Doing Computational Social Science with Python: An Introduction

Damian Trilling

Version 1.2

PDF created January 14, 2019

This book can be obtained from http://papers.ssrn.com/abstract=2737682. The source code of the most recent version can be found at https://github.com/damian0604/bdaca.

Contents

Introduction: What and why?

Social scientists are more and more confronted with the analysis of large-scale datasets. Often, these are data from online sources, and often, they contain some form of textual data. Think of data from social media, but also large archives. Often, this development if referred to as a move towards "computational social science" (Kitchin, 2014; Lazer et al., 2009).

There is a certain overlap with the term "Big Data". But although the latter sounds sexy, it is a somewhat problematic term, because people use it for all kind of data. As scientists, we want a clear definition, but it is hard to tell what the term Big Data actually entails. This book is not about *really* Big Data requiring a whole server farm, but it is about data that is too big to handle manually—think of one million tweets for example. It is about data sets that are small enough to be handled by an ordinary laptop, but often too big to be processed by ordinary programs. Excel does not have an unlimited number of rows, and SPSS and STATA start complaining (or simply stop working) once you have too many cases and variables (see, e.g., Trilling, 2017). If you know R, you are better off, but for some of the tasks we will discuss in this book, Python has a bit more to offer.[1]

This book introduces you to automated content analysis of data that typically comes in amounts that are too voluminous for manual coding and for traditional point-and-click applications: tweets, blogposts, articles from RSS-feeds, etc. We will use the programming language Python, which is very flexible and highly suitable for this end. It also scales very nicely—meaning that you can use it now for some smaller projects, but it can also be used on immense data sets. So, if you will find yourself working on some really Big Data that cannot be handled on a single computer any more, the principles are not too different from what we do in this document (yes, I'm simplifying a bit).

You will be guided through the first steps towards automated content analysis with Python. Note that I wrote this book for an audience of stu-

[1] I do not want to go into the R-vs-Python-for-data-analysis debate here, I use both tools, but for different tasks. Google for it if you want some comparison.

dents in the social sciences, like communication science, political science, or sociology. From a computer science point of view, much more would have to be said, and some things we do in this course might actually be considered bad programming style. But that's not our objective here: This course should enable students of the social science to make some first steps with Python, in order to solve their analytical questions.

This also means that this manual is far from complete. It rather serves as a starting point—and from that point onwards, it's up to you. Once you got the basic ideas and concepts, there are enough resources out there to help you further.

Have fun! And, join me in thanking those who contributed with great ideas or served as guinea pig for earlier versions of this document – which basically applies to all students and colleagues working with earlier versions of this document.

Amsterdam, March 2015 (version 0.1)
Amsterdam, February 2016 (version 0.2)
Amsterdam, January 2017 (version 1.0)
Amsterdam, January 2018 (version 1.1)
Amsterdam, March 2018 (version 1.2)

Damian

Damian Trilling
d.c.trilling@uva.nl
www.damiantrilling.net
@damian0604

How to read this book

Before we start, let me introduce three conventions that I'll use in this book. Python-code, which you will type into the Python interpreter or use in the programs you'll write, is represented in blue:

```
1 print("Hello world!")
```

Commands that you have to type into the Linux command line (also referred to as shell, terminal, or bash) are pink:

```
1 echo Hello world!
```

And any output or data file is represented in gray:

```
1 Hello world!
```

Pay also attention to the line numbering at the left side of the code examples: It helps you see whether a line really ends (and that you should press enter), or whether the paper is just too narrow to print the line on one line:

```
1 print("Hello world!")
2 print("I am in a talkative mood today, I really have so much to say that
      it doesn't fit on one line on paper. But PLEASE, write this as a
      single line if you type me over!")
3 print("Goodbye!")
```

We see immediately that the example above consists of three lines, not five.

Finally, a word about the structure of this book. Part I teaches you the basics and should be read in the structure as presented in the book, as the chapters build on each other.

Part II presents specific techniques and can be read in any order you like. While there are some cross-references between the chapters, they do not necessarily build on each other. However, it assumes that you have all knowledge from Part I, and, most importantly, that you installed everything as explained in Chapter 1.

Nevertheless, there are two ways to work with this book: the complete route and the short route.

The complete route

When you use this book as part of a multi-week course, this is the route you take: You start at the beginning of and end, well, at the end (maybe skipping a bit here and there, maybe changing the order a bit, or maybe using some extra materials in addition.)

The short route

When you use this book as a part of a short one or two day course, then you probably do not have the time to work through all materials. If the course focuses on just a few chapters from Part II, the following preparations might be sufficient (if you do not want to fully read through Part I):

- Make sure you have a basic understanding of the Python language, for example by following an online course like this one: `https://www.codecademy.com/learn/python`
- Install Anaconda (the version for Python 3, not 2) from `https://www.continuum.io/downloads`. Anaconda is a Python distribution that already includes most packages that you might want to use (see also Appendix F). Most likely, you will work using Jupyter Notebooks (Section 3.5), so make sure you know how to start it!

Please note that when you use the short route, you will not be able to follow all instructions and examples in this book to the letter, especially because some of them imply that you are working on a Linux system (which you were, if you followed the approach of the long route, but probably aren't otherwise).

Part I

Basics

Chapter 1

Preparing your computer

This chapter describes how to prepare your computer. We work with so-called virtual machines, which means that we install a program within which you can install an own operating system. This means that

(a) you can play around without the danger of damaging your own system;

(b) you can make use of a lot of cool tools that might not be available for your own system;

(c) everyone in this course has exactly the same environment running, so that you can follow all examples to the letter.

Specifically, we are using Oracle VirtualBox, within which we will install Lubuntu (a lightweight version of Ubuntu, a user-friendly Linux distribution). Before you start, make sure that you have plenty of space available on your laptop, preferably more than 15 GB. (You'd better put your music and photos on an external drive for the duration of this course...)

Before you continue, make sure you have read "How to read this book" on page ix.

Ready to go?

1.1 Install VirtualBox

Download and install VirtualBox from `https://www.virtualbox.org/`. Depending on your computer, download the binary for Windows, MacOS, or Linux.

1.2 Download a Lubuntu-image

Download the Lubuntu image from `http://lubuntu.net`. Install version **18.10 for 64-bit (AMD64)**. The file is called (**`lubuntu-18.10-desktop-amd64.iso`**). Save it at a place of your convenience.

> ! If in the next step, VirtualBox does not even show you the 64 bit option, it might be that using Virtualization is disabled in your computers settings (UEFI, formerly called and still known as BIOS – the basic configuration menu you get when pressing a special key after switching on your computer. Look into your computers' manual or google how to enter it.) Most computers do not even have such an option or have it enabled by default, but it seems that for instance many new Lenovo models have it disabled. In this case, you need to enable it.

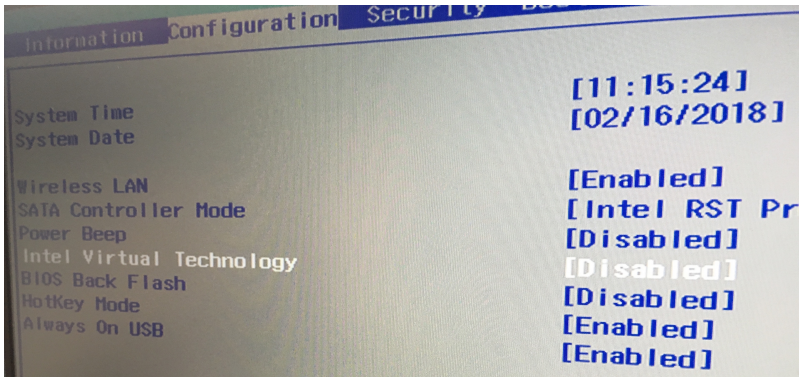


Figure 1.1: The Virtual Technology option must be switched on.

1.3 Create a virtual machine

Open VirtualBox, click on "New", give your image a name, and select first Linux, then Ubuntu 64bit from the list. On the following screens, just leave the default settings and agree to everything—with two exceptions:

- When you are asked how much "base memory" you want to allocate to your machine, you should choose a value of at least 50% of the available memory. The examples in this tutorial are tested on a VM with 2,048 MB RAM. However, especially if you want to work with larger datasets (especially, Elasticsearch and/or INCA), you might need to have 4 GB or more.
- On the next screens, you are asked about a virtual hard disk to create. I strongly recommend you to use a bit more than what is suggested. I

advise you to choose at least 12 GB, but if you have a large hard drive, just go for 15. Chances are lower that you will run out of space later during the course.

! The virtual machine you just created should show up in the main window now. If it doesn't (I had that on one computer once), restart VirtualBox.

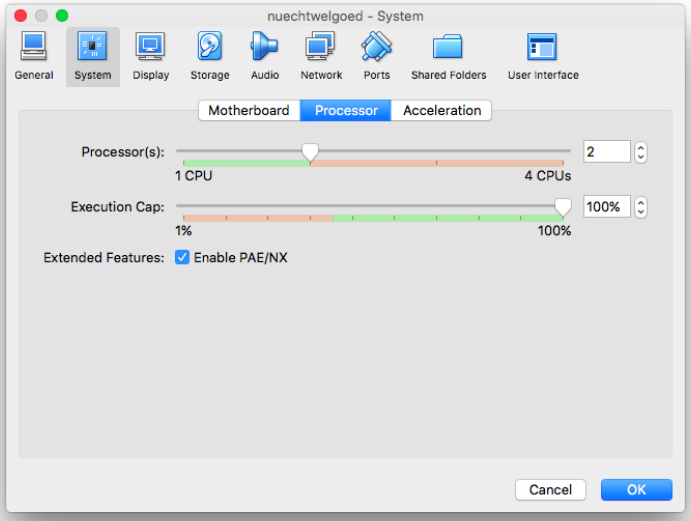


Figure 1.2: Possibly necessary tweaks: Enabling PAE/NX and allowing the use of multiple CPUs in the settings dialogue

Now click on the name of the virtual machine and on "Settings" (Figure 1.2. Under "System"/"Processors", you should see a check box "Enable PAE/NX". If you can enable this, enable it.[1] Confirm with OK. Also, check if you can enable "Display"/"Enable 3D acceleation" (Figure 1.3). These changes might not be strictly necessary, but can in some instances prevent problems.

When you are done, start the virtual machine by clicking on the green arrow in the main window. You will be asked for a location from which to boot. Click on the icon to select a file and select the Lubuntu-image you downloaded.

[1] In rare cases, you might just to have to disable it. So, if you cannot start your VM, you know what to try.

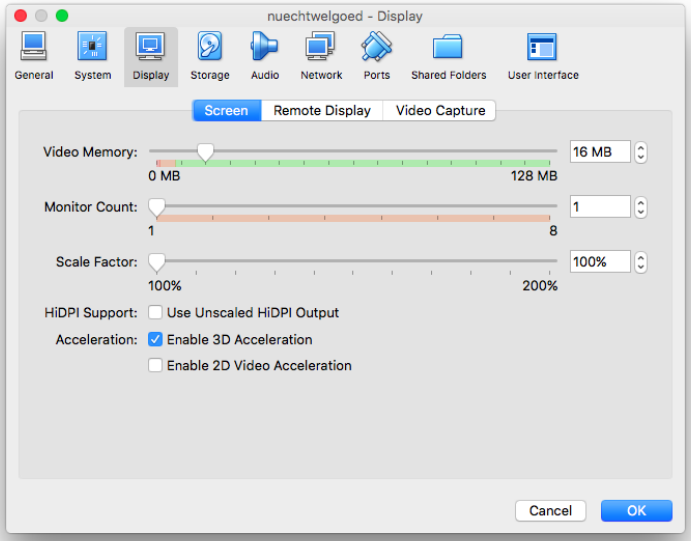


Figure 1.3: Possibly necessary tweaks: Enabling 3D and/or 2D acceleration in the settings dialogue

1.4 Install Lubuntu on your virtual machine

Once your virtual machine is running, you can select whether you want to try out Lubuntu without installing or install it. If you feel like, you can now just play around a bit in the try-out mode, but eventually, you'll need to go for installing as we need some specific tools for our course. So you can as well just select Install now, using the arrow keys.

It may be that the resolution is very low at this point. We will fix this later. If you cannot see the whole dialogue box, click the maximize icon in the upper right corner of the box.

When you select "Install", just follow the instructions. At one point in time, you will be asked whether you want to erase all data from your hard drive (yes, seriously). You can safely agree (remember, that's just the point of the VirtualBox, you have a fenced room in which you cannot damage anything. Only your *virtual* hard drive will be erased, not your real one). You will also be asked whether you want to download updates (yes, but it doesn't really matter) and third-party software (no).

After finishing the installation, your virtual machine will reboot. After clicking on reboot, you can get a message saying "please remove installation medium". You can just press enter, you don't have to remove anything in

your case. You might have to press enter a second time.

Also, it might happen that you see only some lines of text saying something like "Unmounting crypto disks". If this takes longer than a minute or so, try pressing Enter twice.

1.5 Install necessary packages

Once you have installed and rebooted your virtual machine, you will have to install a couple of packages. It might be that your system is running now in a very low resulution, but that will change after the following step. Click on the button on the bottom left (where the traditional Windows start button would be), then on System Tools/QTerminal. Enter the following command:

```
sudo apt update
```

You will be asked for your password (the one you defined during the install process). Just type it and confirm with Enter (no, you do not see anything while you type, that's correct).

You now have updated the list of available packages. To actually install the newest versions of them, enter

```
sudo apt upgrade
```

You will see a long list of packages to be installed, which you – as the last line shows – have to confirm with yes (by typing the letter `y`) followed by Enter). Potentially, you are asked some additional questions, which most likely have to be answered with yes (by typing the letter `y`) followed by Enter.

We will now install the Virtualbox Guest Utils, which solve the potential problem of your screen using a too low resolution.

```
sudo apt install virtualbox-guest-dkms virtualbox-guest-utils virtualbox-guest-x11
```

If you reboot your virtual machine (by clicking on the Start button/logout/reboot or by typing

```
sudo reboot
```

it will display in a higher resolution and you can even run it full screen if you want to (by clicking on View/Switch to Full Screen in the VirtualBox menu).

We will now install some specific Python packages. We will therefore first install pip, a package manager for Python (and also geany, a text editor we will use).

Just type the following command into the LX Terminal, which you already know:

```
sudo apt install python3-pip geany
```

Let us finally install some additional Python modules (by the way, you can use the arrow-up key to get the previous command you typed):

```
sudo -H pip3 install --upgrade pip
sudo -H pip3 install jupyter
sudo -H pip3 install spyder
sudo -H pip3 install nltk
sudo -H pip3 install scikit-learn
sudo -H pip3 install pandas
sudo -H pip3 install statsmodels
sudo -H pip3 install gensim
sudo -H pip3 install requests
sudo -H pip3 install cssselect
```

You should now be able to start Spyder, a tool we will be using for doing some Python programming, by just typing:

```
spyder3 &
```

You also should be able to run Jupyter Notebook, another tool that runs in a Firefox window:

```
jupyter-notebook
```


While working in the VM, you realize that you made a mistake while configuring your keyboard so that some keys don't work as expected? No worries, you can re-configure your keyboard at any time with the following command:
`sudo dpkg-reconfigure keyboard-configuration`
If you have a standard laptop purchased in the Netherlands, it is very likely that you have a US keyboard. In that case, the following configuration is correct:
Model: Algemeen 105 toetsen internationaal (Generic 105 key (intl) PC)
Oorsprong van het toetsenbord: Engels VS (English US)
Indeling: Engels (VS) (English US)
standaard (with Euro on 5)
geen samenstelling
X-server: nee
Of course, if you have a Mac, select the Mac keyboard instead.

From now on, you do not need the Lubuntu image that you downloaded in Section 1.2 any more, you can delete it to free up some space.

1.6 Recap

Make sure you installed *everything* correctly. If you got any error messages, try to find out what went wrong or ask your instructor. In particular, make sure you write down *what* went wrong.

There are so many different system types (and believe me, I never expected *how* many slightly different systems there are until I had dozens of students in these methods courses!), that we cannot cover every eventuality. But once you got your VM is up and running, everything should be the same for everyone in the course.

Do not forget to delete the original image you downloaded (`lubuntu-18.10-desktop-amd64.iso`)

Chapter 2

The Linux command line

2.1 Getting to know it

This chapter provides you with some basic knowledge about how to use the command line of your operating system. Chances are very high that you will need this later, so before diving into Python, I recommend doing this first.

Figure 2.1: The old MS-DOS command line prompt

Let's get started with a generational divide. It greatly helps if you are of the age of the author (I'm born in 1983) or older, as you probably remember the thing in Figure 2.1. The Linux command line (which is basically the same as what you get when you start the Terminal on MacOS, so Mac users, you can use this on your own computer as well!), however, is much, much, much, much, much more powerful than the old MS-DOS thing. If you look on YouTube for "bash" (that's the specific one we are using) or "linux command line", you'll probably get some good tutorials, but let's explore the most basic things together.

- Open a Terminal by clicking on the Start button/System Tools/QTerminal.

- Type the following commands and watch what is happening:

```
pwd
mkdir test
cd test
pwd
ls
echo Hello world > test.txt
ls
cat test.txt
rm test.txt
cd ..
```

What happend? Let's discuss it line by line:

1. `pwd` means "print working directory". It asks the computer to tell you where you are, and you probably got some answer like `/home/damian`, which is your so-called home-directory. Directory is just another word for what you might know as a folder.

2. `mkdir` means "make directory" and creates a subdirectory.

3. `cd` means "change directory" and let you go to the directory in question.

4. ...which, if you don't believe that you have really gone there, can confirm again with `pwd`.

5. `ls` stands for "list" and shows you all files in the directory. Obviously, there isn't a file yet, so...

6. ...let's simply create one. The >-sign basically means that the output of a specific command should be saved under the following filename.

7. Now it's there.

8. Let's have a look at the file,

9. remove it again,

10. and finally go the parent directory (i.e., one level up), so that we are back at where we started. In all commands, `.` always refers to the current directory, `..` to the parent directory.

2.2 Some useful commands for inspecting data

You now already learned how to create and change directories, list the files in a given directory, and how to delete them again. And, of course, in Chapter 1, you used it to install and start programs. One of the really useful things you can do in the command line is inspecting (also very large) files. Imagine you have millions of tweets, then you do not want to load all of them in a program, but maybe only see the first few to get a general idea.

Let's do a small exercise. Lets create a new directory for this exercise, download a training dataset, and have a look at it. I assume that you are in your home directory, you can check that with `pwd`.

```
mkdir exercise1
cd exercise1
wget https://s3.amazonaws.com/sift-sample-data/reviews_complete_en.csv
```

You could as well just have downloaded this file by typing exactly the same URL in your browser, but `wget` allows you to do the same trick from the command line. If you have a really huge file, you'll appreciate this! Later on, you will learn some really cool stuff you can do with wget.

The general scheme of all following commands is that you type the command followed by the file name that you want to view.

You have four tools at your disposal: `head` (which gives you the first lines of a file), `tail` (last lines), `cat` (all lines), `less` (all lines, but with scrolling). By the way, if something goes wrong, you can always cancel by pressing CTRL-C. If you want to learn more about a command like `tail`, just type `man tail`. By default, `head` and `tail` give you 10 lines, but you can specify a different number, thus, you can produce the output below with `head -3 reviews_complete_en.csv`.

```
Review ID,Review Date,Review Content,Listing Title,Neighbourhood,City,
    State,Country,Room Type,Room Price,Room Availability
4055629,2012-10-06,"Very nice accommodation in an aesthetically pleasing
    environment. The apartment is located next to Portobello road which
    makes one feel part of the action. Close to transportation, shopping,
    cafes, restaurants and bars. Thanks Joanna, it was an honor to live in
     your apartment, it made our trip even more alive.",Nottinghill
    Portobello Artist Flat,Kensington and Chelsea,London,England,United
    Kingdom,Entire home/apt,100,338
25329416,2013-11-03,"Me and my friends had such a nice time at Dotti's
    place! She was super relaxed and helpful with all the things we needed
    . The apartment itself was lovely, very homey comfortable and clean.
    The location too was great as it only took a short cycle into the
    centre.I would definitely recommend staying here! Thanks again Dotti
    ",Quiet Pink Studio in the PIJP area,De Pijp - Rivierenbuurt,Amsterdam
    ,North Holland,The Netherlands,Private room,80,10
```

This is a so-called CSV file, a table in which each column is separated by a comma. You probably recognize by now what it contains: AirBnB reviews. As you can see from the first row, the first column contains the ID, the second the date, the third the review itself and so on.

But it gets even cooler: You can also just display a specific column (e.g., only the tweets): `cut -f3 -d, reviews_complete_en.csv`. See man cut for more details. And you can combine these kind of commands through a technique called "pipe", indicated by the |-sign. The command

```
cut -f3 -d, reviews_complete_en.csv | tail -5
```

for example sends the output of the cut-command to the tail-command. If you want to send output to a file instead of the screen, this is done with the >-sign (remember?). For example, try this cool pipe:

```
cut -f3 -d, reviews_complete_en.csv > onlyreviews.csv
```

Now, play a bit around with the commands you learned!

2.3 Recap

In this chapter, you learned some basic command line commands. You should understand how the following commands work:

- pwd
- ls
- cd
- mkdir
- man
- rm
- cat
- head
- tail
- wget

You will encounter these over and over again, so you should have understood the general concept (`cut` was a pretty cool one, but probably too complicated to learn by heart—and hey, that's where `man` is for: you don't have to know all details!).

Chapter 3

A language, not a program

If you come from the SPSS or Stata world, you probably expect to *open SPSS* or *open STATA* and then work within it. There is *one* SPSS, with an icon you can click on.

In contrast, there are many ways to issue Python commands, which is actually a first important lesson: Python is not a program (like SPSS, or Word or Excel are programs), but a language — and there are a bunch of programs one can talk to in this language. Because of this, it is so flexible and can be used in so many contexts.

Therefore, there is no such thing as *opening Python*. Instead, there are several programs in which you can write code in Python and/or execute it. You probably will have your own favorite after a while, but you should be aware of the alternatives. It is partly a matter of taste which one to choose, but especially if you later on want to run a script not on your own computer but, for instance, on a university server, you want to be able to deal with different ways of running Python code.

3.1 A note on different versions

In all examples in this book, we use Python 3. Most likely, it won't matter which exact version you have. The differences between, say, 3.4 and 3.5 are so minor that you are unlikely to notice.

However, this is *not* true for versions starting with a 2. Python 2 is *not* just an old version of Python 3, and the latest version of Python 2, Python 2.7, is still widely used. This is important to know, because if you look on Google or StackOverflow for help, or if you read some of the books and tutorials I cite, then chances are high you come across code written in Python 2.

The differences are not very big. Most notably, in Python 2, you can write `print "Hello"`, while in Python 3, you need to write `print("Hello")`; and in Python 2, it is a bit more complicated to deal with Unicode (i.e., with special characters like umlauts, emoticons and the like).

So, if you see some code and think "hey, they forgot the brackets after print", then it's Python 2 code. Very often, you can just change such little things to make it work with Python 3. There is even a tool for the command line that does this automatically. It's called, not very surprisingly, `2to3`.

A last thing to know is that it is possible (and also common!) to have different versions of Python installed on one and the same computer. For example, it might very well be the case that someone works with both Python 2 and Python 3. For example, MacOS and most Linux distributions already *have* some version of Python installed by default, because they need it for internal workings. Make sure that you don't accidentally use the wrong one.

3.2 The interactive shell

This is the most simple and most straightforward way to just issue some Python commands. Go to the command line and just enter

```
1 python3
```

You just started a so-called python interpreter (or shell), and you can type Python commands, for example

```
1 print("Hello world!")
```

When you are done and want to go back to your normal Linux command line, just type

```
1 quit()
```

You also installed `ipython3`, which is some kind of luxury version of `python3`. It offers some more help and interactive features. It is, for example, better with graphics, and it is easier to copy-paste code.

For a quick look into a dataset or if you just need a calculator (yes, you can simply type `5+2` and stuff like that), starting an interactive shell can be very useful. But it comes with a big disadvantage: You cannot save your code.

In most cases, therefore, you probably want to do something else.

