Introduction to programming for astronomers

Thijs Coenen

September 9, 2014

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. Third the availability of massive amounts of computational power allows increasingly detailed astrophysical simulations to be constructed.

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 provide the basics of programming in Python. Python was chosen for its straight forward syntax, its free availability, the wide range of libraries in its "eco system" and its increasing prominence in physics and astronomy.

1.2 Conventions

This reader uses a number of typographival conventions differentiate normal text, progamming code snippets, commands you enter and progam 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. 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$
```

As you can see in this example the command prompt is bold, while the commands and program output uses normal monospaced type. 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:

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


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

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

When programs require key presses that are not really commands, they are type set as for instance [q] or [control] + [z] if two keys must be pressed at the same time.

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.
Today 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 nowadays the 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 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.

[1] The example shows some directories that are Mac OS X specific: `Applications`, `Developer`, `Library`, `Network`, `System` and `Volumes`.

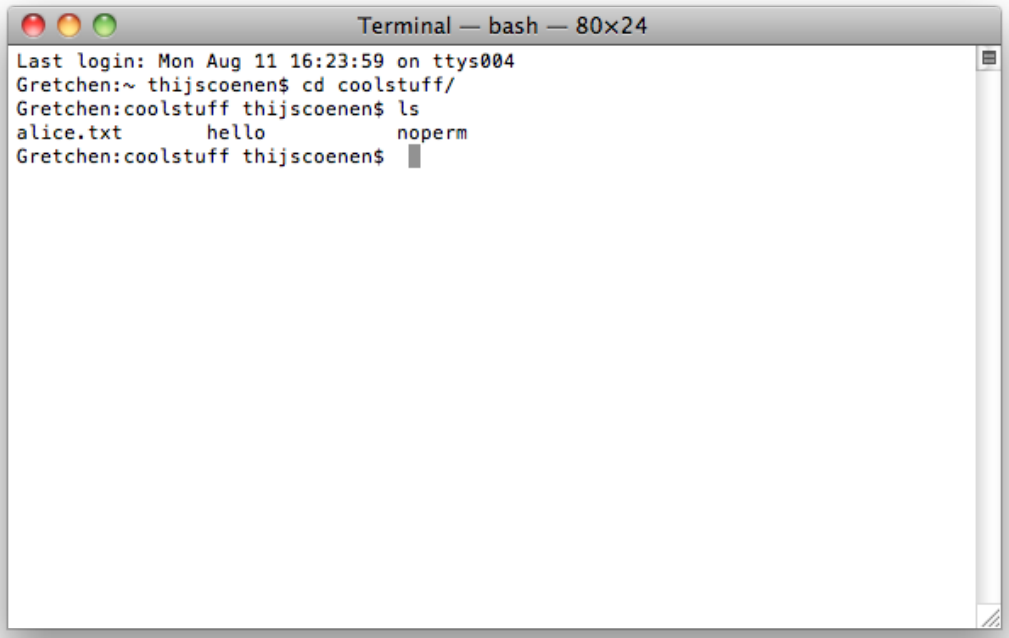

Figure 2.1: A screenshot showing Mac OS X's built-in Terminal program. This program allows you to access the underlying BASH shell on Mac OS X.

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:

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

[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.

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):

