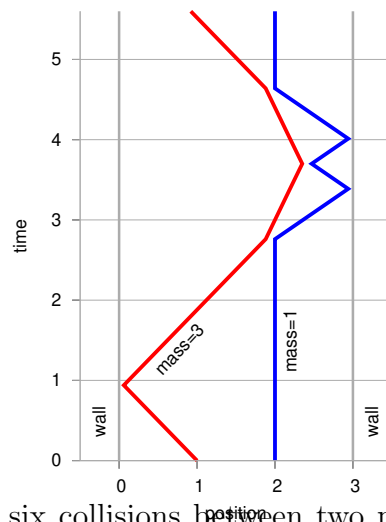


Figure 14: The first six collisions between two particles of masses 3 and 1 and two walls.

22.4 Assessment

This SESSION has a slightly different than SESSIONS 1 to 4 as the emphasis is now on using computing to help you learn about physics, rather than just learning to program.

There is greater importance attached to the “Brief Description” part of the submission than in the earlier sessions, and consequently it will be longer than in the earlier sessions, and will contain a greater discussion of physics directed goals than of the computing ones.

You will be expected to use the “brief description” part of the web-based submission form to explain not only the technical/computational challenges which you faced, but also to devote a greater than normal time to explaining the *physics goals* which you set yourself – perhaps based on the suggestions of Section 22.3 – and what use of your program helped you learn about them. This additional part of the “Brief Description” is the part you need to ‘engage with’ if you are to gain the extra credit for this section.

End of Session 6

22.5 Worked Solution for the Collisions Problem

22.5.1 Sample Code Listing

```
1 # Collision.py
2
3 import matplotlib.pyplot as plt
4 import matplotlib.animation as animation
5 import itertools
6
7
8 class Canvas():
9     """This class creates the canvas object."""
10    def __init__(self):
11        """With self you can access private attributes of
12        the object.
13        """
14        self.size = 20
15        self.blocks = []
16        self.fig = plt.figure()
17        self.ax = self.fig.add_subplot()
18
19    def add_block(self, block):
20        """Every time a block is created it gets put into
21        the array.
22        """
23        self.blocks.append(block)
24
25    def update_blocks(self):
26        """This method moves and draws all blocks."""
27        self.ax.clear()
28        for i, block in enumerate(self.blocks):
29            block.move()
30            block.draw()
31
32    def fix_axes(self):
33        """The axes would change with each iteration
34        otherwise.
35        """
36        self.ax.set_xlim((-self.size/2, self.size/2))
37        self.ax.set_ylim((-1, 1))
38
39    def check_collision(self):
40        """This method checks if blocks are colliding."""
41        combinations = list(itertools.combinations(
42            range(len(self.blocks)), 2))
43
44        for pair in combinations:
45            self.blocks[pair[0]]\
```

```

46         .collide(self.blocks[pair[1]])
47
48
49 class Block():
50     """This class creates the block object."""
51     def __init__(
52         self,
53         canvas,
54         mass,
55         position=0,
56         velocity=0
57     ):
58         self.canvas = canvas
59         self.mass = mass
60         self.position = position
61         self.velocity = velocity
62         # The block is automatically added to the canvas.
63         self.canvas.add_block(self)
64         self.color = "black"
65
66     def move(self):
67         """The block is moved based on the velocity."""
68         self.position = self.position + self.velocity
69
70     def draw(self):
71         """The method to draw the block. Note: if you don't
72         specify color of the block the color of each new
73         elemnt that is plotted will be randomly assigned.
74         """
75         canvas.ax.plot(self.position, 0, "o")
76
77     def collide(self, other):
78         # Watch out with the threshold, this should
79         # be greater than velocity. If, in a single
80         # iteration, blocks move relatively more than
81         # what is the threshold, the collision might
82         # not be triggered.
83         if abs(self.position - other.position) < 0.1:
84             # The velocity after collision is not modelled
85             # correctly. Take a look what happens when
86             # particles going in the same direction collide.
87             # How to fix that?
88             self.velocity *= -1
89             other.velocity *= -1
90
91
92 canvas = Canvas()
93 block1 = Block(canvas, mass=1, position=-2, velocity=0.07)
94 block2 = Block(canvas, mass=1, position=2, velocity=-0.07)

```

```

95 block3 = Block(canvas, mass=1, position=4, velocity=-0.05)
96 block4 = Block(canvas, mass=1, position=-5, velocity=0.05)
97
98
99 def animate(i):
100     """This controls the animation."""
101     print("The frame is:", i)
102     canvas.update_blocks()
103     canvas.check_collision()
104     canvas.fix_axes()
105
106
107 # This calls the animate function and creates animation.
108 anim = animation.FuncAnimation(canvas.fig, animate,
109                                frames=500, interval=10)
110 # This prepares the writer for the animation.
111 writervideo = animation.FFMpegWriter(fps=60)
112 # This saves the animation.
113 anim.save("blocks_animation.mp4",
114           writer=writervideo, dpi=200)

```

Listing 30: Collision.

Advanced Topics

23 Regular Expressions

A sequence of characters that form a search pattern is called a **Regular Expression** or **RegEx**. Regular expressions are used as search patterns and are used to check whether a string contains the specified pattern.