3.3 A text editor plus the command line

You could also write your code in an arbitrary text editor. For example, you could start

```
geany &
```

(or any other text editor you like) and write the following code:

```
#!/usr/bin/env python3
print("Hello world!")
```

Save it as `/home/damian/hello.py` and go back to your linux shell. The first line is a so-called shebang, it tells your computer how to run the program.

Go to your home directory and tell the shell that the file you just wrote is a program and that you are allowed to run it: You tell it that the (u)ser should get the right to e(x)ecute it. Of course, you don't have to do that again if you want to run the program again next time.

```
cd /home/damian
chmod u+x hello.py
```

You can now run the program by typing

```
./hello.py
```


The advantage of this approach is that on *every* system, there is some form of text editor. You do not even need a graphical interface for it. Imagine a really ressource-consuming program you wrote: the great advantage of being able to run it from the command line like this is that you do not have to open any other program next to it. In addition, if you can start your program with a single command like this, it means that you can use a huge variety of Linux tools to actually run it – for example once a day or every hour.

If you pursue this approach, maybe you want to have a look at the "traditional" Linux editors `emacs` and/or `vim`.

3.4 Integrated development environments (IDE)

For daily work, though, there are more helpful ways of developing Python code, so called integrated development environments, or IDEs. We will use Spyder, which is very similar to programs you are familiar with. It looks very much like RStudio and a bit like STATA, so you probably feel a bit at home here.

Just start it like this (or by clicking on it in the menu):

```
spyder3 &
```

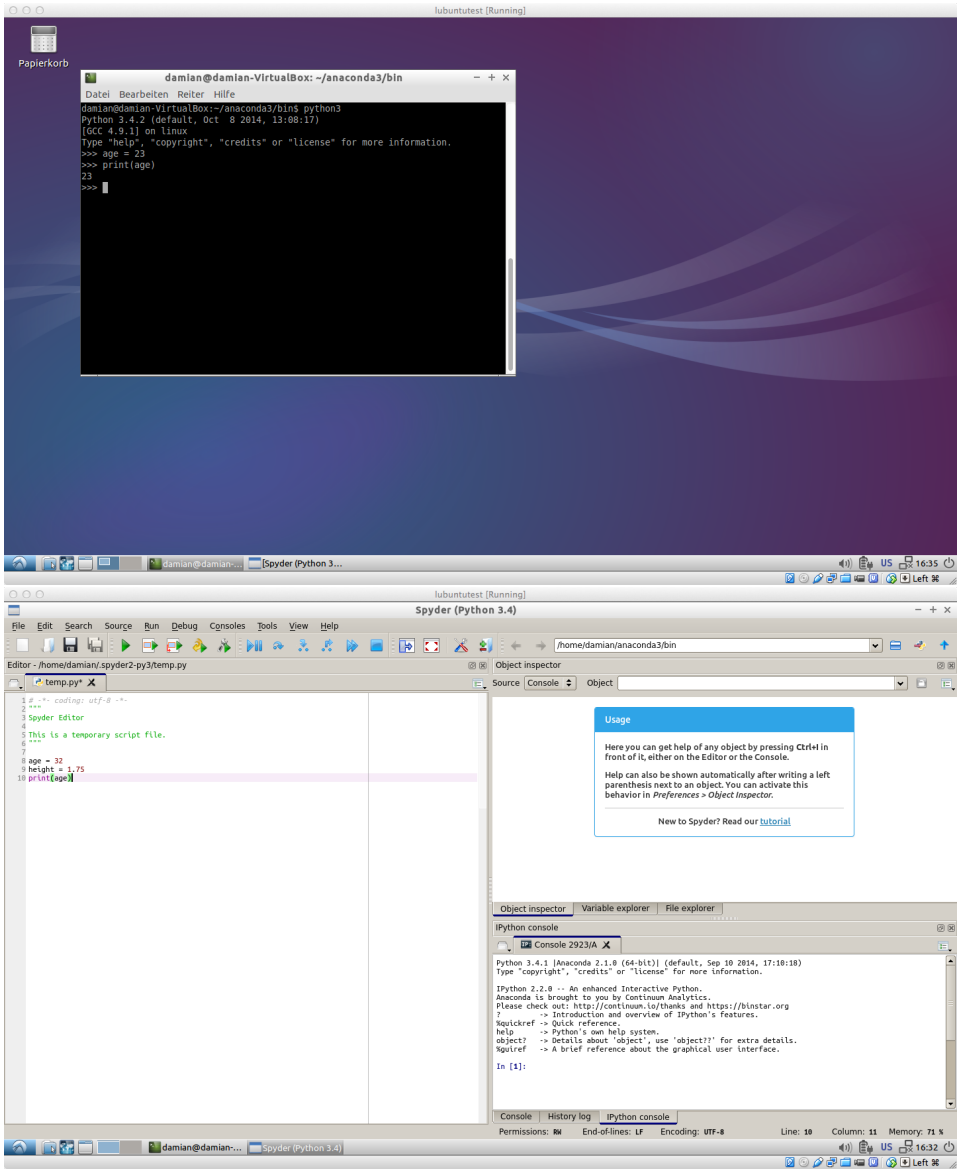


Figure 3.1: Different ways to issue Python commands: By starting the interpreter `python3` in the terminal (top) or by using an integrated development environment (IDE) like spyder (bottom).

It is a good environment to work in, and you will probably do a lot of your work in there. There are other IDEs, most notably one called PyCharm, which I use a lot. It is, however, much more geared towards professional programming and less towards data analysis, so you probably do not want to use it at the start of your Python journey.

3.5 Jupyter Notebook

Especially for data *analysis*, there is a fourth form of issuing Python commands: Jupyter Notebook, formerly known as iPython notebook. It uses the aforementioned iPython, but lets you run it in your web browser. The great advantage is that you can interactively play with the data, and save code *and* output as well as own annotations in one place. You can find some examples for such notebooks here: `https://github.com/damian0604/bdaca/`

You can start Jupyter Notebook from the command line:

```
1 jupyter-notebook
```

It should automatically open a web browser, and lets you do everything from there. For example, in Figure 3.2, you see part of a VAR time-series model conducted in Jupyter notebook.

While this is great for *interactively analyzing* data and keeping track of the output, you might already guess that using Jupyter Notebook is not such a smart way for running programs that take a while to run. For example, if you write a program that downloads a lot of data from websites, maybe running for half an hour or so (as we will do in Chapter 8), you don't want to do that in your browser. But for statistics stuff like we'll discuss in Chapter 12, it is really great. For more examples, have a look at the books by McKinney (2012) and Russel (2013).

3.6 Recap

You should have understand why there is no *one* way of "running Python" and know what the pros and cons of different ways of running Python code are.

For you, the most important ones are:

- Spyder as an allround solution for daily work
- Jupyter Notebook as a useful tool specialized in interactive exploration and analysis of data

And next to that, you should keep in mind that one can also run programs from the command line or starting an interactive Python interpreter on the command line. There will be a point in time when you want to do that.

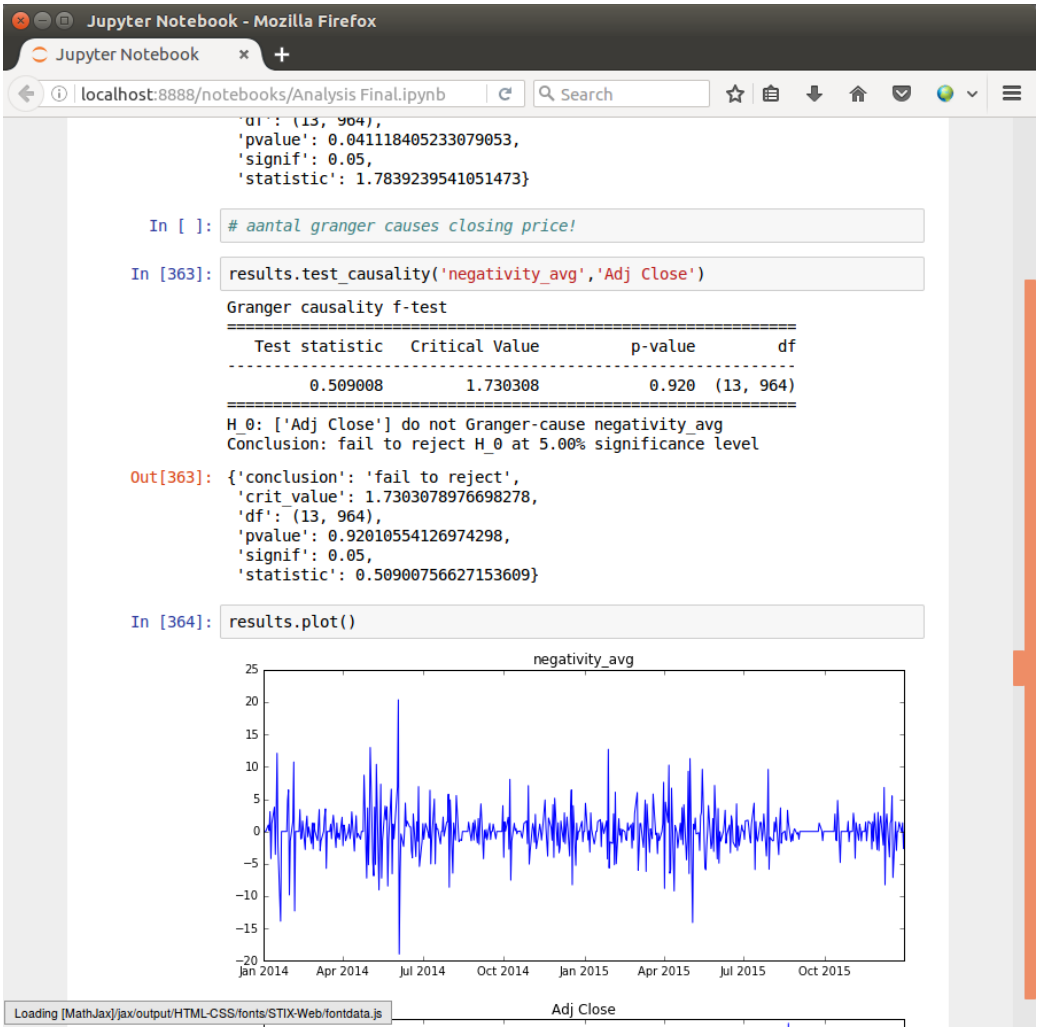


Figure 3.2: Jupyter Notebook lets you run Python code from within your web browser and lets you save it together with the output and your own comments.

Chapter 4

The very, very basics of programming in Python

Before we start writing our first own program, we need to clarify a few concepts. Luckily, they are pretty simple—but nevertheless, it is crucial to completely understand them.

> ! Of course, what I present here is far from complete. I'll mention only the most essential parts that you need to understand to get started. Everything that is not strictly necessary right now or that is pretty intuitive (I guess I don't have to say that mathematical operators like $+$, $-$, $*$, or $/$ work exactly as you'd expect?) will be left out, and a lot is pretty much simplified. That's because our main objective is to get started as soon as possible with some real life data, not to get every detail right as you would do in a computer science course. After all, we are no computer scientists.

So, start an environment of your choice (see Chapter 3) and follow me along with some examples.

4.1 Datatypes

4.1.1 Basic types

When thinking of a *variable*, most social scientists immediately think along the lines of how the term 'variable' is understood in SPSS or STATA datasets (or in methods sections in social science papers, for that matter): Something like age that has one value for each case (each person, each newspaper article, each unit of analysis). That's *not* how we define a variable here. (R users might suffer less from this misunderstanding.)

Instead, we distinguish several types of variables (datatypes), and the basic ones take——in contrast to what you might expect from the SPSS/STATA world—*one and only one* value:

```
age = 33
height = 1.75
male = True
name = "Damian"
```

It is of crucial importance to understand that each of these four variables is of a different type. The first one, age, is an *integer*, or, shorter, an *int*: A whole number without anything behind the comma. If age is an int, then someone cannot be aged 32.3.

The second one, height, is a *floating point number*, or, shorter, a *float*. In contrast to an int, a float can have something behind the comma. Unlike in SPSS, you do not have to specify a number of decimal places, *height* = 1.7527647326573265743648 would be perfectly acceptable.

The third one, a *Boolean value*, or, shorter, a *bool*, can only take two values: True or False. Note that there are no " " around True and that True starts with a capital latter. This is how Python recognizes it is not a ...

... *string*, the fourth data type. A string contains some arbitrary text, which is indicated by surrounding " " or ' '.

4.1.2 Type conversion

Python automatically converts some types for you (for example, luckily, you can divide a float by an int without thinking about their types). But still, you have to be very careful in using the right datatype. For example, you cannot write code like this:

```
numberofguests = "20"
bottlesofbeer = 100
bottlesperperson = bottlesofbeer / numberofguests
```

This does not work, because you cannot divide an int by a string. However, one can convert a string to an int (if the string contains a number):

```
bottlesperperson = bottlesofbeer / int(numberofguests)
print(bottlesperperson)
```

This code would work! By the way, if you want to have it the other way around and make a string from an int, the function is called `str()`.

> ! You might wonder why one would have to convert types at all. Couldn't one just use the right type from the start? That's because a lot of data that we will analyze does not come in the right format. For example, if we have a file with some tweets followed by the number of retweets they

received, then data that we read from the file in principle is text, thus, a string. If we want to do some calculations with the number of retweets, we have to convert it to an int.

4.1.3 Lists and dicitonaries

Imagine we want to calculate the mean age of a group. It would be very inefficient to have a variable for every single person:

```
age1 = 22
age2 = 25
age3 = 23
age4 = 28
age5 = 26
meanage = (age1 + age2 + age3 + age4 + age5) / 5
print(meanage)
```

In fact, we could almost do it more efficiently by hand. Thus, we might want to have something that is more like what is called a variable in SPSS or STATA: something named age that can contain multiple values. Such a datatype is called a *list*. You can have lists of strings, lists of floats, lists of ints, and even lists of lists (and lists of lists of lists...). A list is denoted by `[ ]`, and the entries are seperated by commas. Let's create a list of ints:

```
age = [22, 25, 23, 28, 26]
```

To calculate the mean age, we could now divide the sum of the elements by the number of elements (or, technically speaking, by the *length* of the list). We'll discuss such functions in the next section, but for those who cannot wait, this is how it works:

```
age = [22, 25, 23, 28, 26]
meanage = sum(age)/len(age)
print(meanage)
```

What do you think, of what type are `age`, `sum(age)`, `len(age)`, and `meanage`? If you want to know if you got it right, you can get the answer with `type(meanage)`, `type(sum(age))` and so on.

We can also access a specific item from a list: If we want to know the age of the third person, then we can get it with

```
age[2]
```

Yes, 2, because we start counting with zero[1]! The drawback of our current approach is that you have no idea who the person is that is 23 years old. You could have a second list in the same order:

[1] If you are in a nerdy mood, you can have a look at this classic (brief) text from 1982 explaining why: `http://www.cs.utexas.edu/users/EWD/transcriptions/EWD08xx/EWD831.html`

```
name=["John","Bas","Anne","Sheila","Mark"]
```

Note that this is a list of strings, as we can see from the "-signs. Having multiple lists in a corresponding order is actually a strategy we will work with pretty often, and it is pretty much the same as having a dataset with multiple variables in SPSS or STATA. If we are interested in the name and age of the i^{th} person, we could now find out:

```
i=3
print(name[i])
print(age[i])
```

We can also modify lists by appending new items (such as a new string) to them:

```
mijnlijst = ["element 1", "element 2"]
anotherone = "element 3" # note that this is a string, not a list!
mijnlijst.append(anotherone)
print(mijnlijst)
```

gives you:

```
["element 1", "element 2", "element 3"]
```

If we want to merge all items from a second list into the first one:

```
mijnlijst = ["element 1", "element 2"]
anotherone = ["element 3", "element 4"]
mijnlist.extend(anotherone)
print(mijnlijst)
```

gives you:

```
["element 1", "element 2", "element 3", "element 4]
```

What would have happened if we would have used `.append()` instead of `.extend()` in the last example?

Another option to solve our problem of storing our names and ages is using the last data type that we will discuss: a *dictionary*. As the name says, a dictionary is something where you can look something up——in Python terms, you search for a *key* and get its *value*. A dictionary is denoted by { }, with entries separated by commas, and keys and values by a colon:

```
nameage={"John": 22, "Bas": 25, "Anne": 23,"Sheila": 28 ,"Mark": 26}
```

Note that in this examples, the keys are strings and the values are ints, but they could also have any other data type. If we want to know how old Sheila is, we can retrieve that information:

```
print(nameage["Sheila"])
```

Adding a key to a dict (or changing the value of an existing key) is very straightforward:

```
mydict = {"whatever": 42, "something": 11}
mydict["somethingelse"] = 76
print(mydict)
```

gives you:

```
{'whatever': 42, 'somethingelse': 76, 'something': 11}
```

If a key already exists, its value is simply replaced.

The deeper you dive into data analysis with Python, the more you will appreciate the data structure of a dict: Because each value has a key (thus, some sort of a label), the "meaning" of a specific value is much clearer than with lists. In addition, nested dictionaries can be a great way to analyze and store nested data.

4.2 Functions

We have played around a bit with variables. To do some useful things with them, however, we need something called *functions*. In fact, we already used some: `print()` for example, or `len()` and `int()`. You can imagine a function as a command that in most cases takes some input (or, as we will call it, one or more *arguments*) and does something with it. The argument is supplied between `()`. For example, the function `print()` takes some variable as argument and show its content on the screen. This works for all types you know until now:

```
answer = 42
question = "What's the answer?"
print(question)
print(answer)
```

Note that you can use single quotes inside a string if you use double quotes outside the string (or vice versa!), but don't mix them up, otherwise it is unclear where the string ends. Of course, you do not have to define the variable in advance but also can do so on the fly:

```
print("Hello world")
```

Pay attention to the `" "`, which indicate that you do not supply a variable name that refers to a string, but the string itself. You could as well write:

```
hi="Hello world"
print(hi)
```

This gives exactly the same result. `hi` has no `" "` around it, which indicates that it is a variable name.

Another function that we already used is `len()`. It gives the length of an object, for example, the number of elements in a list or the number of characters in a string. You can actually also use functions within a function:

```
hi="Hello world"
print("The string",hi,"has",len(hi),"characters")
opdetap=["Estaminet","Steenbrugge Blond","Palm"]
print("They serve",len(opdetap),"different beers")
```

Make sure you understand where a `"` is placed and why!

4.2.1 Writing your own functions

There are a lot of useful functions available, either directly or from one of the numerous additional modules that can be loaded into Python. However, you can also write your own functions. Imagine you would want to write a function to calculate the mean age, given a list of integers with ages. Then you could define a function to do this:

```
def averageage(agelist):
    y = sum(agelist)/len(agelist)
    return y
```

We basically define that our function takes some list as input (we called it `agelist`, but we could have chosen any other name). Then, it calculates some variable `y` (again, an arbitrary name) and returns that value.

So, if we have a list

```
ages = [22, 25, 23, 28, 26]
```

we can now simply write

```
averageage(ages)
```

to get the mean age, because we have defined the new function `averageage` before.

Obviously, for such a simple thing, this is not really necessary, because we could have just calculated

```
sum(ages)/len(ages)
```

directly. But if the calculation is more complicated and if we have to do it repeatedly, this can save us a lot of copy-pasting and makes our code more readable.

Once I was teaching this course, a student had the idea of writing a function she named `translatetoenglish()`. It took a Dutch string as argument, passed that one to to Google Translate, and returned the English

translation. So, once the function was defined, you could write something like `print(translatetoenglish("Wat een mooie dag!"))`, which would produce the following output: `What a beautiful day!`. She used it for more serious business, namely to loop over a dataset of multilingual tweets to produce a new dataset in English. What a loop is, is explained in Section 4.4.

4.3 Methods

We have learned that a function is called[2] by simply writing its name and passing some variables as an argument between `()`. Another way of doing something with variables is using a *method*[3]. In contrast to a function, a method is directly associated with an object such as a variable. For example, each string has a built-in method with the name `.lower()`. If you want to change all Capital Letters to lowercase, you can simply call the method by attaching it to the string. See these examples:

```
1 hi="Hello world OUT THERE"
2 print(hi.lower())
3 print("TESTtttt".lower())
```

As you see, methods are very similar to functions, but the way they are called is different. If `lower()` was a function, you would write `lower("TEST")` — but it is a built-in method of each string rather than a function, so you have to call it by writing `"TEST".lower()` instead. Obviously, `.lower()` does not need an additional argument to convert the string to lowercase, as it a method *of the very string it is supposed to work on*, so there is simply nothing between the `()`. Nevertheless, you cannot leave away the `()`, because every function and every method has to end with `()`, otherwise, it would not be executed.

For now, it is not important to know why some things are a method and some are a function.[4] It is important to know, however, that both ways of doing something with your variables exist and how you use them.

Note that functions and methods usually do not change the value of an argument itself but rather return a new object as result. So, if you have a string `mystring = 'HELLO'`, then calling `mystring.lower()` does *not* change mystring itself – it only *returns* the lowercased version of it. You therefore have to assign the result to a new variable (`mylowerstring = mystring.lower()`) or to the old one (`mystring = mystring.lower()`).

[2] To 'call a function' means to execute it.

[3] Strictly speaking, a method is also a function, but hey, this is no computer science course, so we'll allow ourselves to be a bit sloppy (and it relieves me from the burden to explain to you what a *class* is, what I would have to do to tell you the complete story).

[4] In fact, sometimes this is kind of an arbitrary choice made by those who invented Python. Some operations have even been implemented as both function and method.

4.4 For loops

All these functions and methods are mainly interesting if you do not apply them one time, to one string, but hundreds, thousands, or millions of times. In fact, this is the very essence of what we are doing in this course: Splitting up a task in small functions or methods that we then apply repeatedly.

To tell the computer to repeat something, we use a construction called a loop, or more specifically, a *for-loop*. Let's start with an example (when typing it over, start line 3 by pressing SPACE four times):

```
1 alltweets = ["Great lecture at the UvA", "I HATE YOU!", "I want BEER"]
2 for tweet in alltweets:
3     print(tweet.lower())
```

> ! After a line ending with a colon, Python expects an indented block. If you do not do this in a consequent way (for example, you sometimes press TAB, sometimes use 4 spaces, sometimes 3, or none at all, Python will complain and give you an "indention error". Some IDEs and some editors automatically convert TABs to spaces, but to avoid any confusion, the best convention is to simply never press TAB and always use spaces.

What happens? In line 1, we define the list `alltweets`. In line 2, we instruct the computer to take the first element of the list `alltweets` and give that element the name `tweet`. This is done by using the `for`-command. In human language, it might read as "Please repeat the task I will explain in the following indented lines `for` each element `in` the given list. In the task description given in the following block of indented lines, I will refer to the element that you are working on at that time as `tweet`".

Note that the name `tweet` is completely arbitrary chosen, while `for` and `in` are mandatory statements.

The indented line, line 3, is now executed. `tweet` now has the value of the first tweet of the first element of `alltweets`. After the indented block is finished, the program goes back to line 2, takes the next element from the list, assigns the name `tweet` to that next element, executes the indented block, and repeats until all elements have been used.

We could extend our code to produce output that looks a bit more fancy (and do not forget to indent lines 4 to 7 by using four spaces at the beginning of each line):

```
1 i=0
2 alltweets = ["Great lecture at the UvA", "I HATE YOU!", "I want BEER"]
3 for tweet in alltweets:
4     print("Printing tweet number",i,"in lowercase:")
5     print(tweet.lower())
6     print("\n")
7     i = i +1
```

\n denotes a line ending, so we get an empty line. In the last line, we add one to our counter variable `i`. A short form of doing this is `i+=1`, and it's pretty likely that I'll use that shorthand in the lectures.

If you want to loop over a dict, you can do so as follows:

```
1 people = {"Sheila": 28, "Anne": 23, "John": 22, "Bas": 25, "Mark": 26}
2 for k, v in people.items():
3     print(k, 'is', v, 'years old')
```

`k` and `v` are arbitrary names, but it seems intuitive to call the key `k` and the value `v`. We could also write `for name, age in people.items()`

Can you write your own loop using some of the functions and data structures you learned until now?