```
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
```

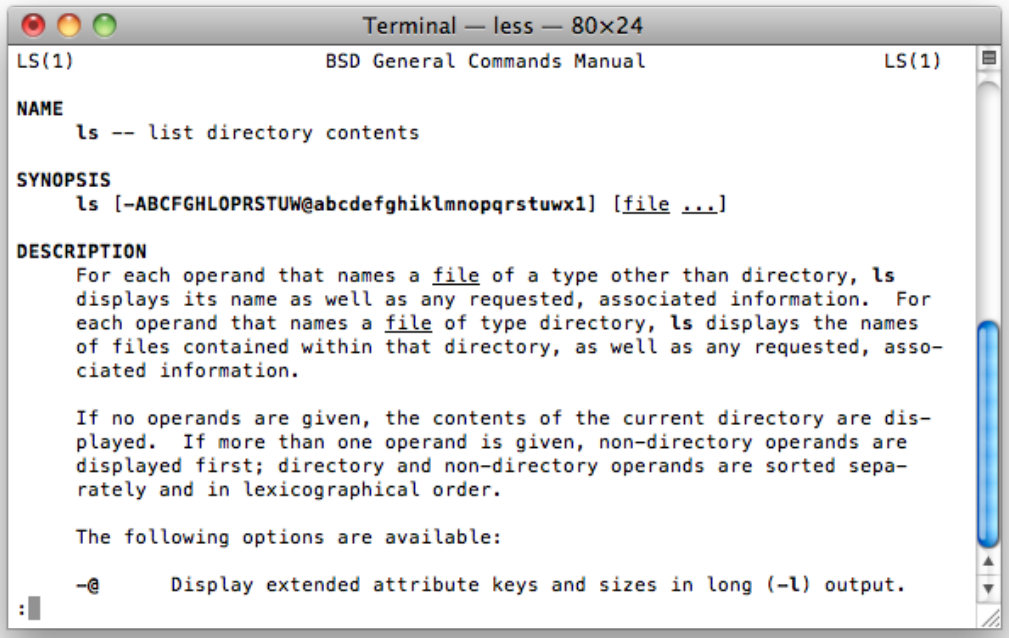

Figure 2.2: The `man` documentation viewer showing part of the documentation for `ls`.

```
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

[3] According to one of my proofreading colleagues.

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 `p`. 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
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:~$
```

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

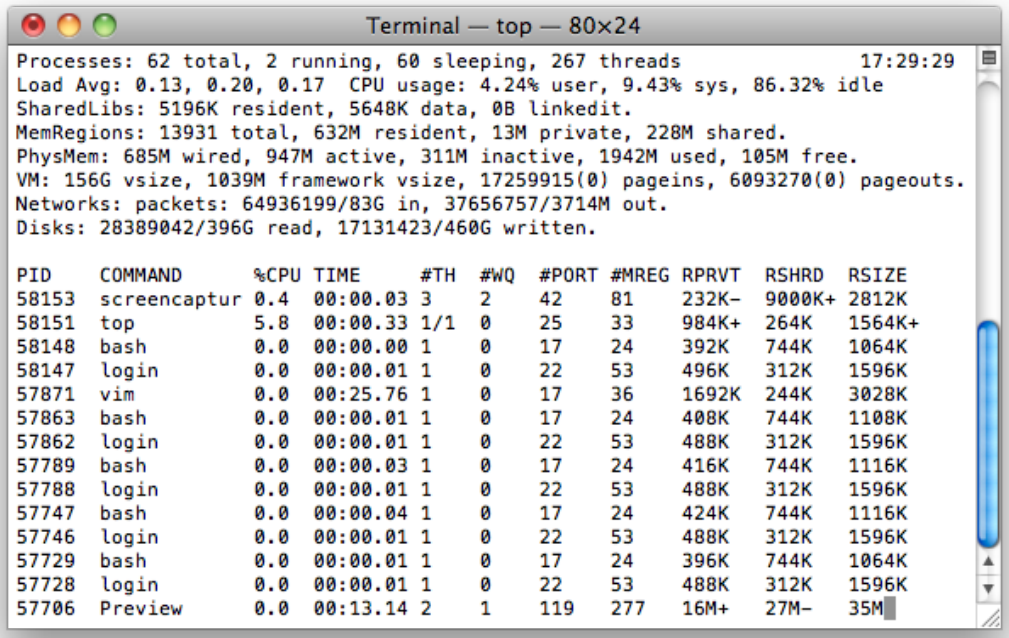

Figure 2.3: This screenshot shows `top` running on a Mac OS X system.

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`.

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
```