To use **RegEx** in **python** the **re** package needs to be imported. Once imported the **re** package will allow the use of regular expressions. The **re** package provides a set of functions which can be used to manipulate strings depending on the result of a regular expression search:

- `findall()` - returns a list containing all matches;
- `search()` - if there is a match, returns a match object;
- `split()` - returns a list, consisting of the string split at each match;
- `sub()` - replaces one (or more) matches with a specified string;

All of the functions of the **re** package rely on regular expressions as search patterns. Regular expressions are formed as a combination of **metacharacters**, **special sequences** and **sets**.

23.1 Metacharacters

Metacharacters are characters which carry a special meaning in regular expressions. A list of the most important metacharacters is given below:

- `[]` - used to form a set of characters,
- `\` - used as an escape sequence for escaping special characters. *e.g.* `\d` - meaning digits,
- `.` - any character except a newline,
- `^` - starts with,
- `$` - ends with,
- `*` - zero or more occurrences of a character,
- `+` - one or more occurrences of a character,
- `?` - zero or one occurrence of a character,
- `{}` - specified number of occurrences,

- | - either or,
- () - capture or group.

```
>>> import re
>>> text = "hello world73"
>>> x = re.findall("[a-j]", text); print(x)
['h', 'e', 'd']

>>> x = re.findall("\d", text); print(x)
['7', '3']

>>> x = re.findall("he..o", text); print(x)
['hello']

text = "hello there"
>>> x = re.findall("^hello", text); print(x)
['hello']
>>> x = re.findall("^Hello", text); print(x)
[]

>>> x = re.findall("ld$", text); print(x)
['ld']
>>> x = re.findall("lo$", text); print(x)
[]

>>> x = re.findall("he.*o", text); print(x)
['hello']
# searched for sequence starting with "he" followed by zero or
# more occurrence of any character and an "o"

>>> x = re.findall("he.+o", text); print(x)
['hello']
# searched for sequence starting with "he" followed by one or
# more occurrence of any character and an "o"

>>> x = re.findall("he.?o", text); print(x)
[]
# two matches between "he" and "o", not zero not one.
# Thus we got not matches.

>>> x = re.findall("he.{2}o", text); print(x)
['hello']
# searched for sequence starting with "he" followed by
# exactly 2 occurrence of any character and an "o"

>>> x = re.findall("Hello|hello", text); print(x)
['hello']
# checked if the string contains either "Hello" or "hello"
```

23.2 Special sequences

Special sequences are formed with a backslash character `\` and one of the characters carrying a special meaning, placed together with a search string. A list of special characters used to form special sequences is given below:

- `\A` - match if specified character(s) are at the beginning of string;
- `\b` - match if specified characters are at the beginning or end of word;
- `\B` - match if specified characters are NOT at beginning or end of word;
- `\d` - returns match where string contains digits (0-9);
- `\D` - returns match where string does NOT contain digits;
- `\s` - returns match where string contains a white space;
- `\S` - returns match where string does NOT contain a white space;
- `\w` - returns match where string contains any word characters (a to Z, digits 0-9, and underscore _);
- `\W` - returns match where string does NOT contain any word characters;
- `\Z` - returns match if specified characters are at the end of the string.

Some special sequences should be used as raw strings to stop `python` from treating them as special (escape) characters inside the regular expression:

```
>>> import re
>>> txt_1 = "Simply the best!"
>>> txt_2 = "The best!"
>>> x = re.findall(r"\bSimply", txt_1)
>>> print(x)
['Simply']
>>> x = re.findall(r"\bSimply", txt_2)
>>> print(x)
[]
>>> x = re.findall("best!\Z", txt_2)
>>> print(x)
['best!']
```

23.3 Sets

Sets are formed by placing a number of characters inside a pair of square brackets, giving them a special meaning. A list is given below:

- `[asd]` - returns match if one of specified characters (**a**, **s**, **d**) if present;
- `[a-f]` - returns match for any character, alphabetically between **a** and **f**;
- `[^asd]` - returns match for any character except **a s** and **d**;
- `[0123]` - match if any of the specified digits (0, 1, 2, or 3) are present;
- `[0-9]` - returns match for any digit between 0 and 9;
- `[0-4][0-2]` - returns a match for any two-digit numbers from 00 and 42;
- `[a-zA-Z]` - returns a match for any character between a and z, lower case or upper case;
- `[+]` - no special meaning in a set, treated as a **+** character. This will return a match for any **+** in a string. The same would be true for other characters used in a set, like: *****, **.**, **|**, **(**, **\$**, and **+**.