4.5 If statements

Another important way of structuring your code is to define that it only has to be executed under specific conditions. Again, the block of code that has to be executed under a condition is indented and introduced by a colon.

```
1 age = 23
2 if age > 25:
3         print("Older than 25")
```

Of course, this does not print anything, as 23 is less than 25. You can also specify different conditions using the `elif` statement:

```
1 age = 23
2 if age >23:
3         print("Older than 23")
4 elif age <23:
5         print("Younger than 23")
6 elif age==23:
7         print("Exactly 23")
```

If you want to have a condition that is met if none of all others are met, you can use `else:`. Note the double == in line 6. This is very important: a single = assigns a value to a variable, a double == compares two variables. Maybe it is obvious, but also note that you can nest such structures and have a condition within a for-loop (and maybe have another loop within the condition).

Can you write a program that takes each element from a list (using a for-loop), and then prints it if it if satisfies a specific condition?

4.6 Recap

You should have no problems explaining what the following datatypes are:

- int
- float
- bool
- string
- list
- dictionary

You should know what functions and methods are and how to use some basic ones like `print()`, `len()` or `.lower()`.

You should have understood that you can write such functions yourself.

And, very importantly, you have to understand for-loops and if-conditions, including the role of indentation to structure your code.

Chapter 5

Retrieving and storing data

We already learned in section 4.1 (Datatypes) how we can internally create some data structures that we can use to further do some analyses. For example, if we wanted to do some analyses with names, ages, and height, we could create three lists like this:

```
1 age = [22, 25, 23, 28, 26]
2 name = ["John","Bas","Anne","Sheila","Mark"]
3 height = [1.83, 1.77, 1.55, 1.76, 1.74]
```

But of course, if we had such a limited amount of data, we wouldn't need to write a program to analyze it. So, we probably want to import the data from an external source. In this chapter, we will describe some ways to to this.

5.1 CSV files

In the easiest secenario, we already have a *table* in which each row contains a case and each column the value for age, name, and height, respectively. The universally acceptable format for such a table are CSV files: Text files in which columns are seperated by commas. Sometimes, a semicolon is used instead, conventions differ. Others use a TAB character instead, which is essentially the same, although one usually calls it a TAB-seperated file then.

You actually already saw a CSV file on page 13. Have a look back there if you forgot. Virtually all programs (including SPSS, Stata, Excel, ...) can read and write CSV files, that's why we will use them very frequently. When creating data with such a program, make sure that you know how they are formatted exactly (e.g., are columns separated by `,` or `;`, are strings encapsulated by `""` or not, ... You can use the tools described on page 13 to inspect the files.

Lets create a text file with the following content:

```
1 John,22,1.83
2 Bas,25,1.77
3 Anne,23,1.55
4 Sheila,28,1.76
5 Mark,26,1.74
```

You can use Geany for this, an editor you find somewhere in the menu, but which you can—surprise, surprise!—also start from the command line with `geany &`. Let's assume we saved this file as `/home/damian/mensen.csv`.

Now we can write a simple Python program to read this data:

```
1  import csv
2  name=[]
3  age=[]
4  height=[]
5  with open('/home/damian/mensen.csv', encoding='utf-8',mode='r',newline='')
       as csvfile:
6      reader = csv.reader(csvfile, delimiter=',')
7      for row in reader:
8          name.append(row[0])
9          age.append(row[1])
10         height.append(row[2])
11 print("Done!")
```

We see a lot of familiar structures: for example, the construction of (empty) lists in lines 2 to 4, the `for`-loop in line 7 to 10. So, let's first focus on the new parts:

- The `import` statement. It simply says that we want to load an external module, in this case one that knows how to read csv files. Import statements are usually written in the very first lines of a python scripts.
- The `with open(...) as:` construction. It indicates that we want to open a file and assigns a name to that so-called *file object*, indicated by `as` (in this case, we decided to call it `csvfile`, which is a completely arbitrary name). The statement ends with a colon, so an indented block has to follow (remember?). When the indented block is over (thus, after line 10), the file is automatically closed again. That's exactly what we want, because by then, we have read all the information and stored it in lists; we don't need the file any more. The arguments passed to the `open()` function specify the file name, but also some other details of how to open it. We'll skip the details for now.
- The `csv.reader()` function. It returns an object[1] (which I decided to call, not very creatively, `reader`) that we can use to actually read the

[1]For those who are interested: More specifically, it returns a so-called generator. You

file. Note that in line 5, we only *opened* the file, we have not *read* a single byte from it yet. In line 5, we actually could have opened any arbitrary file—the `open()` command does not imply a specific data structure. Therefore now, in line 6, we define how to read from the file: Namely, by using the `csv` module. As an argument, we can give more specific information, for example, as done here, that the columns are separated by a `,`.

With your knowledge gained so far, you probably already can explain what in the for-loop in line 7–10 exactly happens. Think about it for a moment before reading further.

OK, the solution.

In line 7, we take the first row from the csv table, as provided by the `reader` object. We call it row. `row` simply is a list of all columns in that specific row. So, when we start, it is the list `["John",22,1.83]`. This means that we can get the value in the first column, `"John"`, by asking for the first element of the list, namely `row[0]`. That's what we do in line 8, where we put this value into the (until now empty) list of `name`s. The same is done with age and height, before we return to line 7 and get the next row from the `reader`.

Now, `row` has the value `"Bas",25,1.77`, and we again append `row[0]` to the list `name`, `row[1]` to the list `age`, and `row[2]` to the list `height`. We continue with this procedure until nothing is left to read from the file.

Let's also check if the data actually looks like what we expected:

```
1 print(name)
2 print(age)
3 print(height)
4 i=2
5 print(name[i],'is',age[i],'years old and',height[i],'meter tall')
```

Great!

Let's now save these columns in to a new file, `test.csv`, but in a different order. To this end, we use the `zip` command that basically glues several lists together. It goes like this:

```
1 output=zip(age,height,name)
2 with open("test.csv",mode="w",encoding="utf-8") as fo:
3     writer=csv.writer(fo)
4     writer.writerows(output)
```

can loop over a generator, but you can do so only once. In every iteration of the loop, it gives you one next element (one more row from the file in our case), and once there are no elements left, it stops. The great advantage of this, especially when dealing with large files, is that you don't have to load the whole file into memory at once.

That's all! Have a look at the file you just created and check if everything went right.

5.2 JSON files

OK, let's do this quickly, because what we will do in the next section is way cooler: We will retrieve JSON-data directly from an Google or Twitter. JSON is a data structure that is very similar to (if not identical to the one of) a Python dict, You already learned on page 24 what a dict is. Have a look back if you do not recall exactly.

Because a JSON file has the same data structure, we can open a JSON file and directly import it into a dict. This is really handy, as a lot of online data is available in that format.

Lets again (with an editor like Geany) create a file with the following content and call it `/home/damian/mensen.json`.

```
{"Sheila": 28, "Anne": 23, "John": 22, "Bas": 25, "Mark": 26}
```

To read this file into a Python dict, we only need to do the following:

```
import json
with open("/home/damian/mensen.json", mode="r", encoding="utf-8") as fi:
       mydict = json.load(fi)
print(mydict)
print("Sheila is ",mydict["Sheila"])
```

Note that both the dictionary name `mydict` and the name of the file object `fi` are completely arbitrary. I often use `fi` (file in) for files I read from and `fo` (file out) for files I want to write to, so that I don't confuse the two.

As you might expect, saving is as easy as opening. We only need to do call the function `json.dump()`, which takes two arguments: the dictionary that we want to save and a file object:

```
import json
ages = {"Sheila": 28, "Anne": 23, "John": 22, "Bas": 25, "Mark": 26}
with open("/home/damian/mensen.json", mode="w", encoding="utf-8") as fo:
       json.dump(ages,fo)
```

Note that in the `open()` function, we now specify `mode="w"` instead of `"r"` — we open a file for writing, not for reading.

> ! Note that opening a file for writing (i.e., with `mode="w"`) creates a new file if it does not exist, **but if it already exists, it immediately deletes all content** that might have been in there.

But as I promised, it get's cooler, because in the next section, we will retrieve JSON objects not from a file, but directly via an API.

5.3 Getting to know APIs

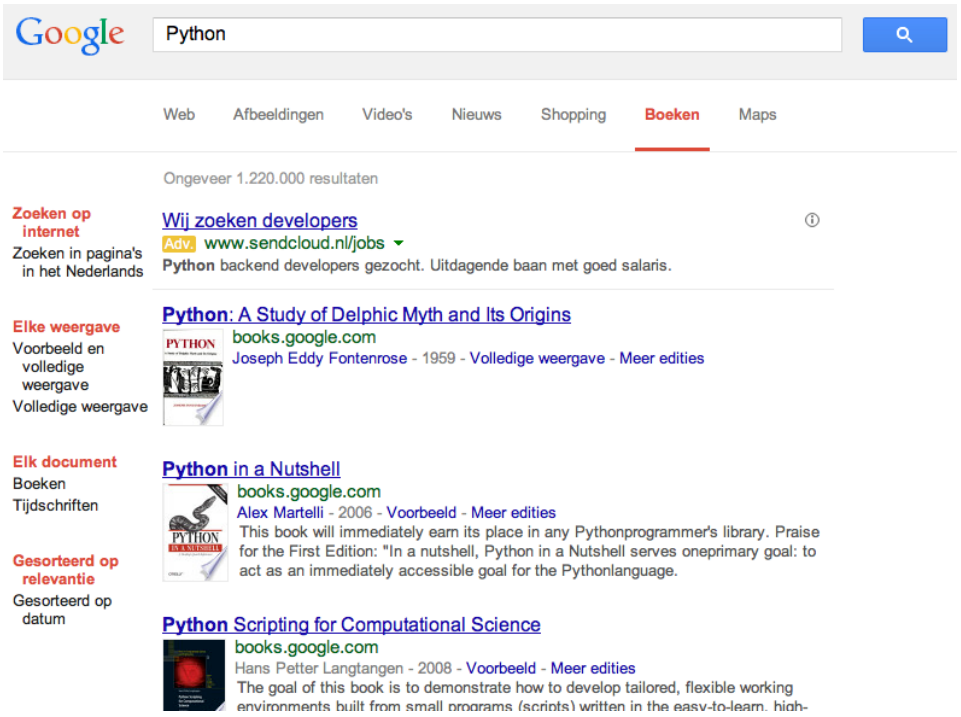


Figure 5.1: The GoogleBooks web interface

Let's assume we want to retrieve data from some online service, for example book descriptions from GoogleBooks. Of course, we could surf to their website, enter a search query, and somehow save the result. This would result in a lot of impracticalities, most notably that the website is perfectly readable and understandable for humans, but may have no meaning for a computer program. As humans, we have no problem understanding which parts of Figure 5.1 refer to the author of a book, what the numbers "2006" and "2008" mean, and on. But it is not trivial to think of a way to explain to a computer program how to identify variables like `author`, `title`, or `year` in the output.

> ! We will learn how to do exactly that in Section 8.3 (Parsing HTML pages). Writing such a parser can be really useful, but it is also error-prone and some kind of a detour, as we are trying to bring some information that has been optimized for human reading *back* to a more structured data structure. Thus, if an API exists, it saves *a lot* of work.

Luckily, however, many online services do not only have web interfaces, but also offer another possibility to access the data they provide: an API

(Application Programming Interface). In our example, we submit our request to GoogleBooks, and the API returns a JSON object with all the relevant data in it, in a neat and organized structure.

Consider the case of GoogleBooks. It offers an extremely simple API, as you do not have to log on (everyone can just use it), and because you call it in a very straightforward way: Assiming that you want to retrieve books that match the search string "python", you simply open the url `https://www.googleapis.com/books/v1/volumes?q=python` — and get a JSON object with all relevant information.

```
1 from urllib.request import urlopen
2 import json
3 from pprint import pprint
4 antwoord=urlopen("https://www.googleapis.com/books/v1/volumes?q=python").
      read()
5 data=json.loads(antwoord.decode("utf-8"))
6 pprint(data)
```

What does the code do? In lines 1 to 3, we import a module to download data from a URL, a module to read JSON data, and (to have some luxury) a module that prints json objects in a more readable way. Nothing spectacular so far.

In line 4, we access the GoogleBooks API and store the response in a variable we gave the arbitrary name `antwoord`. The function `urlopen` opens the URL, and its `.read()` method actually reads the content one gets when accessing the URL. Note that this is very much the same like opening a file and subsequently reading its content.

Until now, `antwoord` is just a bunch of bytes, and we have not made any attempt to somehow interpret it. That's what we do in line 5: `antwoord.decode("utf-8")` transforms the bunch of bytes into a string. However, a string is not really what we want, because we know that the string contains in fact JSON data[2]. And, as we saw in Section 5.2 (JSON files), a JSON string can be directly translated to a Python dict. This is done by the function `json.loads()`. Thus, `data` contains a Python dict with all information provided by the GoogleBooks API based on our search query. In line 6, we simply print that information.

Why `json.loads()` in line 5 and not `json.load()`, as we did in Section 5.2 (JSON files)? Because `json.load()` takes a file object (which we don't have here) as argument, and `json.loads()` a string.

[2]We know these things (that the bytes are a UTF-8 encoded string and that the string contains JSON data) because it is defined like that somewhere in the documentation of the GoogleBooks API. Nevertheless, both things are very common for many APIs, and one could therefore just try it out as well.

Let's have a look at the output `pprint(data)` produced to see what is actually stored in our dict `data`. As you can see, it is a lot, which is why I removed some lines from the following listing to make the picture clearer.

```
{'items': [{'accessInfo': {'accessViewStatus': 'SAMPLE',

<<I REMOVED SOME LINES HERE>>

   'volumeInfo': {'authors': ['Niklas Luhmann'],
        'canonicalVolumeLink': 'http://books.google.nl/books/about/Love.
           html?hl=&id=Hh-kncGf4QkC',
        'categories': ['Social Science'],
        'contentVersion': '0.0.2.0.preview.3',
        'description': 'This short text, originally '

<<I REMOVED SOME LINES HERE>>

        'language': 'en',
        'pageCount': 96,
        'previewLink': 'http://books.google.nl/books?id=Hh-kncGf4QkC&
           printsec=frontcover&dq=inauthor:%22Niklas+Luhmann%22&hl=&cd=1&
           source=gbs_api',
        'printType': 'BOOK',
        'publishedDate': '2010-12-06',
        'publisher': 'Polity',
        'readingModes': {'image': True, 'text': True},
        'subtitle': 'A Sketch',
        'title': 'Love'}},
```

First of all, we see at the very beginning that `data` contains an entry `items`, which is is probably what we need. We could access this part of the dict by typing `data["items"]`.

Now let's see what `data["items"]` consists of. First of all, it seems to be a list of different items (book records, in our case), as `[` denotes the beginning of a list here (in contrast to {, which would imply a dict, see Section 4.1 (Datatypes)). This implies that we could get the i^{th} book by typing `data["items"][i]`.

What information is provided for each item? First of all, we find another dict called `'accessViewStatus'`. This doesn't look very interesting, so let's skip it. Some lines later, however, we see that our first book (and presumably also all the following books) have a dictionary called `'volumeInfo'`. That looks promising!

Now we're there! The dict `data['items'][i]['volumeInfo']` consists of several key-value pairs, so that we could get the language in which the 3^{rd} book was written by typing: `data['items'][2]['volumeInfo']['language']` (remember, we start counting at 0).

While `pprint` can be useful to get a visual representation of small files and objects, it is a bad idea to print it. In that case, it is much better to use commands like `len` and `type` to get an idea of the data, and to use subsets of the data by selecting just one element, such as `firstelement=mylist[0]` or `oneentry = mydict[somekey]`.

If you want to see all keys of a given dictionary `mydict` without having to wade through all the output, then you can get them by calling the method `mydict.keys()`.

Let's put all this into practice and write a small program that gives us some information on books written by a specific author. What about some information on books written by Niklas Luhmann, one of the most important social theorists of the 20th century[3]? We just saw how to access the data we retrieve from the API, but we still have to see how we can transmit a more complicated search string (namely, one specifying that we are interested in one author only).

Let's do some reverse engineering. By playing around with the Google-Books web interface and paying attention to the URLs that appear in your browser's address bar, you can actually infer how the part after `?q=` is constructed for more complicated search strings. I basically surfed to `http://books.google.com`, entered Luhmann as search string, got some results, clicked on the link of one of the results that were actually indeed written by Luhmann, which then gave me the page showed in Figure 5.2. In fact, I saw that entering `inauthor:"Niklas Luhmann"` would have been the correct search string for what I intended, and I also saw that this indeed becomes part of the URL.

Putting all this information together makes it possible to write the following code to retrieve information on some books written by Luhmann:

```
1 luhmann=urlopen('https://www.googleapis.com/books/v1/volumes?q=inauthor:"
      Niklas+Luhmann"').read()
2 niklas=json.loads(luhmann.decode("utf-8"))
3 for boek in niklas["items"]:
4     print (boek["volumeInfo"]["authors"],": ",boek["volumeInfo"]["title"])
```

And why not calculate how long they actually are?

```
1 totalpages=0
2 numberofbooks=0
3 for boek in niklas["items"]:
4     pages=int(boek["volumeInfo"]["pageCount"])
```

[3]Accoring to Wikipedia. Actually, we could also access Wikipedia's API as well to directly retrieve this information. But, to be honest, I just used their web interface. But if I were to collect biographical information on more than 20 sociologists, it would probably already worth learning how to use Wikipedia's API.

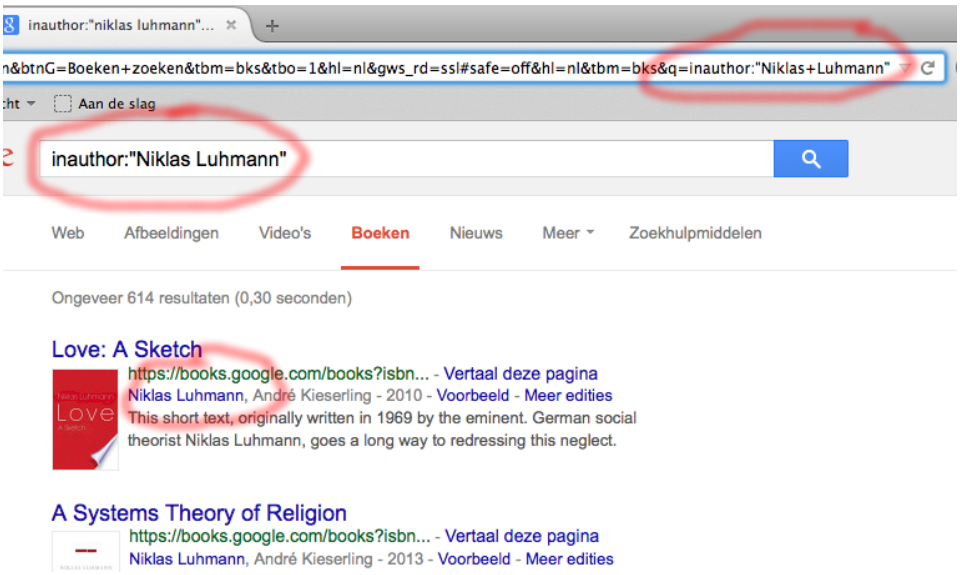


Figure 5.2: Reverse-engineering the GoogleBooks API

```
    print(pages)
    totalpages=totalpages+pages
    numberofbooks=numberofbooks+1
print ("The average length of a book by Luhmann is", totalpages/
    numberofbooks, "pages")
```

? An alternative way of solving the same problem would be the following program. Can you see advantages and disadvantages of both approaches? Which approach would you choose?

```
pagelist=[]
for boek in niklas["items"]:
    pagelist.append(int(boek["volumeInfo"]["pageCount"]))
print ("The average length of a book by Luhmann is", sum(
    pagelist)/len(pagelist), "pages")
```

! There are APIs for a lot of different purposes, it's always worth checking if a source you want to use has one. For example, even the Dutch parliament has one, via which you an access a lot of interesting material: `https://opendata.tweedekamer.nl/nieuws/kamer-maakt-parlementaire-data-toegankelijker`

5.4 The Twitter API

Twitter recently swittched from a 140 character limit to a 280 character limit. The examples below still are only retrieving up to 140 characters.

There is a new option `tweet_mode=extended` *that needs to be passed to the API to retrieve the full text. Instead of using the text field, you can then use the* `extended_tweet` *field and its subfields. For more information, see* `https://developer.twitter.com/en/docs/tweets/tweet-updates` *or the documentation of the libraries you are using.*

5.4.1 Getting all tweets by a given user

For this exercise, we need to import a module named `twitter`.[4] You can easily install it from the command line with

```
sudo pip3 install twitter
```

Do this before you read any further.

There is a lot of information on the Web, an extensive documentation on `https://dev.twitter.com/`, and we will have a number of exercises on the Twitter API in class, so there is no need to go too much into detail here. There are two main differences with the GoogleBooks-example: (1) You do not have to access the API directly via `urlopen()`, but some nice person already wrote a so-called wrapper which does that for you; and (2) you need an account and log on.

To create an account, go to `https://apps.twitter.com/app/new`, register as a developer, create a new app and note down your consumer key, consumer secret, access key and access secret (the latter two are also referred to as access token). See Figure 5.3 for screenshots. Most things should be rather self-explanatory. You can leave the field for a "callback URL" empty, and for "website", just make something up like `idontknowyet.com`.

In the following example, replace ACCESS_KEY etc. with your data (don't forget to put them between `" "`). If you want to retrieve my last 20 tweets as a JSON object (who wouldn't want to do that?), you can simpy do so with a three-line program:

```
from twitter import *
twitter = Twitter(auth = OAuth(ACCESS_KEY, ACCESS_SECRET, CONSUMER_KEY,
    CONSUMER_SECRET))
posts=twitter.statuses.user_timeline(screen_name="damian0604", count=20)
```

If you do not care about my tweets, but rather about more general information about me (like my bio statement or the number of followers I have), you can do so as well:

[4] Another popular module for accessing the Twitter API is called `tweepy`. If you want to dig deeper into using the Twitter API, you might want to check that one out as well and find out which one fits your purposes best.

```
tweep="damian0604"
info=twitter.users.show(screen_name=tweep)
print(tweep,"has",info["followers_count"],"followers.")
print("He is following",info["friends_count"], "people.")
print("This means that his follower-to-following-ratio is",int(info["
    followers_count"])/int(info["friends_count"]))
print("\nThis is what he says about himself:\n")
print(info["description"])
```

To see what type of elements `info` contains, you can either read the documentation on `https://dev.twitter.com/` or use `print` or `pprint` to inspect its structure.

5.4.2 Saving the retrieved data

Once you have retrieved the data from an API, there are several ways to go. You could directly analyze them in some way, but maybe you just want to save them first (in fact, it might be a good idea to sve them anyway for archiving purposes). Combining our knowledge we gained up til know, we can conceive of multiple ways of storing the data. The first possibility, as we have learned, would be to simply dump the Twitter dataset we aquired into a JSON file.

```
with open("/home/damian/someone.json", mode="w", encoding="utf-8") as fo:
        json.dump(info,fo)
```

We could also think of writing a CSV table, in which we store only the relevant information. This is especially useful if we want to further anayze the data using a different program later on. To this end, we could first store all relevant information in lists and then save these lists to a CSV file.

```
import csv
tweets=[]
dates=[]
for p in posts:
        tweets.append(p["text"])
        dates.append(p["created_at"])
output=zip(dates,tweets)
with open("tweetswithdate.csv",mode="w",encoding="utf-8") as fo:
        writer=csv.writer(fo)
        writer.writerows(output)
```

5.5 Listening to the Twitter stream

Many people want to collect all tweets containing a given hashtag. However, it is not possible to get these retrospectively (unless you pay for it). Instead,

you have to do so live (like in the good old days, when you were taping songs from the radio or had a VCR to record television shows). Thus, your computer must run 24/7; and you probably want to consider a lot of things in advance (e.g., the available storage, error handling in case your script fails, etc.). Nevertheless, for the sake of completeness, you find an example of how to listen to the Twitter stream and retrieve everything related to a given search term or hashtag.

