
Programming for Cognitive and Brain Sciences

Release 0.29

Christophe Pallier

Apr 03, 2024

CONTENTS:

1	Foreword	3
2	Software Installation	5
2.1	Instructions for native Windows (without WSL)	6
2.2	Instructions for Windows using the Windows Subsystem for Linux (WSL)	8
2.3	Instructions for MacOS X	9
2.4	Instructions for Ubuntu Linux	11
3	Check your installation	15
3.1	Check R and Rstudio	15
3.2	Check if Python can be executed in a Terminal	16
3.3	Check Git	16
3.4	Check Python	17
3.5	Check basic graphics	18
3.6	Check matplotlib	18
3.7	Check pygame	21
3.8	Check Expyriment	22
3.9	Appendices	22
3.10	Keep your local copy of the course material up to date	22
3.11	Basic surviving skill: how to enter command lines in a Terminal	23
4	Starting from Scratch	25
4.1	First steps	25
4.2	Concepts learned so far	27
4.3	Loops	27
4.4	Two sprites	28
4.5	Conditional execution or branching	29
4.6	First series of exercices	30
4.7	Variables	31
4.8	Second series of exercices	32
5	Fun programming language	35
6	Interacting with a computer (in a Nutshell)	37
6.1	The shell	37
6.2	Disks, Directories and files	39
6.3	filenames, directory structure	39
6.4	Working directory. Absolute pathnames vs. relative pathnames (..)	40
6.5	What is the PATH?	40
6.6	What is a library (or module/package)?	40

7	Running Python	41
7.1	Running a python script from the command line	41
7.2	Testing a short piece of python code	42
7.3	Write code with a text editor (Edit-run cycle)	44
7.4	Using an Integrated Development Environment (IDE)	45
7.5	Perform an interactive data analysis with jupyter-notebook or jupyter-lab	46
7.6	Developping in Python with Emacs	47
8	Coding Exercises	49
8.1	Flow control	50
8.2	Lists	51
8.3	Functions	52
8.4	Strings	53
8.5	Dictionaries	53
8.6	File reading and writing	54
9	Automata and Computers	57
9.1	The Computational Theory of Mind	57
9.2	What is computation anyway ?	58
9.3	The ancestors of the computer: the automata	58
9.4	Formal description of an automaton	58
9.5	Examples of transition diagrams	58
9.6	What is a Computer?	64
9.7	Register machines	65
9.8	The Seven secrets of computers revealed	66
9.9	Programmable computers	67
9.10	Compilation and interpretation	67
9.11	Operating systems	68
9.12	What is a Terminal?	68
10	Representations of numbers, text, images	71
10.1	Representation of integers	71
10.2	Representation of text	76
10.3	Representation of images	80
11	Creating stimuli	83
11.1	Static visual stimuli	84
11.2	Dynamic visual stimuli	94
11.3	Creating and playing sounds	95
11.4	More illusions	96
12	Experiments	97
12.1	Simple reaction times	97
12.2	Decision times	100
12.3	Numerical distance effect	100
12.4	Posner's attentional cueing task	100
12.5	Stroop Effect	100
12.6	A general audio visual stimulus presentation script	102
12.7	Sound-picture matching using a touchscreen	102
12.8	More examples using Expyriment	102
13	Programming a Lexical decision task	103
13.1	Step 1: stimuli in constants	103
13.2	Step 2: read stimuli from a csv file	103
13.3	Select words in a lexical dabatase	104