Some examples of how sets may be used in regular expressions are given below:

```
>>> import re
>>> txt = "Simply the best!"
>>> x = re.findall("[^Ssye!]", txt)
>>> print(x)
['i', 'm', 'p', 'l', ' ', 't', 'h', ' ', 'b', 't']
>>> x = re.findall("[a-zA-Z]", txt)
>>> print(x)
[' ', ' ', ' ', '!']
>>> txt = "23, 12, 45, 68, 43, 30"
>>> x = re.findall("[0-3][0-9]", txt)
>>> print(x)
['23', '12', '30']
```

The sets, metacharacters and special sequences can be used together with different functions from the **re** package to modify strings by searching through, replacing parts of or splitting the string, all based on the results of a regular expression. Let us explore some of these functions and some regular expressions:

- We already saw the `findall()` function in action. This function will return a list containing all matches to the given regular expression. If no matches are found it will return an empty list.

```
>>> import re
>>> txt = "Hello World!"
>>> x = re.findall("\d", txt)
>>> print(x)
[]
>>> x = re.findall("He.*", txt)
>>> print(x)
['Hello World!']
```

- The `search()` function searches through the string for a match. If a match is found the `search()` function returns a match object, but if no match is found the function returns `None`. The match object has methods and properties used to retrieve information, such as: `.span()` (returns a tuple containing the start-, and end positions of the match), `.string` (returns the string passed into the function), `.group()` (returns the part of the string where there was a match), *etc.*

```
>>> import re
>>> txt = "Spain is under heavy rain!"
>>> x = re.search("\d", txt)
>>> print(x)
None
>>> x = re.search("ai", txt)
>>> print(x)
<re.Match object; span=(2, 4), match='ai'>
>>> print(x.string)
Spain is under heavy rain!
>>> x = re.search(r"\bS\w+", txt)
>>> print(x.string)
# search for an upper case "S" character in the beginning of
# a word
Spain is under heavy rain!
>>> print(x.span())
# to print its position
(0, 5)
>>> print(x.group())
# to print the word only
Spain
```

- We can use the `split()` function to return a list which contains the string split at each match. For example, we can split at each white space using the `\s` sequence. We can also control the number of splits

using the `maxSplit` parameter. If the `maxSplit` parameter is set to 1, the string will be split only at the first occurrence.

```
>>> import re
>>> txt = "Spain is under heavy rain!"
>>> x = re.split("\s", txt)
>>> print(x)
['Spain', 'is', 'under', 'heavy', 'rain!']
>>> x = re.split("\s", txt, 1)
>>> print(x)
['Spain', 'is under heavy rain!']
```

- The `sub()` function may be used to replace the matched parts of the string with a supplied text of our choice. We can also control the number of replacements which are to occur by specifying the `count` parameter.

```
>>> import re
>>> txt = "Spain is under heavy rain!"
>>> x = re.sub("\s", "-", txt)
>>> print(x)
Spain-is-under-heavy-rain!
>>> x = re.sub("\s", "?",txt, 2)
>>> print(x)
Spain?is?under heavy rain!
```

24 Working with Text Files

Until now we usually printed the output of our functions to the screen. There are times when we would like to save our data to files for later use. One of the most basic ways of saving data is writing it to text files. Python has built-in functions for creating, writing and reading files.

One of the more important concepts when talking about saving data to text files in python is the **file handle**. When thinking about the **file handle** we can think about the cursor in a visual text editor. Same as with the cursor, the **file handle** determines the position at which the data is read or written in a text file.

Another important concept is the **file access mode**. **File access modes** determine what type of operations are possible on an opened file and directly influence the position of the **file handle**.

24.1 File Access Modes

Python recognises several **file access modes**:

- **read only (r)**. The default mode with which a file is opened. It opens the file for reading and positions the handle at the beginning of the file. If the specified file does not exist an I/O error is raised.
- **read and write (r+)**. File is opened for both reading and writing. The file handle is placed at the beginning. If no file is found the I/O error is raised.
- **write only (w)**. The file is opened for writing. If the file does not exist, it is created. For existing files, the existing data is overwritten. In both cases, the file handle is positioned at the beginning of the file.
- **write and read (w+)**. The file is opened for both reading and writing. If non-existent, the file is created and for existing files the data is overwritten. The file handle is positioned at the beginning of the file.
- **append only (a)**. The file is opened for writing. If the file does not exist, it is created. The file handle is positioned at the end of the file. This means that any data written to the file will be inserted at the end, after the existing data.
- **append and read (a+)** - The file is opened for both reading and writing. If the file does not exist, it is created. The file handle is positioned at the end of the file. This means that any data written to the file will be inserted at the end, after the existing data.

24.2 Opening and Closing Files

The `open()` function is used to open a file object. The function requires two arguments: `open(filename, file_access_mode)`. The second argument of the `open()` function is the file access mode, which was explained above. The first argument is a string used to provide the filename (or path). If the file is in the same directory as the current python prompt or script, only the filename needs to be provided. If the file is not in the same directory, the full path to the file needs to be provided along with filename. As paths usually contain special characters (`\`, `/`) it is common practice to use raw strings to prevent certain characters to be read as special characters (e.g. `/t` in `/temp` could be read as the tab character). Let us look at some examples that could be part of a python script to open some files:

```
1 # the file is in the same directory as this script
2 # and is opened in append mode
3 file1 = open("dummy_file.txt", "a")
4
5 # the file is inside a /tmp directory
6 # we need to use a raw string
7 # we open the file in write and read mode
8 file2 = open(r"/tmp/dummy_file.txt", "w+")
9
10 # if we are on a windows pc, we use a raw string for the path
11 file3 = open(r"C:\system32\dummy_file.txt", "w+")
```

Listing 31: How to open text files.