First of all, we need to install another module (we could also use the Twitter module we used above, but tweepy is easier to use)

```
sudo pip3 install tweepy
```

I will not explain the code below in detail, because it requires knowledge of what a class is[5], but as you probably can see, we listen to the twitter stream and retrieve all tweets with the hashtag #metoo that are send *at this very minute*.

```
from tweepy.streaming import StreamListener
from tweepy import OAuthHandler
from tweepy import Stream

consumer_key = "..."
consumer_secret = "..."
access_token = "..."
access_token_secret = "..."

fo = open('mytweets',mode='w')

class MyListener(StreamListener):
    def on_data(self, data):
        fo.writelines(data)
        return True
    def on_error(self, status):
        print(status)

listener = MyListener()
auth = OAuthHandler(consumer_key, consumer_secret)
auth.set_access_token(access_token, access_token_secret)

stream = Stream(auth, listener)
stream.filter(track=['#metoo'])
```

The file we produce, `mytweets`, basically contains a JSON document per line, where the last line is probably corrupted (as the script above runs infinitely, so we have to kill it at one point, probably in the middle of writing

[5]Unfortunately, we do not have the time to cover classes in this course. If you want to delve into this topic yourself, though, you can find some first hints in Appendix D.3

a tweet). We could for example read it again as follows:

```
import json

tweets = []
lines = open('mytweets').readlines()
for line in lines[:-1]:
    tweets.append(json.loads(line))
```

5.6 Another example: The AmCAT API

Some of you might use AmCAT (Van Atteveldt, 2008), a framework for content analysis. If you have no idea what AmCAT is, just skip this section. If you do, however, you will be delighted to hear that you can access your AmCAT projects via an API and retrieve all your data for further analysis with Python.

First, you have to install the AmCAT Client:

```
sudo pip install amcatclient
```

You need your username, your password, the project number, and the article set number. Then, you can simply adapt the following sample script to your own needs.

```
import json
import csv
from amcatclient import AmcatAPI

api = AmcatAPI('https://amcat.nl', 'USERNAME','PASSWORD')
articles = api.list_articles(project='XXXX', articleset='XXXX')

# articles is a generator, meaning you can read it only once
# that's impractical for us, so we turn it into a list:
articles = [art for art in articles]

#get an idea about the data by printing info about the first item
print(articles[0].keys())
print(articles[0])

#%% save as json
with open("articles.json",mode="w",encoding="utf-8") as fo:
       json.dump(articles,fo)

#%% additionally, make a csv table with some of the data fields
with open("articles.csv",mode="w",encoding="utf-8") as fo:
    writer=csv.writer(fo)
    for art in articles:
```

```
24          # while saving, make sure that the text does not contain
25          # linebreaks ("\n" or "\r")
26          writer.writerow([art["medium"],art["date"],art["text"].replace("\n
                ","  ").replace("\r","  ")])
```

5.7 Recap

You should have understood

- how a CSV file looks like and how it can be read;
- how a JSON file looks like and how it can be read;
- how JSON data can be retrieved via an API;
- how to write the retrieved data to a JSON file;
- how to write the retrieved data to a CSV file.

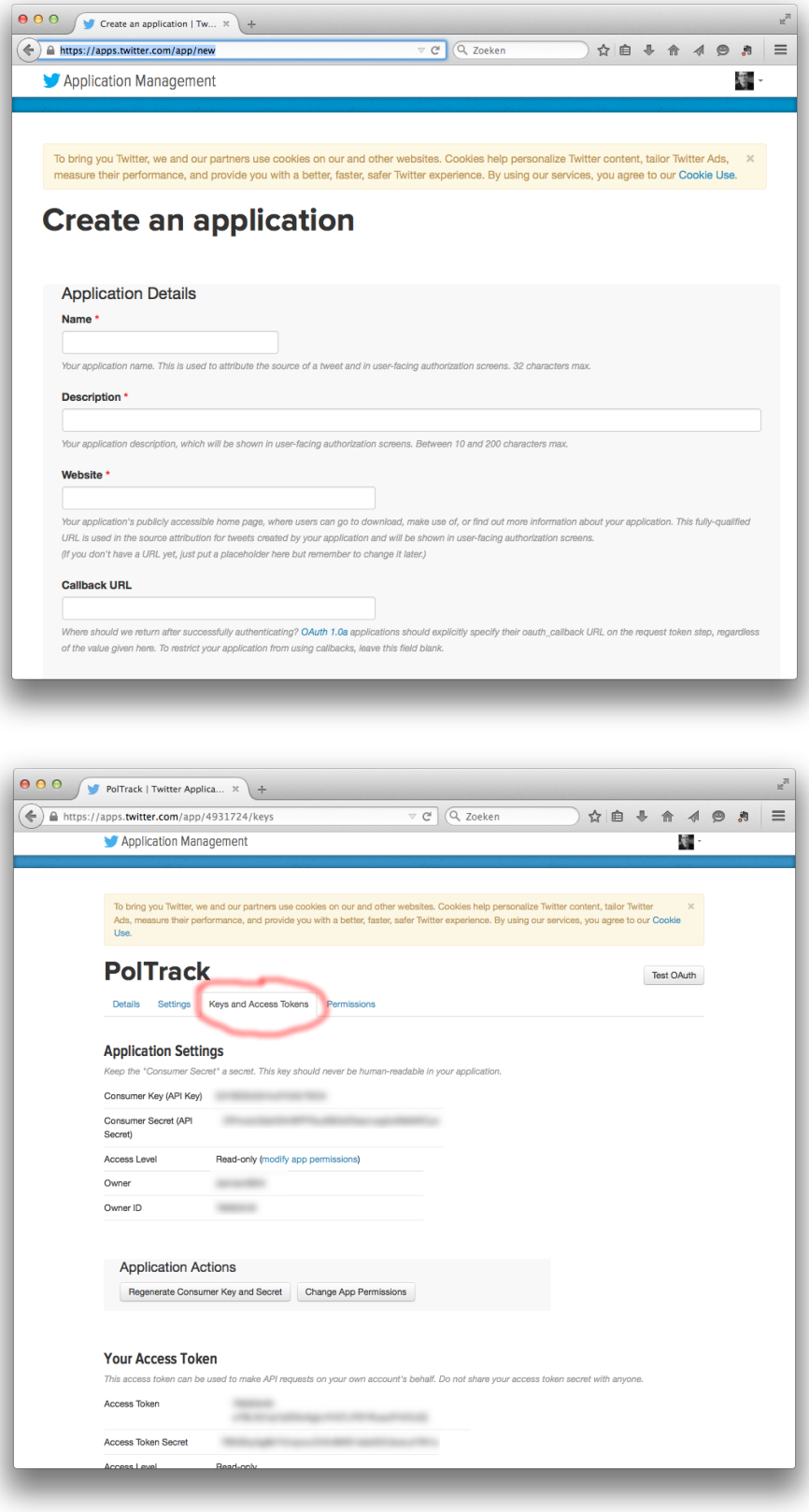


Figure 5.3: Creating a Twitter app and requesting tokens

Part II

Specific techniques

Chapter 6

Sentiment analysis

6.1 Preparation

In this chapter, we are going to conduct a sentiment analysis. You can use your own data for this, you only have to modify the proposed code accordingly. For instance, you can use the examples from earlier chapters about how to use CSV files, JSON files, or APIs and add the sentiment analysis code from the examples below to them.

We will discuss several approaches to sentiment analysis below and use several example datasets. In the end, you should be able to transfer what you learned to own datasets and make an informed decision on the approach to use.

You can also download the whole code for the first example if you want to:

```
1 wget https://raw.githubusercontent.com/damian0604/bdaca/master/examples/ex
    -senti-simple.py
```

6.2 A simple dictionary-based approach

The simplest way to do a sentiment analysis is to simply count the number of positive and negative words in a text. While this has obvious drawbacks (for example, because we do not care about word order, we cannot distinguish between 'good' and 'not good'), it is a good starting point to understand how sentiment analysis works. Moreover, because the approach is so simple, you can easily modify it to measure other things.

First of all, we need a list of negative words and a list of positive words. Let's use the command line to make a new directory and download two files

with such lists from the internet[1]:

```
cd /home/damian
mkdir sentiment
cd sentiment
wget http://www.unc.edu/~ncaren/haphazard/negative.txt
wget http://www.unc.edu/~ncaren/haphazard/positive.txt
```

Let us use some Twitter data to play with. Unfortunately, Twitter does not allow to share datasets with tweets. It does allow, however, to share lists with tweet ids. Via the Twitter API, anyone can then download the whole tweets and reconstruct the dataset.

I prepared a list of tweet ids with tweets about Bernie Sanders in the pre-election period of the US elections before a candidate was nominated. I also prepared a script that automatically downloads the associated tweets.

Download them...

```
wget https://raw.githubusercontent.com/damian0604/bdaca/master/examples/
    sanders.txt
wget https://raw.githubusercontent.com/damian0604/bdaca/master/examples/
    retrievetweets.py
```

...and then open `retrievetweets.py`, where you have to change lines 5 to 8 by entering your API credentials (see Subsection 5.4.1). Now you can run the file, and it will retrieve a set of approximately 900 tweets and save them to a TAB-separated file.

First of all, you should carefully inspect the file to get an understanding of the data structure. For example, you could use the `head` command on the bash command line (see Chapter 2.2):

```
head -5 sanders.tsv
```

Which columns are interesting? Is there a header line (with column names), or do all lines contain data?

Once we have this information, we can make a plan on how to proceed.

1. We can skip the header row with the `next()` function (you didn't know that yet, but now you do).

2. We need a `list` of all tweets. As we know we can get this by starting with an empty list, followed by looping over all rows in the file and appending the value of the appropriate column to that list.

[1] The example presented here is based on the following tutorial: `http://nealcaren.web.unc.edu/an-introduction-to-text-analysis-with-python-part-1/`, and the word lists are originally developed by Wilson, Wiebe, and Hoffmann (2005)

3. We are only interested in the words, not in punctuation. So, we could make a second list where we replace them by a space. This can be done using the `.replace()` method.

```
1  import csv
2  tweetlist=[]
3  processedlist=[]
4  with open("/home/damian/sentiment/sanders.tsv") as fi:
5      reader=csv.reader(fi,delimiter='\t')
6      next(reader)
7      for row in reader:
8          tweet=row[0]
9          tweet_processed=tweet.lower().replace("!"," ").replace("."," ").
               replace("?"," ").replace("'"," ").replace('"'," ").replace
               ('#'," ").replace(':'," ")
10         tweetlist.append(tweet)
11         processedlist.append(tweet_processed)
```

Ok, we have the data and already did the necessary preprocessing (removing unnecessary characters in this case). BTW: You see why we access the element with the index 0 of the list `row` in line 7?

You might actually want to have a look at how the data look like to see that you did not make any mistakes. You can do so with some `print()` functions, or by clicking on the values in the variable explorer (Figure 6.1).

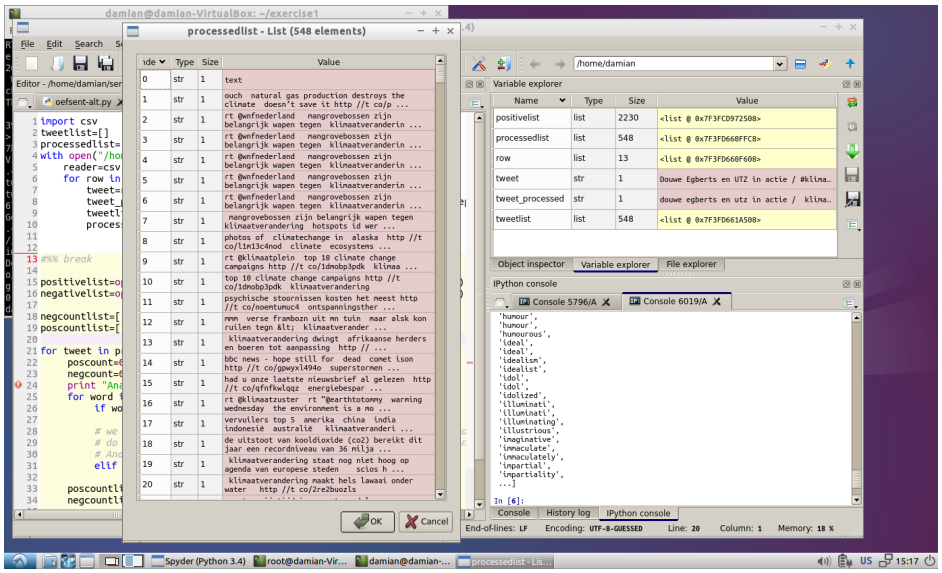


Figure 6.1: Inspecting the data with Spyder's variable explorer

We can now plan our next steps:

4. We need to read a list of positive words
5. We need to read a list of negative words

We already downloaded two of such lists, `positive.txt` and `negative.txt`. Use the bash commands `head` or `cat` to find out how they look like!

Luckily, Python makes it very easy to read a text file in which each line should be treated as a single string into a list of strings:

```
positivelist=open("/home/damian/sentiment/positive.txt").read().splitlines()
negativelist=open("/home/damian/sentiment/negative.txt").read().splitlines()
```

Print the lists and check if they look like you would expect them to look like!

Now we can start with the real analysis. The algorithm is straightforward:

6. Loop over `processedlist`.
7. Within the loop: Set a variable `poscount` that indicates the number of positive words in this specific tweet is 0.
8. Within the loop: Set a variable `negcount` that indicates the number of negative words in this specific tweet is 0.
9. Within the loop: Split each element from that list (thus, each processed tweet) into words and loop over these words
10. Within this inner loop: check wether the word is an element of `positivelist`, and if so, increase `poscount` by one
11. Same for negative words
12. Add counts to some lists
13. Optionally: weigh by the length of the tweet

Translated to Python code, it looks like this:

```
negcountlist=[]
poscountlist=[]
sentilist=[]

for tweet in processedlist:
    poscount=0
    negcount=0
    print ("Analyzing this one:",tweet)
```

```
    for word in tweet.split():
        if word in positivelist:
            poscount+=1
        elif word in negativelist:
            negcount+=1
    print("It contains",poscount,"positive words and",negcount,"negative
        words.")
    poscountlist.append(poscount)
    negcountlist.append(negcount)
    sentilist.append((poscount-negcount)/len(tweet))

print("Average sentiment:",sum(sentilist)/len(sentilist))
```

The only thing we still want to do is to save a dataset for further analysis — maybe with Python, maybe with some statistics package:

```
output=zip(tweetlist,processedlist,negcountlist,poscountlist,sentilist)
with open("/home/damian/sentiment/sanders-analyzed.csv",mode="w",encoding
    ="utf-8", newline="") as fo:
    writer=csv.writer(fo)
    writer.writerows(output)
```

6.3 Vader

There are several nice things to the simple approach demonstrated above: First, it is insightful from a didactical point of view, as it illustrates the basic principles of bag-of-words analyses. Second, it is very flexible in that it can be modified to measure a lot of other things, not only positivity or negativity. Third, it is transparent—everyone can understand the basic algorithm.

However, in a real-world setting, it might be too simplistic. Instead of writing a sentiment analysis algorithm yourself, you can therefore invoke some algorithm that others have developed. For instance, the sentiment analysis module Vader by Hutto and Gilbert (2014) deals with negation, punctuation, intensifiers ("very") and dampeners ("kind of") as well as emoticons (see `https://github.com/cjhutto/vaderSentiment`).

Vader is integrated in NLTK, a Python module that we will use for a number of text processing tasks in this course (see Section 7.2). You already installed NLTK when setting up your VM, but we need to additionally install the Vader sentiment lexicon. We only have to do this once, and we can do it within Python:

```
import nltk
nltk.download('vader_lexicon')
```

It is very easy to use:

```
from nltk.sentiment import vader
senti=vader.SentimentIntensityAnalyzer()
senti.polarity_scores('This is a great day!')
senti.polarity_scores("I don't like this food")
senti.polarity_scores("I love her, but I hate him")
```

The `.polarity_scores` method returns a dictionary with four values. In our example, we get the following results:

```
{'neu': 0.406, 'pos': 0.594, 'neg': 0.0, 'compound': 0.6588}
{'neu': 0.587, 'pos': 0.0, 'neg': 0.413, 'compound': -0.2755}
{'neu': 0.282, 'pos': 0.244, 'neg': 0.474, 'compound': -0.5346}
```

We see that we get for each of the sentences an estimate of its positivity, negativity, neutrality, and a combined measure of those. For detailed information on the workings, have a look at Hutto and Gilbert (2014)!

We see that we could easily implement this function into the script we developed in the previous section. For example, we could do:

```
compoundscores=[]
for tweet in tweetlist:
    compoundscores.append(senti.polarity_scores(tweet)['compound'])
```

Can you write a sentiment analysis script that applies Vader to your own data?

6.4 Alternatives

Let us take a step back and compare the approaches we have seen so far. The first one, the simple dictionary approach, was easy to understand and easy to modify, but not very powerful. The second, Vader, was much more powerful. One disadvantage, however, is that it is only available in English. The third one, Sentistrength, is comparable to Vader, but has as big advantage the support of a number of languages, including Dutch. However, this comes at a price: It is not available for free, making it more difficult to share code with others; and it is written in Java, which requires us to use some tricks to actually use it in Python.

But there are more alternatives. For instance, pattern.nl (De Smedt & Daelemans, 2012) is very easy to use, open source, and it supports multiple languages. However, it's support for Python 3 is still experimental. To install it, you need to do the following:

```
sudo apt install git
sudo apt install libmysqlclient-dev
sudo pip3 install git+git://github.com/clips/pattern.git@development
```

For more information, see `https://www.clips.uantwerpen.be/pattern`.

A last, but increasingly popular method for doing sentiment analysis is using supervised machine learning. How to do that will be discussed in Chapter 10.

6.5 Recap

You should have understood the general working of different approaches to sentiment analysis. You should know about their pros and cons. Practically speaking, you should be able to

- implement a simple, dictionary-based sentiment analysis tool yourself
- integrate a tool like Vader, SentiStrength, or pattern in your own analyses.

Chapter 7

Automated content analysis

Automated content analysis covers a wide range of techiques, and not all of them can be covered within this tutorial (for an overview and more background information, see Boumans and Trilling (2016)). We will start with a first, powerful yet easy deductive approach, in which we basically count how often a pre-defined pattern (a search string, for example) occurs. This technique is known as using *regular expressions*. We will then learn how to take characteristics of human language into account to get closer to the real meaning. The technique we use for this is called *Natural Language Processing (NLP)*. The other approaches to automated content analysis mentioned by Boumans and Trilling (2016) are discussed in Chapters 9 and 10.

7.1 Regular expressions

A regular expression is a *very* widespread way to describe patterns in strings. You probably know wildcards like `*` or operators like `OR`, `AND` or `NOT` that you can use in search strings in a lot of applications. But of course this is rather limited: Maybe you want to say somethhing like "I want all words starting with a letter or a number (but no special character), then a -sign, then again some letters, and after the last dot two or three letters (but not more!) letters. You probably saw that this would be some way of describing an email-adress.

A universal language to formulate such patterns is *regular expressions*. There are some dialects (but the differences are minimal), and you can use them in many editors (!), in the Terminal, in STATA ... and in Python.

In its most simple form, a regular expression is just an ordinary search string:

```
1 python
```

finds ("matches") every occurance of the word "python". But we can also indicate that two different characters are allowed:

```
[pP]ython
```

matches both "python" and "Python". Some other useful things:

- `.` matches *any* character
- `[0-9]` matches a digit
- `[a-z]` matches all lowercase letters
- `[a-zA-Z]` matches all upper- and lowercase letters
- `([tT]witter|[fF]acebook)` matches both Twitter and Facebook and is OK with misspelling them as twitter or facebook.

You probably got the idea. You can also specify how often sth has to occur:

The item *before* has to occur

- `?` 0 or 1 times
- `*` 0 or more times
- `+` 1 or more times

For example,

```
personali[zs]ed-?communication
```

matches

```
personalizedcommunication
personalized-communication
personalisedcommunication
personalised-communication
```

There are actually much more possibilities, so download a regular expression cheat sheet (google or try this one: `http://www.cheatography.com/davechild/cheat-sheets/regular-expressions/`) to build your own one. Also the wikipedia page on regular expressions is pretty good. You can also play around at `http://www.pyregex.com/`.

So, let's try to use regular expressions in Python. There is a whole module for this (its called `re`) that provides a lot of interesting functions for finding and replacing stuff based on regular expressions.

- `re.findall("[Tt]witter|[Ff]acebook",testo)` returns a list with all occurances of Twitter or Facebook in the string called `testo`

- `re.findall("[0-9]+[a-zA-Z]+",testo)` returns a list with all words that start with one or more numbers followed by one or more letters in the string called `testo`

- `re.sub("[Tt]witter|[Ff]acebook","a social medium",testo)` returns a string in which all all occurances of Twitter or Facebook are replaced by "a social medium"

- `re.match(" +([0-9]+) of ([0-9]+) points",line)` returns `None` unless it *exactly* matches the string `line`. If it does, you can access the part between () with the `.group()` method.

The difference between `findall` and `match` is thus that `findall` does not care about what it finds before and after the pattern, while `match` requires the whole line to be matched.

An example to illustrate the use:

```
line="           2 of 25 points"
result=re.match(" +([0-9]+) of ([0-9]+) points",line)
if result:
   print ("Your points:",result.group(1))
   print ("Maximum points:",result.group(2))
```

would produce the following output:

```
Your points: 2
Maximum points: 25
```

This is cool, isn't it? We now know how to extract information from a semi-structured string so that we can store it in variables for further analysis!

On page **??**, you can find some additional information about some regular expression features, especially about the difference between lazy matching and greedy matching, which basically specifies where to "stop" when a regular expression contains something like `.*` (thus, matching an arbitrary number of arbitrary characters, which is in fact much more useful than it sounds).

? Can you write a program (a so-called parser) that takes a semi-structured input file of your choice (maybe something you downloaded somewhere) as input and uses regular expressions to store the data in a structured way in a CSV table?

7.2 Natural language processing

To prepare, we have to install some more ressources. Run the following Python lines:

```
import nltk
nltk.download()
```

A graphical interface is opened (Figure 7.1).

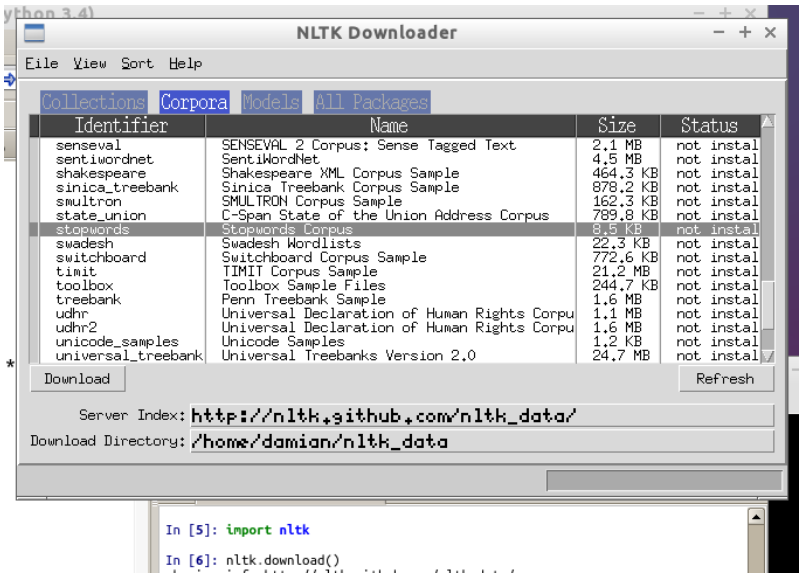


Figure 7.1: Loading additional NLTK-packages

Do *not* download all packages, that will take too much space and you do not need it. Instead, download only the packages “stopwords” , “punkt”, “averaged_perceptron_tagger”, and “maxent_treebank_pos_tagger” from the “all packages” tab. Close the window when done.

Instead of using the graphical interface, you can also install the packages by providing their name as an argument:

```
import nltk
nltk.download('averaged_perceptron_tagger')
```

7.2.1 Stopword removal

Natural language processing is a term that is used to describe a set of techniques used to analyze language written by humans. This comprises simple tasks like the removal of stopwords in order to identify “relevant” words, but also much more complicated things like parsing of a sentence – which

means to automatically identify, for instance, subject, verb and object in a sentence. The book by Bird, Loper, and Klein (2009) gives a comprehensive introduction into NLP with Python. The author also provide the Python package `nltk`, which we use here.