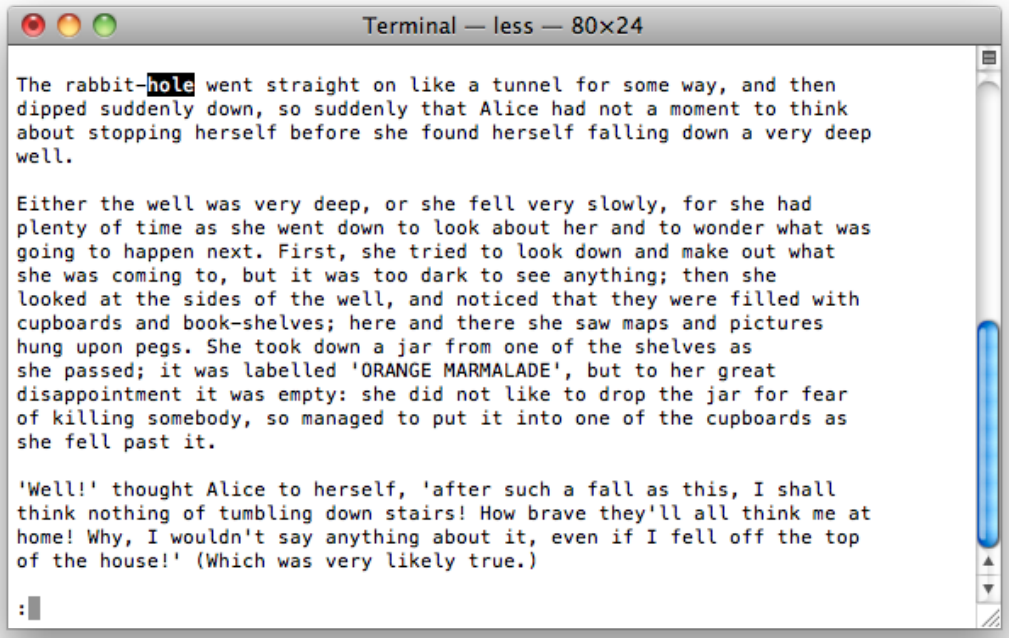

Figure 2.4: The `less` program showing a text file, you can move through the file using the arrow keys.

```
#!/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 [enter]. If there are several matches you can move between them using [n] (or [N] for the reverse direction). Help is accessible by pressing [h] and you can quit `less` by pressing [q]. 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 A). 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 `-l` option to print the filename in stead of the matching text:

```
Gretchen:coolstuff thijscoenen$ grep -li "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 -li "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

3.1 Introduction

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. We will be using Python 2 because it is still the most used version of Python in scientific programming. While eventually most new development will move to Python 3 not all useful libraries are available yet and for the moment Python 2 is a good, pragmatic choice. With Python 2 being supported well into the future (at least through 2020) it is no problem to learn Python 2 now and eventually transition to Python 3.

3.1.1 Source code

The source code of a program is created by a programmer and can be compared to the blue prints of a house. A computer does not directly execute the source code of a program, in stead those instructions are translated to a code that the can be executed by the computer hardware. Python is a high level language which means that it insulates the programmer from the details of the underlying hardware. Python trades speed of execution for convenience and speed of development.

3.1.2 The virtual machine

Unlike some other programming languages programs written in Python are not translated to a machine code version that is directly executed on the computer hardware. In stead Python code is interpreted by another program This program is called a Python interpreter or a virtual machine because it in some ways acts as a computer. In the past only one Python interpreter was available, this interpreter is nowadays referred to as CPython because it was written in the C programming language. Some modern Python interpreters will under certain circumstances translate (or compile) Python code to machine code. The Pypy project, for instance, produces a Python interpreter that will compile pieces of Python code to machine code to speed up their execution (this is still a hard problem, and very much an area of active research).

3.2 Running Python

In this section I will explain how to run the "Hello world!" example, an example without which no programming tutorial is complete. Python code can be executed in two ways, as part of a program or small snippets in the interactive mode of the Python interpreter. Interactive Python interpreters allow testing, experimentation or even interactive analysis (in the style of Matlab or IDL). Many of the basic examples of Python code in this reader will be shown in interactive sessions, while most serious programming is writing actual programs.