The `close()` function is used to close the file and free up the memory space occupied by the opened file. We use the `close()` when we are finished with editing the file or we want to open the file in another file access mode. A common practice when opening files in python is to use the `with` keyword. This ensures that the file is properly closed after we are finished, even if an exception is raised. Using the `with` keyword is preferred over manually freeing up memory by calling `close()`.

Let us see how we can open and close a file object:

```
# we opened the dummy file as a file object f,
# then we closed it manually using close()
f = open("dummy_file.txt", "a")
# ...
f.close()

# we will use the with statement to close the file
# object automatically, even if an exception is raised
with open("dummy_file.txt", "a") as f:
    # ...
```

24.3 Reading

To read from a file we can use three functions:

- `file_object.read([n])` - Returns the read bytes as a string. If an argument `n` is supplied, reads `n` lines. If no argument `n` is supplied it reads the entire file.
- `file_object.readline([n])` - Reads a line of the file and returns it as a string. If `n` is supplied, reads at most `n` bytes. Can only read that one line, even if the length of `n` exceeds the length of the line.
- `file_object.readlines()` - Reads all the lines of the file and returns them as a list where each line is a string element in the list.

24.4 Writing

Two basic functions may be used to write to a text file:

- `file_object.write(str)` - Used to insert the string `str` in a single line in the text file opened as a file object.
- `file_object.writelines(str_list)` - Used to insert multiple strings in the text file opened as a file object. For a list of string elements, each string is inserted in the text file.

24.5 Some Examples

Let us go through a few examples in which we will read and write to some files:

```
1 # we want to read a file which does not exist
2 # in write mode, so the file will be created
3 str_list = ["\tLondon, ", "Zagreb, ", "Tokyo"]
4 f = open("tmp.txt", "w")
5 f.write("Hello \n")
6 f.writelines(str_list)
7
8 # now we need to close the file manually
9 # since we did not use with and since we want to open
10 # the file in read mode
11
12 f.close()
13
14 # now we can change the file access mode to read
15 f = open("tmp.txt", "r")
```

```

16 print("The file reads:")
17 print(f.read())
18 f.close()

```

Listing 32: Reading and Writing into files; Example 1.

The output reads as follows:

```

The file reads:
Hello
    London, Zagreb, Tokyo

```

```

1 # let us try to append to the file and add some more cities,
2 # also let us use the with keyword
3 with open("tmp.txt", "a+") as f:
4     # write new cities:
5     f.writelines(["", New York, " ", "Barcelona, ", "Berlin!"])
6
7 # let us read the file again
8 with open("tmp.txt", "r") as f:
9     print(f.read())

```

Listing 33: Reading and Writing into file; Example 2.

The output reads as follows:

```

Hello
    London, Zagreb, Tokyo, New York, Barcelona, Berlin!

```

```

1 # now let us read the file as a series of string elements in a list
2 with open("tmp.txt", "r") as f:
3     print(f.readlines())

```

Listing 34: Reading and Writing into file; Example 3.

The output reads as follows:

```
['Hello \n', '\tLondon, Zagreb, Tokyo, New York, Barcelona, Berlin!']
```

Lastly, we can use the `tell()` and `seek()` functions to find the position of the file handle and reposition it. The `tell()` function returns an integer as the position of the file handle in the file object represented as the number of bytes, if the file is in binary format, or as the number of characters from the beginning of the file. The `seek(offset, whence)` function allows us to reposition the file handle. The position is computed by adding the `offset` value to the `whence` value. The default value of `whence` is 0 and this denotes the beginning of the file, while 1 means we use the current position of the handle as reference and 2 uses the end of the file as a reference point. When

handling text files that are not in a binary format, `seek()` allows us to reposition the handle relative only to the beginning of the file (or we can place the handle to the very end of the file with `seek(0,2)`).

Let us open a file as binary in write and read mode:

```
>>> f = open("binary_file.txt", "wb+")
>>> f.write(b"abcdefgh01234567")
>>> f.close()
```

Now, we can reopen the newly created file as a binary file in append mode. This will place our handle at the very last byte of the file (16). Then, let us try to reposition the file and find out the new position.

```
>>> f = open ("binary_file.txt", "ab")
>>> f.tell()
16
>>> f.seek(2)
# whence is 0 by default - start of file
2
>>> f.tell()
2
>>> f.seek(4, 1)
# whence is 1 - current position
6
>>> f.tell()
6
>>> f.seek(-3, 2)
# whence is 2 - end of file
14
>>> f.tell()
14
```

Now, let us open the tmp.txt file we created earlier and let us prepend a new string to the beginning of the file:

```
>>> f = open("tmp.txt", "r+")
>>> lines = f.readlines()
>>> lines.insert(0, "Look at this new line !\n")
>>> f.seek(0)
0
>>> f.writelines(lines)
>>> f.close()

>>> f = open("tmp.txt", "r")
>>> print(f.read())
Look at this new line !
Hello
London, Zagreb, Tokyo, New York, Barcelona, Berlin!
>>> f.close()
```

This last example showed us how we can read in existing lines in a file as a list (places handle at the end of the file), append the new lines to the list, reposition the file handle using `seek` and overwriting the existing lines with new ones.

25 Manipulating CSV files with pandas

pandas is a **python** package for data manipulation and analysis. In particular, it offers data structures and operations for manipulating numerical tables and time series. In this introduction section to **pandas**, it is presented how to use **pandas** to handle comma-separated-values (CSV) files.