To find out all the possibilities, have a look at `nltk.org`, but here are some examples.

Let us first consider stopword removal. These are words without a meaning that is specific to a text, such as "a", "the", ... They are usually more disturbing than useful, because they hinder us from seeing the really interesting words. Keeping them in can lead to misleading results if we want to identify key terms (e.g., by means of a word count), if we want to calculate document similarities, or if we want to make a word co-occurance graphs. A very straightforward way to do so would be the following algorithm, which you – with the knowledge you gained so far – actually could have written yourself:

```
t = 'let us clean this up for her'
tn = ""
mystopwords=['and','the','this','a','or','he','she','him','her','us']
for w in t.split():
    if w not in mystopwords:
       tn=tn+w+" "
```

Instead of defining the list in the script itself, we could also read them from a file. Imagine we have a file called `stopwords.txt` in which we put one stopword per line (we could easily create such a file in Geany or any other text editor):

```
he
she
it
we
they
```

We could read this file to a list with

```
mystopwords=[w.strip() for w in open("stopwords.txt",encoding="utf-8").
    readlines()]
```

Especially if we have a very long list, this makes our code much easier to read and to maintain. We could also use one of the pre-defined lists provided by `nltk`, but often, we might want to tweak it to our research interests anyway.

```
from nltk.corpus import stopwords
mystopwords=stopwords.words('dutch')
```

7.2.2 Stemming

Another interesting step is *stemming*. When interpreting text, we usually do not want to distinguish between smoke, smoked, smoking. Stemming reduces all of these forms to their stem `smoke`. Therefore, it is a typical preprocessing step (like stopword removal).

So, we could split a given string into words, then stem each word, and combine them again.

```
from nltk.stem.snowball import SnowballStemmer
stemmer=SnowballStemmer("english")
s="I am running while generously greeting my neighbors"
stems=""
for w in s.split():
    stems=stems + stemmer.stem(w) + " "
```

We can now `print(stems)` to see how the stemmed string looks like.

> ! The last three lines can be replaced by a single line:
> `stems=" ".join([stemmer.stem(w) for w in s.split()])`
> The technique used for it is called *list comprehension* and a really cool feature, see Appendix D.1. In addition, the `.join()` method allows to join elements from a list to a single string. These two things are considered very pythonic and good coding style, so if you do this kind of stuff more often, it is useful to understand the working (but you can as well use the more elaborate way used in the example). In fact, also the stopword removal could be done in this way: Assuming we have a list with stopwords called `mystopwords` and a sentence `s` from which we want to remove all stopwords, we can do so with
> `s2=" ".join([w for w in s.split() if w not in mystopwords])`
> In doing so, we can create a new string `s2` with one single line of Python code in which we first split s into a list of words, then make a new list out of these words in which only those words are included that are not in the stopword list, and finally join this list into a single string seperated by spaces.

7.2.3 Part-of-speech tagging

By parsing sentences, we can find out what grammatical function the words within each sentence has. To this end, NLTK first splits sentences into words (it is called *tokenizing*). But, in addition to what we have done before with the `.split()` method, it attaches a label to each token that identifies the grammatical characteristics of each token.

```
import nltk
sentence = "At eight o'clock on Thursday morning, Arthur didn't feel very
    good."
tokens = nltk.word_tokenize(sentence)
print (tokens)
```

```
5 tagged = nltk.pos_tag(tokens)
6 print(tagged)
```

We see that we get a list of tuples (pairs of two values):

```
1 [('At', 'IN'), ('eight', 'CD'), ("o'clock", 'JJ'), ('on', 'IN'), ('
    Thursday', 'NNP'), ('morning', 'NN')]
```

This is pretty useful, because we can address the items with *slicing*. For example, to get the fifth tuple, we can write `tagged[5]`. Of course, we can also slice the tuple. So, to get only the grammatical function of the word "morning", we can use `tagged[5][1]`. Look up the meaning of the abbreviations like 'NN' up in the documentation. Chapter 5 in Russel (2013) also provides a lot of examples.

7.3 Recap

You should have understood

- How to remove stopwords
- How to stem a sentence
- How to parse a sentence and employ POS tagging
- How to use NLTK

Chapter 8

Web scraping

8.1 General approach

When scraping data from the web, we can distinguish two different tasks:

1. downloading a (possibly large) number of webpages;
2. parsing[1] the content of the webpages.

Both can go hand in hand. For instance, the URL of the next page to be downloaded might actually be parsed from the content of the current page; or some overview page may contain the links and thus has to be parsed first in order to download subsequent pages.

Let us consider the example of a hotel review website. We might, for example, be interested in retrieving all reviews from hotels in Amsterdam. Maybe the site has an overview page that lists all Hotels in Amsterdam. There are (too) many hotels in Amsterdam, so they probably do not fit on one page, but we will ignore that for now and come back to it later.

Our approach, thus, could look as follows:

1. Retrieve the overview page.
2. Parse the names of the hotels and the corresponding links.
3. Loop over all the links, retrieve the corresponding pages.
4. On each of these pages, parse the interesting content (i.e., the reviews, ratings, and so on).

[1] In this context, 'parsing' is just a fancy term for 'extracting meaningful information'

So, what if there are multiple overview pages (or multiple pages with reviews)? Basically, there are two possibilities: The first possibility is to look for the link to the next page, parse it, download the next page, and so on. The second possibility exploits the fact that often, URLs are very systematic: For instance, the first page of hotels might have a URL such as `http://myreviewsite.com/amsterdam/hotels.html?page=1`. If this is the case, we can simply construct a list with all possible URLs:

```
baseurl = 'http://myreviewsite.com/amsterdam/hotels.html?page='
tenpages = [baseurl+str(i+1) for i in range(10)]
```

Afterwards, we would just loop over this list and retrieve all the pages.

But to implement all this, we need to know how to retrieve and subsequently parse a web page.

8.2 Retrieving web pages

There are several ways of retrieving web pages. The most simplest way is maybe using the `requests` module:

```
import requests
htmlsource = requests.get('http://nu.nl').text
```

Usually, you would then want to proceed with parsing the text (see next section), but for testing purposes, you can also save it to a file:

```
with open('/home/damian/Desktop/test.html', mode = 'w') as fo:
    fo.write(htmlsource)
```

Open test.html in Firefox and make sure you downloaded what you wanted to download.

Also, instead of downloading the website again, for later use you can just read it from disk (see also Section 8.4):

```
with open('/home/damian/Desktop/test.html') as fi:
    htmlsource = fi.read()
```

Sometimes, it can be useful to pretend to 'be' a specific web browser. Some websites deliver different content depending on what browser you use (for instance, content delivered to mobile devices might look different). Also, the other party might not like it if you are, in fact, a Python program. You can pretend to be a specific browser by specifyig a so-called *user-agent string*. For instance, the user-agent string of Firefox looks like in the following example (google for more user-agent strings; there are many lists on the Internet):

```
htmlsource = requests.get("http://nu.nl/net-binnen", headers={"User-Agent
    ": "Mozilla/5.0"}).text
```

In fact, the requests library can do much more, including, for instance, handling cookies or transmitting information (like usernames or passwords). For more information, see `http://docs.python-requests.org/en/latest/user/quickstart/`

8.3 Parsing HTML pages

8.3.1 HTML pages as trees

We now know how to retrieve a given web page. But how do we extract the relevant information?

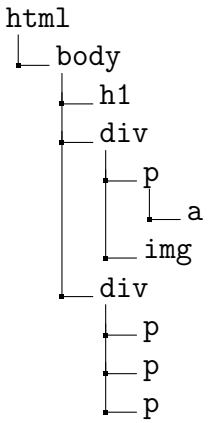
Your first idea might be that we could use regular expressions (Section 7.1) for this. And indeed, that is possible (and, in fact, it can be a last resort when everything else fails). However, using regular expressions to parse a HTML page would be error prone and unnecessarily complicated. We would, in fact, ending up reinventing the wheel: After all, parsing HTML pages is such a common task that there are packages for this. Fun fact: Under the hood, they actually sometimes work with regular expressions, and you can sometimes use regular expressions within them.

However, in order to get started with parsing HTML pages, we need to have a basic understanding of how a HTML page looks like. Here is a simplified example:

```
<html>
<body>
<h1>This is a title</h1>
<div>
<p> Some text with one <a href=bla.html>link </a> </p>
<img src = plaatje.jpg>an image </img>
</div>
<div>
<p> Some more text </p>
<p> Even more... </p>
<p> And more. </p>
</div>
</body>
</html>
```

For now, it is not too important to understand the function of each specific tag (although it might help, for instance, to realize that `a` denotes a link, `h1` a first-level heading, `p` a paragraph and `div` some kind of section).

What is important, though, is to realize that each tag is opened and closed (with `/`). Because tags can be nested, we can actually draw the code as a tree. In our example, this would look like this:



If we represent our web page as such a tree[2], we can describe each location in the page by describing how to walk through the tree to arrive at the correct position. For instance, to describe where to find "Even more text", we could say: start at html, go to body, go to the second div, and take the first p element.

If we translate this verbal explanation to a so-called XPATH, it reads:

```
1 '/html/body/div[2]/p[1]'
```

You can compare the syntax of an XPATH with the path of a given file (`/home/damian/week6/slides.pdf`), where the leftmost element is the "highest" level, and where after each `/`, it indicates to which lower level one has to turn before finally arriving at the file (`slides.pdf`).

If we leave away the `[1]` after `p`, we retrieve a list of all the three texts within the second div – which makes it ease to, for instance, retrieve all reviews from a web page instead of only one specific review.

You might observe that in contrast to how it's generally done in Python, XPATHs do not start counting at 0 (after all, XPATHs are not Python-specific).

Thus, we need a module that transforms our web page into such a tree and allows us to address parts of the tree by their XPATH. Such a module is `lxml`, which will be presented in the next session.

8.3.2 Finding a good XPATH

At first glance, one might think that there is exactly one way to write an XPATH to describe a location on the page. But that's not the case. XPATHs can also contain instructions like 'go to an arbitrary depth of the tree until you find element X' or 'only consider elements with a specific attribute'.

[2]This is also referred to as the Document Object Model (DOM), or a DOM tree.

Therefore, one often has to play around a bit before one finds the most suitable XPATH. On the one hand, one can be not specific enough (for instance, instead of selecting all links to reviews, one would also select all other irrelevant links). On the other hand, one can be too specific (for instance, only selecting the first review instead of all reviews).

In the previous section, we already saw the general structure:

- `/` seperates the different levels. Thus, `/html/body/div` means: go from html to body and from there to div.
- `[]` can be used to indicate which branch of the tree to take if there are several ones with the same name. For instance, if there are five p tags, `p[3]` would select the third.

Useful other elements are:

- `//` (thus, a double slash rather than a single slash) means 'arbitrary depth' (=may be nested in many higher levels, there may be an arbitrary number of higher levels which are not further specified). For instance, `/html/body//p` can be read as: "I don't care which path you take between 'body' and 'p'. Very often, you can start your XPATH with `//` to avoid make it shorter and avoid being too specific.
- `*` means 'everything' (if `p[2]` is the second paragraph, `p[*]` are all paragraphs)
- `[@attribute="whatever"]` lets you select only those tags that contain a specific attribute. For example, instead of just being a short and plain `<div>`, a HTML tag might then look like this:

  `<div class="reviews-single__text">`.

  An XPATH to capture all div tags that have an attribute "class" with the value "reviews-single__text" could look like this:

  `//div[@class="reviews-single__text"]`

Let try this with an example (Figure 8.1).

Go to a website of your choice (because websites change their layout quite often, any specific example is likely to be outdated when you read this).

Select some element that you are interested in, right-click on it, select "Inspect Element") and have a look at the highlighted source code (Figure 8.1).

Now run the following Python code:

```
from lxml import html
import requests

myurl = "https://www.kieskeurig.nl/smartphone/product/3518003-samsung-
    galaxy-a5-2017-zwart/reviews"

htmlsource = requests.get(myurl).text
tree = html.fromstring(htmlsource)
```

Optionally, save the HTML source code to a file and open it in a browser (see Section 8.2).

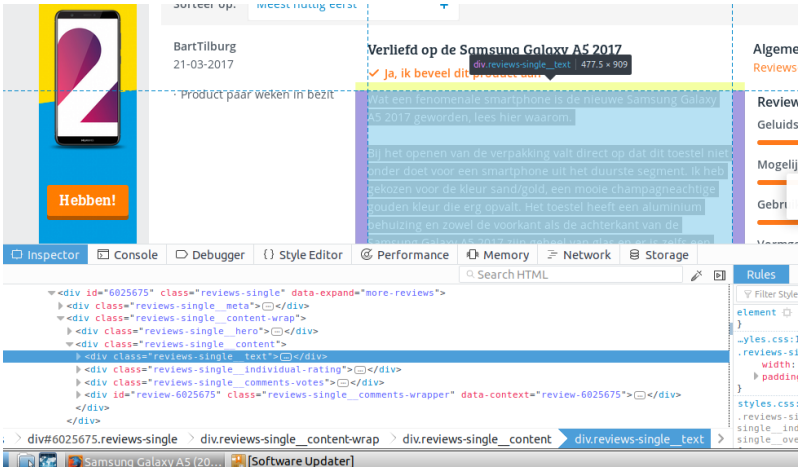
Now, playing begins. You basically need to find an XPATH (using the rules outlined above) that gives you the desired result. Make sure that you do not re-download the page every time, that just causes unnecessary traffic.

```
reviews = tree.xpath('//div[@class="reviews-single__text"]')

# get the text for each of the elements
reviewstext = [r.text for r in reviews]

# alternatively, without leading and trailing whitespace

reviewstext = [r.text.strip() for r in reviews]
```

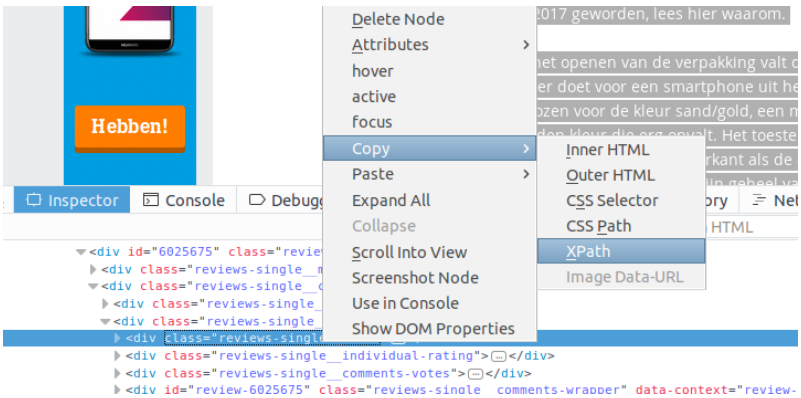
Now, we can do whatever we want with the reviews. For example, let's print an overview:

```
print (len(reviews),"reviews scraped. The first 60 characters of each:")
i=0
for review in reviewstext:
    print("Review",i,":",review[:60])
    i+=1
```

A



B



C

Figure 8.1: Select the part of the website that you are interested in, right-click, and choose "Inspect Element" (A). Have a look at the highlighted part of the HTML source code to see which tags and/or attributes are relevant for constructing your XPATH (B). Alternatively, just copy the XPATH Firefox suggests (C).

8.3.3 Parsing links

The list `reviews` that we created from `tree.xpath` contains the parsed elements in their own, very specific data type. While we cannot do much with them directly, luckily, these datatypes have a couple of properties that contain the information we need.

For instance, as we saw in the example above, they do have a property `.text` that contains the text within the element. That is very often indeed what we are interested in.

But, as outlined in Section 8.1, we sometimes want to retrieve other information, in particular, links.

For example, a website might display the string "My great hotel", which is a link to www.myhotel.com. The underlying HTML code might look like this:

```
...
...
<div class="hotel">
<a href="http://www.myhotel.com">My great hotel</a>

</div>
...
...
```

If we would construct an XPATH such as

```
element = tree.xpath('//div[@class="hotel"]/a')
```

then

```
print(element[0].text)
```

would print "My great hotel", because that's the text between the opening `a` tag and the closing `/a` tag. However, we want to retrieve an attribute of the first tag, namely the value associated with `href`.

Luckily, that's not much more complicated than getting the text:

```
print(element[0].attrib["href"])
```

will print http://www.myhotel.com.

To give another example and let's get the links to all hotels on a website that formats their links in the following way:

```
<a href="http://www.myhotel.com" class="hotel">My great hotel</a>
<a href="http://www.myrestaurant.com" class="restaurant">My restaurant</a>
<a href="http://www.myshabbyhotel.com" class="hotel">My not so good hotel
    </a>
```

Let's further assume that we don't care about where on the page the links are – they can basically be on an arbitrary position on the tree.

We could do so by writing:

```
linkelements = tree.xpath('//a[@class="hotel"]')
links = [l.attrib["href"] for l in linkelements]
```

8.3.4 CSS Selectors as an easy alternative

Another way to select elements within the DOM tree are so-called CSS selectors. Some say that there syntax is easier than the XPATH syntax. For instance, to select all elements that have an attribute `class` with the value `fluid`, we can just write: `.fluid` (with a dot). And to select all `h1` tags, we can just write: `h1` (without a dot).

```
import requests
from lxml.html import fromstring

response = requests.get('https://www.nu.nl/buitenland/5176460/britse-
    regering-zet-23-russische-diplomaten-vergiftiging-skripal.html')

tree = fromstring(response.text)

print ("Getting the headline via attribute")

print('Result via XPATH:')
print(tree.xpath('//*[@class="fluid"]')[0].text)

print('Result via CSS selector:')
print(tree.cssselect('.fluid')[0].text)


print ("Getting the headline via tag")

print('Result via XPATH:')
print(tree.xpath('//h1')[0].text)

print('Result via CSS selector:')
print(tree.cssselect('h1')[0].text)
```

All should print the same.

This already illustrates that for many tasks, it doesn't really matter whether you use XPATHs or CSS selectors. Sometimes, XPATHs can be more powerful (for instance when it makes sense to describe which path on the DOM tree to follow exactly), sometimes CSS selectors are simpler and easier to read. In the end, you need to find out what makes most sense in your project.

For an overview of all possible CSS selectors, see `https://www.w3schools.com/cssref/css_selectors.asp`

Also, as shown in panel C in Figure 8.1, you can directly copy CSS selectors from the Inspect Element function in Firefox.

8.4 Be nice!

When you put together the bits and pieces and end up writing your own scraper (for instance, by collecting all links from some page, follow these links, download and parse the pages you encounter, and repeat the whole process for all links on another page), you need to carefully think about how to minimize the impact this has on the site that you are scraping.

Every time you download a page, you somewhere cause some cost – someone has to pay for the traffic, and someone needs to provide the computing power and memory to serve the page for you. When a human visits a news site, these costs are negligibly small, but if you do not visit the website one or ten times like a human would, but tens of thousands of times, this can change.

It lies beyond the scope of this tutorial to discuss legal implications, but at the very least, it would be pretty unethical to cause unnecessary costs and stress to their servers. If you take it to the extreme, if you run a program that issues an extremely high number of requests to a server and does so with a very high frequency, then you are effectively running an attack on the website in question.

So, what can we do to minimize harm and keep everyone happy?

1. Avoid multiple downloads. For instance, during testing, save the downloaded webpage to disk and read it from there instead of re-downloading it every time you try a new XPATH. Or, when running your scraper, include a functionality to only download the pages that you do not have yet downloaded.

2. Take your time. Between downloading two pages, wait some seconds to act more like a human. You can find the code for waiting randomly between 5 and 10 seconds below.

```
1 from random import randint
2 from time import sleep
3 sleep(randint(5,10))
```

8.5 Alternatives

You can get very far with the approaches sketched in this chapter: regular expressions, the lxml-package, and wget for automated downloading tasks. BeautifulSoup is a python package that lies somewhere in between, as it resembles lxml, but internally uses regular expressions rather than an XPath to parse the HTML.

If you want to write an application that integrates crawling and parsing, you might have a look at the scrapy framework (`http://scrapy.org/`), although it might be a bit of an overkill.

One approach that we haven't covered is how to scrape content from sites that do complicated stuff like interactively loading data while they are visited. For example, think of a news site that uses a JavaScript to display comments: these comments may not be present in the HTML code of the site, but might be dynamically loaded while the user is reading the article. In such a case, one can make use of a framework like Selenium. Selenium allows you to start a browser, click somewhere, and so on (in essence, it just automates what a human would do), and then parse what is displayed in the browser. See Appenix E for more details.

8.6 Recap

You should be able to write your own web scraper. This entails that you

- have a global understanding of how a HTML page looks like, so that you can use this information to build your scraper
- understand that there are several ways to parse a specific piece of information from a page
- know how to use packages like lxml.

Chapter 9

Network visualization

We are often used to representation of data in forms of two-dimensional tables (an Excel or SPSS file, a CSV file). We have already seen that this is not always the most appropriate way to conceptualize data: Hierarchical data, for example, can often be much better represented in JSON format. Another way of representing data is in form of a *network* (Figure 9.1).

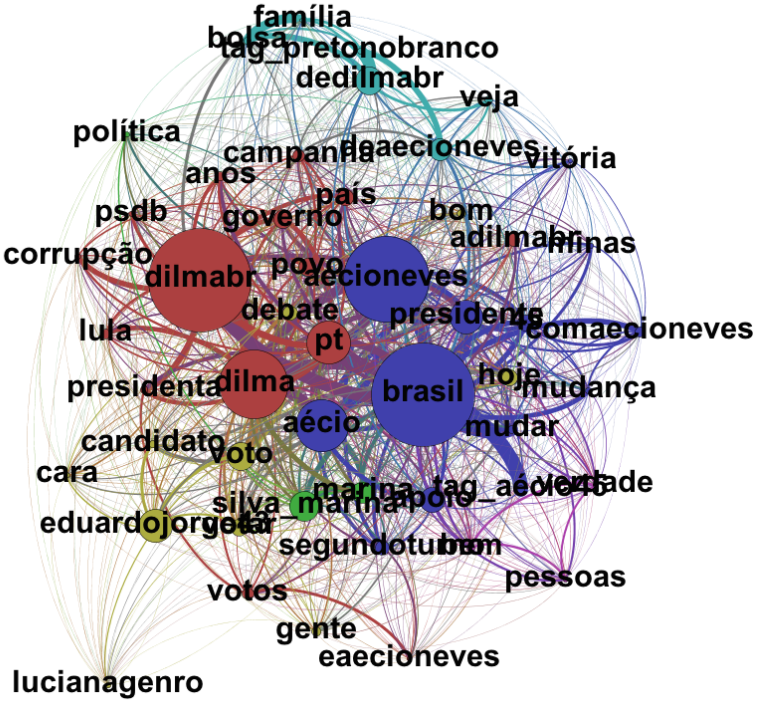


Figure 9.1: A network of co-occuring words

We call the elements in a network *nodes* and the connections between them *edges*. A node can be anything: a word, a person, whatever. Edges van be directed (like an arrow) or undirected. For example, a nework of Twitter users (the edges) could have edges that represent -mentions. These edges would be directed: if A mentions B, that does not imply that B mentions A as well. On a Facebook friendship network, this would be different: Here, we would have undirected edges.

Giving an introduction to network analysis falls outside of the scope of this tutorial, but you will learn how to save your data in a format that can be used to analyze or visualize them.

In particular, we will look how we can construct a network of words that co-occur together in texts, which can be a nice way to conduct some inductive framing analysis (see Boumans & Trilling, 2016; Trilling, 2015).

9.1 Producing a file for analysis with Gephi

Let us considering the following case: We have a list of strings (e.g., a list of tweets) and want to know which words often occur together in the same string (tweet). We could model our data in such a way that each word is a node. We could also attach a weight to each node, so that the node can be displayed bigger the more often the word is mentioned. We could now say that if two words co-occur together, they are connected with a node. Again, we could say that the more often this happens, the thicker the edge should become.

Thus, we basically need

- a list with all words (the nodes) with their frequency (the weight/size of the nodes)
- a list with all connections (the edges, thus, a list of pairs of nodes that are co-occuring in the same string/tweet) with the number of strings/tweets in which they co-occur.

