Introduction to programming for astronomers (fall 2017)

Thijs Coenen   Folkert Huizinga

October 16, 2017

Contents

Chapter 1

Introduction

In astronomy, and outside of it, programming is becoming an increasingly important skill. The general availability of lots of compute power creates new opportunities for astronomy. There are three general areas in astronomy where computers play an increasingly important role. First, the latest generations of observatories produce high data rates that at times need to be searched in real time for interesting signals. Second, there is a move to more open data sharing and public archiving of observational data, which creates opportunities for data mining. Third the availability of massive amounts of computational power allows increasingly detailed astrophysical simulations to be performed.

1.1 Basic Linux and Coding for Astronomy & Astrophysics

The aim of the course *Basic Linux and Coding for AA (5214BLCF3Y)* is to to get you up and running in Linux with some basic knowledge of its command line interface and teach you the basics of programming. This course will use the Python programming language (more specifically Python 2.7 which, despite Python 3 being available, is still the most prevalent in research). Python was chosen for its relatively straightforward syntax, its free availability, the wide range of libraries in its "eco system". For the fields physics and astronomy in particular, Python provides a large number of libraries for scientific computing and is displacing packages like IDL[1] and Matlab[2].

The Linux part of our course will teach you some of the backgrounds and commands that you will need to be productive. More specifically:

- You should attain basic knowledge of the UNIX/Linux file system layout and the use of permissions on UNIX/Linux to control file access.

- You should be able to access the command line (we will be using the BASH shell throughout this course).

- You should be able to manage running processes on a Linux system, to start or stop programs and to find out which processes are running.

- You should be able to navigate the UNIX/Linux file system and find files and programs.

- You should be able to manipulate text files through the command line and search for specific content.

The programming part of this course aims to teach you the basics of programming in general and programming in Python in particular. Because of the limited extent of this course, we will focus our attention on a practical subset of the Python language — and test whether you attain working knowledge of that subset. Specifically:

---

[1] http://www.exelisvis.com/ProductsServices/IDL.aspx
[2] http://www.mathworks.com/products/matlab/

- You should be able to design and implement a properly structured Python program written in the standard Python style. You will also learn to properly document you program, use proper variable and function naming, how to modularize you program using functions, and how to make your program reusable.

- You should be able to read and write simple text data files.

- You should be able create basic plots using Matplotlib.

- You should be familiar with Numpy and Scipy and be able to use these libraries to implement simple scientific calculations efficiently.

A further more general goal is that you develop problem solving skills using programming tools.

- Learn how to find help when programming. That includes how to read the documentation available with Python and its libraries, and which resources are available on-line.

- Develop the skills to, when presented with new data files, inspect that data, create plots and write small tools to work with them.

1.2   Conventions

This reader uses a number of typographical conventions differentiate normal text, programming code snippets, commands you enter and program output. In running text you may come across monospaced examples or names. When these examples are underlined, `like this`, they are (partial) commands or snippets you may enter. When programs require key presses that are not really commands (not followed by an enter), they are type set as for instance $\boxed{\text{q}}$ or $\boxed{\text{control}} + \boxed{\text{z}}$ if two keys must be pressed at the same time. Program output and names of programs are typeset `like this`. Examples of shell usage, interactive Python sessions and Python source code are below. First a shell session:

```
Gretchen:~ thijscoenen$ cd coolstuff/
Gretchen:coolstuff thijscoenen$ ls
Whatever        alice.txt        blah.txt        hello              noperm
Gretchen:coolstuff thijscoenen$ head alice.txt
ALICE'S ADVENTURES IN WONDERLAND

Lewis Carroll


CHAPTER I. Down the Rabbit-Hole

Alice was beginning to get very tired of sitting by her sister on the
bank, and of having nothing to do: once or twice she had peeped into the
book her sister was reading, but it had no pictures or conversations in
Gretchen:coolstuff thijscoenen$
```

The second example is an interactive Python session:

```
>>> l = range(10)
>>> print l[:5]
[0, 1, 2, 3, 4]
>>> 1 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
>>>
```

Here the prompt $>>>$ is followed on the same line by some Python code and followed by some output from that command on the following line(s). The last example is Python source code:

```python
#!/usr/env/bin python
'''
Print a custom greeting.
'''


def say_hello(person):
    print 'Hello {}'.format(person)

if __name__ == '__main__':
    say_hello('Jan Janssen')
```

1.3   Contents of this reader

This reader consists of 3 parts. The first part explains the basics of the Linux command line, the basics of Python and some libraries useful for scientific computing. The second part contains a number of assignments that will be used during the tutorials. Finally, the third part (the appendix) contains some extra background information.

Part I

Reading

Chapter 2

Linux and its Command Line

In this chapter I will quickly introduce you to the very basics of using Linux through its command line interface. After a little history of Linux and operating systems like it, I will explain how you get access to the command line and go over some of the commands you will need. This chapter can be skipped safely if you are already familiar with the Linux command line. The examples in this chapter use the so called Bourne Again Shell (BASH). This shell comes standard with many flavors of Linux or UNIX.

2.1   UNIX and Linux

UNIX is an operating system developed in the early 1970s at AT&T Bell Labs. Unlike some of its contemporaries UNIX was designed as a multi user system from the start, it used a hierarchical file system (see 2.1.2) and consisted of many small programs that could be combined to perform complex operations. UNIX became popular in academia, where UNIX like systems are still used. In the early 1990s Linus Torvalds developed a new UNIX like operating system that he could run on his own PC. This operating system became known as Linux and, while initially developed by volunteers and hobbyists, it was soon picked up by businesses. Nowadays a large fraction of Linux' development is done by very large computer companies like Google, IBM and Samsung. Apple's Mac OS X is another popular UNIX version.

Currently Linux is used on computers ranging from mobile phones (e.g. Google's Android) to the largest super computer clusters. It furthermore runs a large fraction of the Internet's infrastructure (e.g. routers and web servers). Because UNIX was developed before graphical displays became generally available, its earliest interfaces were all text driven. Although graphical user interfaces are available for UNIX and Linux, textual interface persists. This textual user interface is also called a *command line* interface and on UNIX is provided through a so-called *shell*. Several different shells are available, but this chapter will only explain the basics of the Bourne Again Shell (BASH).

2.1.1   Users and Permissions

Because UNIX was designed as a multi user operating system, it has a notion of users. Associated with every user are a level of access to the operating system. The so-called `root` user has full access to a UNIX system while the other users have more restricted access. On your own computer you will likely be able to log in as `root` while on shared computers you will only have access to your own files. Do note that even if you can log in as `root`, it is absolutely a bad idea to do so for day-to-day work. Since the `root` is all-powerful, a mistake while logged in as `root` can be much worse than a mistake while logged in as an ordinary user. Furthermore the software that you use may contain flaws or security problems that become a problem when that software (running as `root`) has full access to the computer. Furthermore each user is also assigned to a group of users that have the same system privileges.

UNIX controls access to files and directories based on ownership and so-called *permissions*. Each file and directory on a UNIX system has an *owner* (one of the users on that system) and permissions. UNIX has three permissions: *read, write* and *execute*. Read permission is needed to be able to view the contents of that file. Write permission is needed on directories to create files in that directory. For files write permission is needed to change or erase them. Execute permission is needed on directories to be able to view their contents and on files to execute them. Permission may also be set at the *group* level or for all *other* users with access to a computer system (see Section 2.2.12).

2.1.2   The File System

UNIX uses a hierarchical file system meaning that directories in it can be arbitrarily nested. Each directory can contain files or sub directories — the former directory is also referred to as *parent* while the latter are referred to as *children*. The directory hierarchy is a tree with the directory at the base of that tree referred to as *root*, denoted as `/`.

When specifying locations, or *paths*, in a file system two different notations may be used. *Absolute paths*, on the one hand, describe the location of a file or directory in absolute terms, i.e. without reference to some other location on the file system. *Relative paths*, on the other hand, specify a location in the file system relative to some other location in the file system (usually the current location of the user). Absolute paths start at the root, i.e. they start with `/`, while relative paths do not.

Because a UNIX system may be shared between many users, each user has his or her own directory to store files in. This special directory is called a *home directory*. Home directories can be found in `/home` on Linux systems and in `/Users` for Mac OS X systems. E.g. my user (`thijscoenen`) has the home directory `/Users/thijscoenen` on a Mac OS X system and `/home/thijscoenen` on a Linux system. The UNIX(-like) operating systems have very specific file system layouts, with some directories assigned to programs or even parts of the operating system itself. Most directories that reside directly in the root directory are in fact system directories and as a normal user you will rarely need access to them. Below, as an example, the contents of my Mac OS X machine's root directory none of which is part of my own files or directories[1]

```
Gretchen:coolstuff thijscoenen$ ls /
Applications    System          cores           mach_kernel     tmp
Developer       Users           dev             net             usr
Library         Volumes         etc             private         var
Network         bin             home            sbin
Gretchen:coolstuff thijscoenen$
```

UNIX also allows so-called *hidden files* that are normally invisible. Any filename or directory name that starts with a dot will be hidden. Hidden files are generally used for settings or for those files that a user does not need access to directly (only through some program). E.g. the settings for the BASH shell are kept in a file called `.bash_profile` or `.bashrc` in your home directory.

2.2   The Command Line

2.2.1   Accessing the Shell

The command line interface, provided by a shell program, is generally accessible through a terminal emulator program. The name terminal derives from the simple computers (terminals) that were used in the past to access large shared computer systems. When you start a terminal emulator you are generally dropped into your home directory. On Mac OS X you can use the "Terminal" program, while on a Linux systems with KDE you use "Konsole", on Linux systems with Gnome "GNOME Terminal" and on Linux systems with Unity the program is also called "Terminal". This

---

[1]The example shows some directories that are Mac OS X specific: `Applications`, `Developer`, `Library`, `Network`, `System` and `Volumes`.
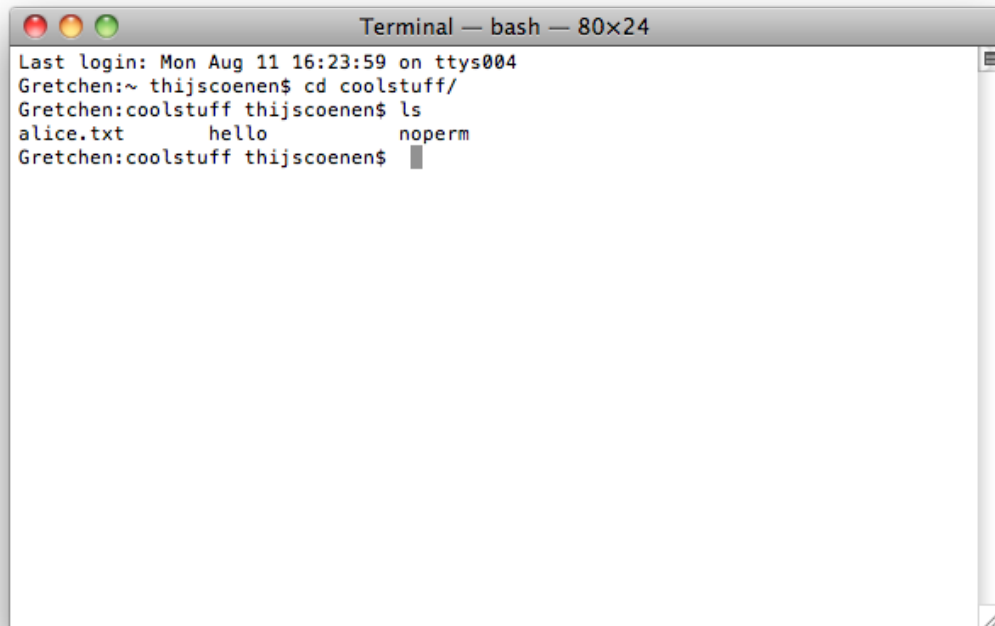
Figure 2.1: A screenshot showing Mac OS X's built-in Terminal program. This program allows you to access the underlying BASH shell on Mac OS X.

chapter will only explain the basics of the BASH shell as it is standard on many Linux systems and on all recent versions of Mac OS X.

2.2.2 Navigating the File System

Assuming you have access to the command line, you will be presented with a so called prompt. What the prompt looks like depends on the version of UNIX or Linux you are using and how it was configured. In my case the command prompt looks like[2]:

```
Gretchen:~ thijscoenen$
```

You can issue your commands by typing them followed by a press of the return key. One of the first things you may ask yourself, is where am I? Usually when you start a new command line session you will be dropped off in your user's home directory. To find out where in the file system you are you use the `pwd` (print working directory) command. The following example shows you what the output looks like for me:

```
Gretchen:~ thijscoenen$ pwd
/Users/thijscoenen
Gretchen:~ thijscoenen$
```

The home directory in these examples has a subdirectory called `coolstuff` that we will access. To change directories to the `coolstuff` directory use the `cd` (change directory) command. In the following example a `pwd` command issued as well to show we have actually changed directory:

---

[2]What your prompt looks like exactly depends on the settings of your shell and is a matter of taste. My computer shows `computername:directory username$`, with `Gretchen` being the name of my computer.

```
Gretchen:~ thijscoenen$ cd coolstuff
Gretchen:coolstuff thijscoenen$ pwd
/Users/thijscoenen/coolstuff
Gretchen:coolstuff thijscoenen$
```

The eagle eyed among you may have noticed the ~ (tilde) appearing in the shell session examples. This is shorthand for your home directory, and your shell will expand it to your full home directory. You can use the tilde in commands to shorten them. To return to your home directory use the cd ~ command.

What does the coolstuff directory actually contain? To get a listing of the contents of a directory you can use the ls command. This command takes many options but the easiest way of using it is as follows:

```
Gretchen:coolstuff thijscoenen$ ls
alice.txt        hello               noperm
Gretchen:coolstuff thijscoenen$
```

Hidden files on UNIX systems have names that start with a dot. There are two special directories that are always present in every directory, the single dot and double dot directories. The former . is shorthand for the current directory and the latter .. is shorthand for the parent directory. By passing the -a option to ls you make hidden files visible.

```
Gretchen:coolstuff thijscoenen$ ls -a
.               ..              alice.txt      hello               noperm
Gretchen:coolstuff thijscoenen$ cd ..
Gretchen:~ thijscoenen$ pwd
/Users/thijscoenen
Gretchen:~ thijscoenen$
```

The ls command has many options, but several are worthy of mention. First, the -l option will list the directory contents with one file or directory per line and also show the size (in bytes) and permissions for each entry. Second, the -h option will show the file sizes in human readable format (so not in bytes for large files, see below for an example).

```
Gretchen:coolstuff thijscoenen$ ls -l
total 312
-rw-r--r--  1 thijscoenen  staff  147731 22 jul 23:27 alice.txt
-rwxr--r--  1 thijscoenen  staff      43  7 aug 18:02 hello
-rw-r--r--  1 thijscoenen  staff      50 11 aug 14:35 noperm
Gretchen:coolstuff thijscoenen$ ls -lh
total 312
-rw-r--r--  1 thijscoenen  staff    144K 22 jul 23:27 alice.txt
-rwxr--r--  1 thijscoenen  staff     43B  7 aug 18:02 hello
-rw-r--r--  1 thijscoenen  staff     50B 11 aug 14:35 noperm
Gretchen:coolstuff thijscoenen$
```

Another nice command is ls -lrt which will list all files and sorted by the time of last access (oldest to newest). This is handy when you want to find the most recently accessed file in a directory as it will be at the end of the ls output.

The ls command can furthermore be used to list the content of a specific directory. Using the previous example I can list the contents of the coolstuff directory from my home directory using the command ls coolstuff. When you want to recursively list the contents of some directory you can use the ls -R command. It shows the content of the current directory, the content of the subdirectories and the content of the subdirectories of each sub directory etc. Because this command produces a lot of output I suggest you try it for yourself.

2.2.3 Running Programs

The shell can only start programs it can find, and the directories where the shell will look for programs are listed in an environment variable (see Section 2.2.13 for an explanation) called `$PATH`. For programs that are on the `$PATH`, i.e. in one of the directories listed by the `$PATH` variable, you can just type their name and press enter (even if the program is in a different directory than you are). The programs `ls` and `cd` are on the `$PATH` (on my system they actually reside in the `/bin` directory). When you try to run a non-existent program or a program that your shell cannot locate, you will see an error like the following:

```
Gretchen:coolstuff thijscoenen$ ls
alice.txt          hello                    noperm
Gretchen:coolstuff thijscoenen$ nosuchthing
-bash: nosuchthing: command not found
Gretchen:coolstuff thijscoenen$
```

This will even happen for programs present in the same directory as you — unless that directory happens to be on the `$PATH`. To remedy this problem, prepend the program name with `./` (this tells your shell to look in the current directory).

```
Gretchen:coolstuff thijscoenen$ hello
-bash: hello: command not found
Gretchen:coolstuff thijscoenen$ ./hello
Hello world!
Gretchen:coolstuff thijscoenen$
```

You may not have sufficient permissions to run just any program on UNIX. To check whether you have permissions to run the program you can just attempt to run it, your shell will report an error if you have insufficient permissions.

```
Gretchen:coolstuff thijscoenen$ ./noperm
-bash: ./noperm: Permission denied
Gretchen:coolstuff thijscoenen$
```

All the previous examples were for programs that run for a short time, if you have long running programs you may want to continue issuing new commands without waiting for some program to finish. You could just start a second (or third etc.) terminal and continue issuing commands in that new terminal. The BASH shell however allows you to start programs in the background so that it is possible to keep issuing commands in that same shell. To start a program in the background just add a space and ampersand `&` to the command. You can move a program from the background to the foreground with the `fg` command:

```
Gretchen:background-demonstration thijscoenen$ ls
runs-1-minute
Gretchen:background-demonstration thijscoenen$ ./runs-1-minute &
[1] 55868
Gretchen:background-demonstration thijscoenen$ pwd
/Users/thijscoenen/background-demonstration
Gretchen:background-demonstration thijscoenen$ fg
./runs-1-minute
One minute passed
Gretchen:background-demonstration thijscoenen$
```

As you can see in the example above a number is shown in the shell after the program was started using the `&`. This number is the process number of the program just started, in Section 2.2.7 these process numbers will be explained. If you started a program in the foreground, but want to move it to the background you first suspend the program and then move it to the background. Suspend the program by pressing [Control] + [z] and then move it to the background by issuing the `bg` command.

```
Gretchen:background-demonstration thijscoenen$ ./runs-1-minute
^Z
[1]+  Stopped                    ./runs-1-minute
Gretchen:background-demonstration thijscoenen$ bg
[1]+ ./runs-1-minute &
Gretchen:background-demonstration thijscoenen$ pwd
/Users/thijscoenen/background-demonstration
Gretchen:background-demonstration thijscoenen$ fg
./runs-1-minute
One minute passed
Gretchen:background-demonstration thijscoenen$
```

In this example you can see that the ⎡Control⎤ + ⎡z⎤ key presses are shown in the shell as ^Z. It is also possible to terminate a program running in the shell by pressing ⎡Control⎤ + ⎡c⎤. Some programs may not react to an attempt to terminate it this way and you may have to resort to the `kill` command described below in Section 2.2.7.

2.2.4   Getting Help

While this reader will get you started using Linux it cannot possibly explain all features of BASH or all programs available by default on a UNIX machine. Fortunately a large amount of documentation is available, some of it accessible directly from the command line and some of it on line. Most command line programs have simple built in help that is usually displayed when no options are specified on the command line or through options as `-?`, `-h` or `--help`.

While many programs provide some simple documentation about themselves UNIX provides a standard command line documentation viewer called `man`. To get help about a specific program just type the command `man program-name`, where you should replace `program-name` with the program that you want help for. Figure 2.2 shows the help for `ls` (accessed by typing `man ls`). The documentation can be navigated using the ⎡up⎤ and ⎡down⎤ arrows, to move up and down by a line, and ⎡space⎤ to jump a full page. The documentation can also be searched by typing a forward slash ⎡/⎤ followed by the word you are looking for and then ⎡return⎤. When there are several matches for a search you can jump to the next match by pressing ⎡n⎤ (or in reverse with ⎡N⎤). The viewer can be quit by typing a ⎡q⎤. More information about navigating in `man` can be accessed by pressing ⎡h⎤ while it is running. The documentation available through `man` are also called "man pages". Some software will be documented using `info`, try running that like you would `man` in case no man pages are available (because the documentation may be written in `info` format).