To install **pandas** use **pip**:

```
pip install pandas
```

To start working with **pandas**, we need a csv file with some data. Let us create a simple csv file with a couple of columns (see Table 7). The separator of the csv file can be various things. For example, comma, space, tab etc. In this demonstration we will use space.

col1	col2	col3	col4
1.00	2.00	3.00	4.00
1.00	2.00	3.00	4.00
3.00	4.00	1.00	4.00
1.00	1.00	1.00	0.00
3.00	3.00	3.00	10.00
4.00	3.00	3.00	4.00
4.00	3.00	3.00	4.00
4.00	3.00	3.00	4.00
4.00	3.00	3.00	4.00

Table 7: Input csv file.

The simple program below loads the csv file with the `.read_csv()` method into a `DataFrame` that simplifies handling and analyzing the data. You can imagine `DataFrame` as a complex container that stores data from the csv file. The `.read_csv` method requires file path of the input file. Additionally, separator can be specified if the separator of the csv file is different than comma. Printing `DataFrame` content can sometimes be messy, especially for big data tables. Methods `.head()` and `.tail()` allow printing the first or last 5 rows of the table.

```
1 import pandas as pd
2
3 file_path = "./file.csv"
4
5 df = pd.read_csv(file_path, sep=" ")
6
7 print(df.tail())
```

Listing 35: Loading csv files with **pandas**.

To create a new column of a DataFrame is very simple. Specify the new column name in square brackets with an arbitrary string and assign a value to it. Output csv file is created with the `.to_csv()` method. To use space as a separator, specify the `sep` argument. If other separator is preferred in the output file, it can be specified here. To ignore writing additional index column use `index=False` argument.

```
1 import pandas as pd
2
3 file_path = "./file.csv"
4 output_path = "./output_file1.csv"
5
6 df = pd.read_csv(file_path, sep=" ")
7
8 df["coolColumn"] = 1
9
10 df.to_csv(output_path, sep=" ", index=False)
```

Listing 36: Creating new columns.

The output file will look like Table 8.

col1	col2	col3	col4	coolColumn
1.00	2.00	3.00	4.00	1.00
1.00	2.00	3.00	4.00	1.00
3.00	4.00	1.00	4.00	1.00
1.00	1.00	1.00	0.00	1.00
3.00	3.00	3.00	10.00	1.00
4.00	3.00	3.00	4.00	1.00
4.00	3.00	3.00	4.00	1.00
4.00	3.00	3.00	4.00	1.00
4.00	3.00	3.00	4.00	1.00

Table 8: Output csv file of the first program.

The columns of the table can be accessed in the same way. The following example shows how to create a new column as a combination of the existing columns. The resulting column of this program will be the sum of the first and double second column. The individual columns can be treated as lists. We can compute mean value, sum, standard deviation (`std`), *etc.*

To access a single value of the column put the index in the square brackets.

```
1 import pandas as pd
2
3 file_path = "./file.csv"
4 output_path = "./output_file2.csv"
5
6 df = pd.read_csv(file_path, sep=" ")
```

```

7
8 df["coolColumn"] = df["col1"] + df["col2"]*2
9
10 # This prints the sum of the "combinedColumn"
11 print(sum(df["combinedColumn"]))
12 # This prints the first element of the "combinedColumn"
13 print(df["combinedColumn"][0])
14 # This prints the standard deviation of the "combinedColumn"
15 print(df["combinedColumn"].std())

```

Listing 37: Accessing columns.

The resulting column of this program will be the sum of the first and double second column as shown in Table 9.

col1	col2	col3	col4	combinedColumn
1.00	2.00	3.00	4.00	5.00
1.00	2.00	3.00	4.00	5.00
3.00	4.00	1.00	4.00	11.00
1.00	1.00	1.00	0.00	3.00
3.00	3.00	3.00	10.00	9.00
4.00	3.00	3.00	4.00	10.00
4.00	3.00	3.00	4.00	10.00
4.00	3.00	3.00	4.00	10.00
4.00	3.00	3.00	4.00	10.00

Table 9: Output csv file of the second program.

A very useful method is `sort_values`. It sorts the whole DataFrame based on a given column. `inplace` parameter forces writing to the DataFrame `df` without explicit assigning. If `inplace` was not set to `True`, the DataFrame would not be rewritten.

```

1 import pandas as pd
2
3 file_path = "./file.csv"
4 output_path = "./output_file3.csv"
5
6 df = pd.read_csv(file_path, sep=" ")
7
8 df.sort_values(by="col1", inplace=True)
9
10 print(df.head())
11
12 df.to_csv(output_path, sep=" ", index=False)

```

Listing 38: Sorting DataFrame.

The whole output csv file is now sorted according to the specified column name as shown in Table 10.

col1	col2	col3	col4
1.00	2.00	3.00	4.00
1.00	2.00	3.00	4.00
1.00	1.00	1.00	0.00
3.00	4.00	1.00	4.00
3.00	3.00	3.00	10.00
4.00	3.00	3.00	4.00
4.00	3.00	3.00	4.00
4.00	3.00	3.00	4.00
4.00	3.00	3.00	4.00

Table 10: Output csv file of the sorted DataFrame.