The GDF file format is a format that allows us to store exactly this information. Have a look at the following example. In lines 1, it is defined how the following node definitions are formated: a variable called `name`, which is a string (that's what they mean with VARCHAR), followed by a variable `width` which has the format double (a type of number, we would call it a float). And indeed, you see that the following lines contain strings (the words which are the nodes) followed by the size of the node (the frequency of the words). In line 12, it is defined that the following lines contain the edge definitions: two strings (node1 and node 2, followed by the edge weight.

For example, we see that the words “coffee” occurs three times in total (line 2), and one time together with the word “beer” (line 13).

```
1  nodedef>name VARCHAR, width DOUBLE
2  coffee,3
3  beer,2
4  i,4
5  and,1
6  with,1
7  friend,1
8  having,1
9  like,3
10 am,1
11 my,1
12 edgedef>node1 VARCHAR,node2 VARCHAR, weight DOUBLE
13 coffee,beer,1
14 i,beer,2
15 and,beer,1
16 with,friend,1
17 coffee,with,1
18 i,and,1
19 having,friend,1
20 like,beer,2
21 am,friend,1
22 i,am,1
23 i,coffee,3
24 i,with,1
25 am,having,1
26 i,having,1
27 coffee,and,1
28 like,coffee,2
29 am,coffee,1
30 with,my,1
31 i,friend,1
32 like,and,1
33 am,with,1
34 having,with,1
35 i,my,1
36 having,coffee,1
37 i,like,3
38 coffee,friend,1
39 having,my,1
40 am,my,1
41 coffee,my,1
42 my,friend,1
```

As the format is so simple, we can write a program that produces such a file. In fact, we have already done so for the first part: it is in fact nothing else than a CSV file with frequency counts.

For the second part, we need a module that tells us wich words co-occur together. In other words, we want to know all *combinations* of words in a given string. Not very surprisingly, this method is called `combinations`. See the example below:

```
1 from itertools import combinations
2 words="Let's combine stuff!".split()
3 print ([e for e in combinations(words,2)])
```

produces:

```
1 [("Let's", 'combine'), ("Let's", 'stuff!'), ('combine', 'stuff!')]
```

We already see that it is probably wise to do some preprocessing (to remove the punctuation for example, but you can also think of stopword removal, stemming, or converting to lowercase. We will leave that for know, you already know how to do this and can add that yourself.

There is one thing we still have to consider: We have an undirected network and do not want to distinguish between `('combine', 'stuff!')` and `('stuff!', 'combine')`. We can construct the following minimal example, where we print a list of edges, based on co-occurrances of words in a set of three tweets. Do you see how we use lines 8 and 9 to turn around `('stuff!', 'combine')` if we already have `('combine', 'stuff!')` in our dict of edges that we construct? Also look at line 10, where we specify that a combination is only added to the dict (in line 11) if both nodes are not identical.

```
1  from collections import defaultdict
2  from itertools import combinations
3  tweets=["i am having coffee with my friend","i like coffee","i like coffee
       and beer","beer i like"]
4  cooc=defaultdict(int)
5  for tweet in tweets:
6      words=tweet.split()
7      for a,b in set(combinations(words,2)):
8          if (b,a) in cooc:
9              a,b = b,a
10         if a!=b:
11             cooc[(a,b)]+=1
12 for combi in sorted(cooc,key=cooc.get,reverse=True):
13     print (cooc[combi],"\t",combi)
```

This gives you:

```
1 3       ('i', 'coffee')
2 3       ('i', 'like')
3 2       ('like', 'coffee')
4 2       ('i', 'beer')
5 2       ('like', 'beer')
6 1       ('i', 'am')
7 1       ('am', 'friend')
```

You know have all building blocks you need to write a program that produces a GDF file. You would need to add

- something that reads your input from a file (e.g., a CSV file of tweets or a JSON file of speeches or a set of individual TXT files with newspaper articles)
- some preprocessing
- maybe some filter to filter out nodes and edges that occur very few times
- something to write the output to a file rather than printing to the screen.

The GDF file your program produces can be opened in Gephi (or some other network visualization or analysis tool). I made a screencast where I quickly show how to use Gephi. You can watch it here: `https://streamingmedia.uva.nl/asset/detail/t2KWKVZtQWZIe2Cj8qXcW5KF`.

Rather than producing a GDF file and opening it in Gephi, you can also do network analysis in Python itself, for example with the module `networkx`. This can be very interesting, especially for really large networks (which you maybe cannot open with a graphical tool like Gephi).

Further reading: In fact, you can do a lot of great network analysis stuff in Python itself, for instance using Python modules such as `networkx` or `igraph`. There are many tutorials available, for instance here: `https://github.com/vtraag/4TU-CSS/blob/master/presentations/traag/notebook/Network.ipynb`

9.2 Recap

This chapter taught you how to deal with co-occurrences and similar structures from a network point of view and how to visualize them using Gephi. More in general, you also should have seen how one can write own programs to create specific file formats, even if they are not natively supported.

Chapter 10

Supervised machine learning

The vastness of the topic makes it impossible to cover it extensively in this course. Nevertheless, as last part, we will take a very small and superficial glimpse in the world of supervised machine learning. The example illustrates the general principle — and should give you a starting point for your own investigations. We will use scikit-learn (Pedregosa et al., 2011), a packages that is very widely used and for which you will find a lot of tutorials and usage examples on the web.

For this demonstration, we use a dataset from the Internet Movie Database (Maas et al., 2011). It contains in total 50000 reviews, separated into a test dataset and a training datasset (25000 each), half of them positive, half of them negative.

Download and unpack the dataset:

```
cd /home/damian
wget http://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz
tar -xzf aclImdb_v1.tar.gz
rm aclImdb_v1.tar.gz
```

You now have a folder `aclImdb` in your home directory. Take a look at the structure of the dataset, maybe read the documentation that is included.

You should probably see rather quickly that there are two folders with an identical structure, `train` and `test`. Within each of them, there are two folders `pos` and `neg`, with a lot of plain text files within them. Have a look at them.

First of all, we need the training dataset and the test dataset. Lets conceptualize them as a list of tuples where all positive reviews get a 1, and all negative ones a -1:

```
reviews=[("This is a great movie",1),("Bad movie",-1), ... ...]
test=[("Not that good",-1),("Nice film",1), ... ...]
```

Of course, we do not want to insert our 50000 reviews by hand, but read them from the files we downloaded. We did something similar already with for example the LexisNexis articles or the lists of positive or negative words. Let's do it using the package `glob` that makes our live easier here:

```
from glob import glob

reviews=[]

for file in glob ("/home/damian/aclImdb/train/pos/*.txt"):
    with open(file) as fi:
        reviews.append((fi.read(),1))

for file in glob ("/home/damian/aclImdb/train/neg/*.txt"):
    with open(file) as fi:
        reviews.append((fi.read(),-1))
```

After that, we do exactly the same for the test dataset. You only have to replace `reviews` with `test` in the code above, and `train` with `test`.

Once we have done this, and thus have two lists, `reviews` and `test`, each of them with a length of 25000, we can train and test the classifier — after importing the necessary modules from the package scikit-learn (Pedregosa et al., 2011), of course:

```
from sklearn.naive_bayes import MultinomialNB
from sklearn.feature_extraction.text import CountVectorizer
from sklearn import metrics
```

We use a Bag-of-Word representation of all reviews rather than the whole reviews. We only want to know the frequency of each word in each review. While we could calculate this ourselves, scikit-learn integrates this nicely in the workflow (they even remove stopwords for us):

```
vectorizer = CountVectorizer(stop_words='english')
train_features = vectorizer.fit_transform([r[0] for r in reviews])
test_features = vectorizer.transform([r[0] for r in test])
```


Note that `[r[0] for r in reviews]` is a short form of writing something like the following (it is called "list comprehension", see Appendix D.1):

```
newlist=[]
for r in reviews:
    newlist.append[r[0]]
```

It gives us a list of the reviews themselves, without the scores – and `r[1]` would give us only the scores.

Training the machine learning algorithm (we chose the Multinomial Naïve Bayes variant here, there are others as well – read the docs and check out which one is best!) takes only two lines:

```
# Fit a naive bayes model to the training data.
nb = MultinomialNB()
nb.fit(train_features, [r[1] for r in reviews])
```

Let's now use the classifier we just trained to predict the classification of our test dataset:

```
predictions = nb.predict(test_features)
actual=[r[1] for r in test]
```

(of course, we already know the actual, real classes, we just put them in a list in the second line for easier comparison)

We could compare the two lists `predictions` and `actual` by hand to find out in where our classifier is right or wrong. But we can also just calculate the AUC, a measure of accuracy:

```
print(metrics.accuracy_score(actual,predictions,normalize=True))
```

We can now play around and see how the classifier classifiers some new data:

```
newreviews=["What a crappy movie! It sucks!",
         "This is awsome. I liked this movie a lot, fantastic actors",
         "I would not recomment it to anyone.",
         "Enjoyed it a lot"]

new_features=vectorizer.transform(newreviews)
predictions = nb.predict(new_features)
print(predictions)
```

10.1 Comparing different classifiers and vectorizers

Usually, when we de supervised machine larning, we want to compare different classifiers. For example, in the last section, we used a Naïve Bayes classifier. But we could use a logistic regression as well, or a so-called support vector machine. It is common to run several of these models and then compare their *precision* and *recall*.

Let us assume that the goal of training above-mentioned classifier is to build an app that shows the user only films we can expect to receive positive ratings. There are two things that we want to achieve: We want to find as many as possible positive films (recall), but we also want that the selection we found *only* contains positive films (precision).

Precision is calculated as $\frac{TP}{TP+FP}$, where TP are true postivies and FP are false positives. For example, if our classifier retrieves 200 articles that

it classifies as positive films, but only 150 of them indeed are positive films, then the precision is $\frac{150}{150+50} = \frac{150}{200} = 0.75$.

Recall is calculated as $\frac{TP}{TP+FN}$, where TP are true postivies and FN are false negatives. If we know that the classifier from the previous paragraph missed 20 positive films, then the recall is $\frac{150}{150+20} = \frac{150}{170} = 0.88$.

In other words: Recall measures how many of the cases we wanted to find we actually found. Precision measures how much of what we have found actually is correct.

Often, we have to make a trade-off between precision and recall. For example, just retrieving *every* film would give us a recall of 1.0 (after all, we didn't miss a single positive film). But on the other hand, we retrieved all the negative films as well, so precision will be extremely low. It can depend on the task at hand whether precision or recall is more important.

Another thing we can vary in our classifier is the vectorizer we used. In our example, we used a count vectorizer, which simply means that our *features* (which is just a fancy word for independent variables) are simply the frequency counts of the words. This approach has the disadvantage that some words will be very frequent in almost all articles, while others occur in only a few articles. Arguably, such words are more informative and therefore, their occurrence should weigh more heavily. The tf·idf scheme does exactly that: it weighs the term frequency (TF) by the inverse document frequency (IDF), i.e. the number of texts ("documents") in which the word ("term") occurs at least once.

Scikit-learn allows us to use tf·idf scores instead of simple counts by using another vectorizer[1]:

```
1 from sklearn.feature_extraction.text import CountVectorizer,
      TfidfVectorizer
2 # instead of:
3 myfirstvectorizer=CountVectorizer(stop_words='english')
4 # we could do:
5 mysecondvectorizer=TfidfVectorizer(stop_words='english')
```

Below, you find an example script that combines everything that we discussed in these sections. There are more elegant ways of doing this, for example by writing functions instead of repeating the code that calculates the evaluations, but for illustration purposes, I decided to leave it as it is:

```
1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3 from glob import glob
4 from sklearn.naive_bayes import MultinomialNB
```

[1]For details, see `http://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html`

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn import metrics

reviews=[]
test=[]

print("Constructing training dataset")

# glob gives you a list of filenames that match a specific criterion
# in this case, all .txt-files in the postivie training folder

for file in glob ("/home/damian/aclImdb/train/pos/*.txt"):
    with open(file) as fi:
        reviews.append((fi.read(),"1"))
nopostr=len(reviews)

print ("Added",nopostr,"positive reviews")

for file in glob ("/home/damian/aclImdb/train/neg/*.txt"):
    with open(file) as fi:
        reviews.append((fi.read(),"-1"))
nonegtr=len(reviews)-nopostr
print ("Added",nonegtr,"negative reviews")

print("Constructing test dataset")

for file in glob ("/home/damian/aclImdb/test/pos/*.txt"):
    with open(file) as fi:
        test.append((fi.read(),"1"))
noposte=len(test)
print ("Added",noposte,"positive reviews")

for file in glob ("/home/damian/aclImdb/test/neg/*.txt"):
    with open(file) as fi:
        test.append((fi.read(),"-1"))
nonegte=len(test)-noposte
print ("Added",nonegte,"negative reviews")

#%% Checking the data
# Thus, we got two lists, reviews and test.
# Both contain tuples (pairs of two values: The first is the review,
# the second the classification: 1 or -1)

# We can easiliy verify this by looking at a random entry:
print(reviews[244])
# or
print('The following review\n\n\n',reviews[244][0],"\n\n\nis evaluated as
    ",reviews[244][1])

```

```
53 #%% Training the classifier
54
55 print("Training classifier...")
56
57 # Generate BOW representation of word counts
58 vectorizer = CountVectorizer(stop_words='english')
59 #alternatively, you could provide a list of stop words yourself
60 train_features = vectorizer.fit_transform([r[0] for r in reviews])
61 test_features = vectorizer.transform([r[0] for r in test])
62
63 # Fit a naive bayes model to the training data.
64 nb = MultinomialNB()
65 nb.fit(train_features, [r[1] for r in reviews])
66
67 #%% testing the classifier
68 # Now we can use the model to predict classifications for our test
       features.
69 predictions = nb.predict(test_features)
70 # and also put the 'true' results from the test dataset in a list
71 actual=[r[1] for r in test]
72
73 # now we can compare whether the predicted values and the actual values
       match.
74 # We could write the output to a tab seperated file to see what matches:
75 with open("/home/damian/agreement.tab", mode='w') as fo:
76     fo.write("actual\tpredicted\tfirst words\n")
77     for i in range(len(predictions)):
78         fo.write(str(actual[i])+"\t"+str(predictions[i])+"\t"+test[i
              ][0][:50]+"\n")
79
80 # That can be helpful for inspecting where sth goes wrong, but we can
81 # also calculate some measures of performance immediately:
82
83 print('Accuracy:')
84 print(metrics.accuracy_score(actual,predictions,normalize=True))
85
86 print('Precision:')
87 print(metrics.precision_score(actual,predictions,pos_label='1', labels =
       ['-1','1']))
88 print('Recall:')
89 print(metrics.recall_score(actual,predictions,pos_label='1', labels =
       ['-1','1']))
90
91 # Note that Precision is not a symmetric measure.
92 # If we want the precision for retrieving a NEGATIVE review # instead of a
       positive we get a different value:
93
94 print('\t side note: Precision if we are interested in retrieving negative
       reviews:')
```

```
print('\t',metrics.precision_score(actual,predictions,pos_label='-1',
    labels = ['-1','1']))
print('\t side note: Recall if we are interested in retrieving negative
    reviews:')
print('\t',metrics.recall_score(actual,predictions,pos_label='-1', labels
    = ['-1','1']))

# back to normal

print('F1-score:')
print(metrics.f1_score(actual,predictions,pos_label='1', labels =
    ['-1','1']))
print('Confusion matrix:')
print(metrics.confusion_matrix(actual,predictions))


#%% Now, let's play around and see how well it would work on
# new unseen data:
newreviews=["What a crappy movie! It sucks!",
           "This is awsome. I liked this movie a lot, fantastic actors",
           "I would not recomment it to anyone.",
           "Enjoyed it a lot"]
newdata=vectorizer.transform(newreviews)
predictions = nb.predict(newdata)
print(predictions)
for i in range(len(predictions)):
    if predictions[i]==1:
        print(newreviews[i],'\nis probably about a GOOD movie\n')
    elif predictions[i]==-1:
        print(newreviews[i],'\nis probably about a BAD movie\n')
```

We used a Naïve Bayes classifier, but we could as well use a different one, for example a logistic regression[2]

You can run a logistic regression classifier just as you would run a NB classifier[3]:

```
from sklearn.linear_model import LogisticRegression
logreg = LogisticRegression()
logreg.fit(train_features, [r[1] for r in reviews])
predictions = logreg.predict(test_features)
```

[2]You can find one explanation of the difference here: https://www.quora.com/What-is-the-difference-between-logistic-regression-and-Naive-Bayes?share=1. Basically, in contrast to a logistic regression, in a Naïve Bayes classifier, all features are assumed to be uncorrelated.

[3]We could get more info, see http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html. For example, if you are interested in coefficients:

```
predictionsproba = logreg.predict_proba(test_features)
print([j for i in logreg.coef_ for j in i])
```

A last one would be a support vector macine (SVM) (for more info, see `http://scikit-learn.org/stable/modules/svm.html`)

```
from sklearn import svm
mysvm = svm.SVC()
mysvm.fit(train_features, [r[1] for r in reviews])
predictions = mysvm.predict(test_features)
```

Note that the names I assigned to each classifier (`nb`, `logreg`, `mysvm`) are completely arbitrary.

10.2 Saving the trained model

In theory, you could just train the model everytime you want to use it. However, that does not only take unnecessary time and ressources, but it also requires you to have the training data at hand. Therefore, it can be useful to save your vectorizers and classifiers. It is important to realize that you need to save both: After all, the vectorizer determines which word is mapped to which internal numeric representation, that is used by the classifier.

Once you have fitted both vectorizer and classifier, you can save them as follows (assuming you have called your instance of the vectorizer `vectorizer` and your instance of the classifier `nb`):

```
import pickle
from sklearn.externals import joblib

pickle.dump(vectorizer,open("myvectorizer.pkl",mode='wb'))
joblib.dump(logreg, 'myclassifier.pkl')
```

Then, later on, instead of fitting a new vectorizer, you can simply load the old one and use it

```
import pickle
vectorizer = pickle.load(open("myvectorizer.pkl",mode='rb'))
new_features = vectorizer.transform([listwithnewdata])
```

As you see, you do not do any `.fit_transform` any more, because the vectorizer is already fitted (that was why we saved it, after all).

Also the classifier can be loaded again very easily and be used immediately for prediction

```
from sklearn.externals import joblib
nb = joblib.load('myclassifier.pkl')
predictions = nb.predict(new_features)
```

Further reading: You'll easily find more examples and other methods on `http://scikit-learn.org`, but also in a lot of other tutorials on the internet, for instance `https://www.kaggle.com/abhishek/approaching-almost-any-nlp-problem-on-kaggle`

Chapter 11

Unsupervised machine learning

In the previous chapter, we were had *labeled* data, i.e. we had an outcome which we could predict. In other words, we applied supervised machine learning. But sometimes, we do not have an outcome to predict.For instance, we want to group the films into topics, but we have no idea which topics exist to begin with. The sci-kit learn package (Pedregosa et al., 2011) offers a lot of unsupervised methods, like principal component analysis or cluster analysis. Have a look at their website for usage examples.

In this chapter, however, we will focus on one specific form of unsupervised machine learning: topic modeling. The basic idea is to find some latent constructs (topics), that are present in a set of documents, so that one gets both a list of topics and a dataset indicating the extend o which these topics are present in each of the documents.

While there are several methods, we will focus here on a very popular one: Latent Dirichlet Allocation.

11.1 Latent Dirichlet Allocation (LDA)

Topic Modelling therefore is a form of *unsupervised* machine learning. To get to know it, we will use the gensim package (Řehůřek & Sojka, 2010).

Furthermore, let us assume you have a list of lists of words (!) called `texts`:

```
articles=['The tax deficit is higher than expected. This said xxx ...', '
    Germany won the World Cup. After a']
texts=[art.split() for art in articles]
```

which looks like this:

```
[['The', 'tax', 'deficit', 'is', 'higher', 'than', 'expected.', 'This', '
    said', 'xxx', '...'], ['Germany', 'won', 'the', 'World', 'Cup.', '
```

```
    After', 'a']]
```

I'll take the movie reviews from the last section as a test case (without the ratings, which are irrelevant now), but of course you can use any collection of text files.

```
from glob import glob
texts=[]
for file in glob ("/home/damian/aclImdb/train/pos/*.txt"):
    with open(file) as fi:
        texts.append(fi.read().split())
```

Pay attention to the `.split()` method, which makes that we now have a list of lists of words, just as we wanted.

We now let gensim create a BOW representation of the texts — just as in earlier examples, but each packages has a slightly different syntax for that. Luckily, you don't have to remember that, that's where the documentation and example scripts that come with each package are for.

```
from gensim import corpora, models
# Create a BOW represenation of the texts
id2word = corpora.Dictionary(texts)
mm =[id2word.doc2bow(text) for text in texts]
```

We can now train the LDA models:

```
lda = models.ldamodel.LdaModel(corpus=mm, id2word=id2word, num_topics=100,
     alpha="auto")
```

Of course, you can specify any other number of topics. Most people use something like 50 or 100 topics, though. Let's print the most characteristic words for each topic:

```
for top in lda.print_topics(num_topics=NTOPICS, num_words=5):
    print ("\n",top)
```

Only one thing left to do: Calculate the scores for each topic per document and save them to a file. I chose a tab-separated one:

```
scoresperdoc=lda.inference(mm)
with open("topicscores.tsv","w",encoding="utf-8") as fo:
    for row in scoresperdoc[0]:
        fo.write("\t".join(["{:0.3f}".format(score) for score in row]))
        fo.write("\n")
```

This is the general principle, but of course, there are a lot of ways of tuning it. To start with, some stopword removal would be advisable - and we did not even remove punctuation or all the weird HTML tags, or bother about converting it to lower case. However, you already learned all this and should be able to write a good script by integrating the structure from above with other techniques.

For some suggestions, see the documentation of gensim.

One particular thing you might want to try, though: Just as in the case of supervised machine learning, also for unsupervised machine learning, you might want to try to use a tf·idf vectorizer instead of the default count vectorize. In gensim, you can do so by slightly modifying the script presented above:

```
from gensim import corpora, models
# Create a BOW represenation of the texts
id2word = corpora.Dictionary(texts)
mm =[id2word.doc2bow(text) for text in texts]

# create a tf-idf representation
tfidf = models.TfidfModel(mm)

# use that tfidf-representation instead of the pure counts
lda = models.ldamodel.LdaModel(corpus=tfidf[mm], id2word=id2word,
    num_topics=100, alpha="auto")
```


There is a more detailed tutorial available online at `https://github.com/damian0604/bdaca/blob/master/ipynb/ic2s2.ipynb`

There is also a nice tool called `pyLDAvis` that – when run inside a Jupyter Notebook – allows you to interactively explore a topic model. Rather than printing topics and words, you can hover with your mouse over them and even get the topics projected in a two-dimensional space to see how big and how similar they are (see Figure 11.1).

For our above model, we could invoke `pyLDAvis` as follows:

```
import pyLDAvis
import pyLDAvis.gensim
vis_data = pyLDAvis.gensim.prepare(lda,mm,id2word)
pyLDAvis.display(vis_data)
```

11.2 Recap

You should be able to explain what supervised and unsupervised machine learning are and give some examples. More practically, you should be able to use the following widely used packages to conduct such analysis:

- scikit-learn
- gensim

In particular, you should be able to not only train models and apply them to new, unseen data, but you should also know how to do a basic evaluation of the performance of the model.

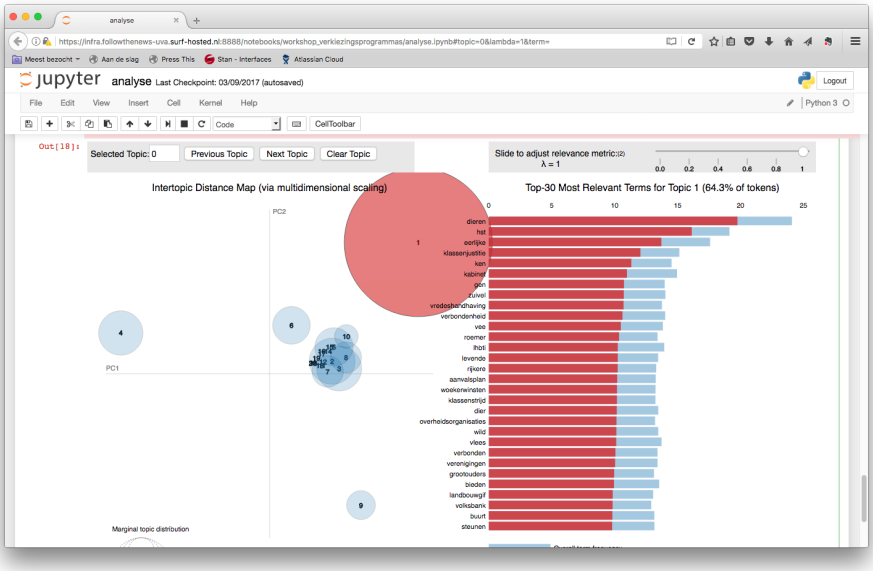


Figure 11.1: `pyldavis` allows you to interacitvely explore a topic model in your browser.

Chapter 12

Statistics with Python

One of the things you have learned so far is how to output data in a format that suits your needs. For example, no matter what your input data are, you can always save the results of your program in a CSV file that you can open with R, Stata or SPSS for further analysis. And indeed, it can make sense to use different tools in different stages of a project.

However, if you only want to calculate a mean and a standard deviation or even conduct a regression, this is actually not neccessary – and, let's face it, is pretty cool if your program just does *everything* in a single run. In addition, Python is actually pretty good in doing statistics and used by many data scientists for this purpose. In this chapter, I'll briefly mention some modules that are useful for this, so that you know where to look further.

12.1 numpy & scipy

Numpy (Van der Walt, Colbert, & Varoquaux, 2011) is a package that provides a lot of mathematical functions. It can, for example, be used to calculate means (ok, that's boring, you could do that yourself), standard deviations, correlations, and much, much more. It also offers some specific data types that are more efficient for some calculations than the build-in lists. It goes a long with the similar package SciPy (Jones, Oliphant, Peterson, et al., 2001).