On line there are several good resources for help with UNIX problems, that can be very helpful because the man pages are quite technical and at times hard to read. The very basic UNIX commands tend to have Wikipedia pages with examples. Stack overflow, a community site about programming problems (see http://stackoverflow.com), can be helpful as can be the related Stack Exchange site about UNIX and Linux (see http://unix.stackexchange.com). A nice site that can explain some shell commands is Explain Shell (see http://explainshell.com), it allows you to cut and paste a command and it then shows you an explaination of it. The Linux Documentation Project (see http://www.tldp.org) is also useful. A note about using Google to find documentation on the Internet: not everything you find will be correct so refrain from just copying whatever you find!

2.2.5   Copying, Moving and Removing Files

To copy files, or directories use the `cp` command. The general shape of the command is `cp source destination`. If the `source` is a file and the `destination` is also a filename the contents of `source` will be copied over the `destination` file. If `destination` does not exist yet `source` will be copied to a file with the specified destination as name. If `destination` is a directory the `source` file will be copied to it. If the source is a directory itself the copying will fail (these examples use `cat` to show file contents):

```
 ● ● ●                  Terminal — less — 80×24
LS(1)                     BSD General Commands Manual                    LS(1)

NAME
     ls -- list directory contents

SYNOPSIS
     ls [-ABCFGHLOPRSTUW@abcdefghiklmnopqrstuwx1] [file ...]

DESCRIPTION
     For each operand that names a file of a type other than directory, ls
     displays its name as well as any requested, associated information.  For
     each operand that names a file of type directory, ls displays the names
     of files contained within that directory, as well as any requested, asso-
     ciated information.

     If no operands are given, the contents of the current directory are dis-
     played.  If more than one operand is given, non-directory operands are
     displayed first; directory and non-directory operands are sorted sepa-
     rately and in lexicographical order.

     The following options are available:

     -@        Display extended attribute keys and sizes in long (-l) output.
:
```

Figure 2.2: The man documentation viewer showing part of the documentation for ls.

```
Gretchen:cp-demonstration thijscoenen$ ls -l
total 16
drwxr-xr-x  2 thijscoenen  staff  68 21 aug 22:40 directory_a
-rw-r--r--  1 thijscoenen  staff  16 21 aug 22:40 file_a.txt
-rw-r--r--  1 thijscoenen  staff  16 21 aug 22:40 file_b.txt
Gretchen:cp-demonstration thijscoenen$ cat file_a.txt
This is file A.
Gretchen:cp-demonstration thijscoenen$ cat file_b.txt
This is file B.
Gretchen:cp-demonstration thijscoenen$ cp file_a.txt file_c.txt
Gretchen:cp-demonstration thijscoenen$ ls -l
total 24
drwxr-xr-x  2 thijscoenen  staff  68 21 aug 22:40 directory_a
-rw-r--r--  1 thijscoenen  staff  16 21 aug 22:40 file_a.txt
-rw-r--r--  1 thijscoenen  staff  16 21 aug 22:40 file_b.txt
-rw-r--r--  1 thijscoenen  staff  16 21 aug 22:41 file_c.txt
Gretchen:cp-demonstration thijscoenen$ cat file_c.txt
This is file A.
Gretchen:cp-demonstration thijscoenen$ cp file_a.txt file_b.txt
Gretchen:cp-demonstration thijscoenen$ cat file_b.txt
This is file A.
Gretchen:cp-demonstration thijscoenen$ cp file_a.txt directory_a/
Gretchen:cp-demonstration thijscoenen$ ls directory_a/
file_a.txt
Gretchen:cp-demonstration thijscoenen$ cp directory_a whatever
cp: directory_a is a directory (not copied).
```

```
Gretchen:cp-demonstration thijscoenen$
```

To copy a directory and all it contains use the -r option of cp:

```
Gretchen:cp-demonstration thijscoenen$ cp -r directory_a/ whatever
Gretchen:cp-demonstration thijscoenen$ ls whatever/
file_a.txt
Gretchen:cp-demonstration thijscoenen$
```

To just move a file to a different filename or location use the mv command. Simple use of mv takes the shape mv source destination. If source is a file and destination does not exist yet source will be moved there (after this operation source will cease to exist). If destination does exist, and is a file, it will be overwritten. If destination is a directory the file source will be moved to that directory. If source itself is a directory it will be moved to destination, with similar rules:

```
Gretchen:cp-demonstration thijscoenen$ ls -l
total 24
drwxr-xr-x  3 thijscoenen  staff  102 21 aug 22:42 directory_a
-rw-r--r--  1 thijscoenen  staff   16 21 aug 22:40 file_a.txt
-rw-r--r--  1 thijscoenen  staff   16 21 aug 22:42 file_b.txt
-rw-r--r--  1 thijscoenen  staff   16 21 aug 22:41 file_c.txt
drwxr-xr-x  3 thijscoenen  staff  102 21 aug 22:45 whatever
Gretchen:cp-demonstration thijscoenen$ mv file_a.txt file_b.txt
Gretchen:cp-demonstration thijscoenen$ ls -l
total 16
drwxr-xr-x  3 thijscoenen  staff  102 21 aug 22:42 directory_a
-rw-r--r--  1 thijscoenen  staff   16 21 aug 22:40 file_b.txt
-rw-r--r--  1 thijscoenen  staff   16 21 aug 22:41 file_c.txt
drwxr-xr-x  3 thijscoenen  staff  102 21 aug 22:45 whatever
Gretchen:cp-demonstration thijscoenen$ mv whatever directory_a/
Gretchen:cp-demonstration thijscoenen$ ls -l
total 16
drwxr-xr-x  4 thijscoenen  staff  136 21 aug 22:57 directory_a
-rw-r--r--  1 thijscoenen  staff   16 21 aug 22:40 file_b.txt
-rw-r--r--  1 thijscoenen  staff   16 21 aug 22:41 file_c.txt
Gretchen:cp-demonstration thijscoenen$ mv directory_a/ file_b.txt
mv: rename directory_a/ to file_b.txt: Not a directory
Gretchen:cp-demonstration thijscoenen$
```

As you can see trying to move a directory to an existing file will fail.
Removing files is done with the rm command. Generally you just use the rm somefile command. If somefile is a directory this will fail because only *empty* directories are removed with rmdir.

```
Gretchen:cp-demonstration thijscoenen$ ls -l
total 16
drwxr-xr-x  4 thijscoenen  staff  136 21 aug 22:57 directory_a
-rw-r--r--  1 thijscoenen  staff   16 21 aug 22:40 file_b.txt
-rw-r--r--  1 thijscoenen  staff   16 21 aug 22:41 file_c.txt
Gretchen:cp-demonstration thijscoenen$ rm file_b.txt
Gretchen:cp-demonstration thijscoenen$ ls -l
total 8
drwxr-xr-x  4 thijscoenen  staff  136 21 aug 22:57 directory_a
-rw-r--r--  1 thijscoenen  staff   16 21 aug 22:41 file_c.txt
Gretchen:cp-demonstration thijscoenen$ rm directory_a/
rm: directory_a/: is a directory
```

```
Gretchen:cp-demonstration thijscoenen$ rmdir directory_a/
rmdir: directory_a/: Directory not empty
Gretchen:cp-demonstration thijscoenen$
```

If you want to remove a directory and all its content (whether files or sub directories) you can use the `rm -rf *`. Note though that this is a very dangerous command, if not the "most dangerous command ever"[3], as it will recursively (the `-r` option) remove everything (matched by the `*` "glob pattern") and skip any questions (the `-f` option). For more information on glob patterns see Section 2.2.9. Note that `rm -rf *` cannot be undone!

2.2.6 Creating Directories

Because in the previous section it was shown how to remove directories you may ask yourself how you create them in the first place. UNIX has the `mkdir` command for that. You can create a directory with the command `mkdir somedirectory` where some directory is the name of the directory to be created. It is also possible to create several directories in one go by listing their names after `mkdir` separated with spaces. If you have to create a directory inside of several that is possible using the command `mkdir -p` followed the full path you want to create.

```
Gretchen:mkdir-examples thijscoenen$ ls
Gretchen:mkdir-examples thijscoenen$ mkdir dir1 dir2 dir3
Gretchen:mkdir-examples thijscoenen$ ls
dir1        dir2        dir3
Gretchen:mkdir-examples thijscoenen$ ls -l
total 0
drwxr-xr-x  2 thijscoenen  staff  68 21 aug 23:28 dir1
drwxr-xr-x  2 thijscoenen  staff  68 21 aug 23:28 dir2
drwxr-xr-x  2 thijscoenen  staff  68 21 aug 23:28 dir3
Gretchen:mkdir-examples thijscoenen$ mkdir -p container/subdir/subdir/whatever
Gretchen:mkdir-examples thijscoenen$ cd container/subdir/subdir/whatever
Gretchen:whatever thijscoenen$ pwd
/Users/thijscoenen/mkdir-examples/container/subdir/subdir/whatever
Gretchen:whatever thijscoenen$
```

2.2.7 Listing and Killing Processes

On UNIX each running program consists of one or sometimes more processes that are identified by *process numbers* often abbreviated to PID. To get an idea of the running programs use the `top` command. This will show you a continually updating list of running processes, sortable by for instance memory use or processor use. When your computer is stuck it is quite often possible to find the offending program by looking through the most active processes in `top`. As with `man` you can exit `top` by pressing q̄. Figure 2.3 shows a screenshot of `top`. You can get a list of the currently running processes using the `ps` command, which unlike `top` will not continually update - it is like an `ls` for processes. Run without options `ps` only shows the processes running in the current shell while you can get a list of all processes running in all shells for a certain user using `ps -u username` (with `username` replaced with relevant username). For example, while logged in on a LOFAR[4] compute node as `coenen`:

```
coenen@locus048:~$ ps
  PID TTY          TIME CMD
14099 pts/2    00:00:00 bash
15518 pts/2    00:00:00 ps
```

---

[3]According to one of my proofreading colleagues.

[4]The Low Frequency Array, a large radio telescope array operating at low radio frequencies that has its core in the Netherlands.

```
●●●                    Terminal — top — 80×24
Processes: 62 total, 2 running, 60 sleeping, 267 threads           17:29:29
Load Avg: 0.13, 0.20, 0.17  CPU usage: 4.24% user, 9.43% sys, 86.32% idle
SharedLibs: 5196K resident, 5648K data, 0B linkedit.
MemRegions: 13931 total, 632M resident, 13M private, 228M shared.
PhysMem: 685M wired, 947M active, 311M inactive, 1942M used, 105M free.
VM: 156G vsize, 1039M framework vsize, 17259915(0) pageins, 6093270(0) pageouts.
Networks: packets: 64936199/83G in, 37656757/3714M out.
Disks: 28389042/396G read, 17131423/460G written.

PID    COMMAND      %CPU TIME      #TH  #WQ  #PORT #MREG RPRVT  RSHRD  RSIZE
58153  screencaptur 0.4  00:00.03 3    2    42    81    232K-  9000K+ 2812K
58151  top          5.8  00:00.33 1/1  0    25    33    984K+  264K   1564K+
58148  bash         0.0  00:00.00 1    0    17    24    392K   744K   1064K
58147  login        0.0  00:00.01 1    0    22    53    496K   312K   1596K
57871  vim          0.0  00:25.76 1    0    17    36    1692K  244K   3028K
57863  bash         0.0  00:00.01 1    0    17    24    408K   744K   1108K
57862  login        0.0  00:00.01 1    0    22    53    488K   312K   1596K
57789  bash         0.0  00:00.03 1    0    17    24    416K   744K   1116K
57788  login        0.0  00:00.01 1    0    22    53    488K   312K   1596K
57747  bash         0.0  00:00.04 1    0    17    24    424K   744K   1116K
57746  login        0.0  00:00.01 1    0    22    53    488K   312K   1596K
57729  bash         0.0  00:00.01 1    0    17    24    396K   744K   1064K
57728  login        0.0  00:00.01 1    0    22    53    488K   312K   1596K
57706  Preview      0.0  00:13.14 2    1    119   277   16M+   27M-   35M
```

Figure 2.3: This screenshot shows top running on a Mac OS X system.

```
coenen@locus048:~$ ps -u coenen
  PID TTY          TIME CMD
14098 ?        00:00:00 sshd
14099 pts/2    00:00:00 bash
15229 ?        00:00:00 sshd
15230 pts/3    00:00:00 bash
15266 pts/3    00:00:00 top
15522 pts/2    00:00:00 ps
coenen@locus048:~$
```

Another useful option to ps is -A which will select all processes, also not necessarily running in a terminal.

The process numbers can be used to terminate unresponsive programs or those that are otherwise stuck. The kill command allows you to terminate processes by process number:

```
Gretchen:background-demonstration thijscoenen$ ./runs-1-minute &
[1] 59380
Gretchen:background-demonstration thijscoenen$ kill 59380
Gretchen:background-demonstration thijscoenen$ fg
-bash: fg: job has terminated
[1]+  Terminated              ./runs-1-minute
Gretchen:background-demonstration thijscoenen$
```

Some processes may still not terminate, you can then send them a stronger message using the -9 (KILL) signal. E.g. to kill process 101 using the KILL signal: kill -9 101.
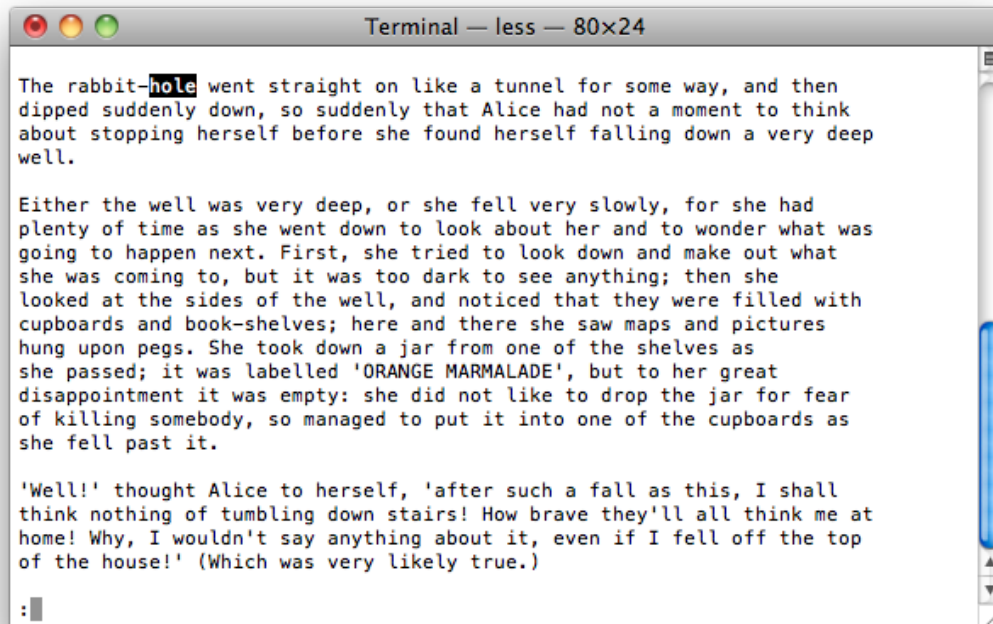
Figure 2.4: The less program showing a text file, you can move through the file using the arrow keys.

2.2.8 Dealing with Text Files

You will inevitably have to work with text files, because, among other reasons, programming source code are text files, many of UNIX's settings are in small text files in your home directory and quite a few scientific data sets are encoded in text files. Luckily UNIX has many tools to work with text files from the command line. The first command is cat which will show the contents of a text file in your terminal. Since cat does not paginate, this will likely cause your terminal to scroll. In the following example I look at the contents of the hello file (which is a Python script and therefore text):

```
Gretchen:coolstuff thijscoenen$ ls
Whatever        alice.txt       hello               noperm
Gretchen:coolstuff thijscoenen$ cat hello
#!/usr/bin/env python
print "Hello world!"
Gretchen:coolstuff thijscoenen$
```

If the file you are working with is large, then using cat to look at its content is inconvenient. By using the more or less commands you will be presented with a so-called pager that allows you to view and move around in the text file without scrolling. The less command allows you to move in both directions through the file using the up and down arrow keys. You can search for words by pressing a forward slash ⃞/⃞ followed by the word you are looking for and then ⃞enter⃞. If there are several matches you can move between them using ⃞n⃞ (or ⃞N⃞ for the reverse direction). Help is accessible by pressing ⃞h⃞ and you can quit less by pressing ⃞q⃞. This is very similar to the man works because it actually uses less to show the actual man page (visible in the title bar of the window in Figure 2.2).

If you are only interested in a quick peek at the contents of some text file, you can use the `head` or `tail` commands to, respectively, look at the first few lines or last few lines of a text file. Similarly if you want to find out the number of words or lines in a file the `wc` (word count) command is useful, without options it shows the number of lines, the number of words and the number of bytes in the file.

```
Gretchen:coolstuff thijscoenen$ head alice.txt
ALICE'S ADVENTURES IN WONDERLAND

Lewis Carroll


CHAPTER I. Down the Rabbit-Hole

Alice was beginning to get very tired of sitting by her sister on the
bank, and of having nothing to do: once or twice she had peeped into the
book her sister was reading, but it had no pictures or conversations in
Gretchen:coolstuff thijscoenen$

Gretchen:coolstuff thijscoenen$ wc alice.txt
    3335   26444  147731 alice.txt
Gretchen:coolstuff thijscoenen$ wc -l alice.txt
    3335 alice.txt
Gretchen:coolstuff thijscoenen$
```

The last example (the `wc -l alice.txt` command) shows you how to count only the number of lines in a file. A oft used file format to encode tabular data is a so called Comma Separated Values (CSV) file (white space or tab separation are also used frequently). The lines in these files generally correspond to one data point, you can thus get a quick feel for the size of the data set by using `wc -l` on that file.

So far we have only treated looking through files, not editing or creating them. Many different text editors are available on UNIX, spanning the gamut from editors that run in your terminal and use arcane commands to more modern text editors with a graphical user interface. Note though that in this context modern does not always mean more efficient! The choice of editor is personal and there is not enough space in this reader to explain how to use them all. Instead, I have added a chapter to the appendix that lists some good text editors (see Appendix **??**). For now I will just mention that you will want a text editor that is focused on programmers as they allow you to work more efficiently.

To search a file for a certain piece of text you can use the `grep` command. To find out on which lines the Cheshire cat appears in the text we use `grep` with the `-n` option to print line numbers and `-i` option to perform case insensitive matches:

```
Gretchen:coolstuff thijscoenen$ grep -ni "Cheshire" alice.txt
1313:'It's a Cheshire cat,' said the Duchess, 'and that's why. Pig!'
1319:'I didn't know that Cheshire cats always grinned; in fact, I didn't know
1433:was a little startled by seeing the Cheshire Cat sitting on a bough of a
1440:'Cheshire Puss,' she began, rather timidly, as she did not at all know
2068:'It's the Cheshire Cat: now I shall have somebody to talk to.'
2100:'It's a friend of mine--a Cheshire Cat,' said Alice: 'allow me to
2144:When she got back to the Cheshire Cat, she was surprised to find quite a
```

If you want more context to each match use the `-B` option to specify how many lines *before* a match should also be printed and `-A` for the number of lines *after* a match (output truncated after the first two matches to save space):

Gretchen:coolstuff thijscoenen$ grep -B 1 -A 2 -i "*Cheshire*" alice.txt

'It's a Cheshire cat,' said the Duchess, 'and that's why. Pig!'

She said the last word with such sudden violence that Alice quite
--

'I didn't know that Cheshire cats always grinned; in fact, I didn't know
that cats COULD grin.'