To access row or a range of rows use the `.iloc` method. The arguments for this method go inside the square brackets. If a single integer is passed, the method returns a single row. If two integers separated by colon are passed, the method will return multiple columns ranging from the first to the second integer.

```
1 import pandas as pd
2
3 file_path = "./file.csv"
4
5 df = pd.read_csv(file_path, sep=" ")
6
7 print(df.iloc[0:2])
```

Listing 39: Accessing rows of DataFrame.

Higher statistical moments with **pandas** are simple to compute as well. For example, to get a covariance matrix of the **DataFrame**, use the `.cov()` method. To get unbiased variance use the `.var()` method.

```
1 import pandas as pd
2
3 file_path = "./file.csv"
4
5 df = pd.read_csv(file_path, sep=" ")
6
7 cov = df.cov()
8 var = df.var()
9
10 print("covariance matrix\n", cov, "\n")
11 print("variance\n", var, "\n")
```

Listing 40: Variance and Covariance.

This short overview of **pandas** just scratches the surface of the possibilities within the package. Interested readers that would like to learn more about **pandas** are referred to the **pandas** documentation.

Appendix

A Submitting Your Work

Submission of work will be done over a simple web-based Google form. The link to upload the form is: <https://forms.gle/TGddTz2pUdu5mtBr6>.

Clicking on the link should take you to the form shown in Fig. 15.

The figure displays two views of a Google Forms interface for a 'Self-Assessment'. The left view shows the top of the form with a header image of a desk with a coffee cup and papers. Below the header, the title 'Self - Assessment' is followed by instructions: 'Here, you can upload your self-assessment code for different Sessions.' and a Google account selection prompt for 'hj348@cam.ac.uk'. The right view shows the main form fields: 'Session' (a dropdown menu), 'Title' (a text input), 'Description' (a text input), 'Email' (a text input), 'Name' (a text input), and 'CRSid' (a text input). Below these fields is an 'Upload file' section with a note 'Before uploading, please merge all of the scripts into a single file.' and an 'Add File' button. At the bottom of the form, there is a 'Submit' button and a 'Clear form' link. The footer includes the text 'A copy of your responses will be emailed to the address that you provided.', 'Never submit passwords through Google Forms.', 'reCAPTCHA', and 'This form was created inside University of Cambridge. Report Abuse'.

Figure 15: Self-Assessment Form for submission of your work.

It is expected from each participant to upload a single file representative of your work for each session. The form should require you to supply:

- Your e-mail address. This is important to be entered correctly since an automated e-mail confirming your submission will be sent to this address once the 'submit' button is clicked;
- Your full name;
- Your CRSid number;
- The session you are submitting your work for;
- A title for the task which you set yourself. It should describes your program in a few words;

- A brief description of the task you set yourself. For example: ‘I set myself the following additional goal. I decided to write a program to [do this] and extended/modified it to [do this additionally] until it could [do all of this]. My solution is in the file called [script].py’;
- A Python or Text file with all of the code you would like to submit. Please note, you should copy-paste all scripts into a single file. For example:

```
'''This is a collection of files for the Session 1.'''


# Here begins file1 [name].py
The source code of the file1

# Here begins file2 [name].py
The source code of the file2

# Here begins file3 [name].py
The source code of the file3

# Here begins file4 [name].py
The source code of the file4
```

If all is in order, after you click Submit button you should be presented with a message resembling that shown in Figure 16 (a). You should also receive the confirmation via email shortly after. The e-mail should look like Figure 16 (b). If all that happens, your submission was a success and you can continue to the next task. If you suspect any problem with submission, contact the lecturer.



Self - Assessment

Your response has been recorded.

This form was created inside of University of Cambridge. [Report Abuse](#)

Google Forms

Thanks for filling out [Self - Assessment](#)

Here's what was received.

Self - Assessment

Here, you can upload your self-assessment code for different Sessions.

Email *

Name *

CRSid *

Session *

1

Figure 16: Successfully submitted form will: a) present you with the page of successful submission; b) automatically send you an email with the details of your submission.

B Some Useful UNIX Commands

This guide, based on a University computing service leaflet, is intended to be as generally valid as possible, but there are many different versions of Unix available within the University, so if you find a command option behaving differently on your local machine you should consult the on-line manual page for that command. Some commands have numerous options and there is not enough space to detail them all here, so for fuller information on these commands use the relevant on-line manual page.

The names of commands are printed in **bold**, and the names of objects operated on by these commands (*e.g.* files, directories) are printed in *italics*.

Index

1. **cat** - display or concatenate files
2. **cd** - change directory
3. **chmod** - change the permissions on a file or directory
4. **cp** - copy a file
5. **find** - find files of a specified name or/and type
6. **grep** - searches files for a specified string or expression
7. **help** - display information about bash built-in commands
8. **less** - scan through a text file page by page with enabled backward navigation
9. **ls** - list names of files in a directory
10. **mkdir** - make a directory
11. **more** - scan through a text file page by page
12. **mv** - move or rename files or directories
13. **pwd** - display the absolute path of your current directory
14. **rm** - remove files or directories
15. **which** - shows the path of an executable

Detailed Description

1. **cat** – display or concatenate files

cat takes a copy of a file and sends it to the standard output (i.e., to be displayed on your terminal, unless redirected elsewhere), so it is generally used either to read files, or to string together copies of several files, writing the output to a new file.

cat *ex*

displays the contents of the file *ex*.

cat *ex1 ex2 > newex*

creates a new file *newex* containing copies of *ex1* and *ex2*, with the contents of *ex2* following the contents of *ex1*.