```
1 import numpy as np
2 >>> x = [1,2,3,4,3,2]
3 >>> y = [2,2,4,3,4,2]
4 >>> np.mean(x)
5 2.5
6 >>> np.std(x)
7 0.9574271077563381
```

```
>>> np.corrcoef(x,y)
array([[ 1.        ,  0.67883359],
       [ 0.67883359, 1.        ]])

>>> from scipy import stats
>>> stats.skew(x)
0.0
>>> stats.kurtosis(x)
-0.942148760330578
```

12.2 matplotlib

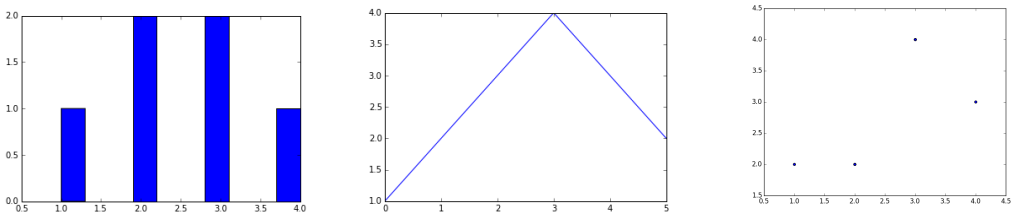


Figure 12.1: Examples of plots generated with `matplotlib`

While there are tools that make fancier graphics (take a look at the Python module `seaborn`; or use `ggplot2` in R), `matplotlib` (Hunter, 2007) is really useful if you just want to make a quick histogram, line graph, or scatter plot (Fig. 12.1).

```
import matplotlib.pyplot as plt
x = [1,2,3,4,3,2]
y = [2,2,4,3,4,2]
plt.hist(x)
plt.plot(x,y)
plt.scatter(x,y)
```

12.3 pandas & statsmodels

Pandas (McKinney et al., 2010) is a framework that allows statistical modelling in Python. Those of you who know R will see a lot of similarities. Within pandas, data are stored in a table, very much like a Stata or SPSS dataset – and yes, it is indeed called exactly as it is called in R: a *dataframe*. As you see below, a dataframe basically consists of different columns with a

name, which consist of a list of data. Pandas can read directly from a CSV file, but as you can see below, you can also construct a dataframe yourself.

While pandas has a lot of nice functions to explore and manage datasets, it is – for our purposes – especially powerful when combined with statsmodels (Seabold & Perktold, 2010). Statsmodels allows provides a wide range of models, just like what you would expect from SPSS, Stata, or R.

To run an OLS regression, for example, you only have to specify which columns are the dependent and the independent variables.

```
import pandas as pd
import statsmodels.formula.api as smf
df = pd.DataFrame({"income": [10,20,30,40,50], "age": [20, 30, 10, 40,
    50], "facebooklikes": [32, 234, 23, 23, 42523]}
# alternative: read from CSV file:
# df = pd.read_csv('mydata.csv')

myfittedregression = smf.ols(formula='income ~ age + facebooklikes', data=
    df).fit()
print(myfittedregression.summary())
```

prints a regression table like you would expect from any statistics program:

```
OLS Regression Results
==============================================================================
Dep. Variable:                 income  R-squared:                       0.579
Model:                            OLS  Adj. R-squared:                  0.158
Method:                 Least Squares  F-statistic:                     1.375
Date:                Mon, 31 Oct 2016  Prob (F-statistic):              0.421
Time:                        18:11:40  Log-Likelihood:                -18.178
No. Observations:                   5  AIC:                             42.36
Df Residuals:                       2  BIC:                             41.19
Df Model:                           2
Covariance Type:            nonrobust
==============================================================================
coef    std err          t      P>|t|      [95.0% Conf. Int.]
------------------------------------------------------------------------------
Intercept      14.9525     17.764      0.842      0.489     -61.481    91.386
age             0.4012      0.650      0.617      0.600      -2.394     3.197
facebooklikes   0.0004      0.001      0.650      0.583      -0.002     0.003
==============================================================================
Omnibus:                        nan  Durbin-Watson:                   1.061
Prob(Omnibus):                  nan  Jarque-Bera (JB):                0.498
Skew:                        -0.123  Prob(JB):                        0.780
Kurtosis:                     1.474  Cond. No.                     5.21e+04
==============================================================================
```

Pandas is a *huge* framework, especially in combination with Jupyter Notebook (Section 3.5). It is extremely popular in the world of data science, but also in areas like finance.

However, it would be kind of useless to cover it extensively in this intro, as others have already done so. You can have a look at the books by McKinney (2012) and Russel (2013).

You can also find some iPython notebooks with typical analyses with pandas, ranging from correlations and t-tests to regression models and time-series analysis, here: `https://github.com/damian0604/bdaca`.

12.4 Recap

Things you should remember:

- There are multiple packages for doing statistical analysis like you know them from programs like SPSS, Stata, or R.
- pandas offers you R-like data structures.
- There are online ressources with a lot of examples.

In particular, once you arrived at this part of this book, you can run *your whole workflow* in Python — from collecting the data until the final analysis. This obviously has major advantages compared to switching between environments.

Chapter 13

Further reading

The following books provide the interested student with more and deeper information. They are intended for the advanced reader and might be useful for your individual projects (or, maybe, a thesis):

- Russel, 2013. Gives a lot of examples about how to analyze a variety of online data, including Facebook and Twitter, but going much beyond that. Because social media APIs have undergone multiple changes in the last years, the examples might be slightly outdated. A PDF of the book can be downloaded for free on `http://www.webpages.uidaho.edu/%7Estevel/504/Mining-the-Social-Web-2nd-Edition.pdf`
- Bird et al., 2009. This is the official documentation of the NLTK package that we are using. A newer version of the book can be read for free at `http://nltk.org`
- McKinney, 2012: Another book with a lot of examples. A PDF of the book can be downloaded for free on `http://it-ebooks.info/book/1041/`.
- VanderPlas, 2016: More on the numeric and data handling side. While the book itself is not freely available, it is accompagnied by a set of interesting Jupter Notebooks that show a lot about data handling, visualization, and machine learning: `https://github.com/jakevdp/PythonDataScienceHandbook`

In the last years, some other tutorials, partly similar to this one, have been published. See for exampe:

- Jürgens & Jungherr, 2016. Provides examples and code for non-pogrammers to analyze Twitter data using Python and R.

Part III

Appendices

Appendix A

Exercise 1: Describing an existing structured dataset

A.1 Downloading the data

In this exercise, you will do some basic descriptive analyses of an existing dataset. Mazières, Trachman, Cointet, Coulmont, and Prieur (2014) scraped data from online porn sites, resulting in two datasets with metadata about almost two million amateur porn videos. We will work with one of these datasets, consisting of all metadata on all videos posted on `http://xhamster.com` from its creation in 2007 until February 2013. The authors made the dataset available on on `http://sexualitics.github.io`, and you can find a description of it in Figure A.1.

We do the exercise together in class, but make sure you have downloaded the dataset before class. You can do so as follows (but, of course, replace "damian" with your own user name):

```
cd /home/damian
mkdir pornexercise
cd pornexercise
wget pornstudies.sexualitics.org/data/xhamster.json.tar.gz
tar -xzf xhamster.json.tar.gz
```

The `wget` command downloads the dataset. It is compressed, so we have to uncompress it, which is done by the `tar` command (most of you probably are used to having `.zip` files for compressed or archived data, which is essentially the same; `.tar.gz` is more common among the nerdier part of the population). Lets check if everything went right:

```
ls -lh
```

should give you an output like this:

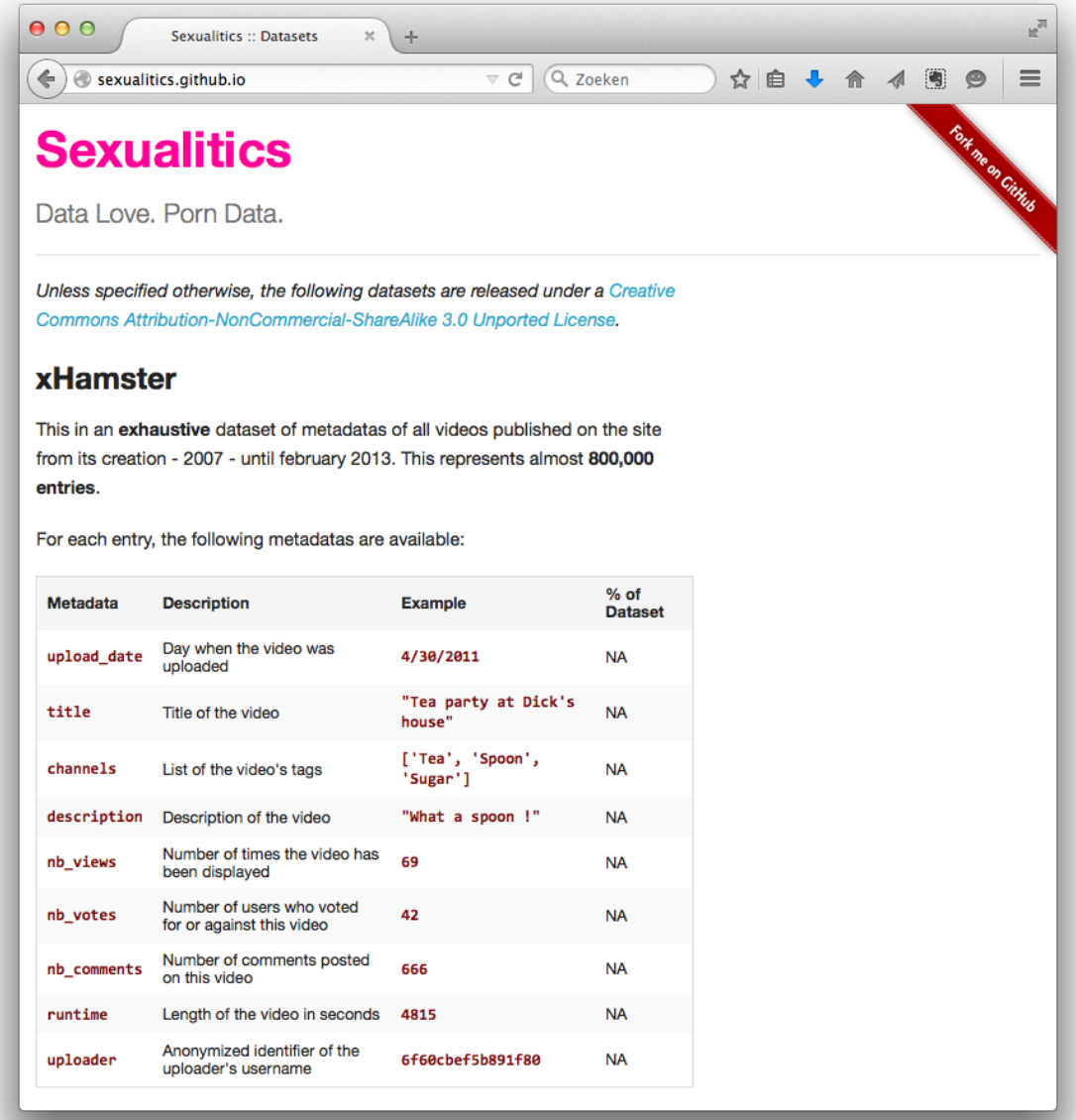


Figure A.1: The discription of the dataset.

```
damian@damian-VirtualBox:~/pornexercise\$ ls -lh
total 284M
-rw-r--r-- 1 damian damian 229M feb 8 2014 xhamster.json
-rw-rw-r-- 1 damian damian 55M feb 8 2014 xhamster.json.tar.gz
```

You see that the compressed file is 55MB large, but the uncompressed one

is more than four times as large. Let's delete the compressed one, we don't need it any more:

```
rm xhamster.json.tar.gz
```

A.2 The tasks

Start with having a look at Figure A.1. It is important to understand the structure of the data: Which fields are there, how are they named, and what do they contain? For example, we see that the field "channels" contains a *list* of different tags, while "nb_votes" seems to contain an *integer*. Ready to go? Let's do some work:

1. Print the title of each video.
2. (a) What are the 100 most frequently used tags?
   (b) What is the average number of tags per video?
3. What tags generate the most comments/votes/views?
4. What is the average length of a video description?
5. What are the most frequently used words in the descriptions?

A.3 Hints

The following hints will help you solving the exercise.

Task 1

You haven't learned yet how to handle JSON-files. This is explained in chapter 4 in detail. For now, you only have to know that you can read the data into a Python dict with the following command:

```
import json
with open ('/home/damian/WHATEVERTHEFILEISCALLED.json') as f:
    data=json.load(f)
```

You now have a dict called data. This dict consists of other dicts, each of which contains information on one video clip. We can loop over the keys and values of the outer dict to get the data we want:

```
for key, value in data.items():
  print(value["title"])
```

You could print the key as well (try it!), if you are interested in the number of the clip to which the title belongs.

You know from Figure A.1 that the key we are interested in is called "title".

That's it!

Task 2

There is a module called Counter that allows you to count elements in a list.

Consister this example to print the two most frequent words in a list:

```
from collections import Counter

l = ['hi','hi','hi','bye','bye','hi','doei','hoi']
c = Counter(l)
print(c.most_common(2))
```

This gives as a result:

```
[('hi', 4), ('bye', 2)]
```

To solve this task, you probably want to create an empty list for the tags first, and then loop over your data to fill the list. After you have this list, you can use a counter to determine the frequencies.

If you do this on *really* big datasets, there is a replacement for Counter called bounter, see `https://github.com/RaRe-Technologies/bounter`

Task 3

You might want to construct a dict with the categories as keys and an int containing the number of comments. Then you loop over your dataset and each time you encounter the same category, you add the number of comments to the dictionary entry. However, the first time you encounter a category, it does not have a key yet. We can use a so-called defaultdict, that automatically creates a key if it isn't there yet.

```
from collections import defaultdict
commentspercat=defaultdict(int)
```

And then, loop over the dataset and add `int(value["nb_comments"])` to the dict.

Not all items might have comments and have a value of 'NA' instead. Thus, referring to `int(data[item]["nb_comments"])` might fail. You can specify what should happen in such cases by using a try-except construction:

```
try:
    commentspercat[category]+=int(value['nb_comments'])
except:
    pass
```

This just means that if the line fails, it just continues without any error message.

Task 4

This is basically the same what you've already done...

Task 5

You can transform a string to a list of words by using the split method.

```
"this is a test".split()
```

results in

```
["this", "is", "a", "test"]
```

(useful for applying a Counter...)

On the next page, you will find the solution.

A.4 Solution

Try first to do it yourself, but if you cannot find out how to do it, you can have a look at the example solution below.

```
import json
from collections import Counter

#Right path?
with open("/home/damian/pornexercise/xhamster.json") as fi:
    data=json.load(fi)


#%% task 1: print titles
for k,v in data.items():
    print (v["title"])




#%% task 2a en 2b: average tags per video and most frequently used tags

alltags=[]
i=0
for identifyer,clip in data.items():
    i+=1
    alltags.extend(clip["channels"])

print(len(alltags),"tags have been used to describe the",i,"different
    videos")
print("Thus, we have an average of",len(alltags)/i,"tags per video")
c=Counter(alltags)
print (c.most_common(100))




#%% task 3: What porn category is most frequently commented on?
from collections import defaultdict
commentspercat=defaultdict(int)
for identifier,clip in data.items():
    for tag in clip["channels"]:
        try:
            commentspercat[tag]+=int(clip["nb_comments"])
        except:
            pass

print(commentspercat)

# or nicer:
for tag in sorted(commentspercat, key=commentspercat.get, reverse=True):
```

```
44      print(tag, commentspercat[tag])
45
46
47
48  #%%task 4: average length of text
49  length=[]
50  for identifier,clip in data.items():
51      length.append(len(clip["description"]))
52
53  print ('Average length',sum(length)/len(length))
54
55
56  #%%task 5: most frequently used words
57
58  # most frequently used words
59  allwords=[]
60  for identifier,clip in data.items():
61      allwords.extend(clip["description"].split())
62  c2=Counter(allwords)
63  print(c2.most_common(100))
```

Appendix B

Exercise 2: Analyzing your Whatsapp-Chats

B.1 Get the data

In this exercise, you will play around with some data from your own Whatsapp chats.

Whatsapp allows you to email a chat to yourself. Do that and unzip the file. It should contain a file called `_chat.txt`. In Python, you can read this file into a list of strings, each string representing one line:

```
1 with open('_chat.txt') as f:
2     data = f.readlines()
```

Print it, calculate its length and so forth to get some idea. As you'll see, each element of the list looks a bit like this:

```
1 05-12-16 18:57:07: Damian: Ben denk ik pas rond 2130 of zo thuis\n
```

Use

- slicing (i.e. `somearbitratystring[:6]` or `somearbitratystring[6:12]`) to retrieve only the first 6 letters, or the next 6, ...
- `.split(":")[0]` or `.split(":")[1]` or ... to split the string into a list of substrings at every `:`-sign and select only the 0^{st}, 1^{st}, ... of these substrings

to select the meaningful information (i.e., seperate time stamps from user names from text). Save this info to new lists (or dicts, if you prefer).

Put all of this in a for-loop to iterate over all lines.

Then, analyze the new lists (e.g., count most frequently used words or the like).

First try it yourself, but if you do not manage to get it work, you'll find a solution on the next page.

B.2 Solution

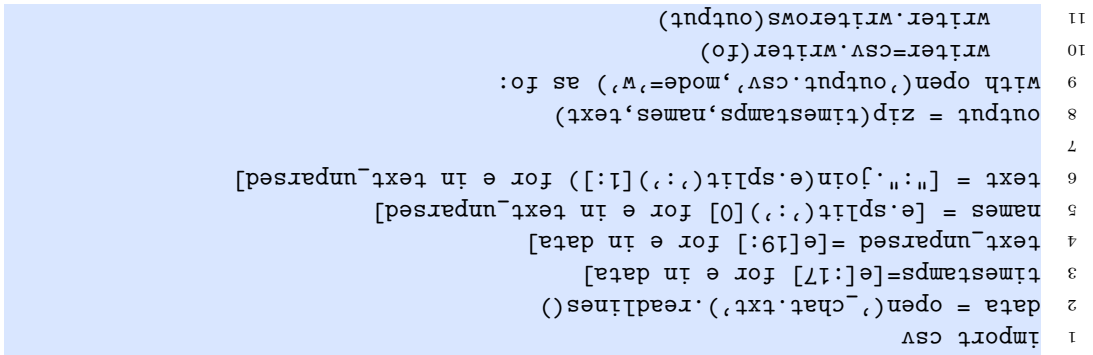
This is one possible solution, using list comprehensions (see Appendix D.1). You can also write more verbose code. For instance, instead of writing

```
timestamps=[e[:17] for e in data]
```

you can also write

```
timestamps = []
for e in data:
    timestamps.append(e[:17])
```

The complete solution (rotated):



Appendix C

Exchanging files by mounting your host system

One of the nice things of using a Virtual Machine is that you cannot break anything on your own computer. The obvious downside is that you also cannot access files on it. Of course, you can always mail stuff to yourself or use some other workaround.

But if you *really* want to break the isolation and have access to folders on your computer, here is how it works.

The process of connecting a device (a USB stick, a harddisk, ...) and assigning it a name is called "mounting". For example, when you insert a USB stick in a Mac or Linux computer, it might be (nowadays, usually automatically) mounted on `/media/mystick` or something similar.

What you want to do, is mounting a directory from your "real" computer, the so-called host, in the file system of the VM. Select the folder you want to share in Virtual Box (see Figure C.1). Select permanent, but NOT auto-mounting (screenshot). Remember the name of the share (in my example, Desktop)

Within the VM, now create a folder where you want the content of your host folder to appear. For example, this could be:

mkdir /home/damian/myrealcomputer

Now, use the following line to mount the share (in my example, it is called Desktop) to that folder:

```
1 sudo mount -t vboxsf -o uid=$UID,gid=$(id -g) Desktop /home/damian/
    myrealcomputer/
```

Voilà!

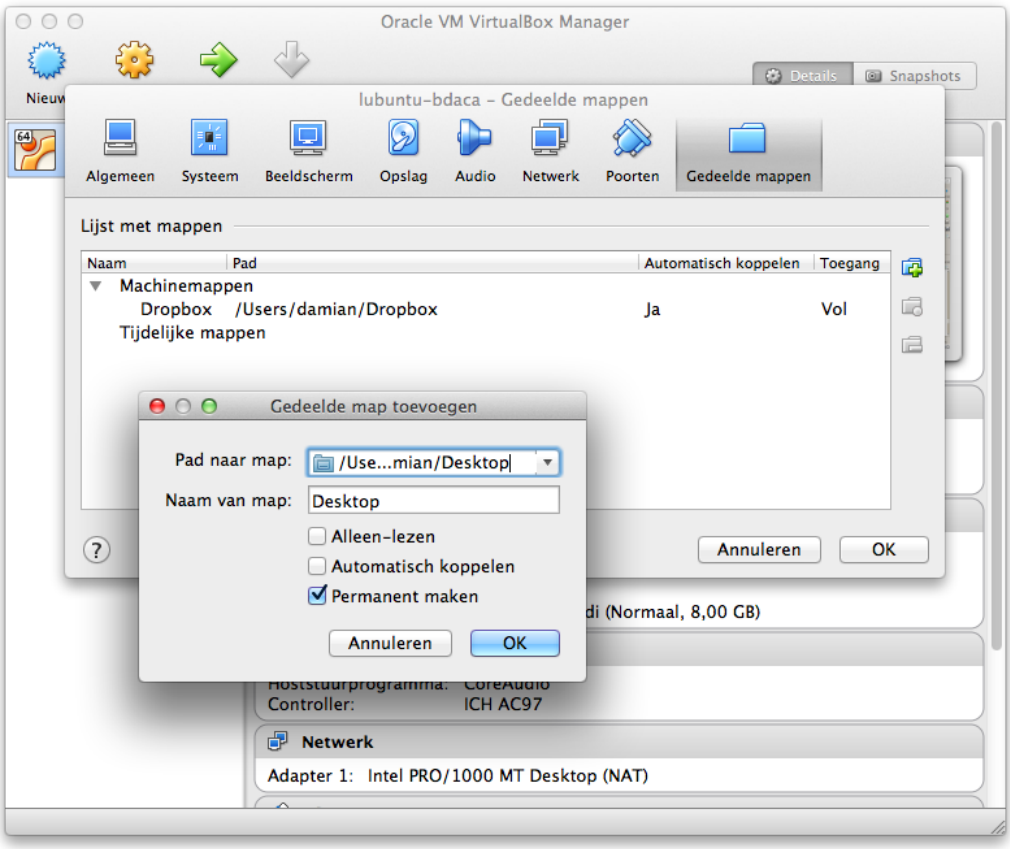


Figure C.1: Configuring VirtualBox to make it possible to mount folders from the host system

Appendix D

Some more Python concepts

This appendix summarizes a couple of cool Python features that are not covered as part of a specific technique this book deals with. They rather will be useful in various circumstances, and if you want to deepen your Python knowledge, it is strongly advised to make use of them.

D.1 List comprehensions

List comprehension offer a a short form of getting information from one list into another. For example, you may want to apply a function to each element of a list, select only specific elements of a list, etc.

You can think of list comprehensions as for-loops written in one single line, without the need to create an empty list and .append() things to it.

Take the following example, in which we want to convert a list of strings to lowercase and strip leading and trailing whitespace:

```
names = [" Joanna", "THEO", " Anne "]
names_cleaned = []
for name in names:
    names_cleaned.append(name.strip().lower())
```

Using list comprehensions, we can write instead:

```
names_cleaned = [name.strip().lower() for name in names]
```

We can even add an if condition to it:

```
names_cleaned_female = [name.strip().lower() for name in names if not name
    .startswith('T')]
```

And if we want to, we do not even need to create a new variable name for the list: we can also use the old one and overwrite it:

```
names = [name.strip().lower() for name in names]
```

The big advantage of list comprehensions is that they are so concise: they just take one single line, and you do not have to deal with indention. This can make your code more readable and is considered good programming style.

D.2 Generators

In Python, we have a lot of so-called *iterables*: things we can loop over. A prominent example are lists.

However, lists have a drawback: They can be big and need to be stored in memory. But often, when looping over a list, we do not actually need the *whole* list – we only need to be able to get the *next* element, and care neither about all preceeding elements nor about the total number of elements.

Here, generators come into play.

Actually, in Section 5.1 you already used one: You learned how to iterate over the lines of a csv file using a csv reader.

This reader was a generator. Rather than reading the whole file, it only reads one line at a time. This means that if we have a table with 10 million rows and 1,000 columns, but we only need one of the columns, while reading, we only read 1,000 values at a time. Therefore, at no point in time, we have the whole $10,000,000 \times 1,000$ table in memory.

We can make our own generator by defining a function, as we already did in Section 4.2.1. The only difference is that instead of `return`, we use `yield`.