3.2.1 Whole programs

To be re-structured as per plan. 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 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 to your shell the text `Hello world!`. 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 (if they were programmed to do so). 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 Section 3.4.

3.2.2 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 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
```

3.2.3 Interactive 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.

3.3 Variables and data types

A variables are objects in your program that have a name and a value that can change. You refer to a variable through its name. To define a variable you "assign" a value to a name in your program, this uses the assignment operator = (the *single* equals sign). In the following example the variables `a`, `b`, and `c` are all assigned values.

```
>>> a = 10
>>> b = 'abacadabra'
>>> c = None
```

To access a variable you use its name, in general a variable's name is visible only in certain parts of a program. The rules that govern this visibility are the rules of *scope* (explained in some detail in Section **??**) If a variable is not accessible, either because it was not defined, or you are trying to access it from the wrong scope, you will get a `NameError` or `AttributeError`.

```
>>> a
10
>>> d
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'd' is not defined
>>>
```

Associated with any value in a program is a *type* or *data type*. This is how Python keeps track of how a value should be treated. Unlike some languages it is possible to change the type of Python variable at any point. Changing the type of a variable in Python really means that the variable name is pointed to another value somewhere. The built-in `type` function can always be used to find out what type is associated with a variable (or value), this can come in handy when debugging:

```
>>> type(a)
<type 'int'>
>>> type(b)
<type 'str'>
>>> type(c)
<type 'NoneType'>
>>>
```

3.4 Basic data types

Python provides many data types, the most basic of which will be treated in this section. To represent numbers it has integers, longs, floating point and complex numbers. Pieces of text are represented as strings (or "Unicode strings"). Strings are an example of a *sequence type* because they can comprise of any number of values (characters in the case of a string). Tuples and lists are two other very basic sequences, both can contain any value, but the former cannot be changed after its definition whilst a list can be. Python also has dictionaries, which are an example of a mapping type, equivalent to hash tables in other

languages. Truth values are represented as so-called booleans (after an English mathematician George Boole who invented Boolean logic). There is also a very special value in Python, `None`, that is used to signify the absence of a value, it is of type `NoneType`.

3.4.1 Representing numbers

Python has several data types to represent numbers. Integers, known as type `int`, are used to represent whole numbers (both positive and negative). Floating point numbers, of type `float`, are used to represent numbers with a decimal point. The way that integers are implemented means that when they overflow, i.e. when their value becomes too large for the amount of memory associated with them, Python will switch to representing them using the `long` type.

```
>>> a = 10
>>> b = 5.5
>>> type(a), type(b)
(<type 'int'>, <type 'float'>)
>>>
```

3.4.2 Doing calculations

Python implements operators for simple arithmetic: addition `+`, subtraction `-`, multiplication `*`, division `/` are defined for both integers and floats (but see below for a common mistake involving division). The `%` operator calculates the remainder of a division. You can raise numbers to some power using the `**` operator or the `pow` function.

```
>>> 1 + 1
2
>>> (3 * 6 + 6) / 8
3
>>> 4 ** 0.5, pow(4, 0.5)
(2.0, 2.0)
>>>
```

Division by zero will raise a `ZeroDivisionError` which will terminate your program if unhandled (see Section **??** for error handling).

```
>>> 1 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
>>> 1.5 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: float division by zero
>>>
```

Common sources of errors involving simple arithmetics are misunderstanding the precedence rules of the operators and the truncation of integer division. Problems with operator precedence can easily be fixed by adding parentheses (so that you control the order in which operations happen). When two integers are used in a division the result will be an integer and the fractional part will be truncated. If you are interested in the fractional part you should convert either or both integers to floating point number(s) before performing the division.