2. **cd** – change directory

cd is used to change from one directory to another.

cd *dir1*

changes directory so that *dir1* is your new current directory. *dir1* may be either the full pathname of the directory, or its pathname relative to the current directory.

cd

changes directory to your home directory.

cd ..

moves to the parent directory of your current directory.

3. **chmod** – change the permissions on a file or directory

chmod alters the permissions on files and directories using either symbolic or octal numeric codes. The symbolic codes are given here:

u	user	+	add permission	r	read
g	group	–	remove permission	w	write
o	other	=	assign permission explicitly	x	execute, access

x gives execution permission for files and access permission for folders. The following examples illustrate how these codes are used.

chmod **u=rw** *file1*

sets the permissions on the file *file1* to give the user read and write permission on *file1*. No other permissions are altered.

chmod u+x,g+w,o-r file2

alters the permissions on the file *file2* to give the user execute permission on *file2*, to give members of the user's group write permission on the file, and prevent any users not in this group from reading it.

chmod u+w,go-x dir1

gives the user write permission in the directory *dir1*, and prevents all other users having access to that directory (by using **cd**. They can still list its contents using **ls**.)

chmod g+s dir2

means that files and subdirectories in *dir2* are created with the group-ID of the parent directory, not that of the current process.

4. **cp** – copy a file

The command **cp** is used to make copies of files and directories.

cp file1 file2

copies the contents of the file *file1* into a new file called *file2*. **cp** cannot copy a file onto itself.

cp file3 file4 dir1

creates copies of *file3* and *file4* (with the same names), within the directory *dir1*. *dir1* must already exist for the copying to succeed.

cp -r dir2 dir3

recursively copies the directory *dir2*, together with its contents and subdirectories, to the directory *dir3*. If *dir3* does not already exist, it is created by **cp**, and the contents and subdirectories of *dir2* are recreated within it. If *dir3* does exist, a subdirectory called *dir2* is created within it, containing a copy of all the contents of the original *dir2*.

5. **find** – find files of a specified name or type

find searches for files in a named directory and all its subdirectories.

find . -name '*.cc' -print

searches the current directory and all its subdirectories for files ending in *.cc*, and writes their names to the standard output. In some versions of Unix the names of the files will only be written out if the **-print** option is used.

find /local -name core -user user1 -print

searches the directory */local* and its subdirectories for files called *core* belonging to the user *user1* and writes their full file names to the standard output.

6. **grep** – searches files for a specified string or expression

grep searches for lines containing a specified pattern and, by default, writes them to the standard output.

grep *motif1 file1*

searches the file *file1* for lines containing the pattern *motif1*. If no file name is given, **grep** acts on the standard input. **grep** can also be used to search a string of files, so

grep *motif1 file1 file2 ... filen*

will search the files *file1*, *file2*, ... , *filen*, for the pattern *motif1*.

grep -c *motif1 file1*

will give the number of lines containing *motif1* instead of the lines themselves.

grep -v *motif1 file1*

will write out the lines of *file1* that do NOT contain *motif1*.

7. **help** – display info about bash built-in commands

help gives access to information about built-in commands in the **bash** shell. Using **help** on its own will give a list of the commands it has information about. **help** followed by the name of one of these commands will give information about that command. **help history**, for example, will give details about the **bash** shell history listings.

8. **less** – terminal pager program

less is used to view the contents of a text file. It is similar to **more** with an extended functionality to navigate the file both forwards and backwards.

less *file.txt*

visualizes the contents of *file.txt* in terminal. User can navigate line by line of the file with *arrow keys*, or page by page with *Page up* and *Page down* keys. If the content of the file is changing while the user is reading it, **shift + f** command can be use to refresh and tail new input in the *follow* mode. To exit the *follow* mode use **ctrl + c**. To exit **less** press *q*.

9. **ls** – list names of files in a directory

ls lists the contents of a directory, and can be used to obtain information on the files and directories within it.

ls *dir1*

lists the names of the files and directories in the directory *dir1*, (excluding files whose names begin with *.*). If no directory is named, **ls** lists the contents of the current directory.

ls -a *dir1*

will list the contents of *dir1*, (including files whose names begin with *.*).

ls -l *file1*

gives details of the access permissions for the file *file1*, its size in kbytes, and the time it was last altered.

ls -l *dir1*

gives such information on the contents of the directory *dir1*. To obtain the information on *dir1* itself, rather than its contents, use

ls -ld *dir1*

10. **mkdir** – make a directory

mkdir is used to create new directories. In order to do this you must have write permission in the parent directory of the new directory.

mkdir *newdir*

will make a new directory called *newdir*.

mkdir -p can be used to create a new directory, together with any parent directories required.

mkdir -p *dir1/dir2/newdir*

will create *newdir* and its parent directories *dir1* and *dir2*, if these do not already exist.