--

Each match is separated by a line with two dashes --. If you are only interested in knowing whether a file contains a match you can use the -H option to print the filename and the matching text:

Gretchen:coolstuff thijscoenen$ grep -Hi "*Cheshire*" *.txt
alice.txt
Gretchen:coolstuff thijscoenen$

grep is a very powerful tool that can also match patterns, it is useful to read through the man pages to see what more it can do for you.

2.2.9 Glob Patterns

The BASH shell supports filename expansion, that is it will expand special wildcard symbols and match them to filenames. This is also called *globbing*, and the pattern that is matched *glob pattern*. I will first explain the parts that make up a glob pattern, then show you how you use them with BASH.

* Matches any text in filename.

? Matches zero or one unspecified character.

[ABC] Allow one character from a specified set of characters to be matched, in this case the characters A, B and C make up that set. Will match a single character only, not ABC.

[0-9] Will match a number in the range 0 through 9.

[a-z] Match any lower case character in the range a through z.

[A-Z] Match any upper case character in the range A through Z.

[A-Za-z] Match any upper case character in the range A through Z or any lower case character in the range a through z.

\ Escape character, use this if you should match one of the characters that have a special meaning in a glob pattern. E.g. if you want to match the *, your glob pattern should include * so that the * itself matched and not expanded.

! This will negate a pattern. E.g. if you want to match anything not a lower case character use ![a-z].

Glob pattern can then be used in combination with other programs, what follows is a simple example that shows how ls can be combined with some glob patterns to look for some specific files.

```
Gretchen:glob-demonstration thijscoenen$ ls
101.txt                604.txt                ABC.txt
123.txt                709.dat                ABC123.txt
Gretchen:glob-demonstration thijscoenen$ ls [0-9]*
101.txt         123.txt         604.txt         709.dat
Gretchen:glob-demonstration thijscoenen$ ls [!0-9]*
ABC.txt                ABC123.txt
Gretchen:glob-demonstration thijscoenen$ ls [5-9]*
604.txt         709.dat
Gretchen:glob-demonstration thijscoenen$ ls *dat
709.dat
Gretchen:glob-demonstration thijscoenen$ ls 1[2468]*
123.txt
Gretchen:glob-demonstration thijscoenen$
```

2.2.10   Finding Files

A common problem is, how do I find some specific file? The naive approach, use `ls` and move around the file system until you find what you were looking for, is time consuming and . Beyond the glob patterns that your shell provides (see Section 2.2.9) the UNIX `find` command is useful. With `find` you can look through a directory hierarchy for filenames matching a certain pattern or certain properties. You can then run some commands for each file that was found. At its simplest you can use `find` to look for a certain file called `needle.txt` in this example:

```
Gretchen:find thijscoenen$ find . -name "needle.txt"
./haystack/hay/hay/hay/needle.txt
./haystack/straw/needle.txt
Gretchen:find thijscoenen$
```

As you can see two files were found, but what if we know that the file we need contains the word `gold`? It turns out you can run other programs on each found file using the `-exec` option of `find`. In the following example `grep` is used to check whether a match contains `gold`:

```
Gretchen:find thijscoenen$ find . -name "needle.txt" -exec grep -Hi "gold" {} \;
./haystack/straw/needle.txt
Gretchen:find thijscoenen$
```

It is now clear that the file we needed was `./haystack/straw/needle.txt`. In the commands that follow the `-exec` each occurrence of {} is replaced with the matching file name and each command must be followed by \;. The `grep` command was explained in Section 2.2.8.

2.2.11   `gzip` and `tar`

To conserve disk space it is sometimes a good idea to pack files more densely, there are several tools to do so. If you have used Windows so called zip files may be familiar. Although utilities to handle zip files exist (`zip` and `unzip`) on UNIX, you are more likely to come across `.gz` or `.bz2` files. Below I give examples of how to deflate and reinflate a file using `gzip`, note that I use a so-called pipe `|` and `grep` to only show filenames containing `alice`. Pipes are a way of sending output from one program to another, which are explained in Section 2.2.14. The following example shows that the Alice in Wonderland story can be packed to only about 36 % of its original size:

```
Gretchen:coolstuff thijscoenen$ ls -l | grep alice
-rw-r--r--  1 thijscoenen  staff  147731 22 jul 23:27 alice.txt
Gretchen:coolstuff thijscoenen$ gzip alice.txt
Gretchen:coolstuff thijscoenen$ ls -l | grep alice
```

```
-rw-r--r--  1 thijscoenen  staff  53596 22 jul 23:27 alice.txt.gz
Gretchen:coolstuff thijscoenen$ gunzip alice.txt.gz
Gretchen:coolstuff thijscoenen$ ls -l | grep alice
-rw-r--r--  1 thijscoenen  staff  147731 22 jul 23:27 alice.txt
Gretchen:coolstuff thijscoenen$
```

The `bzip2` and `bunzip2` programs work similarly and may achieve a slightly higher compression than `gzip`, but the latter is used more often.

When you download software or data you will come across so-called tarballs, which are files that can encapsulate many separate files or even directory full hierarchies. The name tarball derives from the `tar` utility that can create or unpack them. The name `tar` itself derives from tape archive (archiving to tape was and still is cheaper than archiving to disk). The example that follows shows how you create (`-c` option) a tarball that is compressed using `gzip` (`-z` option) how you can list its contents (`-t` option) and how you can extract it again (`-x` option):

```
Gretchen:coolstuff thijscoenen$ ls
Whatever        alice.txt       hello           noperm
Gretchen:coolstuff thijscoenen$ tar -czf cool.tar.gz *
Gretchen:coolstuff thijscoenen$ tar -tzf cool.tar.gz
Whatever
alice.txt
hello
noperm
Gretchen:coolstuff thijscoenen$ rm Whatever alice.txt noperm hello
Gretchen:coolstuff thijscoenen$ ls
cool.tar.gz
Gretchen:coolstuff thijscoenen$ tar -xzf cool.tar.gz
Gretchen:coolstuff thijscoenen$ ls
Whatever        cool.tar.gz     noperm
alice.txt       hello
Gretchen:coolstuff thijscoenen$ rm cool.tar.gz
Gretchen:coolstuff thijscoenen$
```

Note that some tarballs are compressed using `bzip2` in which case you can use the `-j` option in stead of `-z`.

2.2.12   Permissions Continued

In Section 2.1.1 users and their permissions were introduced. UNIX has several commands that let you manage file permissions and ownership. UNIX also allows a user to temporarily acquire more privileges. We will start by demonstrating how file permissions can be changed. The owner of a file and a user with root privileges can change its permissions. The first step is to check the permissions, run for example `ls -l` to get a file listing that shows the permissions in the first column of the output.

```
Gretchen:chmod-demo thijscoenen$ ls -l
total 16
-rw-r--r--  1 thijscoenen  staff  68 27 aug 22:02 demo.py
----r--r--  1 thijscoenen  staff  15 27 aug 22:00 nopriv.txt
drwxr-xr-x  2 thijscoenen  staff  68 27 aug 22:12 subdir
```

The first column can be further split, taking `demo.py` as an example we get the file type `-`, owner permissions `rw-`, group permissions `r--` and finally the permissions for everyone else `r--`. The file type can be: `-` for a file, `d` for a directory and `l` for a "symbolic link". In our example we are dealing with a normal file. The user permissions are listed in the order read, write and execute using

the abbreviations rwx. If a permission is not granted the permission will be replaced with a dash. In our example the owner of the file has read and write permissions but no execute permissions: rw-. The group permissions in the example are only read permissions: r--. All other users are also only able to read the file: r--. The second column in the output of ls -l is not interesting for our current purposes, but the third is: it shows the owner of the file. In this case that user is thijscoenen. The following column shows the group that the file is assigned to, in this case the group staff. The next column shows the file size in bytes, followed by a column that shows the last time the file was changed. The last column shows the actual file or directory name.

To change the permissions of the user, group and all other users the file owner (or a sufficiently privileged user like root) can use chmod. This command takes as options a string like ugo+rwx that specifies which user should gain or lose which permissions and finally as arguments the files for which the permissions should be changed. The options should be read as: the owner, the group and all others (ugo), gain (+) the permissions to read, write and execute (rwx). If we want to make the demo.py file executable for its owner, the following command will do the trick:

```
Gretchen:chmod-demo thijscoenen$ chmod u+x demo.py
Gretchen:chmod-demo thijscoenen$ ls -l
total 16
-rwxr--r--  1 thijscoenen  staff  82 27 aug 22:29 demo.py
----r--r--  1 thijscoenen  staff  15 27 aug 22:00 nopriv.txt
drwxr-xr-x  2 thijscoenen  staff  68 27 aug 22:12 subdir
Gretchen:chmod-demo thijscoenen$
```

If we want to disallow other users to see what is in the subdir directory we can remove (-) the execute permissions on that directory for its group and all other users:

```
Gretchen:chmod-demo thijscoenen$ chmod go-x subdir/
Gretchen:chmod-demo thijscoenen$ ls -l
total 16
-rwxr--r--  1 thijscoenen  staff  82 27 aug 22:29 demo.py
----r--r--  1 thijscoenen  staff  15 27 aug 22:00 nopriv.txt
drwxr--r--  2 thijscoenen  staff  68 27 aug 22:12 subdir
Gretchen:chmod-demo thijscoenen$
```

You can also specify several files by just typing their names or by using the appropriate glob patterns. To recursively change the permissions of some directory hierarchy use the -R option, for example to change the permissions on subdir and everything it contains to be completely open to everyone: chmod -R ugo=rwx subdir. In this last command we use = to just set the permissions. Old examples explaining chmod may show numerical permissions, but they are not necessary to manage the permissions and will not be treated here.

To change the owner of a file use the chown command. Using requires that you are either the owner or are otherwise sufficiently privileged. To change the owner of demo.py from the examples to root use the following command:

```
Gretchen:chmod-demo thijscoenen$ chown root:staff demo.py
chown: demo.py: Operation not permitted
Gretchen:chmod-demo thijscoenen$ sudo chown root:staff demo.py
Password:
Gretchen:chmod-demo thijscoenen$ ls -l
total 16
-rwxr--r--  1 root         staff  82 27 aug 22:29 demo.py
----r--r--  1 thijscoenen  staff  15 27 aug 22:00 nopriv.txt
drwxrwxrwx  2 thijscoenen  staff  68 27 aug 22:12 subdir
```

As you can see there is a small hitch, the user thijscoenen cannot change the ownership to root and group staff, because that user is not privileged enough. The second time chmod is preceded with

sudo, which means execute the command that follows as a super user (root on most systems). To prove that you in fact are allowed to acquire root privileges you are asked to enter the password that belongs to that user. After acquiring the privileges chown to root becomes possible. While this example is a bit silly sudo is very important when installing software. A lot of software is installed system wide and that means somewhere in the system directories of UNIX. Those directories are not writable for normal user, a problem that is solved with sudo. Note: if you are using a university computer you will most likely not have the ability to use sudo, because most system administrators will not give out root privileges to just anyone — on your own Mac or Linux machine that should be no problem.

2.2.13   Environment Variables

Some of the settings of your BASH shell are put in so-called *environment variables*. A *variable* is a name (or an identifier) associated with some stored value that can be changed. We already came across an environment variable called $PATH, its value is a list of directories that are searched for executables. The BASH echo $PATH command will echo (show) the value of the variable $PATH, the $ is needed to signify that the following name is a variable (the example below shows what goes wrong without the $):

```
Gretchen:~ thijscoenen$ echo $PATH
/Library/Frameworks/Python.framework/Versions/2.7/bin:/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:
/usr/texbin:/usr/X11/bin
Gretchen:~ thijscoenen$ echo PATH
PATH
Gretchen:~ thijscoenen$
```

As you can see $PATH contains multiple directories separated with a colon :. When you change $PATH you should know that the directories are searched for a executable program in the same order as they are specified in $PATH.

To set a variable to a certain value use the export command, e.g. to set $GREETING to Good morning use the following command:

```
Gretchen:~ thijscoenen$ export GREETING="Good morning"
Gretchen:~ thijscoenen$ echo $GREETING
Good morning
Gretchen:~ thijscoenen$
```

This variable will be defined as long as your BASH session exists. If you are running several BASH sessions, the variable will only be defined for that one session where it was defined. To define a variable for all future BASH session you should edit the settings file for your BASH sessions. This file is probably called .bashrc (Linux) or .bash_profile (Mac OS X) in your home directory. Be careful editing these files, getting it wrong may make your shell inoperable.

If you want to know which environment variables are defined in your shell and what their values are you can use env. Running this command will result in a list of environment variables and their values. Because this is a long list I suggest you run this command on your own computer.

2.2.14   UNIX Pipes

So far we have treated a number of simple commands and utilities available on the command line. The power of the UNIX command line partially derives from the ability to connect different commands. The output of one program can be sent to another program using a so-called pipe. A pipe is signified with the | character. E.g. to use grep to look for a file called .bash_profile in my home directory:

```
Gretchen:~ thijscoenen$ ls -la | grep ".bash_profile"
-rw-r--r--  1 thijscoenen  staff     573 22 jul 23:38 .bash_profile
Gretchen:~ thijscoenen$
```

This may be a very simple example but the principle generalizes easily. The next example shows how you can search through a directory with some type of pulsar data files, looking for the dispersion measure, sorting by dispersion measure and then looking at only the first 10 lines:

```
Gretchen:00000143 thijscoenen$ ls *.inf | wc -l     313
Gretchen:00000143 thijscoenen$ cat *.inf | grep "Dispe" | sort -n -k 6 | head -n 10
 Dispersion measure (cm-3 pc)           =  2.6
 Dispersion measure (cm-3 pc)           =  2.65
 Dispersion measure (cm-3 pc)           =  2.7
 Dispersion measure (cm-3 pc)           =  2.75
 Dispersion measure (cm-3 pc)           =  2.8
 Dispersion measure (cm-3 pc)           =  2.85
 Dispersion measure (cm-3 pc)           =  2.9
 Dispersion measure (cm-3 pc)           =  2.95
 Dispersion measure (cm-3 pc)           =  3
 Dispersion measure (cm-3 pc)           =  3.05
Gretchen:00000143 thijscoenen$
```

Understanding how you can combine different UNIX command line utilities can help you automate boring tasks. This document is not about BASH programming so I will stop here now.

2.2.15   Output Redirects

The output of programs that shows up in your shell was written to "standard out" or "standard error" (a program should use the former for normal output while the latter can be used for errors or warnings). If you want these messages in a text file, you could copy the text from your shell into a text editor and save a file, but that is a lot of work. In stead you can just redirect standard out to a text file with a > sign:

```
Gretchen:coolstuff thijscoenen$ ls -l > blah.txt
Gretchen:coolstuff thijscoenen$ cat blah.txt
total 312
-rw-r--r--  1 thijscoenen  staff        0 15 aug 16:35 Whatever
-rw-r--r--  1 thijscoenen  staff   147731 22 jul 23:27 alice.txt
-rw-r--r--  1 thijscoenen  staff        0 28 aug 01:31 blah.txt
-rwxr--r--  1 thijscoenen  staff       43  7 aug 18:02 hello
-rw-r--r--  1 thijscoenen  staff       50 11 aug 14:35 noperm
Gretchen:coolstuff thijscoenen$
```

If you want to capture both standard out and standard error in a file you can add 2>&1 to the end of the redirect command:

```
Gretchen:~ thijscoenen$ python divide_by_zero.py
Hi I'll divide by zero!
The result is
Traceback (most recent call last):
  File "divide_by_zero.py", line 2, in <module>
    print "The result is", 1/0
ZeroDivisionError: integer division or modulo by zero
Gretchen:~ thijscoenen$ python divide_by_zero.py > output1.txt
Traceback (most recent call last):
```

```
  File "divide_by_zero.py", line 2, in <module>
    print "The result is", 1/0
ZeroDivisionError: integer division or modulo by zero
Gretchen:~ thijscoenen$ cat output1.txt
Hi I'll divide by zero!
The result is
Gretchen:~ thijscoenen$ python divide_by_zero.py > output2.txt 2>&1
Gretchen:~ thijscoenen$ cat output2.txt
Hi I'll divide by zero!
The result is
Traceback (most recent call last):
  File "divide_by_zero.py", line 2, in <module>
    print "The result is", 1/0
ZeroDivisionError: integer division or modulo by zero
Gretchen:~ thijscoenen$
```

In this example you can see that Python sends the text associated with an `ZeroDivisionError` to standard error and the rest of the messages to standard out. The file `output1.txt` only contains the text that went to standard out, while the file `output2.txt` contains both the text for standard out and standard error. In case you do not want to overwrite an existing file, but append to the new output to it replace the $>$ with $>>$.

Chapter 3

Python

The programming course of choice for this course is Python. It was originally developed by Guido van Rossum at the "Centrum voor Wiskunde en Informatica" (CWI) in Amsterdam and was made public in 1991. Nowadays Python is no longer developed at the CWI, but van Rossum is still leading its development. The Python syntax allows for concise code and the language is relatively easy to learn whilst still being powerful. Part of Python's popularity stems from its large standard library (Python is "batteries included") and the availability of a large number of external packages. For scientific programming in libraries like Numpy, Matplotlib and Scipy make Python a powerful tool.

There are two flavors of Python available at the moment, Python 2 and Python 3. At the time of this writing Python 3 is makig inroads into scientific programming, but there is still a large amount of legacy software written in Python 2. While Python 2 and 3 are not fully compatible they are very similar and it is even possible to use a subset of Python 2 and 3 that can run on both versions.

3.1   Programming Languages, Source Code, and Python

Computer programs can be created in a variety of different programming languages. The most hardware specific language is called *machine language*, it is the language that is directly executed by the computer hardware. To write programs in machine language a programmer would have to be intimately familiar with the details of the hardware, making the creation of even simple programs laborious. Fortunately programs can also be created in more abstract programming languages, that are then translated to machine language automatically using special computer programs. These special programs, compilers or interpreters (see below), know the details of the underlying hardware and how to interact with it.

The instructions in some programming language that specify a program's behavior, are called its *source code*. As mentioned above, this source code is later translated to machine code. Python is a *high level* programming language because it shields programmers from the underlying hardware almost completely, while other *lower level* programming languages may require knowledge of the hardware to a larger degree. The C programming language, for instance, requires one to do memory management (unlike Python).

The translation from source code to machine code can be performed in several ways. When the source code is translated to machine code in one go before the program is executed, the translation is called *compilation* and the program that does the translation a *compiler*. The aforementioned C language is generally compiled. When source code is translated instruction by instruction each time the program is run, it is said to be *interpreted* and the program that performs the translation an *interpreter*. The standard way of running Python programs uses an interpreter. The most well known Python interpreter, the one that is probably already installed on your computer if it runs Linux or Mac OS X, is referred to as CPython because that interpreter itself is written in C.

The short description of interpretation and compilation given above necessarily skips over many

nuances and complications. Python code is actually compiled to *byte code* that is executed in a *virtual machine* — so-called because that program acts like virtual computer hardware. Several of the newer Python implementations mix interpretation with compilation in different ways that are (far!) outside the scope of this reader.

3.2   Running Python Programs

Python programs or scripts can be executed from the command line by starting the interpreter and passing it the name of the relevant Python file. We start with the classic "Hello world!" example without which no programming tutorial is complete. Start your text editor of choice, enter the following line of Python code and save it as `hello.py`.

```
print 'Hello world!'
```

This is about the simplest Python program you can produce that still does something. To run it enter the `python hello.py` command at your command prompt. Your shell session should look somewhat like mine:

```
Gretchen:python thijscoenen$ python helloworld.py
Hello world!
Gretchen:python thijscoenen$
```

As you can see all the program does is output the text `Hello world!` to your shell. The `print` statement sends the text that follows to the so-called "standard out", in this case your shell. Like all command line programs, Python programs can also accept command line options.
The Hello world example demonstrates one Python statement, `print`, that sends some text to the screen. In this case what it sends to screen is the *string* 'Hello world!'. A string is a basic data type of Python (representing a string of characters). The basic Python data types are treated in Chapter 4.