```
def firstsquares(n):
    for i in range(n+1):
        yield i**2
```

We can than use this generator:

```
for v in firstsquares(10):
    print(v)
```

The efficieny of this becomes clear when we want to do something like calculating the sum of these numers: We can just calculate `sum(firstsquares(10))` *without actually having to store a list of all these numbers in memory.* After all, if you add up 10 numbers, you only need two numbers at a time: the sum up til now and the next number.

Just like we can have list comprehensions (Section D.1) as a shorter version to create lists, we can also use so-called *generator expressions* to simplify the creation of generators:

```
g = (x*x for x in range(10))
```

To get the next square number, we can just do

```
next(g)
```

However, we could also iterate over it in a loop:

```
g = (x*x for x in range(10))

for square in g:
    print(square)
```

Note that you can only iterate over an iterator once!

D.3 Classes

There is one big (really big) topic that is *not* covered in this book: the concept of a *class*. A class is a sort of blueprint and a core concept of object oriented programming. We actually even worked with it before, without diving into the details of a class. For instance, when we did

```
from collections import Counter
mycounter = Counter()
```

or

```
import csv
with open(test.csv') as fi:
    myreader = csv.reader(fi)
    ...
```

then Counter and csv.reader are classes, and mycounter and myreader are *instances* of these classes.

Defining an own class allows you to create objects that contain exactly those properties and methods that you need for your task at hand, and are something you might want to look into before beginning a bigger project that aims at reuseability of the code.

Explaining the exact workings lies outside the scope of this book, but you can find many tutorials and examples online, for instance here: `http://www.jesshamrick.com/2011/05/18/an-introduction-to-classes-and-inheritance-in-python/`

Nevertheless, maybe the following example can illustrate the general principle. `self` refers to the specific instance of a class. In other words, in the example below, `self.weight` is the weight of the specific animal we are talking about, not the weight of all animals. `__init__` is a function that is executed when the class is instantiated. All functions that are defined within a class are available as methods in each instance of the class.

```
class animal():
    def __init__(self,weight=10):
```

```
        # when an animal is born, give it a weight of 10 kg unless
            specified otherwise
        self.weight=weight
    def eat(self):
        self.weight+=.2
        print('im eating')

# This is an example of inheritance:
# a cat is an animal (and has all methods and properties of an animal),
# but we can add new functions or overwrite old ones:

class cat(animal):
    def purr(self):
        print('purring')


myownpet = animal() # we create an instance of an animal

myownpet.eat()      # functions defined in the class can be called as
    methods

mysecondfatpet = animal(weight = 2000) # let's get pet elephant

print('I have a pet weighing {} kilo'.format(mysecondfatpet.weight))

poes = cat(weight = 5) # and a cat.

poes.eat() # can eat even though we didn't define it in the cat class.
# That's thanks to inheritance!

print(poes.weight)

poes.purr()

myownpet.purr() # doesn't work
```

D.4 String formatting

Often you want to combine information in a string. For instance, you want to insert the result of a calculation into a string that is printed.

The not-so-elegant way is to simply print these things seperately. For instance:

```
result = 5 + 7
print("He ate", result, "peanuts")
```

The nicer way of doing this is using the string formatting method. Basically, you use curly brackets in a string to indicate where a value is to be inserted, and specify the value afterwards.

```
print("He ate {} peanuts".format(result))
```

Or, using multiple arguments and specifying their order:

```
v1 = 5
v2 = 7
result = 12
print("He ate {2} peanuts in total, first {0} and then {1}".format(v1,v2,
    result))
```

The nice thing is that this offers a lot of extra possibilities to specify how the string is to be formatted exactly. For instance, to get two decimals:

```
points = 19
total = 22
print('Correct answers: {:.2%}'.format(points/total))
```

Have a look at the official documentation for more details: `https://docs.python.org/3.4/library/string.html#format-examples`

Appendix E

Web scraping with Selenium

By Marthe Möller

Sometimes, web scraping as explained in Chapter 8 cannot be used, because content is dynamically fetched or created. For example, maybe some comments are only retrieved by some JavaScript after the user has scrolled down. Here, Selenium comes into play.

Selenium is a tool which you can use to automate what a human would do when browsing the web. For example, with Selenium it is possible to let the computer open a web browser, go to `www.google.com`, enter the name of your favorite cartoon character as a search term, click the "search" button, take a screenshot of the results page and then click the "next page" button. This appendix will show you how to use Selenium so that your computer will do exactly that. . This way, it will introduce you to working with the tool.

To use Selenium, you must first install the Selenium package by typing the following in your terminal:

```
sudo pip3 install selenium
```

You also have to download the geckodriver. You can find it here: `https://github.com/mozilla/geckodriver/releases`

In most cases, the geckodriver-v0.13.0-linux64.tar.gz version is the one you need, but there are versions for Mac and Windows as well. Save it in the directory in which you are working and unpack it, either by double-clicking on it or by using the `tar -xzf` command.

You can consider copying it to a system directory so that it is available no matter in which working directory you are.

```
cd path-to-the-folder-where-geckodriver-is-saved
sudo cp geckodriver /usr/local/bin
```

From the Selenium packageFrom the Selenium package, you must then

import some modules in your Python script. You also need to import the time module, as it is a useful tool when working with Selenium (this is standard module and you do not need to download anything to import it):

```
1 from selenium import webdriver
2 from selenium.webdriver.common.keys import Keys
3 from selenium.webdriver.support.ui import WebDriverWait
4 from selenium.webdriver.support import expected_conditions as EC
5 from selenium.webdriver.common.by import By
6 import time
```

Now, we can instruct our script to open a web browser and go to `www.google.com`.

```
1 driver = webdriver.Firefox()
2 driver.get("https://www.google.com")
```

In line 1 we tell the computer that we want to work with Firefox as a driver. Of course, if you prefer Google Chrome or some other driver, you can replace Firefox with the name of that browser.[1]

In line 2 we tell the computer to use the driver and to go to Google. If you execute these two lines of code, you will see that your computer will open Firefox and go to Google.

Next, we need the computer to type in a word in the Google search field. To do this, we must first tell the computer with which element in the web page we want to work. In this case the element that we want to work with is the white area in which we can type our search word. With Selenium, there are different ways in which you can find elements. One of them is defining elements by their class name. Similar to finding a webpage element's Xpath (see Section **??**), you can find an element's class name by right-clicking the page and by clicking "inspect element" (Figure E.1).

In the case of Google, the search field has class "sbtc".

```
1 element = driver.find_element_by_class_name("sbtc")
```

In addition to finding elements by their class name, it is possible to find them using other classifiers as well (for example, by their Xpath!). Other manners to find elements will be discussed later on, but for more information about different manners to detect elements using Selenium, see `http://selenium-python.readthedocs.io/locating-elements.html`.

Now, let's enter a search word in the search field. For this, use the Keys module that you imported earlier. The Keys module allows you to automate

[1] There is also a driver called PhantomJS, which is a so-called "headless browser", which means that it does not display any browser window; in fact, it doesn't display anything. This can be very useful if you want to run a selenium script on a server without a graphical display, or if you want to run selenium on the background.

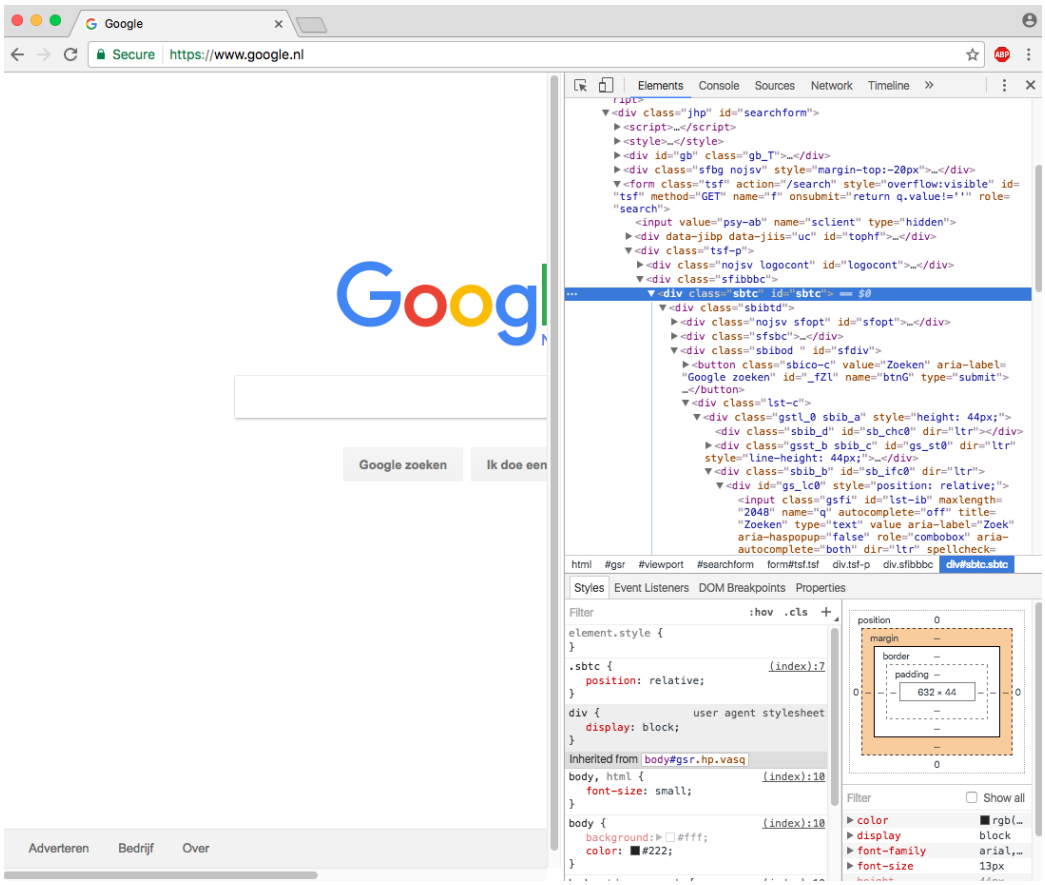


Figure E.1: Looking up an element's class.

everything that a human would type with the keyboard.

```
element.send_keys("TinTin")
```

To instruct the computer to press search, you could use class names again to detect the "search" button and then instruct the computer to click it. However, you can also just press Enter instead of clicking "search", as many humans also do. For this, you can use the Keys module again.

```
element.send_keys(Keys.RETURN)
```

Now, you want to go to a webpage about TinTin. However, before you can find a link to such a webpage, you must make sure that the Google webpage is fully loaded and that all the search results are visible, otherwise the driver might not be able to find such a link. For this, you need to write a loop.

```
try:
```

```
2     element2 = WebDriverWait(driver, 10).until(EC.
          presence_of_element_located((By.ID, "rso")))
3     element3 = driver.find_element_by_partial_link_text("Tintin")
4
5     element3.click()
6     time.sleep(10)    driver.save_screenshot("/home/Marthe/
          screenshot_adventures_TinTin")
7
8 finally:
9     driver.quit()
```

In line 2 and 3, you tell the computer to locate an element by its ID (you can find the ID of an element in the same way as you can find an element's class), which in this case is the ID of the page's results section, namely "rso". In line 2, you also tell the driver to wait for maximum 10 seconds until this element is present. This is called an explicit wait. [2] If the driver does not find the element in 10 seconds, it will quit the browser (of course, you can insert any number of seconds that you want to allow the driver to wait before quitting the browser).

Once the driver has found element2, it will search for a new element on the webpage, namely a link to a webpage with more information about TinTin. However, we do not have a specific link in mind, we just want to click some link to a webpage about TinTin. Luckily, with Selenium you can search for a link by a part of its text. If the driver finds a link that contains the word "TinTin", we should get to a webpage that contains the information we need. So, in line 5 and 6, you instruct the computer to find a link that contains the word "Tintin" and to click it.

The computer has now found a link containing the word "Adventures" and clicked it. It turns out to be a link to a Wikipedia page about the adventures of TinTin. Before you can take a screenshot of this page, let's make sure that the whole page is loaded. This is done in line 7. As we want to wait for the page to load, but we do not have specific element in mind that we want to find, you can use another method to let the driver wait. By using the time function, you can let the driver pause for a number of seconds. This function can also be useful to conceal the fact that there is no human, but in fact an automated script accessing the content.

[2] In addition to letting the computer wait until an element has appeared, you can also let the driver wait for a fixed number of seconds until it starts searching for an element. This is called an implicit wait. The advantage of an explicit wait is that as soon as the driver found the element, it will continue to the next line of the code. If you use an implicit wait, the computer will always wait the number of seconds you instructed it to wait, even if the element takes less time to load. Take a look at this webpage if you want to learn more about waits: `http://selenium-python.readthedocs.io/waits.html`

Now that the page has loaded, you can take a screenshot in line 8. In the command, you also need to specify where you want the computer to save the image and under what name. If you run this line of code, you will see that a screenshot of the webpage appears in your images folder. Finally, you can close the browser in line 9.

The entire code will look as follows:

```
1  from selenium import webdriver
2  from selenium.webdriver.common.keys import Keys
3  from selenium.webdriver.support.ui import WebDriverWait
4  from selenium.webdriver.support import expected_conditions as EC
5  from selenium.webdriver.common.by import By
6  import time
7  driver = webdriver.Firefox()
8  driver.get("https://www.google.com")
9  element = driver.find_element_by_class_name("sbtc")
10 element.send_keys("TinTin")
11 element.send_keys(Keys.RETURN)
12 try:
13     element2 = WebDriverWait(driver, 10).until(EC.
           presence_of_element_located((By.ID, "rso")))
14     element3 = driver.find_element_by_partial_link_text("Tintin")
15     element3.click()
16     time.sleep(10)
17     driver.save_screenshot("/home/Marthe/screenshot_adventures_TinTin")
18 finally:
19     driver.quit()
```

With Selenium you can automate a lot more things, such as saving a file, uploading a file, or downloading texts or links from a webpage. It is also possible to deal with pop-up screens, such as screens with information about cookies. You can find more information on Selenium here: `http://selenium-python.readthedocs.io/index.html`.

Two common geckodriver error messages

Python might throw the following error message: 'geckodriver' executable needs to be in PATH. If this happens, enter the following code in the terminal:

```
1  export PATH=$PATH:/path-to-the-folder-where-geckodriver-is-saved
```

Then, restart Spyder (or whatever Python interpreter you still have opened) and the problem should be solved.

Alternatively, and better if you want to keep using the geckodriver in the long run, you can just save the geckodriver in a directory that is always accessible:

```
cd path-to-the-folder-where-geckodriver-is-saved
sudo cp geckodriver /usr/local/bin
```

Another common error message is this: Permission denied: 'geckodriver.log'. To solve this, create an empty text document using an editor like Geany, name it 'geckodriver.log' and save it in the same directory as where geckodriver is saved. Alternatively, you can create an empty file with the following command in your terminal:

```
touch /path-to-where-the-file-is-saved/geckodriver.log
```

Then change line 3 in the code to this:

```
driver.webdriver.Firefox(log_path="/path-to-where-the-file-is-saved/
    geckodriver.log")
```

and the error message should disappear.

Appendix F

Installing Python on other systems

Before you install Python outside of the virtual machine, read this so that you know what you are doing. While you could just go to `http://python.org` and download Python for your operating system, this may or may not be a good choice. First, Python might already be installed (which you could find out by typing `python` or `python3` in your Terminal). Second, even if it is not, if you prefer one-stop-shopping above dealing with packages, you might like Anaconda (see below).

Keep in mind that one can have multiple versions of Python on one computer, and you don't want to litter your system with a lot of different versions that you then mix up accidentally.

In this book, we use a virtual machine with Linux as operating system. However, if you want to install Python somewhere else, chances are high that it isn't Linux. Within Linux, we used the system's package manager `apt-get` to install Python itself (and maybe some other programs). We then used `pip` to install specific Python packages. This requires you to know which packages you want (that's why we installed a bunch of them on page 7), but on the other hand, if you forgot one, it doesn't really matter because you can easily just install it later on.

All very easy, it seems, but to install some (few) packages via pip one needs specific programs like, for instance, a C compiler. You probably didn't even realize this, because on Linux, this stuff is usually present, and if not, it can be easiliy installed via `apt-get`. MacOS *sometimes* has the necessary stuff installed (usually if you have installed XCode via the App Store). But on a common Windows installation, things can quickly become complicated (unless you know exactly what you are doing).

Because people do not really like dealing with all this stuff, there is an

easier solution: Anaconda `https://www.continuum.io/downloads`.

Anaconda is a platform that includes Python together with a lot of commonly used scientific Python packages preinstalled. In addition, it has its own package manager, `conda`, which can solve some dependencies `pip` cannot resolve.

So, if you want to use Python outside of the virtual machine that we used in this book, you might want to give Anaconda a try.

One important characteristic of anaconda is that it installs an own Python installation in your home directory. To quote from their documentation:

```
On Windows this might be a path such as C:\Users\Jane Smith\anaconda\bin\
    python.
On macOS this might be a path such as /Users/jsmith/anaconda/bin/python.
On Linux this might be a path such as /home/jsmith/anaconda/bin/python.
As well as anaconda, the folder in your home directory might be named
    anaconda2 or anaconda3.
```

Because you might *also* have a version of Python already installed on your system (at least on Linux and MacOS, this is generally the case), you have to make sure that you actually run the correct version: If you just type

```
python
```

in your Terminal, then you probably will *not* start the anaconda version – and thus be unable to use the packages installed with anaconda. You would have to explicitly refer to the location where anaconda is installed.

References

Bird, S., Loper, E., & Klein, E. (2009). *Natural language processing with Python*. Sebastopol, CA: O'Reilly.

Boumans, J. W., & Trilling, D. (2016). Taking stock of the toolkit: An overview of relevant autmated content analysis approaches and techniques for digital journalism scholars. *Digital Journalism*, *4*(1), 8–23. doi: 10.1080/21670811.2015.1096598

De Smedt, T., & Daelemans, W. (2012). Pattern for Python. *The Journal of Machine Learning Research*, *13*, 2063–2067.

Hunter, J. D. (2007). Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, *9*(3), 90–95. doi: 10.1109/mcse.2007.55

Hutto, C. J., & Gilbert, E. (2014). Vader: A parsimonious rule-based model for sentiment analysis of social media text. In *Eighth international aaai conference on weblogs and social media.*

Jones, E., Oliphant, T., Peterson, P., et al. (2001). *SciPy: Open source scientific tools for Python.* Retrieved from `http://www.scipy.org/`

Jürgens, P., & Jungherr, A. (2016). A tutorial for using Twitter data in the social sciences: Data collection, preparation, and analysis. *SSRN*. doi: 10.2139/ssrn.2710146

Kitchin, R. (2014). Big Data, new epistemologies and paradigm shifts. *Big Data & Society*, *1*(1), 1–12. doi: 10.1177/2053951714528481

Lazer, D., Pentland, A., Adamic, L., Aral, S., Barabási, A.-L., Brewer, D., ... van Alstyne, M. (2009). Computational social science. *Science*, *323*, 721–723. doi: 10.1126/science.1167742

Maas, A. L., Daly, R. E., Pham, P. T., Huang, D., Ng, A. Y., & Potts, C. (2011). Learning word vectors for sentiment analysis. In *Proceedings of the 49th annual meeting of the association for computational linguistics: Human language technologies* (pp. 142–150). Portland, Oregon, USA: Association for Computational Linguistics. Retrieved from `http://www.aclweb.org/anthology/P11-1015`

Mazières, A., Trachman, M., Cointet, J.-P., Coulmont, B., & Prieur, C. (2014). Deep tags: toward a quantitative analysis of online pornogra-

phy. *Porn Studies*, *1*(1-2), 80-95. doi: 10.1080/23268743.2014.888214

McKinney, W. (2012). *Python for data analysis*. Sebastopol, CA: O'Reilly.

McKinney, W., et al. (2010). Data structures for statistical computing in Python. In *Proceedings of the 9th python in science conference* (Vol. 445, pp. 51–56).

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, *12*, 2825–2830.

Řehůřek, R., & Sojka, P. (2010). Software framework for topic modelling with large corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks* (pp. 45–50). Valletta, Malta: ELRA. (`http://is.muni.cz/publication/884893/en`)

Russel, M. (2013). *Mining the social web. Data mining Facebook, Twitter, LinkedIn, Google+, GitHub, and more* (2nd ed.). Sebastopol, CA: O'Reilly.

Seabold, S., & Perktold, J. (2010). Statsmodels: Econometric and statistical modeling with Python. In *9th Python in science conference.*

Trilling, D. (2015). Two different debates? Investigating the relationship between a political debate on TV and simultaneous comments on Twitter. *Social Science Computer Review*, *33*(3), 259–276. doi: 10.1177/0894439314537886

Trilling, D. (2017). Big Data, Analysis of. In J. Matthes, C. S. Davis, & R. F. Potter (Eds.), *The international encyclopedia of communication research methods*. Wiley. doi: 10.1002/9781118901731.iecrm0014

Van der Walt, S., Colbert, S. C., & Varoquaux, G. (2011). The NumPy array: A structure for efficient numerical computation. *Computing in Science & Engineering*, *13*(2), 22–30. doi: 10.1109/mcse.2011.37

Van Atteveldt, W. (2008). *Semantic network analysis: Techniques for extracting, representing, and querying media content*. Charleston, SC: BookSurge.

VanderPlas, J. (2016). *Python data science handbook: Essential tools for working with data*. Sebastopol, CA: O'Reilly.

Wilson, T., Wiebe, J., & Hoffmann, P. (2005). Recognizing contextual polarity in phrase-level sentiment analysis. In *Proceedings of the conference on human language technology and empirical methods in natural language processing* (pp. 347–354).