11. **more** – scan through a text file page by page

more displays the contents of a file on a terminal one screenful at a time.

more *file1*

starts by displaying the beginning of *file1*. It will scroll up one line every time the *return* key is pressed, and one screenful every time the *space* bar is

pressed. Type **?** for details of the commands available within **more**. Type **q** if you wish to quit **more** before the end of *file1* is reached.

more -n file1

will cause *n* lines of *file1* to be displayed in each screenful instead of the default (which is two lines less than the number of lines that will fit into the terminal's screen).

12. **mv** – move or rename files or directories

mv is used to change the name of files or directories, or to move them into other directories. **mv** cannot move directories from one file-system to another, so, if it is necessary to do that, use **cp** instead – copy the whole directory using **cp -r oldplace newplace** then remove the old one using **rm -r oldplace**.

mv name1 name2

changes the name of a file called *name1* to *name2*.

mv dir1 dir2

changes the name of a directory called *dir1* to *dir2*, unless *dir2* already exists, in which case *dir1* will be moved into *dir2*.

mv file1 file2 dir3

moves the files *file1* and *file2* into the directory *dir3*.

13. **pwd** – display the name of your current directory

The command **pwd** gives the full pathname of your current directory.

14. **rm** – remove files or directories

rm is used to remove files. In order to remove a file you must have write permission in its directory, but it is not necessary to have read or write permission on the file itself.

rm file1

will delete the file *file1*. If you use

rm -i file1

instead, you will be asked if you wish to delete *file1*, and the file will not be deleted unless you answer **y**. This is a useful safety check when deleting lots of files.

rm -r *dir1*

recursively deletes the contents of *dir1*, its subdirectories, and *dir1* itself, and should be used with suitable caution.

rm -rf *dir1*

is like **rm -r**, except that any write-protected files in the directory are deleted without query. This should be used with even more caution.

15. **which** – identify the location of an executable

which command takes one or more arguments and for each of them, it prints the full path of the executable to standard output that would have been executed if this argument had been entered into the terminal. It does this by searching for an executable or script in the directories listed in the environment variable PATH. For example:

which *less*

returns the location of the file that is executed when *less* command is called.

C Some Useful Windows Commands

This list of commands is intended for the Windows `cmd` shell. If you are using PowerShell on Windows, refer to Section [B](#).

In case of problems, please inform the instructor in order to update or correct the list.

Ending any command with a flag `/?` shows the documentation of the command.

Index

1. **cd** - change directory
2. **copy** - make a directory
3. **del** - remove files
4. **dir** - list contents of a directory
5. **find** - find files of a specified name or type
6. **mkdir** - make a directory
7. **rename** - rename a directory
8. **rmdir** - remove directories

Detailed Description

1. **cd** – change directory

cd is used to change from one directory to another.

cd *dir1*

changes directory so that *dir1* is your new current directory. *dir1* may be either the full pathname of the directory, or its pathname relative to the current directory. The full pathname in Windows should be written between double quotes.

cd

changes directory to your home directory.

cd ..

moves to the parent directory of your current directory.

2. **copy** – copy a file

The command **copy** is used to make copies of files.

copy *file1 file2*

copies the contents of the file *file1* into a new file called *file2*. **copy** cannot copy a file onto itself.

copy *file1 folder1*

creates a copy of *file1* with the same name in the *folder1*.

copy *file1.txt + file2.txt file3.txt*

merges *file1.txt* and *file2.txt* into a new file *file3.txt*.

3. **del** – remove files

del is used to remove files. In order to remove a file you must have write permission in its directory, but it is not necessary to have read or write permission on the file itself.

del *file1*

will delete the file *file1*. If you use:

del *A**

you will delete all the files that start with A in the current directory.

del file1 file2 dir1

deletes both files with a single command.

4. **dir** – list names of files in a directory

dir lists the contents of a directory, and can be used to obtain information on the files and directories within it.

dir *dir1*

lists the names of the files and directories in the directory *dir1*. If no directory is named, **dir** lists the contents of the current directory.

dir *dir1 /a*

will list all the contents of *dir1*.

dir *dir1 /b*

will list the contents of *dir1* in bare format, which removes the typical header and footer information, as well as all the details on each item, leaving only the directory name or file name and extension.

dir *dir1 /d*

will list only directories in *dir1*.

5. **find** – searches files for a specified string or expression

find searches for a string in a file or files, and displays lines of text that contain the specified string.

find "string" file1.txt

returns lines where the string appears in *file1.txt*.

find "string" *.txt

returns the lines for all the text files that contain the string.

find /i "string" *.txt

adding /i in front of string makes the search case insensitive.

6. **mkdir** – make a directory

mkdir is used to create new directories. In order to do this you must have write permission in the parent directory of the new directory.

mkdir newdir

will make a new directory called *newdir*.

7. **rename** – renames a file or a directory

rename is used to rename files.

rename file1 file2

will rename the *file1* into *file2*. File names can be input in the form of full pathname as well.

8. **rmdir** – deletes a directory

rmdir is used to delete existing directory.

rmdir dir1

will remove an empty *dir1*. If the directory is not empty and you would like to remove it anyway use **rmdir dir1 /s**

instead. Additional flag **/q** will execute the command in quiet mode. The **/q** flag works only if **/s** is also specified.