3.2.1   UNIX and the Shebang

It is possible on UNIX to avoid having to type `python` in front of the programs you want to run. To do so make the script executable (using the `chmod` command and add a so-called *shebang* to it. The shebang line informs UNIX which interpreter to use to run your program. The shebang must be the first line of a script and it looks as follows:

```
#!/usr/bin/env python
```

3.3   Interactive Python Sessions

Python can also be started without giving it a program to execute by entering the command `python` in your shell. Python will start a new session in interactive mode, you can recognize this mode by the prompt Python shows you: >>>. At this prompt you can enter some Python code to be executed. Recreating the previous example looks like:

```
Gretchen:python thijscoenen$ python
Python 2.7.3 (v2.7.3:70274d53c1dd, Apr  9 2012, 20:52:43)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print 'Hello world!'
Hello world!
>>> exit()
Gretchen:python thijscoenen$
```

As you can see the output of the program is the same as before and it directly follows the line of code you wrote. In the following line the interactive interpreter is stopped using `exit()`. There is another way of stopping the interactive interpreter by pressing ⸢Control⸥ + ⸢d⸥. The interactive interpreter gives immediate feedback, which makes it well suited for quick experimentation. As a side note: the interactive mode of Python is also referred to as the *Python REPL* for "read eval print loop" because it takes user input (the *read* step), evaluates it (the *eval* step) then prints the result (the *print* step) before "looping" back to the first step.

3.4   Using Ipython Notebook

While Python comes with a built-in REPL, there are better ways of interacting with Python. IPython in particular is very convenient, it allows you to interact with Python and with the shell it is running under. IPython can be run in a terminal window, or you can run it as an IPython Notebook. In that latter mode IPython starts a local web server that you interact with through a web browser. The current version of IPython Notebook is called Jupyter. IPython Notebooks notebooks are also able to display graphs using Matplotlib (see Chapter 10), mathematical equations (using LaTeX notation) and Markdown[1] formatted text.

When IPython Notebook is properly installed you can start it from the command line using `ipython notebook`. IPython Notebook will open a browser window and present you with a list of notebooks already open in the current directory (where you started IPython Notebook). Figure 3.1 shows a IPython Notebook session that displays a plot created using Matplotlib. The notebook viewer part of IPython Notebook is evolving into a more general tool named *Jupyter*. Jupyter can currently also be used with the R statistical programming language and the Julia programming language.

In an IPython notebook you can enter Python code and then executed it by pressing ⸢shift⸥ + ⸢enter⸥. An IPython Notebook session behaves much like a "normal" interactive Python session but IPython adds a few magic functions. To invoke these IPython specific functions prepend the input line with a `%`. The previously mentioned example notebook shows the command that allows Matplotlib graphs to be shown (`%maplotlib inline`). Help about the available magic functions can is available by issuing the `?` command. The shell can be used directly from IPython Notebook by prepending a BASH command with an `!`. For example to use `ls` type `!ls`.
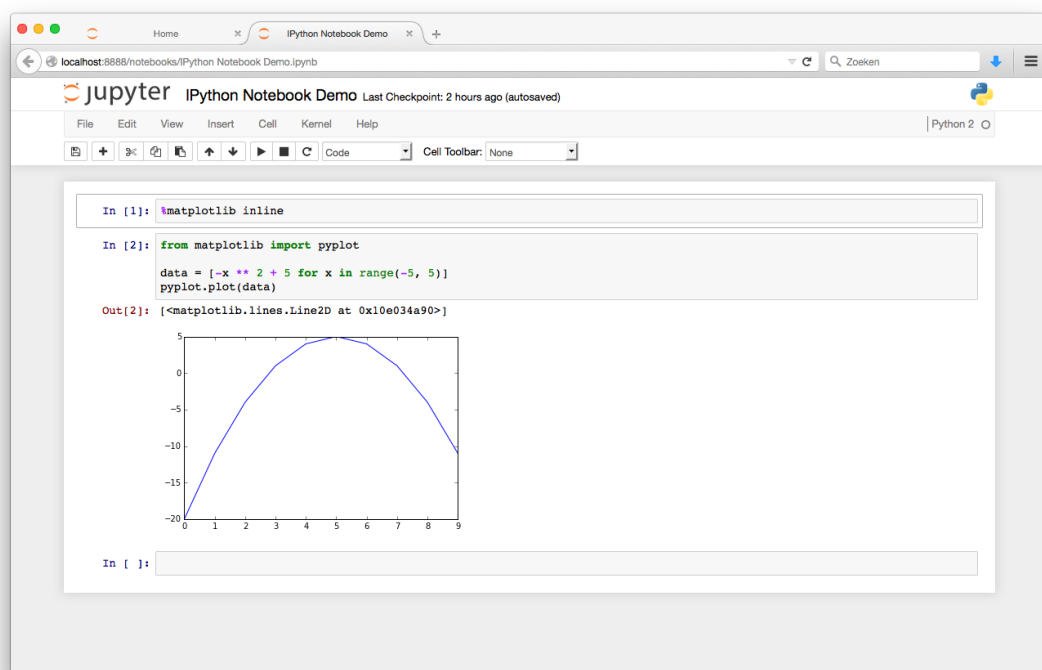
---

[1]`http://daringfireball.net/projects/markdown/`

Figure 3.1: An example Ipython Notebook session demonstrating the ability to display graphs in notebooks.

Chapter 4

Variables and basic data types

The last chapter described several ways in which you can run snippets of Python, while this chapter describes what variables and data types are. It furthermore explains the basic data types of Python and give some examples of how they are used.

4.1 Variables and how to define them

Let's say you want to calculate the travel time of light from the Sun to Earth in minutes. As was shown in the last chapter you can start Python in its interactive mode and enter the calculation. Using 150 million kilometers for the distance of the Sun to Earth and 300 thousand kilometers per second as the speed of light. The division operator in Python is written as a single forward slash /. Remember to press ⏎Enter after entering a line of Python code.

```
>>> (150000000 / 300000) / 60.
8.333333333333334
```

Writing small snippets like this is fine for little experiments. You may, however, need the speed of light more often than just once, and repeatedly entering its value is tedious and error prone. It is better to enter the value once and associate a name with it, that you can then refer to when you need that value again. Variables in Python allow you to do just that, you choose a name and assign a value to it. Python's *assignment operator*, represented by a single equals sign =, is used to assign a value written to the right of the operator to a name written to the left of that operator. Python has no concept of constants and it is therefore always possible to change the value associated with a name by assigning it a new value (Python has no constants, unlike some other programming languages). Using variables the previous snippet can be rewritten using variables as follows:

```
>>> c = 300000
>>> d = 150000000
>>> seconds_per_minute = 60.
>>> (d / c) / seconds_per_minute
8.333333333333334
```

This example may be contrived but is shows you how variables are defined and then used in a calculation. Arithmetic will be described in some detail below.

4.2 Data types

One important property of values was skipped over in the previous section, namely their type. In Python every value has a type which describes what the value represents. Are we dealing with a number, a piece of text, a list of things, or something completely different? This is important because in memory these are all represented as a series of ones and zeros!

In Python, when you assign a new value to a variable the type of the value that the variable points to can change. Python is said to have dynamic types. Not all programming languages share this property. The C language, for instance, does not allow you to change the type associated with a variable and is therefore said to have static types. In C a variable is of a certain type, in Python it is the values that have a type. In Python a variable is properly understood as a label pointing to some value, and the type of a variable is really is the type of the value pointed to.

In Python it is always possible to find out what type is associated with a variable by using the `type` function. Functions will be explained in more details later, but for now you can see them as named pieces of a program that operate on a set of inputs to provide some return some outputs. The type function can be used, "invoked" or "called" in programming jargon, by writing its name and adding parentheses around some input value(s). The type function can be called as `type(someval)` to find the type associated with the `someval` variable. Some examples below (reusing the previous definitions for the `c` and `seconds_per_minute` variables):

```
>>> type(10)
<type 'int'>
>>> type(c)
<type 'int'>
>>> type(seconds_per_minute)
<type 'float'>
```

This example shows two types of variables, both representing numbers, the so-called *integer* numbers (called `int` for short) and floating point numbers (`float` for short). These are two of the basic data types available in Python that will be described in more detail below.

While Python is dynamically typed, it is strict in the way you operate on values of different types. Trying to, for instance, add a piece of text to a number using the addition operator `+` will result in a so-called `TypeError`. The following snippet shows what happens when you add a string (the data type that represents text) to an integer:

```
>>> '1' + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  TypeError: cannot concatenate 'str' and 'int' objects
```

Some programming languages, such as JavaScript or BASH, will perform an implicit type conversion to arrive at a result for this kind of operations — Python does not. While the Python behavior may seem strict, it is consistently applied and that makes programs easier to debug.

4.3 Mutable and immutable values in Python

When a value can be changed after it is created it is called *mutable*. When a value cannot be changed after it is created it is called *immutable*. Even "immutable variables" (in quotes because the values are immutable) can get a new value by assigning them a new value. What will not happen, however, is that the underlying value is changed.

A simple example of immutable values is the Python string data type used to represent text. If you use the `upper` function associated with a string you will get an upper case version of that string, but the original string will not change, as is shown in the following snippet:

```
>>> s = 'Hello world!'
>>> s.upper()
'HELLO WORLD!'
>>> s
'Hello world!'
```

A similar simple example of a mutable data type is the Python `list` data type, which acts as a container for an arbitrary number of values. The `sort` function associated with lists will change the original list and, unlike the `upper` function shown above, it will not return a new copy of the list.

```
>>> l = [10, 5, 9, 0]
>>> l
[10, 5, 9, 0]
>>> l.sort()
>>> l
[0, 5, 9, 10]
```

Mutability versus immutability is frequent source of misunderstandings and errors in Python, so be careful.

4.4   Literals

This section introduces one last piece of jargon (for now) before continuing on with more practical examples. In the previous sections you saw how variables are defined and you were given a high level description of why types matter. What you also saw was how values are represented in snippets of Python source. The representation of a value in source code is called a *literal*. You saw integer literals, floating point literals, string literals and a list literal in the previous sections.

4.5   Numbers and basic arithmetic in Python

In Section 4.2 two of Python's data types, namely integers and floating point numbers, made a brief appearance. Integers represent the whole numbers including zero and negative numbers, while floating point numbers approximate the real numbers. I say "approximate" because in Python floating point numbers generally use 64 bits, allowing for only 2^{64} different numbers to be represented. Python's integers are special in that they can change internal representation in such a way that allow arbitrarily large numbers to be represented (or in the words of the Python documentation its integers numbers represent values "which are limited only by available memory"[1]).

Integers have a number of literal notations that you may come across. All previous examples showed integers literals written in their decimal form, there are also octal and hexadecimal forms. Octal and hexadecimal representations use respectively base 8 and base 16 to write the numbers. Octal digits run `0` through `7` while hexadecimal digits run `0` through `9` and then `A` through `F`. To differentiate these notations from decimal notatation octal numbers are preceded by `0` and hexadecimal numbers are preceded by `0x`.

```
>>> # octal (note the leading 0):
... 011
9
>>> # hexadecimal (note the leading 0x):
... 0xff
255
```

Hexadecimal notation is quite often used in more low level (i.e. closer to the hardware or less abstract) code to represent numerical constants. You will certainly come across memory adresses written in hexadecimal notation in Python error messages. The `#` signs, also known as a hash or octothorpe, is used to signify a *comment*. Comments start after the `#` and extend to the end of the line, they are added to help programmers better understand the code and are ignored when the Python program is executed. The ... in this example are the way Python interactive sessions show that several lines were entered before they were executed and the result shown.

Like integers floating point numbers have several notations, they can be in their decimal form using a decimal dot to differentiate them from integers and they can be in scientific form. The following example shows these notations:

---

[1]See `https://www.python.org/dev/peps/pep-0237/`.

```
>>> # decimal notation:
... 3.1415
3.1415
>>> # scientific notation:
... 3e+5
300000.0
>>> 6.7e-11
6.7e-11
>>> 1e-2
0.01
>>> 1E-2
0.01
```

Python supports a small set of standard operators for numbers, while a larger set of mathematical functions are available through a special module called `math`. This section only treats the former, deferring the latter.

Addition in Python uses the `+` operator, subtraction uses the `-` operator, multiplication uses the `*` operator while raising to the power uses the `**` operator. Standard precedence order is used in Python meaning that raising to the power is performed before a multiplication, which in turn is performed before a subtraction or addition. The full list of operator precedences is available from the Python website[2]. If in doubt, use parentheses to explicitly control the order in which the operators are applied yourself.

The division operator was deliberately not mentioned yet, because there are a few important details you should know about. The normal division operator in Python written as `/` has behavior that depends on the types that it operates on. Divide two floating point numbers and it will return a floating point number as answer, when it operates on a floating point number and an integer number it will return a floating point number, but when it operates on two integer numbers it will return an integer and throw away the remainder (the answer is *truncated*). This behavior is inherited from the C programming language and is a frequent source of bugs in Python programs. In Python 3 (*i.e. the version of Python that this reader is emphetically not about*) the division operator was split into two operators. The first division operator written `/` returns a floating point number regardless of whether it operates on floating point numbers or integers. The second division operator written `//` always truncates regardless of whether it operates on floating point numbers or integers. This behavior can be turned on also for Python 2 by adding the line

```
from __future__ import division
```

to the top of your programs (before all other import statements). Other than the divsion operator(s) of Python there is also the modulo operator written `%`. This operator returns the remainder of a division. Some examples of the default behavior of the Python division operator are shown below:

```
>>> 10 / 2
5
>>> 10 / 3
3
>>> 10 % 3
1
>>> from __future__ import division
>>> 10 / 2
5.0
>>> 10 // 2
5
>>> 10 / 3
3.3333333333333335
```

---

[2]See `https://docs.python.org/2/reference/expressions.html#operator-precedence`.

```
>>> 10 // 3
3
```

To solve truncation issues in programs that do not use the special `import` line you must manually convert the type of the operands. Python has built in functions to convert to and from strings, integers and floats. The `float` function converts its input to a floating point number and the `int` function converts to an integer. By converting one of the numbers in a division to a `float` the answer will not be truncated:

```
>>> float(3)
3.0
>>> 10 / float(3)
3.3333333333333335
>>> int(3.1415)
3
```

4.6   Strings in Python

Text of any length is represented as a string (type `str`). String literals can be delimited with single quotes (`'abc'`), but double quotes are also allowed (`"abc"`). A number of characters cannot be represented directly in a string literal and must be written as a so-called *escape sequence*. These sequences start with a \ and the character that follows is interpreted in a special way. An example is the tab character which may look like several spaces but is in fact only one character \t, the new line character is another example \n. A table of string escape sequences can be found in the section about string literals in the Python documentation[3]

Beyond the single and double quotes you can also come across triple quoted strings in Python source code (useful because they can contain new lines and quotes without escaping them) or strings broken up across several lines, see the following string definitions:

```
# Several different string literals
s1 = 'abcdefg'
s2 = "abcdefg"
s3 = '''This string
contains 'quotes' and
new lines!
'''
s4 = """This is also allowed"""
```

The normal Python 2 strings use 8 bits to store each character and thus only 256 characters can be encoded. That is clearly not enough to handle all the world's characters or languages directly. Unicode was designed to solve this problem once and for all. It allows all the world's known characters to be represented and it can thus be used throughout the world. Python has a special Unicode string data type (`unicode`). Unicode literals are delimited with `u'abc'` and special characters can be entered using Unicode escapes that start with \u. The escape sequence for the Euro-sign, for example, is \u20ac.

```
>>> u = u'abcd\u20ac'
>>> type(u)
<type 'unicode'>
>>>
```

---

[3]See `https://docs.python.org/2/reference/lexical_analysis.html#strings`.

4.6.1 Indexing and slicing

Each individual character in a string has an index that can be used to access it. These indices start at 0 and run to `n-1` for a string of `n` characters length. For example, you can access the first character of a string `s` as `s[0]` and its second character as `s[1]`. If you use an index that is too high, Python will raise an `IndexError`. As a convenience Python allows negative indices that are used as indexes starting at the end of the string. The last character of a string `s` can be accessed as `s[-1]` while the second-to-last can be accessed as `s[-2]`, etc.

To extract more than just one character from a string the slice operator is used. For a string `s` it is written as `somestring[startindex:endindex]`, where `startindex` and `endindex` are respectively the index of the letter where the slice starts and the index of the character before which the slice ends. To extract a sub string starting at the first character the `startindex` can be left out, similarly if the `endindex` is left out the sub string runs to the last character.

```
>>> s = 'abcdefg'
>>> type(s)
<type 'str'>
>>> s[0]  # normal indexing
'a'
>>> s[10]  # accessing a non-existant character
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>> s[-1]  # accessing the last character
'g'
>>> s[0:5]  # accessing the first 5 characters
'abcde'
>>> s[:5]  # idem
'abcde'
>>> s[5:]  # access starting at the 6th character
'fg'
```

Python implements many standard string manipulations as special functions connected to the string objects[4]. Below some examples of useful string manipulations; more are available in Python[5]. As was mentioned before, Python strings are immutable and can thus not be changed after they are created. These functions therefore create new strings in stead of changing the original strings they were called on.

```
>>> s.startswith('abc')
True
>>> s.endswith('xyz')
False
>>> s.upper()
'ABCD EFG'
>>> s.isdigit()
False
>>>
```

Because a sequence of characters are contained in strings and unicode strings, they are so-called *sequence types*. Indexing and slicing operations are also supported by other sequence data types in Python like tuples and lists (see below).

---

[4]I am avoiding the term *method* here, because that is associated with object oriented programming. But since Python is object oriented and its strings objects these functions are properly called methods.

[5]See the documentation at `https://docs.python.org/2/library/stdtypes.html#string-methods`

4.7   Tuples

A tuple is a container for several other values, whose types can be mixed in a tuple. The separate
values contained in a tuple can be accessed by their indices. Because a Python tuple is immutable,
it is not possible to assign new values to some position in a tuple. If you do try to assign new
values to a tuple a `TypeError` will be raised. In source code tuple literals are usually enclosed in
parentheses `()` and the values in the tuple separated by commas. Note that you can leave the
parentheses out but not the comma(s). Like strings, tuples are a sequence type. The following
example shows some operations on tuples.

```
>>> t1 = (1, 2, 3, 'abc')
>>> type(t1)
<type 'tuple'>
>>> t2 = 1,
>>> t2
(1,)
>>> type(t2)
<type 'tuple'>
>>> t3 = (1)
>>> t3
1
>>> type(t3)
<type 'int'>
>>> t1[0]
1
>>> t1[0:]
(1, 2, 3, 'abc')
>>> t1[0] = 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>>
```

4.8   Lists

Like tuples and strings, lists are a sequence type that can contain any number of values of any type.
Unlike tuples, lists can be changed after they are created (both their contents and their lengths
can be changed), they are therefore *mutable*. Lists literals are enclosed in square brackets `[]` and
the values in a list are separated by commas. Lists like strings support indexing and slicing to
access values or extract parts of a list. Like strings, lists have many useful methods and because
lists are mutable they are changed by these methods. The following example demonstrates some
list methods:

```
>>> l = [1, 2, 3, 4]
>>> type(l)
<type 'list'>
>>> l.pop(0)
1
>>> l
[2, 3, 4]
>>> l.append(1)
>>> l
[2, 3, 4, 1]
>>> l.sort()
```

```
>>> l
[1, 2, 3, 4]
```

4.9   Dictionaries

Dictionaries are used to store key-value pairs, that is they map a certain key to the value that was stored with that key. Unlike lists that use indices to access stored values dictionaries use keys. Values can be any Python object but the keys have to be "hashable" (what that means exactly we will skip for now, but in practice integers, floats and strings can used as keys). Unlike tuples and lists, dictionaries are not ordered and you cannot speak of its first or last element in a dictionary. Trying to use a key that is not present in the dictionary will result in a KeyError being raised. Dictionaries correspond to *hash tables* in other programming languages.

```
>>> d = {3: True, 1: 'a', 5: 123}  # a dictionary literal
>>> type(d)
<type 'dict'>
>>> d[3]  # accessing an existing key
True
>>> d['no such key']  # accessing a non-existant key
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'no such key'
>>> del d[1]  # removing a key-value pair (using the key)
>>> d
{3: True, 5: 123}
>>>
```

4.10   Sets

Related to dictionaries is the Python set datatype. It can be used to represent sets of values where order does not matter, and where you want to be able to access elements quickly. Like the keys to a dictionary the values added to a set must be hashable. Sets are convenient when you are interested in mathematical set operations, Python sets offer special functions and operators for operations like taking the intersection or union of two sets. The example below shows some of these functions but you are encouraged to look at the Python documentation for more.

```
# define a set and add some elements:
s1 = set()
s1.add(1)
s1.add(4)
print s1  # note: you cannot assume this is ordered

# define a set starting from a list of elements:
s2 = set([6,4,7, 7, 7, 10])
print s2  # note: elements are not repeated

# some set operations:
print s1.intersection(s2)
print s1.union(s2)
print s1.symmetric_difference(s2)

# or equivalently:
print s1 & s2
```

```
print s1 | s2
print s1 ^ s2

# A small trick to make list of unique elements:
l = [1, 4, 6, 2, 6, 1, 10, 4]
print list(set(l))
```

4.11 Booleans

Python has a special data type to represent truth values so called booleans (type `bool` for short). Booleans can only take one of two values, a boolean is either `True` or `False`. The results of comparisons are booleans (for more see Section 5.1):

```
>>> b = True
>>> type(b)
<type 'bool'>
>>> 1 > 2
False
>>> 2 > 1
True
>>>
```

4.12 NoneType

To signify a missing value Python uses the special `None` object of type `NoneType`. This is very useful to differentiate a missing value `None` versus for instance an empty string `""` or an empty list `[]`.