```
>>> a = 1
>>> b = 2
```

```
>>> a / b
0
>>> a / float(b)
0.5
>>>
```

Note: floating point numbers are not real numbers, they are almost always approximations of the real numbers you want. E.g. the decimal number `0.1` is impossible to represent precisely using floating point numbers (representing bank balances as floating point numbers is therefore a bad idea).

3.4.3 Representing text

Text of any length is represented as a string (type `str`). Each individual character in a string has an index, starting at `0`, which can be used to access it. E.g. for a string `s` you can access the first character as `s[0]`. If you use an index that is too high Python will raise a `IndexError`. A special property of Python is that it allows negative indices to be used, where `-1` is the last character of a string. Furthermore the slice operator can also be used to extract a sub string from some string. The general notation that allows you to take sub strings is `somestring[startindex:endindex]`. 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 so called string methods (why they are called methods will become clear when *classes* are treated). Below some examples of useful string methods and how you use them, but more are available see the Python documentation[1]: Python strings cannot be changed after they are created (a string is called `immutable` for that reason) so these methods do not modify the original string, they create a new one.

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

[1] `https://docs.python.org/2/library/stdtypes.html#string-methods`

A note about terminology, the term "string literal" is used for the representation of a string in Python source code. Strings are not the only data types for which a "literal" notation exists (the numbers in the previous examples that were written out are also integer and floating point literals). 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 also a common example `\n`. A table of string escape sequences can be found in the section about string literals in the Python documentation[2]
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 strings contain characters of 8 bits, that allow only 256 characters to be encoded. That is clearly not enough to handle all the world's characters. Unicode was designed to represent all the world's known characters and thus provide a standard that can be used throughout the world. Python has a special Unicode string type (`unicode`). Unicode literals are delimited with `u'abc'` and special characters can be entered using Unicode escapes that start with `\u`, e.g. the escape sequence for the Euro-sign is `\u20ac`.

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

Because a sequence of characters are contained in the strings or unicode strings, they are sequence types.

3.4.4 Tuples

A tuple is a sequence type as it allows you to put several values in one object and unlike strings these need not just be characters. Tuples are not limited to storing only one type, types can be mixed. Tuples are immutable and if you try to modify them an `TypeError` will be raised. In source code tuples 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).

```
>>> t1 = (1, 2, 3, 'abc')
>>> type(t1)
<type 'tuple'>
>>> t2 = 1,
>>> t2
(1,)
>>> type(t2)
<type 'tuple'>
>>> t3 = (1)
```

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

```
>>> 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
>>>
```

3.4.5 Lists

Like tuples and strings, lists are sequence types that can contain any number of values of any type. Unlike tuples, lists can be changed after they are created (both their content and their length can be changed), they are *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]
```

3.4.6 Booleans

Python has a special data type to represent truth values so called booleans (type `bool`). 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 3.5.1):

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

3.4.7 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 `[]`.

3.4.8 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. Dictionaries are an example of a mapping type. Dictionaries correspond to hash tables in other programming languages. 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. To access a value in a dictionary use the associated key and as you would an index for a list. If a key you try to access is not present in the dictionary a `KeyError` will be raised:

```
>>> 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}
>>>
```

3.4.9 Type conversion

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` is raised (for example `'abc'` 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 3.6.3.

3.4.10 The `in` operator

No description of strings, tuples, lists or dictionaries would be complete without mentioning the `in` operator. This operator is used to test whether a certain value is present in a string, tuple or list. In the case of dictionaries `in` checks only the keys (not their associated values).

```
>>> s = 'abcdefg'
>>> 'z' in s
False
>>> t = (10, 4, 5, 23)
>>> 23 in t
True
>>> a = range(10)
>>> a
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> 5 in a
True
>>> 11 in a
False
>>> d = {1: 'a', 10: 'b', 11: 'd'}
>>> 'a' in d
False
>>> 11 in d
True
>>>
```

Different data type (or data structures) can have different access speeds. Generally searching an key in a dictionary (or a value in the related set data type) is fast, while looking for an element in a string, tuple or list is slow.

3.4.11 The `len()` function.

The built-in `len()` function can be used with strings, tuples, lists and dictionaries to find the number of elements they contain. Given a variable `x` of any of these data types `len(x)` will return the number of elements in `x`.

3.5 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. Before we can explain how code is conditionally executed we must explain how conditions can be expressed.