4.13 Basic type conversions

As was mentioned above Python is quite strict about its types and usually does not convert between different types automatically. Strings cannot be added to integers, mathematical functions will not work on strings, etc. When you use the wrong types typically a `TypeError` is raised. Fortunately you can convert between different types by using the right functions. To convert a string or a float to an integer use the `int()` function, to convert something to a float use `float()`, to convert to string use `str` and to convert to a Unicode string use `unicode()`. If such a conversion is not possible, a `ValueError` will be raised (for example 'xyz' cannot be converted to a floating point number while '1.5' can.

```
>>> int(1.5)
1
>>> int('1.5')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '1.5'
>>> float('1')
1.0
>>> float(2)
2.0
>>> str(4.576)
'4.576'
>>>
```

For tuples, lists and dictionaries similar functions are available. The conversion functions for tuples and lists, respectively `tuple()` and `list()` expect a so-called *iterable* (we forego a exact definition at the moment but mention that an iterable contains several values that can be accessed one-by-one in a process called iteration). Dictionaries are a bit more complicated, as they need key-value pairs, so the `dict()` function expects an iterable of key-value pairs (so each object in the iteration needs to have two values). This "iterable of key-value pairs" is in practice often a list of tuples or a list of lists, where the inner tuples or lists have the key as the first element and the corresponding value as the second element.

```
>>> list('1234')
['1', '2', '3', '4']
>>> tuple('abcd')
('a', 'b', 'c', 'd')
>>> tuple([1, 2, 3, 4])
(1, 2, 3, 4)
>>> list((1, 2, 3, 4))
[1, 2, 3, 4]
>>> dict('1234')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: dictionary update sequence element #0 has length 1; 2 is required
>>> dict([(1, 'a'), (5, 'b')])
{1: 'a', 5: 'b'}
>>> list({1: 'a', 3: 'c'})
[1, 3]
>>>
```

The last example, converting a dictionary to a list, discards the values in the dictionary being converted because iteration over a dictionary returns only its keys. We will revisit iteration in Section 6.

Chapter 5

Conditional execution

When some piece of Python code is only executed when a certain condition is met that piece of code is *conditionally executed*, conditional execution is a fundamental concept in any programming language. This chapter explains how conditions are expressed in Python and used to control program flow.

5.1 Boolean expressions

An expression that evaluates to either `True` or `False` is a called *boolean expression*. Comparisons are common boolean examples of a boolean expressions

```
>>> a = 10
>>> b = 5
>>> a == b  # Check whether a equals b
False
>>> a != b  # Check whether a does not equal b
True
>>> a > b  # Check whether a is greater than b
True
>>> a >= b  # Check whether a is greater than or equal to b
True
>>> a < b
False
>>> a <= b
False
```

5.2 Boolean operators `and`, `or` and `not`

Boolean expressions can be combined using the `and` and `or` keywords and negated using the `not` keyword. When you have two booleans (or boolean expressions) the `and` operator will result in `True` if both booleans are `True` else the result will be `False`. The `or` operator results in `True` if either or both the booleans are `True` else the result will be `False`. The `not` operator negates a boolean turning `True` into `False` and vice versa.

```
>>> t = True
>>> f = False
>>> t and t
True
>>> t and f
```

```
False
>>> t or t
True
>>> t or f
True
>>> not t
False
>>> not f
True
```

When several boolean expressions are combined using the `and`, `or` or `not` the resulting expression is still a boolean expression because it evaluates to `True` or `False`.

5.3   Truthy and falsy

While Python has proper booleans `True` and `False` there are several other values that are treated as true or false in the context of a conditional. These are sometimes called *truthy* for values that are treated as `True` and *falsy* for values treated as `False`. Empty lists, dictionaries, sets and strings are falsy, whilst they are truthy when they have elements. In the following example the `bool` built-in function is used to test the truthiness of an empty list and a list with elements:

```
>>> l1 = []
>>> l2 = [1, 2, 3, 4]
>>> bool(l1)
False
>>> bool(l2)
True
```

5.4   `if`, `elif` and `else`

The `if` statement allows the conditional execution of a block of code. In Python blocks of code are defined by indenting them. The following trivial example shows that `if` followed by a `True` or `False` will either always be executed or never:

```
if True:
    print 'This will be printed'
if False:
    print 'This will never be printed'
```

Using a boolean expression in stead of `True` or `False` directly allows you to test a condition and only execute the following block if the boolean expression evaluates to `True`.

```
x = 10
y = 20

if x > y:
    print 'x is greater than y'
else:
    print 'x smaller than or equal to y'
```

This example also demonstrated the `else` clause, which will be executed only when the preceding condition (defined by the `if` statement) is `False`. When several conditions must be tested you could do that by repeatedly using `if` statements, but Python also has the `elif` statement. The `elif` statement can not occur by itself, it is always preceded by another `elif` statement or an `if` statement. Similarly `else` can only be used following an `elif` or `if` statement.

```
x = 10

if x < 0:
    print 'x is negative'
elif x == 0:
    print 'x is zero'
else:
    print 'x is positive'
```

There is a difference between using several `if` statements and using only one `if` and one or more `elif` statements. In the former case all conditions are checked while in the latter case the checking stops as soon as a condition is `True`.

Chapter 6

Loops and iteration

6.1 Introduction

When you need to repeat some instructions several times you can create a so-called loop to do this. There are two types of loops in Python, `while`-loops and `for`-loops. The former executes a block of code as long as a certain condition holds true, and the latter performs an action for each of the vales of an object like a list or dictionary. Each time the code block is executed is also called an *iteration*. The process of accessing the elements in a list is also called iteration, so the term can be a confusing. This chapter explains how the basic sequence types of Python can be iterated over.

6.1.1 The `while` loop

The `while` loop is executed as long as the condition at the beginning of the `while` loop is `True`. This condition is given as a boolean expression much like those used with the `if` and `elif` statements. The code that is to be repeatedly run is put in an indented block, again like the code blocks for `if` and `elif` statements. To break out of a loop you can use the `break` statement, when break is encountered the current loop is stopped immediately. If a loop is nested inside another loop breaking out of the inner loop will not break out the outer loop. The `continue` statement stops the current iteration immediately and jumps to the next iteration. A feature of loops in Python, one that is not shared with many other languages, is the possibility to add an `else` statement to a loop. The block of code associated with the `else` statement will be executed only if the `while` loop exits normally (that is without the use of `break`).

```
i = 0

while i < 10:
    print i
    i = i + 1
else:
    print 'i is equal or larger than 10'

while True:
    pass  # Infinite loop, nevers stops (unles you kill the program) !!!
```

6.2 The `for` loop and iteration

The Python `for` loop always iterates over some object, i.e. it accesses all the items in that object one-by-one[1] and runs a block of code. When all elements have been accessed, or a `break` statement

---

[1]It is possible in Python to write functions or classes that can be iterated over without them having elements, much like the built-in `xrange` function. In this section we will not consider these types of iterables.

was encountered the iterations stops. The `continue` statement can be used with `for` loops as well, and functions in the same way as it does for `while` loops. Like `while` loops it is possible to use the `else` statement in combination with a `for` loop. The block of code associated with the `else` statement is executed when all items were accessed without encountering a `break` statement.

```python
s = 'abcdefg'

for character in s:
    if character == 'a':
        continue  # 'a' will never be printed because of the continue
    elif character == 'z':
        break  # if a 'z' is encountered the loop is exited
    else:
        print character
else:
    print 'There was no z in this string.'
```

Objects, like lists, that can be iterated over are called *iterable*. While in some languages `for` loops always use indices they should be avoided in Python (unless you really need the indices themselves). It is possible to iterate over the indices of for instance a list by using the `range()` function.

```python
s = 'abcdefg'

for i in range(len(s)):
    print s[i]
```

6.3   Iteration over basic Python types

6.3.1   Iterating over lists

```python
l = ['abc', 'def', 'efg']

# print elements of l one-by-one
for element in l:  # preferred!
    print element

for i in range(len(l)):
    print l[i]

# print index and corresponding element, element-by-element
for i, element in enumerate(l):  # preferred!
    print i, element

for i in range(len(l):
    print i, l[i]
```

6.3.2   Iterating over lists of tuples

```python
# Iteration over a list of tuples:
l1 = [('a', 'b'), ('x', 'y'), ('p', 'q')]

# print pairs of elements:
for c1, c2 in l1:  # preferred!
    print c1, c2
```

```python
for tup in l1:
    print tup[0], tup[1]


# This style of iteration also works with more than two element tuples
l2 = [(1, 2, 4), (2, 3, 5), (6, 2, 9)]


# print the elements of each 3-tuple:
for a, b, c in l2:
    print a, b, c
```

6.3.3 Iterating over strings

```python
s = 'abcdef'


# printing characters in s one-by-one
for character in s:  # preferred!
    print character


for i in range(len(s)):
    print s[i]


# printing index and corresponding character, character-by-character
for i, character in enumerate(s):  # preferred!
    print character, i


for i in range(len(s)):
    print i, s[i]
```

6.3.4 Iterating over dictionaries

```python
# dictionary iteration:
d = {1: 'a', 0: 'b', 6: 'c'}


# iteration over dictionary keys:
for key in d:
    print key


# iteration over dictionary values:
for value in d.values():
    print value


# iteration over key-value pairs:
for key, value in d.items():  # preferred (more readable)!
    print key, value


for key in d:
    print key, d[key]
```

Chapter 7

Simple File Input and Output

Python has a built-in `File` object that is used to represent an open file. To open a file you use the `open` function and to close it you use the `close` method of file objects (examples below). A file can be opened in one of several *modes*, it can be opened for *reading*, for *writing* or for *appending*. Reading leaves a file unchanged, while writing will change the file's content and appending will add to the end of a file.

7.1 Text files

The `open` function takes two arguments, first the name of the file as a string, and second the mode that the file should be opened in as a string. That latter string can be one of: `'r'` for reading, `'w'` for writing, `'a'` for appending. These modes work for textual data, binary data has special modes that will not be treated here — their documentation is available from the Python website. The `open` function returns a file object. Depending on the mode it is opened in this file object can be written to or read from using respectively the `write` or `read` methods of the file object.

Since operating systems have limits on the number of files that can be simultaneously opened, you must close files that you open. Python will automatically close open files when your program terminates, but that may be too late to avoid errors.

In the next example a small text file is created, then written to, and finally closed. The file object in this example must be assigned to a variable, `f` in this case, because you need a reference to it so that you can write to it and close it. The second part of the example shows how all data can be read from a text file.

```python
# Open a file for writing and close it again:
f = open('smallfile.txt', 'w')
f.write('Some text for this file\n')
f.close()


# Now open the example file for reading, and print out the file content:
f = open('smallfile.txt', 'r')
print f.read()
f.close()
```

Note, this is the simplest example of writing and then reading a file, but it is not the recommended way of doing it — see below for a better technique.

Modern Python programs use the `open()` function in conjunction with the `with` keyword. The `with` keyword in the following example will ensure that the open file is closed no matter what, when the block defined by `with` is left. You will no longer need to close the file manually. A further nice property of Python `File` objects is that they can be iterated over (in the case of text files). Iterating over an open text file will access the lines in that file one-by-one. The following example demonstrates both the `with` statement and iteration over a text file.

```python
with open('alice.txt', 'r') as f:
    count = 0
    for line in f:
        count = count + 1

print count
```

Chapter 8

Program Structure Basics

The previous chapters have shown you parts of Python, but not what a Python program actually looks like. This chapter will explain how you can organize a program in such a way that it is readable, reusable and well documented.

A well structured program is a modular program with constituent parts that each have well defined purposes and is as independent from the rest of the program as possible. By writing program this way programmers can solve problems by breaking them up in to smaller more manageable parts. Modularity furthermore allows parts of the program to be reasoned about independently. Python has several mechanisms to achieve modularity, it has functions, classes, modules and packages. This chapter will explain how functions and modules can be used to create a modular program. Reusability is also enabled by modularity because the independent parts of a program can be included in other programs.

While programs are executed by computers they are written, adapted, and maintained by humans which implies that source code must be written to be read *and* understood by humans. Readability of code is affected by many factors; clear structure, descriptive naming schemes, adhering to language idioms, useful comments, clear documentation, and so on. This chapter will explain how Python code can be documented and give some recommendations for naming schemes.

8.1 Functions

Functions are a way to name a block of executable code, much like a variables are a way of naming values, and like variables allow you to reuse a value, functions allow you to reuse a blocks of code. They take a number of inputs, perform some operations and return some outputs. The input variables that appear in the function definition are called the *parameters* of that function, while the actual values that are sent to the function are called *arguments*. A function definition starts with the `def` keyword, followed by the name of that function, a list of parameters enclosed by parentheses, a colon and finally an indented block of code called the *function body*. In Python you do not need to declare the data type of the parameters to a function. The first example shows two simple function definitions:

```python
def say_hello():
    '''Print 'Hello!'.'''
    print 'Hello!'

def is_even(n):
    '''Return True if n is a even.'''
    return n % 2 == 0
```

The `say_hello` function in this example takes no arguments (since there is no list of arguments

between parentheses) and it has no return values[1] (since there is no `return` statement). When you use the first function it will print `Hello!` to the standard output. The second function `is_even` takes only one argument, a number n, and returns `True` or `False` depending on whether the number is even or not. Both functions contain a triple quoted string with an explanation of what the function does. These types of strings, so-called docstrings, will be explained in Section 8.5.

The next example shows how you can filter a list of numbers so that the resulting list only contains even numbers:

```python
numbers = [1, 2, 3, 4, 5, 6, 7, 8]

def is_even(n):
    '''Return True if n is a even.'''
    return n % 2 == 0

def filter_even_numbers(l):
    '''Return a list of even numbers in l.'''
    out = []

    for number in l:
        if is_even(number):
            out.append(number)
    return out

print 'The even numbers are:', filter_even_numbers(numbers)

print out  # This will fail with an NameError because out is unknown here.
```

The final line of this little program demonstrates that functions can define variables that are invisible to the rest of the program. This allows you to reuse obvious variable names in different contexts without having to worry that you already used them somewhere else. The following section will explain the rules around visibility and how you can use these rules to create a nice modular program.

8.2 Scopes and namespaces

The `NameError` in the last example of the previous section showed that variables can be defined inside of a function that remain invisible to the rest of a program. The extent of program where the variable is visible is called its *scope*. In Python functions, classes and modules all create their own scopes. Associated with each scope is a namespace. These namespaces maintain a mappings from names to objects accessible in that scope.

Scopes are nested. When an object is accessed by name, Python first checks the so-called *local scope* for that object. The local scope is the scope associated with the currently executing function. If that check fails, Python will move to enclosing scopes and check those. If the name is not found in any of the scopes Python raises a **NameError**.

The scopes that are checked during name lookup are (in order): The local scope, any enclosing function or class scopes, the module *global scope* and finally the outermost scope that contains the Python built-ins. Since functions and classes nest (you can define functions inside functions or classes, and vice versa), there may be several scopes enclosing the local scope before the module global scope is reached. The module global scope contains names that are not defined inside any function or class, but are inside that module. These *global variables* are accessible throughout the whole module they were defined in, even from enclosed scopes.

---

[1]Beware that Python considers the output of a function without a return statement to be `None`.

8.2.1   Note on global variables

Variables defined in the module global scope are accessible everywhere in that module. While it may seem appealing to have all your variables in the global scope so that you can access them everywhere it is a bad idea, really! In a large program that uses many global variables every part of the program has the potential to affect every other part of the program, and therefore the program will be hard to understand, debug or adapt.

When a name that exists in the global scope is assigned to from a enclosed scope, that object in the global scope is not changed. In stead, the name is defined in the local scope making the global object inaccessible from the local scope. To circumvent this problem Python has the `global` keyword. If you are only interested in the value of a global without needing to change it you can leave the `global` keyword out. Trying to change the value of a global variable without using the `global` keyword will result in a `UnboundLocalError`. The **global** keyword is rarely needed, and should be avoided. The following example shows the behavior of global variables.

```python
a = 10  # in global scope


def add(a, b):
    # Note: a and b are now local variables of this function
    # Note: the local a variable is something else than the global one
    a = a + 10  # all working on local variables!
    return a + b


def change_a():
    # To change the value of a global variable from some function's scope
    # use the global keyword of Python.
    global a  # Is needed!
    a = a + 10


def two_a():
    # Note: you can access a global variable to read its value without the
    # global keyword.
    return 2 * a


print a  # prints 10
print add(2, 2)  # prints 14
change_a()
print a  # prints 20
print two_a()  # prints 40
```

8.3   Classes

Before explaining what classes are, it should be noted that object oriented programming is not a subject that is treated by the Basic Linux and Coding for Astronomy and Astrophysics course. What follows in this section is here only to provide some context.

Classes are a way of defining objects that contain both data and functions that work with that data. They facilitate a style of programming called Object Oriented Programming (OOP). Python allows object oriented programming but it does not require it. The Python list that figured in many examples already is an example of a class, it combines data (the elements of the list) with functions that operate on that data (like the `list.sort` member function).

Much as a number like `3.1415` is of data type `float` a certain object like `[1, 3, 4]` is said to be an *instance* of the class list.

```python
# An example of a class definition and how to use it:
class Human(object):
    def __init__(self, name, age): # The special initializer member function
        self.name = name
        self.age = age

    def say_hello(self):
        print 'Hello my name is {0} and I am {1} years old'.format(
            self.name, self.age
        )

    def say_goodbye(self):
        print 'Good bye!'


# Creates an instance of Human called j, this calls the Human.__init__ member
# function behind the scenes.
j = Human('Jan', 50)


# Call the Human.say_hello function. Note how the self parameter is not given
# a value by the programmer, Python handles self behind the scenes.
j.say_hello()  # will print: Hello my name is Jan and I am 50 years old.
```

This example shows a simple class definition and how that class definition can be used to create instances. The magic __init__ function is called when an instance of the class is created. As you can see in this example the programmer never supplies a value for self — it is handled automatically by Python which uses self to keep track of the different instances of the class Human.

An important concept in Object Oriented Programming (OOP) is *inheritance*. Using inheritance you can create a hierarchy of classes that inherit some of their properties from other classes. A class that inherits some of the properties from another class is called a *sub class* of that class. Similarly the class that is inherited from is called *base class*. If a certain attribute (data member or member function) is not defined on a class, Python will check the base classes in search of that attribute. If the attributed cannot be found on any of the base classes Python raises an **AttributeError**. To continue the previous example we introduce a Teacher class which inherits properties from the Human class:

```python
class Teacher(Human):
    def say_hello(self):
        print 'Hello dear students.'

    def teach(self):
        print 'Blah blah blah'


# Because Teacher does not define a __init__ function instantiating the
# Teacher class will call __init__ on the base class, so Human.__init__ is ]
# called (because of inheritance).
k = Teacher('Kees', 40)


# The say_hello function was overridden on the Teacher class so it will
# be called on the Teacher class.
k.say_hello()  # prints: Hello dear students.
k.say_goodbye()  # prints: Good bye!
```

This example demonstrates what happens to instances of a sub class when the methods of this sub class are called. If it defines the same member functions as its base class, the version on the sub

class is used. If no definition is given in the sub class, Python will look for that member function on a base class and call that.

8.4   Modules

A Python file is called a module. Modules can be run as programs (as was demonstrated in Section 3.2 or imported. To import a module means to load its definitions so that they may be used in the program you are working on (or in a Python shell). The load for instance the `math` module that contains more advanced mathematical functions than what is normally available use the `import math` statement. This will create a name `math` in your program (or Python shell) that points to the `math` module. It is also possible to import a module under a different name, or just some functions from a module:

```python
import math
import cProfile as profile  # imports the cProfile module and names it profile
from os.path import join  # imports only the join function from os.path


# and now a really bad and lazy idea !!!
from pylab import *  # import all definitions from pylab into the global scope
```

The last import statement of the above example is a really bad idea because it indiscriminately pulls all definitions in the `pylab` module into your program (which may overwrite some of the other definitions). You should only ever import those things you need, or a whole module in its own name space (that is give it a name and access all other functions via that name using the . notation). To access the cosine function from the `math` module in this example using the unambiguous `math.cos` notation.

8.5   Documentation

Python code can be documented in several ways. Code with a logical flow, clear modularity, and well chose, descriptive, names is referred to as self documenting (the holy grail of well written code). Comments, i.e. lines starting with a `#`, are used to explain the purpose of sections of code, the reasons it was written a certain way, or to give some extra context. Python also allows so-called `docstrings` which are triple quoted strings at the top of a module, function, class, or class method that explain what their purpose is. Docstrings can easily be turned into nice documentation using one of Python's documentation tools[2]. Docstrings should explain how to use the module/class/function/method, without a need to read the underlying code.

The following example shows a simple module with a module docstring and two function definitions with their own docstrings (for the purposes of this text assume that this module is named `simple.y`).

```python
#!/usr/bin/env python
'''
Some trivial string manipulations.
'''


def is_palindrome(word):
    '''
    Check whether a word is a palindrome.
    '''
    # the [::-1] slice goes through a string in reverse
    return word == word[::-1]
```

---

[2]See, for instance, `http://sphinx-doc.org/`.
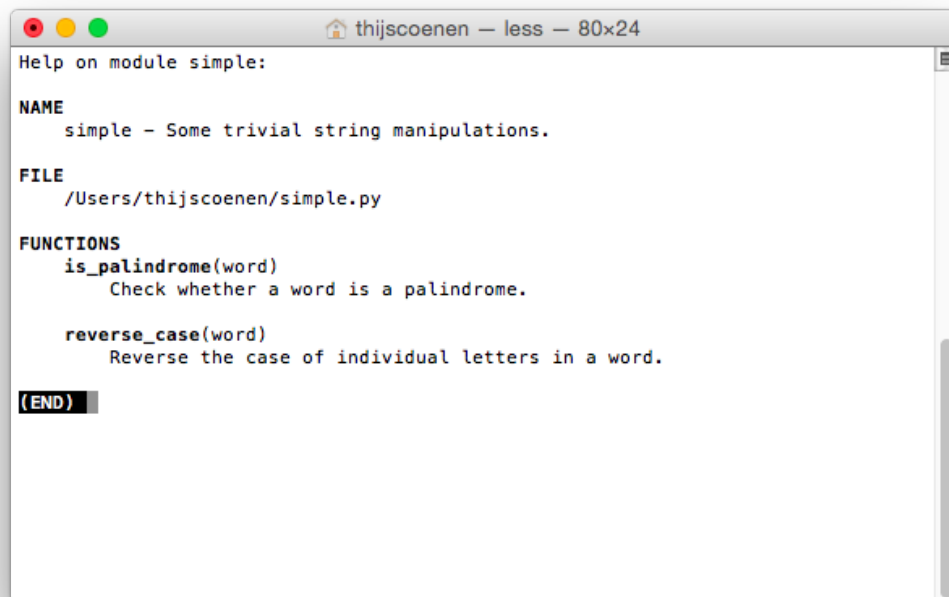
Figure 8.1: The Python built-in `help` function in action viewing the module documentation of the `simple.py` module.

```python
def reverse_case(word):
    '''
    Reverse the case of individual letters in a word.
    '''
    # create a list with the case reversed letters:
    case_reversed = []
    for c in word:
        if c.isupper():
            case_reversed.append(c.lower())
        elif c.islower():
            case_reversed.append(c.upper())
        else:
            case_reversed.append(c)

    # turn the case reversed list in string and return it:
    return ''.join(case_reversed)
```

You can import a module in an interactive Python session, and use the built-in `help` function to view that module's documentation. The documentation will appear in a viewer as shown in Figure 8.1.

```python
>>> import simple
>>> help(simple)
```

8.6   The `if __name__ == '__main__':` trick

Python modules can be used in two contexts. They can be executed directly or they can be imported by other Python modules. The only way of differentiating these two cases is to check a special variable `__name__` whose value will differ based on the context. If a module is executed directly the value of that variable will be the string `'__main__'`. If the module is imported its value will be the name of the module. This variable is automatically set by the Python interpreter.

Chapter 9

Style

A programming language's syntax constraints how you can write a program, but as a programmer you still have considerable freedom. Even when implementing the same algorithm different programmers will have code that looks differently; it has a different *style*.

While source code describes what a program should do in a language that a computer can execute or translate and execute, it is first and foremost meant to be worked with by humans. As such it should be readable and clearly communicate the programmer's intent. Code that is hard to understand is more likely to contain bugs and harder to work with.

When it comes to code style novelty is bad as it distracts. The most important rule for coding is that your programs should be internally consistent. Use one naming scheme and stick to it. Use one style of capitalization and stick to it. Use white space consistently. When extending other peoples' code, adhere to the style of that code for your extension. Code that has no internally consistent style is unpredictable and hard to work with.

Another way in which your code can be novel (in the bad sense) is if it does not adhere to the idiomatic style of a language. For Python that style is documented in a document called PEP 8. What follows below are some highlights from PEP 8 combined with a suggestions for a naming scheme. You should read through PEP 8 and try to stick to that style of Python.

9.1 Code layout

9.1.1 Indentation

Python has so-called *significant white space*. Indentation is used to delimit blocks of code (like loop bodies and function bodies). While it is possible to indent using a mix of tabs and spaces, this is a source of errors and should be avoided. The vast majority of Python programmers use four spaces to indent, and we will require that style of indentation for this course. It is possible to configure your text editor in such a way that pressing the tab key adds four spaces in stead of one tab character. Most editors understand the Python syntax and help you indent correctly.

9.1.2 Usage of white space

Like nicely readable text is broken up into paragraphs using white space, it is a good idea to do the same for your code. Use single empty lines inside functions to guide the eye through the different parts of your algorithms. If these parts are complicated, precede them with a small comment explaining the what the code is intended to do.

9.1.3 Line length

Lines of Python code should be no longer than 79 characters. This limit is a holdover from older computer systems that used IBM punch cards with only 80 columns. It is still a useful rule because

long lines of code are harder to read (just like normal text). You can occasionally break this rule if a certain line is not much longer than the limit and would break in a weird place.

9.1.4   No deeply indented code

Deeply nested code is hard to read and probably trying to do too much at once. Strive to not nest much deeper than three levels deep (not counting the indentation associated with function bodies or class definitions. Consider splitting up deeply nested code into functions. Inside loops the `continue` statement is sometimes useful to limit how deep the code is nested.

9.2   Naming things

You should take care to use descriptive names when naming objects in your code. If you use abstract or really short names, it will be more work to read your code because you have to remember what everything means.
Some suggestions:

- **functions:** Use verbs for function names.

- **variables:** Use nouns for variable names. For containers such as tuples, lists, dictionaries and sets use plural nouns. For normal single objects like floats and strings use singular nouns.

- **classes:** Use nouns.

9.2.1   Avoid built-in names

Python has a number of built-in names, such as `print` or `for`. These names should not be used as identifiers in your code. Sometimes avoiding built-in names is hard, in that case you can add an underscore to the end of the identifier like `class_` or `id_`. A full list of built-ins can be found in the Python documentation. Note that most competent text editors use special colors for Python built-ins.

9.3   Capitalization

Python has a proper way of using capitalization, which we will adhere to in this course. Note that the Python standard library was written over many years meaning that some of the older parts of it may not follow the standard rules regarding capitalization (the `unittest` library is a good example).

- **variables:** Use lowercase with underscores as needed for readability like `greeting`.

- **functions:** Use lowercase with underscores as needed for readability like `say_hello` or `sort`.

- **classes**: User capitalized words like this: `CapitalizedWords`.

- **function arguments**: User lower case characters.

9.4   Comments

Comments help to explain code. They should not be explaining trivial things, nor should every line of code be commented. Ideally code is self explanatory and no comments are needed. If a particular stretch of code is hard to understand, or otherwise tricky, precede it with a comment explaining what the code does.

9.4.1 Avoid commented-out code

During development it is quite common to comment out certain pieces of code. This is fine, but do not keep this code around. Remove it before handing it in[1], or giving it to someone else. The problem with commented-out code is that it does not get executed, and can therefore contain all kinds of mistakes and errors. It adds nothing useful to the program, is a source of misunderstanding or at best a distraction.

---

[1]Do not hand in homework with commented out code, we will be strict about this rule during the course Basic Linux and Coding for Astronomy and Astrophysics.

Chapter 10

Matplotlib

10.1 Introduction

Matplotlib is the standard when it comes to plotting Python. There are other packages available, but none is as full featured and widely used as Matplotlib. Matplotlib was inspired by Matlab's plotting facilities, which at times makes its API (Application Programming Interface) feel a bit foreign to an experienced Python programmer. What follows are a few examples of standard plotting tasks in Matplotlib to get you started. Extensive documentation is available online. One note about older Matplotlib documentation, some of it uses `pylab`, which you should avoid as it makes scripts hard to maintain (look for examples using `pyplot` in stead).

10.2 Simple use of `pyplot`

You can plot in several ways using Matplotlib. The simplest way is to use the `pyplot` module and allow it to keep track of which figure and plot you are working on. The first two code examples demonstrate this way of using `pyplot` (screenshots are shown in Figure 10.1). Matplotlib includes a built-in viewer, that can be invoked using the `pyplot.show()` function. This viewer can be used to zoom in on a plot and to save it. Below the code that creates the scatter plot using the `pyplot.scatter` function. It expects one or two lists (or arrays) of coordinates, one for the x-coordinates and one for the y-coordinates, these lists (or arrays) must be of the same length.

```python
from matplotlib import pyplot
from random import normalvariate

# generate data:
X1 = [normalvariate(0, 2) for i in range(1000)]
Y1 = [normalvariate(0, 2) for i in range(1000)]

# create plot:
pyplot.scatter(X1, Y1, color='k', marker='+', label='Gaussian')
pyplot.legend()
pyplot.title('Distribution')
pyplot.xlabel('x-axis')
pyplot.ylabel('y-axis')
pyplot.show()
```

The `marker` keyword argument of `pyplot.scatter` sets the marker to be used for each point. Possible markers are o for circles, . for dots, + for plusses, x for crosses, * for stars and several other possibilities that can be found in the Matplotlib documentation. The color of the points is set using
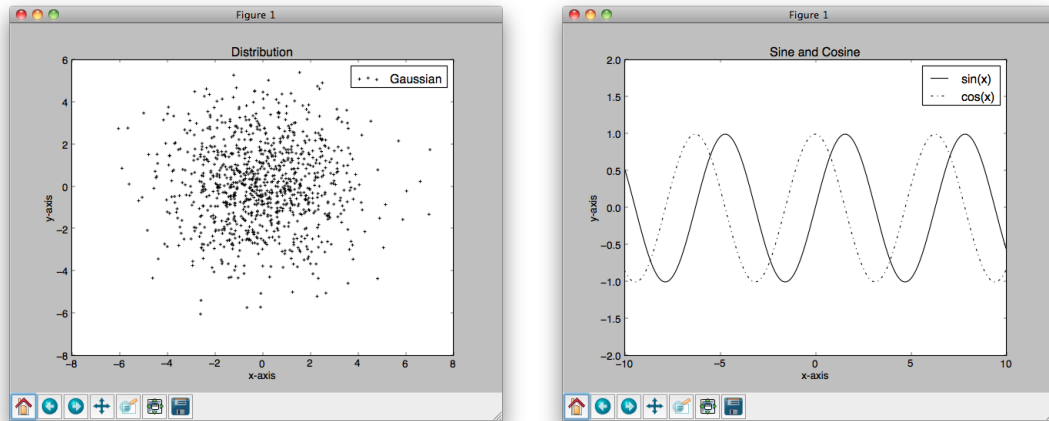
(a) Simple scatter plot using `pyplot.scatter`.

(b) Simple line plot using `pyplot.plot`.

Figure 10.1: These two screenshots show the interactive viewer included with Matplotlib. The interactive viewer allows you to zoom in on the data and save the plots as well.

the `c` keyword argument, you can pass it any color that Matplotlib understands[1]. The `pyplot.title` function sets the title for the entire image and `pyplot.xlabel` and `pyplot.ylabel` set respectively the label for the horizontal axis and for the vertical axis. The legend was created by labeling the data that was plot (the `label` keyword argument of `pyplot.scatter` and subsequently using the `pyplot.legend` function.

The second example (below) shows how line plots can be constructed using the `pyplot.plot` function. Just like `pyplot.scatter` this function expects one or two lists (or arrays) of numbers to be used as coordinates. The color and style of each line can be set in several ways but here we have used the shorthand notation for a black line and a black dot-dashed line, respectively 'k-' and k-. Note that this example uses Numpy arrays, and functions to create the data.

```python
import numpy
from matplotlib import pyplot

x = numpy.linspace(-10, 10, 201)

pyplot.plot(x, numpy.sin(x), 'k-', label='sin(x)')
pyplot.plot(x, numpy.cos(x), 'k-.', label='cos(x)')
pyplot.ylim([-2, 2])
pyplot.xlabel('x-axis')
pyplot.ylabel('y-axis')
pyplot.legend()
pyplot.title('Sine and Cosine')
pyplot.show()
```

Axes can be limited to a certain range using `pyplot.xlim` and `pyplot.ylim` functions to respectively constrain the horizontal and vertical axes.

The third example shows a simple historgram created using the `pyplot.hist` function. Unlike the plots in the earlier examples, which were shown using the interactive plotting component of Matplotlib, the third example saves the figure directly to a PDF (Portable Document Format) file. Using PDF is a good idea because Matplotlib saves your plot as a vector image which will look sharp and clean even when enlarged.

---

[1] `http://matplotlib.org/api/colors_api.html`
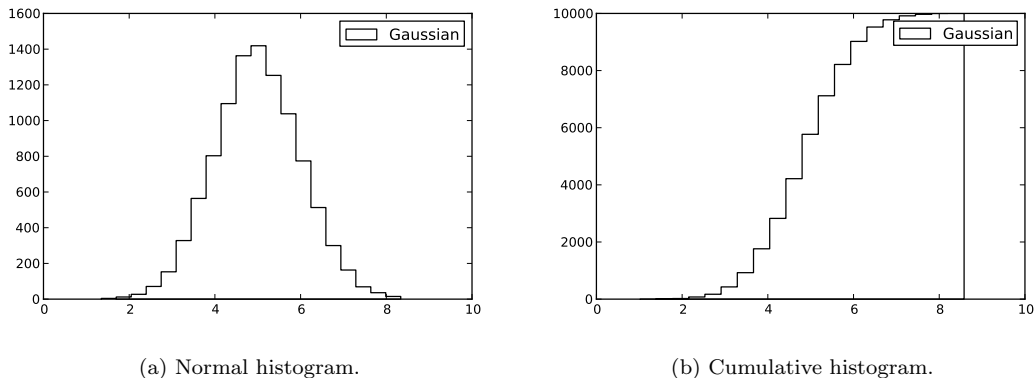
(a) Normal histogram.

(b) Cumulative histogram.

Figure 10.2: Simple histograms created with `pyplot.hist` and saved as a PDF file using `pyplot.savefig`. The cumulative plot on the right was created using the `cumulative=True` keyword argument to the `pyplot.hist` function.

```
import numpy
x = numpy.random.normal(5, 1, 10000)


pyplot.hist(x, 20, histtype='step', color='k', label='Gaussian')
pyplot.xlim(0, 10)
pyplot.legend()
pyplot.savefig('pyplot-histogram.pdf')
```

The first argument to `pyplot.hist` is a list or array of number for which a histogram is constructed. The second, optional, argument is the number of bins that should be created (20 in this example). The `histtype` keyword argument controls how the histogrgam looks and can be set to `bar`, `barstacked`, `stepfilled` or `step`. The color of the histogram is further controlled using the `color` keyword argument. The histogram can be turned into a cumulative histogram setting the `cumulative` keyword argument of `pyplot.hist` to `True`. Similarly the `normed` keyword argument can be set to `True` to normalize the histogram such that the histogram will sum to 1. The `orientation` keyword argument that defaults to `'vertical'` argument controls the direction of the bars in the histogram.

As is visible in Figure 10.2 the default position of the legend can be inappropriate. The `loc` keyword argument to the `pyplot.legend` function controls the position of the legend in the figure and can take the following values: `'best'` (which is default), `'upper right'`, `'upper left'`, `'lower left'`, `'lower right'`, `'right'`, `'center left'`, `'center right'`, `'lower center'`, `'upper center'`, and `'center'`.

10.3   Multiple plots in one figure

To add several plots to one figure recent versions of Matplotlib have the `pyplot.subplots` function to create an array of subplots while older versions have the less convenient `pyplot.subplot`.

10.3.1   Using `pyplot.subplots`

The following example shows how to create a figure with two subplots in it.

```
import numpy
from matplotlib import pyplot


fig, axes = pyplot.subplots(nrows=1, ncols=2)
X = numpy.linspace(-10, 10, 201)
```
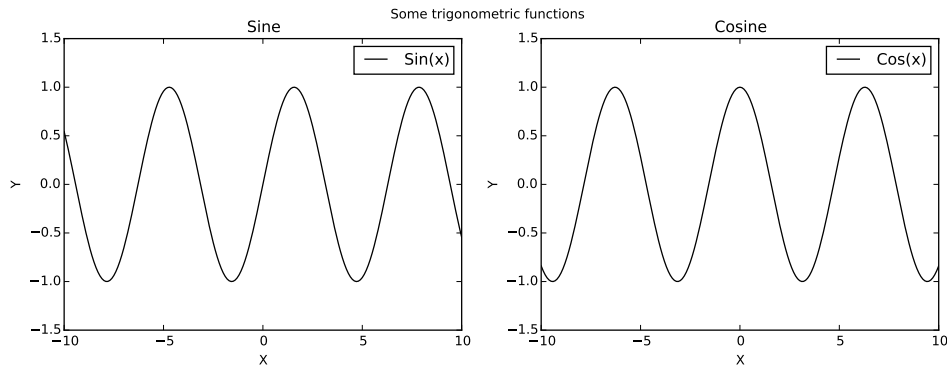
Figure 10.3: Two subplots created using `pyplot.subplots`.

```
axes[0].plot(X, numpy.sin(X), 'k-', label='Sin(x)')
axes[0].legend()
axes[0].set_title('Sine')
axes[0].set_ylabel('Y')
axes[0].set_xlabel('X')
axes[0].set_ylim([-1.5, 1.5])

axes[1].plot(X, numpy.cos(X), 'k-', label='Cos(x)')
axes[1].legend()
axes[1].set_title('Cosine')
axes[1].set_ylabel('Y ')
axes[1].set_xlabel('X')
axes[1].set_ylim([-1.5, 1.5])

w, h = fig.get_size_inches()
fig.set_size_inches(2 * w, h)
fig.suptitle('Some trigonometric functions')
fig.savefig('subplots-example-1.pdf')
```

The `fig` object a reference to the figure as a whole (an instance of `matplotlib.figure.Figure`) and the `axes` object is an array of `matplotlib.axes._subplots.AxesSubplot` instances. To set the limits and labels on the individual plots you can no longer use the `xlabel`, `ylabel`, `xlim` and `ylim` functions of `pyplot`, instead use the `set_xlabel`, `set_ylabel`, `set_xlim` and `set_ylim` methods of the individual plots. The example furthermore demonstrates how the size of the figure can be found and changed using the `get_size_inches` and `set_size_inches` methods of the figure. If you want to set an overall title for the figure use the `suptitle` method of the figure object. It should be noted that `pyplot.subplots` will return a two-dimensional array if both `nrows` and `ncols` are larger than one, a one-dimensional array if either `nrows` or `ncols` is one or just a reference to a subplot if both `nrows` and `ncols` are one.