3.5.1 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
```

3.5.2 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`.

3.5.3 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
```

3.5.4 `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`.

3.6 Loops and iteration

When a block of code should be executed several times it can be executed in a so-called loop. Python has two types of loops: `while` loops and `for` loops. Iteration is the process of accessing elements one-by-one that is closely related to the `for` loop.

3.6.1 The `while` loop

The `while` loop is executed as long as the condition at the beginning of the `while` loop is `True`. To exit from a loop early you can use the `break` statement. When you want to return to the top of the block of code in a loop you can use the `continue` statement. A feature of loops in Python 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) !!!
```

3.6.2 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[3] and runs a block of code. When all elements have been accessed, or a `break` statement was encountered the iterations stops. 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 only when iteration stops without having hit a `break` statement.

```
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.

```
s = 'abcdefg'

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

[3] 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.

3.6.3 Examples of iteration

In this section the iteration of basic Python types is shown. We start with an example of a list of strings:

```
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]
```

Iteration over a string:

```
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]
```

Iteration over a list of tuples:

```
# 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

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
```

Iteration over a dictionary:

```
# 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]
```

3.7 Organizing a program

3.7.1 Functions

Functions are a way to name a block of executable code, in general functions take arguments as inputs and have return values as output. Functions are defined using the `def` keyword followed by the name of the function, a list of arguments in parentheses and a colon. The block of code that belongs to a function is indented, much like the block of code belonging to an `if` statement or loop is indented.

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

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

say_hello()  # calling function say_hello without arguments
is_even(10)  # calling function is_even with one argument, the integer 10
```

The `say_hello` function in the example takes no arguments (there is no list of arguments between parentheses) and it has no return values (there is no `return` statement). All the first function does when you use it, or in the proper jargon *call it*, is that it prints `Hello!` to the standard out. The second function takes only one argument `n` a number and determines whether that number is even or not, it returns `True` or `False`. In the next example we combine a loop, a function and an if statement to only print even numbers from a list (using the previous definition of `is_even`).

```
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.
```

This example also demonstrates the principle of *scope*. The variables `l` and `out` are only "known to Python" inside of the `filter_even_numbers` function. That is their scope is limited to that function and if you try to access it outside of that function Python will raise a `NameError`.

3.7.2 Classes

Classes are a way of grouping data and functions that work on that data together, a class describes the properties of all objects that belong to that class. Objects of a certain class are said to be instances of that class. 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).

```
# The class definition:
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
        )

# 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*). To continue the previous example we introduce a `Teacher` class:

```
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.
```

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.

3.7.3 Modules and packages

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:

```
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. Packages can be compared to libraries in other languages and they contain create importable hierarchies.

3.8 Scope

Above the `scope` of a function was mentioned. The rules of scope determine which variables, or in general things with a name, are accessible and also where they are accessible. In Python functions, classes and modules have their own scopes, and these can be nested. When a name is accessed it is first searched in the local scope, then in one (or more) enclosing scopes until the scope of the whole module is reached. This module scope is called the global scope. When the name cannot be found in the global scope either, the Python built-ins will be searched. A failure to find the name in the built-ins will finally lead to an `NameError`.
Variables in the global scope are accessible everywhere, so changing them anywhere potentially affects your whole program. 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.

```
a = 10  # in global scope
```

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

Table 3.1: Reserverd keywords in Python 2.7, these keywords cannot be used as names for objects or variables. This list can be accessed from Python itself by importing the `keyword` module and printing `keyword.kwlist`.

```
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
```

When a name that exists in the global scope is assigned to, that object in the global scope is not changed. In stead, the name is defined in the local scope making the object making the global object inaccessible from the local scope. To circumvent this problem Python has the `global` keyword as was demonstrated in the example above. 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 `global` will result in a `UnboundLocalError`.