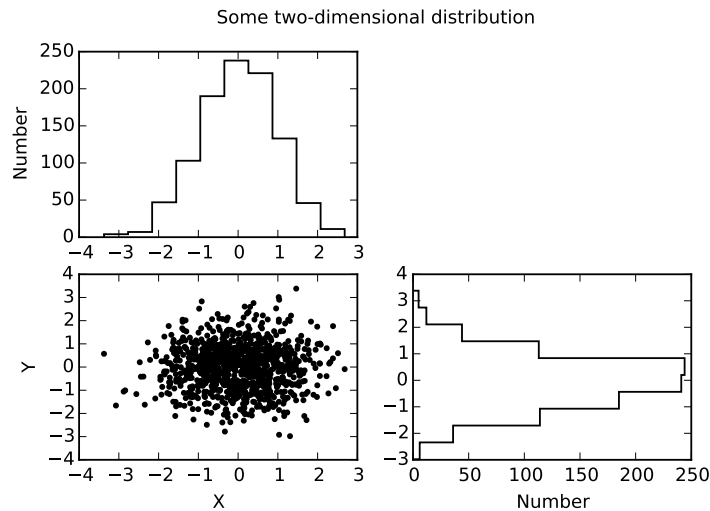## 10.3.2   Using `pyplot.subplot`

While `pyplot.subplots` returns subplot and figure references, `pyplot.subplot` does not, in stead it activates one of the subplots and all subsequent drawing commands go to that subplot. Below a simple example showing a scatter plot and two one-dimensional histogram (see also Figure 10.4).

```
from matplotlib import pyplot

X = numpy.random.normal(0, 1, 1000)
```

Figure 10.4: Three subplots created using `pyplot.subplot`.

```
Y = numpy.random.normal(0, 1, 1000)

pyplot.suptitle('Some two-dimensional distribution')
pyplot.subplot(2, 2, 1)
pyplot.hist(X, 10, orientation='vertical', histtype='step', color='k')
pyplot.ylabel('Number')
pyplot.subplot(2, 2, 3)
pyplot.scatter(X, Y, marker='.', color='k')
pyplot.xlabel('X')
pyplot.ylabel('Y')
pyplot.subplot(2, 2, 4)
pyplot.hist(Y, 10, orientation='horizontal', histtype='step', color='k')
pyplot.xlabel('Number')

pyplot.savefig('subplot-example-1.pdf')
```

The arguments to `pyplot.subplot` are: the number of rows, the number of columns and *plotnumber* of the subplot to be activated. The *plot number* numbering scheme starts with the top left plot and moves through the other plots in that row before moving to the next row, etc. These arguments can be supplied separately as is done in the example above, or a shorthand can be used in special circumstances. Provided that each of the parameters to `pyplot.subplot` is smaller than 10, the numbers can be combined into one parameter. To activate the top left plot in an table of plots with two rows and two columns you can use the shorthand `221`, where each position in the number will be treated as one of the arguments — the call then becomes `pyplot.subplot(211)`. This shorthand is used quite often in examples in the Matplotlib documentation (but is not very *Pythonic*).

10.4   Customizing tickmarks

While Matplotlib conveniently creates tickmarks in appropriate locations for your figures, you can also control them yourself using `pyplot.xticks` and `pyplot.yticks`. The following example shows how the position, text, and rotation of tickmarks and their labels can be controlled (see Figure 10.5). For more precise control see `matplotlib.ticker` in the Matplotlib documentation.
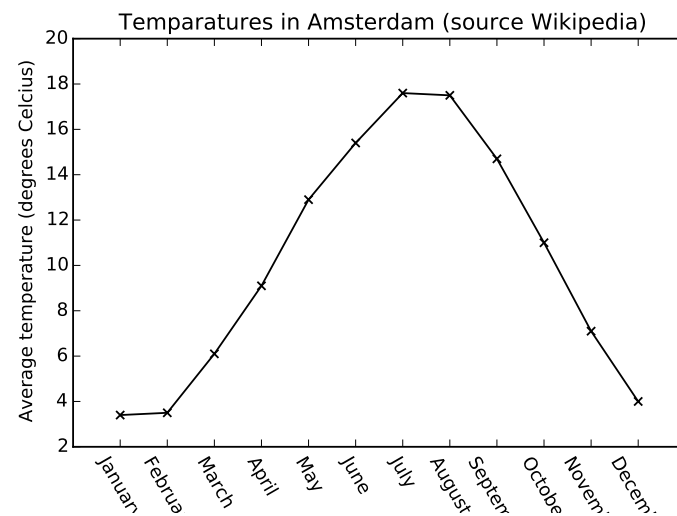
Figure 10.5: Controlling tickmark label position, text and rotation.

```python
import calendar
from matplotlib import pyplot

months = list(calendar.month_name)[1:]
temps = [3.4, 3.5, 6.1, 9.1, 12.9, 15.4, 17.6, 17.5, 14.7, 11.0, 7.1, 4.0]

pyplot.scatter([i for i in range(12)], temps, marker='x', color='k')
pyplot.plot([i for i in range(12)], temps, 'k-')
pyplot.xticks([i for i in range(12)], months, rotation=-60)
pyplot.xlim([-1, 12])
pyplot.title('Temparatures in Amsterdam (source Wikipedia)')
pyplot.ylabel('Average temperature (degrees Celcius)')
pyplot.savefig('xticks-1.pdf')
```

Chapter 11

Error handling and some debugging tricks

When programming errors are inevitable, how you deal with them influences the speed of development and correctness of your program. Python uses exceptions as a way to communicate errors, knowledge of exceptions allows you to detect and handle errors appropriately. In this chapter we will go over the basics of exceptions and show you how you can use assertions (explained below) to more quickly find problems in your programs.

11.1 `try`, `except` and `else`

Python has *exceptions* that are *raised* whenever an error occurs. When an exception is not handled it will cause the Python program to crash. Fortunately Python allows you to detect the occurrence of exceptions and to act on them using the `try` and `except` statements. These statements always occur together. The `try` clause contains a statement or a piece of your program that can potentially fail, while the `except` clause contains the code that deals with such an expected failure when it occurs.

Beyond the `try` and `except` keywords there is also an `else` keyword that can be added to the try-except construct. The code contained in the else clause is only executed when no error occurs. The example below shows a try-except-else construct for a trivial program.

```python
try:
    val = 1 / 0
except ZeroDivisionError:
    print 'Zero division occurred - handled it - no problem.'
else:
    print 'No exception ocurred.'
```

When you use the try-except construct it is a good idea to capture only those errors that you expect. In the case of the example above you may expect a `ZeroDivisionError` and handle that problem. If any other exception occurs, the code will still stop executing with a traceback allowing you to investigate the problem. If you write a program that use try-except constructs to suppress all errors it will be hard to debug your program, or to make it correct. The example below shows you how *not* to write your error handling:

```python
import math

def function_with_errors():
    """Three different errors."""
    1 / 0
```

```
        'abcd'[0] = 'x'
        math.log10(0)


# This is bad as it hides errors:
try:
        function_with_errors():
except:
        print 'See no problems at all!'
\end{python}
```

If you do **not** know what errors to expect, just run your code **and** when it crashes take note of the \texttt{Error} **or** \texttt{**Exception**} that you **is** raised. It may indicate a problem that can check **for** at run time using a **try**-**except** construct. Since you saw what error occurs you can write error handling code that handles only exactly that problem (**in** an appropriate **except** clause). Your program may still crash when other errors occur, but at least you will have the information you need to solve them.

Do note that **not** all problems should be addressed by using the \texttt{**try**} **and** \texttt{**except**} statements. Some are better avoided altogether by actually fixing the underlying cause of the problem.

Raised exceptions are objects themselves that can be manipulated **in** the \texttt{**except**} clause, but to do so you need to assign them to a local variable. Python has the \texttt{**as**} keyword **for** that. The following example tries to create a directory. Trying to create a directory that already exists normally raises an exception that stops your program, but **in** the example below that problem **is** dealt **with** using a specific \texttt{**except**} clause. This clause **not** only looks **for** \texttt{**OSError**} exceptions, but it also assigns the error to a local variable \texttt{e} **and** inspects it. If the error number indicates a pre-existing directory the error **is** handled. Otherwise it **is** re-raised using the \texttt{**raise**} keyword. Re-raising an error may lead to a crash, but at least you are **not** hiding problems **in** your program.

```
\begin{minted}{python}
import os
import errno


try:
    os.mkdir('coolstuff')
except OSError as e:
    if e.errno != errno.EEXIST:
        raise  # unknown errors are re-raised here
```

11.1.1   A note about Python 2 syntax

Using as to assign an exception to a local variable is safer than the original (older and not recommended) syntax. In Python 2 it is possible to check for several types of errors in one except clause. The most obvious syntax for that is wrong as the example below demonstrates.

```
# THIS IS AN EXAMPLE CONTAINING A CLASSIC PYTHON 2 BUG


# This does not look for both ZeroDivisionError and RuntimeError. Instead it
# redefines what RuntimeError means by assigning ZeroDivisionError to it!
```

```
try:
    val = 1 / 0
except ZeroDivisionError, RuntimeError:
    print type(RuntimeError)
```

The example program above is not testing for two types of exceptions, it only tests for ZeroDivision-Error and assigns any ZeroDivisionError that is caught to the variable RuntimeError as can be seen by running the program:

```
Gretchen:coolstuff thijscoenen$ python exception-handling-3.py
<type 'exceptions.ZeroDivisionError'>
Gretchen:coolstuff thijscoenen$
```

For completeness the correct version of the last snippet is given below.

```
try:
    val = 1 / 0
except (ZeroDivisionError, RuntimeError) as e:
    print 'Either a ZeroDivisionError or a RuntimeError occured'
    print 'It was a', type(e)
```

11.2   Assertions

Before describing assertions it is good to know that they are no substitute for proper error handling. Having said that, assertions are a quick way to help you write correct programs. An assertion is a way of stating your assumptions about a porgram in a way that Python will check for you. If an assumption does not hold your program will stop executing with an AssertionError. An AssertionError is useful because it tells you explicitly which of your assumptions does not hold. As before, when the try/except/else construct was described, the idea is that a clear failure is more useful than a program that hides problems, is hard to reason about, and becomes unpredictable. Assertions start with the assert keyword and are followed by a Boolean expression. Below are a few examples of assertions:

```
>>> assert True
>>> assert False
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  AssertionError
  >>>
  >>> a = 10
  >>> b = 12
  >>> c = list()
  >>> assert a > 0
  >>> assert a > b
  Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    AssertionError
  >>> assert type(c) == type([])
```

The following example shows how you can use assertions to check that the inputs to a function that calculates factorials are infact integers and that they are 1 or larger:

```
def factorial(n):
    """
    Calculate the factorial of n.
```

```
    """
    assert type(n) == type(1)
    assert n > 1
    fact = 1
    for i in range(n, 1, -1):
        fact *= i
    return fact


if __name__ == '__main__':
    print factorial(4)
    print factorial(3.1415)
```

When you run this program you get the following outputs:

```
thijscoenen@gretchen: errors$ python assertion_example.py
24
Traceback (most recent call last):
  File "assertion_example.py", line 14, in <module>
    print factorial(3.1415)
  File "assertion_example.py", line 5, in factorial
    assert type(n) == type(1)
AssertionError
```

At the start of this section it was mentioned that assertions are not a substitute for proper error checking. They are a way to help spot problems during development. One of the reasons why assertions are no substitute for proper error checking is that it is possible to run a Python program while ignoring assertions. The following example shows you how:

```
thijscoenen@gretchen: errors$ python -O assertion_example.py
24
Traceback (most recent call last):
  File "assertion_example.py", line 14, in <module>
    print factorial(3.1415)
  File "assertion_example.py", line 8, in factorial
    for i in range(n, 1, -1):
TypeError: range() integer start argument expected, got float.
```

As you can see the assertions are ignored and the range function complains that it got a floating point number in stead of an integer.

Chapter 12

Recap and further reading

12.1 Course goals

The goals for the Basic Linux and Coding for Astronomy & Astrophysics course have been described in the Introduction to this reader. Because all of Python cannot be learned from scratch in the amount of time allowed for the course, a practical subset of Python was defined that you must understand to pass. The following subsections list the parts of Python you are expected to understand. In this context "understand" means that you can explain them in your own words and that you can apply that knowledge when designing and implementing programs.

12.2 A practical subset of Python

You are expected to know the basic Python data types, what they are used for, how to convert them into each other. You are expected to know how to iterate over the various sequence and mapping types (both by using indices and by direct iteration) and how to perform simple operations on them (including arithmetic). Furthermore you should know how you find the type of some value or variable.

- the assignment operator

- number types: floating point numbers and integers

- sequence types: strings, tuples, and lists

- mapping types: dictionaries

- the set datatype

- Booleans and `None` type

- the literal notations for these data types (!)

- indexing and slicing of strings, tuples, and lists

- immutable versus mutable data types

- the `type` function

- the `int`, `float`, `str`, `list`, `dict`, and `tuple` functions

- `list.sort`, `list.append`, `list.extend`, and `list.pop` methods

- `min`, `max` and `len` functions

You are expected to understand Boolean expressions, Boolean operators and comparison operators.

- `and`, `or` and `not`

- Truthy and Falsy as applied to strings, tuples, lists and dictionaries

- the comparison operators (like `==` etc.)

- the `bool` function

- the `in` operator as applied to strings, tuples, lists, dictionaries and sets

You should know how to conditionally or repeatedly execute a blocks of code. You should also know the different ways you can iterate over strings, tuples, lists, dictionaries, and sets.

- `if`/`elif`/`else`

- `for`/`else` and `while`/`else`

- the `break` statement

- the `continue` statement

- the `range` function

- the `enumerate` function

You should be able to write a properly structured Python program. Meaning that the program is modular — for the purposes of this course modular at the level of functions and modules is enough (you are not expected to understand classes and packages are not necessary). While designing a program you should be able to break up a problem into smaller parts that are more easily solved. The functions you implement should have clear purposes and be as independent of the rest of the program as possible — meaning do not use the global scope to communicate state between functions, make sure functions only solve one part of the problem. You should understand what scope is, how scopes nest, which Python constructs create scopes of their own and which do not. Your modules should allow reuse of their contents so you should understand how you can prevent code from running at import of a module. Furthermore you should be able to document your code and know the difference between documentation, i.e. docstrings, and comments in your code.

- the `def` keyword and function definition

- understand the difference between positional arguments and keyword arguments

- understand how you can give default values to function arguments

- what scope and namespaces are and how they are used

- the `if __name__ == '__main__':` trick

- docstrings (for modules and for functions) and how they differ from comments

- what a Python module is

Last but not least you should be able to write programs in proper Python style, which includes proper use of white space, descriptive naming schemes, comments, and indentation (4 spaces). In general you should follow the Python style rules as laid out in PEP 8.

12.2.1 Numpy and Matplotlib

You are expected to be able to create plots with `Matplotlib` and use `Numpy` arrays during the homework exercises. You will not be expected to know them by heart during the exam. You are, however, expected to be able to explain at a high level the use of arrays and what makes them different from lists.

12.3   More Python resources

Python has a large community that produces a large number of libraries and a large amount of documentation. Various learning resources are available for free on-line.

- For the absolute beginner there is the *Learn Python the hard way course* by Zed Shaw. It assumes very little computer knowledge and takes small steps in teaching.

- As an alternative to the Python parts of this reader, you can also read the relevant chapters of *Think Python* by Allen B. Downey. This book is an appropriate resource for learning Python 2. It is a more complete introduction to Python than can be given in this course.

- If you already know quite a bit of Python the *Intermediate Python* website is a nice resource to dive deeper in more advanced Python constructs.

Part II

Assignments

Chapter 13

Finding Pulsars from the command line

13.1 Introduction

Pulsar surveys in radio produce large numbers of potential pulsars, these then have to be inspected. For this exercise you are given a large number of potential pulsars. For each potential pulsar you will find a `.bestprof`-file and a `.png`-file. The former is a text file with meta data while the latter is a diagnostic plot. Below I will use detection to describe one of these potential pulsar signals (whether a real pulsar or not). The data set should contain a pulsar that you have to find using command line tools.

The data you need is available from the `http://www.astroprog.nl` website.

13.1.1 Grading

While this exercise does not count towards your final grade, it is required to be handed in.

13.1.2 The assignments

- Start a report to be handed in as a A4-sized PDF file. Make sure you include your name and student number on the first page of your report.

- Download the files `*.tar.gz` (containing a number of potential pulsar candidates) and copy it to a directory where you can work.

- Unpack the tarballs, you will now be presented with a number of sub directories containing the images and metadata files.

- Open one of these `.png` images from the command line. If you don't know which command to use to open these images, perform a web search to find out which command will do that. (Add the command you used to the report.)

- Now that you have seen the plot, open the metadata (hint: the bestprof files are text files). Looking through this file you will see that the reduced chi-squared is an entry in the header, we will need these later. How many entries are there in the header? (Add the number to your report.)

- How many candidates are there in the complete data set? (Add the answer to your report.)

- Add the picture of the brightest profile to your report (the brightest candidate will have the highest reduced chi squared). You should use the meta data to find the brightest candidate. Also add to the report how you found the brightest one.

- Now add the 83th brightest profile you report and explain how you found it. (The reader chapter about BASH contains a hint as to how you could do this.)

Chapter 14

Parsing text files (Homework for week 3)

The KNMI (Royal Dutch Meteorological Institute) offers a dataset for download that covers weather observations from the Dutch weather stations. In this assignment you will build a program to load the data that the KNMI offers and convert it to appropriate Python data types. The data is available from the KNMI website at: `http://projects.knmi.nl/klimatologie/daggegevens/selectie.cgi` In building this program you will encounter the basic Python datatypes, some flow control and iteration. The goals for this week are to:

- read a text data file (provided by KNMI),

- use the basic Python types and conversions between them,

- properly name variables in your program, and

- to structure a simple program.

This exercise will be graded as follows, you can earn at most 9 points by completing the exercise and handing in on time. Your grade g, awarded on a scale from 1 to 10, is calculated as follows:$g = 1 + s$, where s are the points you earned. For this exercise the points are awarded for the following things:

(3 points) Style, in this case appropriate variable naming, code layout and comments.

(4 points) For completing Section 14.1 correctly.

(2 points) For completing Section 14.2 correctly.

Note you cannot use pre-built CSV readers of any kind, this exercise is about building a simple one yourself.

14.1   Assignment part 1

- Go to the KNMI webpage, and download all the data you can. Make sure that the data you download starts at 1901 01 01 and runs through to today, for all weather stations in the Netherlands. Save the text file to an appropriate directory. The file you download will in the 100 MB - 200 MB size range.

- When building a program it is convenient to start with a small amount of data so that you program runs quickly when you are still looking for errors. Use one of the commandline tools you learned about and grab the first 500 lines of the data file and write them to a new file. You will now have an appropriately small dataset that contains both a header and a data part.

- Start a new Python file, possibly using the template from Section 14.3, name the file `read_data.py`. Make sure to add you name and student number to this file.

- Look up how you can access a file line-by-line (see Chapter 7). Find out how you can split a string on a certain character. Using this write a simple program that opens the small data file, skips the header, reads every line, splits it on an appropriate character and turns the strings into numbers.

- Start a document as well (A4-sized PDF). How do you deal with missing data and why? What are appropriate values if you choose to use a number to represent missing data? (Add to the report, refer to this question.)

- Change the program you wrote so far. The function `read_file` should return a list of lists of numbers. Where each inner list corresponds to line in the data file with appropriate values for missing data.

- The header consists of 3 parts. Separate these out into several files that we will use in building a programs to read the header.

- Build a program that can read the part of the header where the stations and their locations are defined. The format of this data should be a list containing one list for each station. That list for each station should contain 5 entries (station number, longitude, latitude, altitude, its full name) in appropriate data types. Explain in you report why you chose the data types that you did. Call this program `station_locations.py`.

- Build a program that can read the part of the header that explains the meaning of the column names. The output of this program should be a dictionary mapping the column name to its description. Call this program `column_meaning.py`.

- Build a program to read the line of column headers, its output should be a list of column names.

14.2   Simple analysis part 2

If your program that reads the data entries and skips the header works on the small data file. Create a new Python program called `analysis.py`, copy to it the aforementioned program and adapt it to answer the following questions.

- Find the wettest day in your data set, find the hottest day. Where were these records broken? Note these can be found by looping through the data set and keeping track of the highest encountered value (of e.g. temperature) and the date that it occured at.

- Compare the summers in "De Kooy" with those in "Valkenburg". Calculate monthly averages for min, max temperature and the amount of precipitation for 2016. Add a table to your report with these monthly average for 2016.

14.3 Hints

Below you find a template that you can use as a start for this exercise. If you choose to use it, copy it to a Python source file and run it. It should give no errors before you move to the rest of the exercise. Note that copying and pasting the program below will probably not work.

```python
#!/usr/bin/env python
"""
Parse KNMI data and do simple analysis.
"""
# this program scaffold shows you how you can structure a program


def read_knmi_data_file(filename):
    """Read KNMI data"""
    print 'Loading the file', filename
    data = []
    # you write the data reading part here
    return data



def main(filename):
    """Run program for exercise of week 2"""
    data = read_knmi_data_file(filename)
    # you can work with the data here



if __name__ == '__main__':
    main('knmi-data.txt')
```

Chapter 15

Simple plotting

The goals for this week are to:

- apply the rules for program structure in Python,

- produce some basic plots using `matplotlib`.

This exercise will be graded as follows, you can earn at most 9 points by completing the exercise and handing in on time. Your grade g, awarded on a scale from 1 to 10, is calculated as follows:$g = 1 + s$, where s are the points you earned. For this exercise the points are awarded for the following things:

(3 points) Style: proper naming scheme, documentation,

(3 points) For completing Section 15.1 correctly.

(3 points) For completing Section 15.2 correctly.

15.1 Assignment part 1

This part of the exercise is a warm up of what is to follow. For this part of the exercise you will create a parser for a climate reconstruction by the Berkeley Earth project.

- Create a report for this week, where you can add the plots you created. As always you should use a A4 sized PDF file.

- Download the data available from `http://berkeleyearth.lbl.gov/auto/Global/Land_and_Ocean_complete.txt` and save it to a file called `Land_and_Ocean_complete.txt`.

- As you can see by opening the text file in a text editor, the structure of the file is much simpler than that of the KNMI data of last week. Note that there are two data sets in the file, read only the first.

- Starting from the scaffold provided below, write a program to read the data and plot the monthly average temperature anomaly. See the points below for the steps this entails.

- Fill the function body of `read_data` with code that opens the file, skips the header in that file, reads in all the data (i.e. iterates over all lines and converts the columns to appropriate data types) and returns the data that was read. You can represent the missing values as not-a-number: `float('nan')`.

- Fill the function body of `plot_data` with code that makes two lists, one for the months since january 1850 (you are allowed to just count them from 0), and a list for the monthly temperature anomaly. You can now use the `plot` function from `pyplot` to create a plot of the monthly temperature anomaly since January 1850.

- Make sure to properly label the plot you create before handing in.

- Make sure to add docstrings to your program.

- Make sure that your program runs correctly when the data file is placed in the same directory.

- Add the plot you created to your report.

```python
#!/usr/bin/env python
from matplotlib import pyplot


def read_data(filename):
    pass


def plot_data(data):
    pass


def main(filename):
    data = read_data(filename)
    plot_data(data)


if __name__ == '__main__':
    main('Land_and_Ocean_complete.txt')
```

15.2   Assignment part 2

In this part of the exercise you will fix and refactor the script you wrote last week to read the KNMI data and plot it. In this part of the exercise you are allowed to skip reading the header and just use your knowledge about the different columns in the data.

- Below a program scaffold is provided that you can use to fix the KNMI data file parser of last week. Note that the third part of the header is already parsed by the program scaffold.

- Using your own code of last week and optionally the provided scaffold, build a parser for the KNMI data file of last week. This time around we are not modifying the data file. Note that the filename `chunk.txt` is a placeholder for the full data file.

- Add a function body to the `plot_max_temperature` that plots the maximum temperature at a certain weather station (identified by station number).

- Make sure to properly label the plot you create before handing in.

- Make sure to add docstrings to your program.

- Make sure that your program runs correctly when the data file is placed in the same directory.

- Add the plot you created to your report.

```python
#!/usr/env/bin python


def read_station_locations(file_obj):

    # look for start of the first part of the header
    for line in file_obj:
        if line.startswith('# STN'):
            break

    # parse the station locations until an empty line is reached
    station_locations = []
    for line in file_obj:
        if line.startswith('#') and not line[1:].strip():
            break

        # parsing code goes here:
        print 'STATION LOCATION', line

    return station_locations


def read_abbreviations(file_obj):
    # parse abbreviations until an empty line is reached
    abbreviations = {}
    for line in file_obj:
        if line.startswith('#') and not line[1:].strip():
            break

        # parsing code goes here
        print 'ABBREVIATOIN', line

    return abbreviations


def read_column_names(file_obj):
    """Read column names from KNMI data file"""
    column_names = [cn.strip() for cn in next(file_obj).split(',')[1:]]

    return column_names


def read_data(file_obj):
    """Read the weather observations from the KNMI data file"""
    pass  # parse the body of the data file here


def read_knmi_data_file(filename):
    """Fully parse the KNMI data file"""
    with open(filename, 'r') as f:
        station_locations = read_station_locations(f)
        abbreviations = read_abbreviations(f)
        column_names = read_column_names(f)
```

```python
        data = read_data(f)

    return station_locations, abbreviations, column_names, data


def plot_max_temperature(data, station_number):
    """Given its number plot max temperature over time at a weather station"""
    pass


def main():
    """Parse the KNMI data file and plot the max temperature at the Bilt."""
    station_locations, abbreviations, column_names, data = \
        read_knmi_data_file('chunk.txt')
    plot_max_temperature(data, station_number=260)


if __name__ == '__main__':
    main()
```

Chapter 16

Week 5: Arrays and Jupyter

This week we will explore a new data type, the *array*, provided by the Numpy library. Numpy is an absolutely foundational component of the to scientific computing stack in Python. Numpy provides arrays and fast numerical operations on those arrays. Numpy arrays furthermore are used in packages like Scipy as a data format.

The exercise for this week is provided as a Jupyter Notebook (formerly known as an Ipython Notebook). You are expected to hand in your homework as an Jupyter notebook (again clearly marked with your name and student number).

16.1 The exercise

- Download the Jupyter Notebook available from
  `http://www.astroprog.nl/assets/week5/assignment-week-5.tar.gz.`

- Unpack the tarball and start Jupyter in that directory.

- Start the notebook, follow the instructions therein.

Chapter 17

Redetecting Kepler exoplanets

For this assignment you will use data from the Kepler telescope and the NASA exoplanet archive[1].
With the Kepler data you can re-detect previously discovered planets.
Goals for this week:

- demonstrate an ability to design and implement simple programs,

- demonstrate an ability to collect and access tabular data (in text formats),

- access data in FITS files, and run some data analysis on them (fitting and creating periodograms in this case).

This exercise will be graded as follows, you can earn at most 9 points by completing the exercise and handing in on time. Your grade g, awarded on a scale from 1 to 10, is calculated as follows:$g = 1 + s$, where s are the points you earned. For this exercise the points are awarded for the following things:

(3 points) For proper programming style and a programs that are clearly structure.

(2 points) For the assignment in Section 17.1.

(4 points) For the assignment in Section 17.2.

17.1   Part 1: creating an all sky plot

In this part of the assignment you will create a plot of the positions on the sky of the known exoplanets. For this part of the assignment you have to hand in a program, and add to your report the all sky plot you create (with for reference the data file that you used to create it, so that it may be checked).

- The NASA exoplanet archive also provides data on all known exoplanets. This data can be downloaded as simple text files like the ones you processed before. See `https://exoplanetarchive.ipac.caltech.edu/`.

- **Assignment:** Download the sky positions of the known exoplanets, and write a parser for these data. You are allowed to use a library functionality to do the parsing. There are many libraries you may choose to use, like: the Python standard library `csv` module, the Numpy `genfromtxt` function, the Pandas `read_csv` funtion, etc.
  Be sure to include all data files you use in your submission. As a tip: for this exercise you do not need to download all the more than twenty columns from the Exoplanet data archive table.

- **Assignment:** Create an all sky plot of the confirmed exoplanets in the archive using the Hammer projection in Matplotlib. You may have to convert the right ascension and declination to degrees first.

---

[1]See `http://exoplanetarchive.ipac.caltech.edu/`

17.2 Re-detecting planets in Kepler data

For this assignment you will re-detect previously known exoplanets in the Kepler data. For this part of the assignment you have to hand in a program and a report.

- The Kepler mission observed a specific path of sky (the "Kepler field") for several years, repeatedly measuring the magnitudes of the stars in that field[2] The lightcurves created by Kepler show exoplanet transits, and by tracking these over a long time the orbits of these exoplanets can be reconstructed. Each of the stars in the Kepler field has a Kepler ID (sometimes called KOI for Kepler Object of Interest) that can be used to retrieve the data files, these IDs should not be confused with the numbering of the Kepler discoveries (given to planets like Kepler 5b).

- Kepler data files are in the FITS (Flexible Image Transport System) format that is used often in Astronomy. If you want to open FITS files in a Python program you should use the Astropy library (you may come across mentions of PyFits, but that is an older library). Astropy is a large library with a lot of functionality beyond reading FITS files. FITS files contain both header and data sections. The header generally contains information about the observed data and the data section the actual data.
  Assignment: Check that you have Astropy installed. If it is not, you you must install it.

- Find the earliest Kepler exoplanet discoveries, we will use one of these because they easiest to detect in the data (which is exactly why the were detected first by Kepler). The data can be downloaded from the Mikulski Archive for Space Telescopes (MAST)[3].
  Assignment: Download some data files for the Kepler-7b planet. Be sure to choose data files with lightcurves whose baseline does not jump around too much and that shows a number of dips (transits) so that it is easy to reconstruct the orbital period of the planet.
  Note: the website can generate plots for these data files so that you can check them before downloading.

- **Assignment:** Write a function to open the FITS file and extract the raw time-series data and not the baseline corrected one — because part of this assignment is to correct the baseline of the data. Create a plot of the photometry data, make sure your time axis is correct. To access the data in the FITS file use the `astropy.io.fits` module.

- **Assignment:** Inspect your time-series and choose a range of data that has no breaks, and filter out any not-a-number values as these will cause problems in the following analysis.

- Scipy provides a way of creating Lomb-Scargle periodograms, we will use that functionality (do not use the Lomb Scargle functions from other libraries like AstroML). Make sure you find the Lomb-Scargle periodogram function in Scipy.

- Scipy's implementation of these periodograms expects data that is centered around zero. The raw Kepler data is not like that.
  Assignment: Fit a low order polynomial function to the data (so that the planet transits are not also fit) and subtract this from the data. You will now have data that is close to being centered around zero. Furthermore subtract the median of the data to get even closer to zero.
  If you did this part of the assignment correctly you will still clearly see the planet transits while the baseline is now centered around zero.

- **Assignment:** Use Scipy's periodogram functionality to find the period of the planet's orbit. Create a plot of the periodogram and indicate where the planet's signal can be found. Include this plot in your report, and make sure to also include the period you extracted and the

---

[2]Hardware failure meant that it has since moved to observing other patches of sky.
[3]See `https://archive.stsci.edu/kepler/`.

actual period of the planet (from literature, or in this case it is admissible to quote the value you may find on Wikipedia). Provide relevant plots.

- **Assignment:** Extend your program such that it can process several light curve data files in one run of the program. Show that it works for several planets and several light curves per planet. Provide relevant plots.

17.3   Handing in

Check that the following things are included in your submission:

- a report describing your data sources and methods,

- plots you were asked to create,

- all code you wrote to perform your data analysis (able to be executed),

- any ancillary files needed to run your programs.

Furthermore, check that your code conforms to proper style and that it has a clear structure. Check that your code does not depend on hard coded paths, as those will not be present on the computer used for grading — make everything work with relative paths (and structure the tarball that you hand in such that the program will run without having to move files around).

Chapter 18

Stellar density in a globular cluster

For this assignment we will look at the globular cluster 47 Tucanae (or NGC 104). The data is available from the NASA Hubble space telescope archive.
Goals for this week:

- demonstrate an ability to design and implement simple programs,

- access data in FITS files, and run some data analysis on them (fitting and creating periodograms in this case).

This exercise will be graded as follows, you can earn at most 9 points by completing the exercise and handing in on time. Your grade g, awarded on a scale from 1 to 10, is calculated as follows:$g = 1 + s$, where s are the points you earned. For this exercise the points are awarded for the following things:

(3 points) For proper programming style and a programs that are clearly structured.

(2 points) For the assignment in Section 18.1.

(2 points) For the assignment in Section 18.2.

(2 points) For the assignment in Section 18.3.

18.1   Part 1: Showing FITS image data with sky-coordinates

In this part of the assignment you will download a FITS image and create a figure showing it, plotted along the correct axes (i.e. using sky coordinates).

- Go to the Mikulski Archive for Space Telescopes (MAST) Hubble pages and search for NGC 104. Have a look at the results page and see what is available.

- **Assignment:** For the purposes of this exercise you should download the following FITS file: `https://archive.stsci.edu/cgi-bin/mastpreview?mission=hst&dataid=IC2R02050` .

- **Assignment:** Use the FITS functionality from Astropy to open the FITS file. Access the image data in the FITS file and create a plot of it using Matplotlib as we have done before. You will have to adjust the range of values that are shown by Matplotlib. Look at the documentation of `pyplot.imshow`, there are relevant keyword arguments to adjust this range.

- The image you just created has no sky coordinates along the axes, instead it has pixel coordinates. Astropy contains a module `astropy.wcs` which can deal with the *World Coordinate System*, which allows you to create plots with proper sky-coordinates.

- **Assignment:** Read through the documentation for the `astropy.wcs` module, available on `http://docs.astropy.org/en/stable/wcs/index.html`. It shows you how you can add the sky coordinates to a plot of the image data. You should look through the FITS header and confirm that it contains a WCS section. It is that section that describes how the image pixels are projected on the sky.

- **Assignment:** Create a plot of the globular cluster NGC 104 that shows the stars in the cluster and has axes labeled with sky coordinates. Make sure your plot also shows the masked regions (so do not remove the `NaN`-values yet).

18.2   Part 2: Detecting stars

In this part of the exercise you will use the `photutils` package to detect the stars in the image. Unless noted otherwise you should perform the following analysis using pixel coordinates.

- **Assignment:** Install the `photutils` package.

- **Assignment:** The `photutils` package expects data with the background subtracted and without `NaN` values. Use Numpy to mask `NaN`-values and determine the median of the image. Subtract the median from the image and after that replace the `NaN`-values with zero.

- **Assignment:** Use `DAOStarFinder` from `photutils` package to detect the star positions in the image.

- **Assignment:** Plot the detected stars over the image of the NGC 104 globular cluster.

18.3   Part 3: Finding the cluster center

In this section of the assignment you will perform some simple analysis (histograms and function fits) to find the approximate cluster center. In this part of the exercise we will again perform our analysis in pixel coordinates.

- **Assignment:** Use the `numpy.histogram` function to create a two histograms. One for the x-positions of the detected stars, and one for the y-positions. (We ignore the distortion in the image, and assume the cluster to be circular.) Use on the order of 40 bins for each of these histograms. Create an array with bin centers from the bin edges returned by `numpy.histogram`.

- **Assignment:** Read the documentation to the `scipy.optimize.curve_fit` function that allows you to fit functions to data.

- **Assignment:** Create a function for a one-dimensional Gaussian add a term to be able to fit a constant background $Ae^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} + b$. *As a hint, you can use the following function definition as a start:*

```
def gaussian(x, A, mu, sigma, b):
    pass
```

- **Assignment:** For both the X and Y star position histograms perform a fit. Fit the Gaussian model you created to the bin values and bin centers. Provide an initial set of parameters for the `curve_fit` fucntion using the `p0` keyword argument. You can estimate values for `A`, `mu`, `sigma` and `b` values from a plot of the histograms. Provide plots of your histograms with the fitted functions drawn over them, like the plots in Figure 18.1.

- *Tip: The fits will be better if you skip the first and last few bins in the histograms. These bins are lower artificially because they are partially masked. Because of this they throw off the fitting algorithm.*
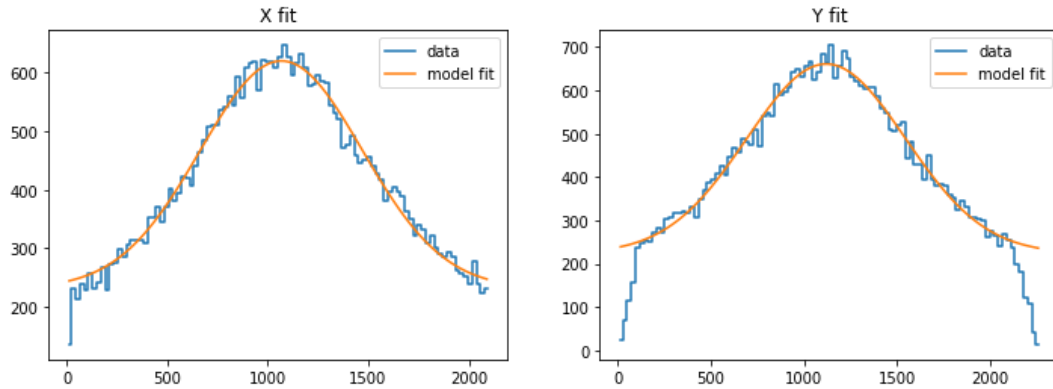
Figure 18.1: Example of the fits to X and Y star position histograms.

- **Assignment:** Finally create a histogram of the pixel radii (from the cluster center) at which the stars were detected. Explain the break in this histogram that is present at a radius of about 1000 pixels (put the answer to this question in your report).

18.4 Handing in

Check that the following things are included in your submission:

- a report describing your data sources and methods,

- plots you were asked to create,

- all code you wrote to perform your data analysis (able to be executed),

- any ancillary files needed to run your programs.

Furthermore, check that your code conforms to proper style and that it has a clear structure. Check that your code does not depend on hard coded paths, as those will not be present on the computer used for grading — make everything work with relative paths (and structure the tarball that you hand in such that the program will run without having to move files around).