3.9 Naming

3.9.1 Reserved keywords and built-ins

Every programming language contains certain words that cannot be used as names (or identifiers) of object because they have a special meaning in that language. In Python trying to assign to a reserved keywords will result in a `SyntaxError`. E.g. when trying to assign to the keyword `for`:

```
>>> for = 10
  File "<stdin>", line 1
    for = 10
        ^
SyntaxError: invalid syntax
>>>
```

The full list of reserved keywords can be found in Table 3.1 Beyond the keywords Python also has a number of other built-in object, which are mostly functions but also include a number of errors and exceptions. You should also avoid using these names (and some, like `None` will also result in `SyntaxErrors` if you assign to them). Redefining the built-ins that can be assigned to will make your code hard to read and likely cause defects. The full list of these built-ins can be found by running 'dir(__builtins__)' in for example an interactive Python session. Note that this list does not include the names of the standard library modules that are available.

3.10 Error handling

Python has *exceptions* that are *raised* whenever an error occurs, when an exception is not handled it will case 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. The `try` clause contains a statement that can potentially fail, while the `except` clause contains the code that deals with such a failure. When no exception occurs the `else` clause is executed.

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

In this example you see that the `except` clause is only looking for `ZeroDivisionError`, because they are the only errors expected from the code in the `try` clause. It is possible to look for generic exceptions by using `Exception` with the `except` statement. Doing this however will make your code hard to debug because you may be handling different errors than you expect. It may be better to crash when unexpected problems occur in stead of having your program behave in unpredictable ways.

3.11 File input

Python has a built-in function to open files, imaginatively called `open()`. When Python successfully opens a file the `open()` function will return an open `File` object which can then be written to or read from depending on the mode that the file was opened in (see the example python session below).

```
>>> f = open('alice.txt', 'r')
>>> f
<open file 'alice.txt', mode 'r' at 0x100468540>
>>>
```

Files opened for reading that contain text can be accessed line-by-line by iterating over the file object, a silly example that counts lines in the file `alice.txt`:

```
f = open('alice.txt', 'r')  # open 'alice.txt' for reading
count = 0
for line in f:  # iterate over the open file object
    count = count + 1
file.close()  # always close the file object after use
print count
```

Appendix A

Text editors

- **Kate** and **Gedit** are the editors included with, respectively, the KDE and Gnome desktop environments for Linux. Both are graphical editors that support syntax highlighting in many programming languages. Kate, like all things KDE, is very configurable. (**Recommended.**)
- **Xcode** is Apple's integrated development environment for Mac OS X. While it supports Python, it includes many features that are not needed for Python. (**Avoid, unless you use it for other languages already.**)
- **Textmate** is a capable text editor for Mac OS X, previously one of the favorites for that system. Version 1.x is available commercially while version 2 is available as open source software. (**Recommended.**)
- **Sublime** is a modern grahical text editor for Mac OS X that supports many programming languages. (**recommended**).
- **IDLE** is the editor and interactive Python shell combination included with Python by default. IDLE is cross platform, not integrating well with some of its hosts. The interactive shell can be very slow when your program prints a large amount of text. IDLE may also complicate debugging because at times you will come across tracebacks that point to IDLE's code. Since IDLE only supports Python syntax highlighting you will end up learning another editor in the long run. (**Avoid, unless required by some course.**)
- **Emacs** is a very capable and customizable editor. It was originally a text based editor although there are also graphical versions like XEmacs. The learning curve for Emacs is steeper than graphical editors like Kate or Gedit. Emacs will also work in a terminal connected to a remote computer system, this allows you to do programming work on a remote system that does not allow graphical connections. (**Recommended.**)
- **vi** or **vim** are very capable and customizable editors that have a *very steep learning curve*. Vi(m) uses very concise commands that can be combined to perform complex tasks. The vim, "vi improved", editor has more features than vi. Like Emacs, vi(m) will work in a terminal and allow you to do programming on systems without graphical displays. (**Recommended, if you have the time to learn it.**)
- **pico** and **nano** are simple text based editors one of which is usually available on a remote Linux systems. Use these editors only for quick small edits on remote systems, and only if you cannot use a more powerful editor like emacs and vi(m). (**Avoid for anything other than emergencies.**)