13.4	Automatising database searches with R and Python	104
13.5	Generate nonwords	104
13.6	Create a stimuli file	105
13.7	Use <i>sys.argv</i> to pass the name of the file containing the list of stimuli	105
13.8	Improving the pseudowords	105
13.9	Data analysis	105
13.10	Auditory Lexical Decision	106
13.11	Finally	106
14	Data Analyses	107
14.1	Permutation tests	107
14.2	Bootstrap	107
14.3	Basic Data Analysis with R	108
14.4	Comparing means using Easy ANOVA (Analysis of Variance)	108
14.5	Frequency Analysis	108
15	Lexical Statistics	109
15.1	Zipf law	109
15.2	Benford's law	110
15.3	Neuroimaging	110
16	Using HTML for webpages	111
16.1	HTML Basics	111
16.2	Simple shapes	112
16.3	Exercises	117
17	Using JavaScript	119
17.1	Combining shapes with JS	119
17.2	Modifying elements with innerHTML	121
17.3	Modifying elements with pure JS	121
17.4	Drawing on canvas.	122
17.5	Using JsPsych	122
17.6	Practice: color-detection task	125
18	Regular Expressions	131
18.1	Tutorial	131
18.2	Example in Cognitive Science research	131
18.3	More on the Theory	131
19	Simulations	133
19.1	Monte Carlo Estimation	133
19.2	Fractals	134
19.3	Formal systems	134
19.4	Artificial Neural networks	135
20	Hopfield Networks	137
20.1	Model	137
20.2	Computing the weight matrix	138
20.3	Encoding images	139
20.4	Simulations	140
20.5	Discussion	140
21	Web Scraping	141
22	Tools to do Reproducible Science	153

22.1	Two tools: literate programming and version control	153
22.2	Lesson 101: how to compare files or directories	154
22.3	Introducing git	155
22.4	Working with remotes	159
23	Resources to learn Git	163
24	How to solve problems	165
24.1	How to think like a programmer: lessons in problem solving	165
24.2	How to report bugs effectively	168
24.3	<i>How to solve it</i> by Polya	169
24.4	Computer Science Distilled	169
25	Writing clean code	171
25.1	Names	171
25.2	Functions	171
25.3	Successive refinement	172
25.4	General	172
25.5	Comments	172
25.6	Testing	173
26	Building abstractions with recursive functions and higher-order functions	175
26.1	Recursive functions	175
26.2	Higher-order functions	176
26.3	Reference	177
27	Sending TTL triggers	179
27.1	Parallel Port	179
27.2	DLP-IO8-G	180
27.3	Python	181
27.4	Reading an input line	182
27.5	Arduino	183
27.6	Raspberry Pi	183
28	Cogmaster Lectures	185
28.1	Course description	185
28.2	Prerequisites	186
28.3	Projects	186
29	Projects	189
30	Resources	191
30.1	Learning Python3	191
30.2	Programming skills	191
30.3	Books relevant to Cognitive and Brain Sciences Programming	192
30.4	Stimulus/Experiment generation modules	192
30.5	Data analyses, Statistics in Python	192
30.6	Simulations	193

PDF version of this document

<https://media.readthedocs.org/pdf/pcbs/latest/pcbs.pdf>

Git repository on github (contains slides, solutions to exercises, and more)

<https://github.com/chrplr/PCBS>

Discord (discussion forum)

<https://discord.gg/WEQWWmqZ>

Self-evaluation quizz (do you need to attend PROG101?):

<https://forms.gle/iq7CN41DZMA6XJC59>

FOREWORD

Students in cognitive/neuro science need to learn programming in order to:

- have a basic understanding of how computers work because of the importance of the Computational Theory of Mind in Cognitive Science.
- prepare and run experiments on humans or animals, analyze data, run computational simulations, ...
- automate the boring stuff such as, repetitive work on files, web scraping, ...

This book gathers various lectures given at the Master in Cognitive Science in Paris (“*Cogmaster*”) over the last 15 years.¹ (Students enrolled in the current course, *Programming for Cognitive and Brain Sciences*, will find more information about it in the chapter *Cogmaster Lectures*).

This book can also be used as a self-teaching tool. Depending on the Reader’s prior programming experience and interests, different sections are relevant. I have tried to keep the sections as independent as possible to facilitate the study, but you will find some cross referencing when needed. Although we cover many topics, some are only touched superficially; the reader should definitely check the *Resources* section to go further.

The latest version of this document is always available at <https://pcbs.readthedocs.io> and can be downloaded as a single pdf file.

Dear Reader, suggestions for corrections are very appreciated! You can just send me an email at christophe.pallier@gmail.com, or open an issue on the PCBS [github repository](#).

Christophe Pallier <<http://www.pallier.org>>

¹ These lectures come from various courses (*Atelier d’experimentation humaine*, *Atelier d’Analyse de données expérimentales*, *Introduction to R*, *Introduction to Programming with Python*, ...) taught by me and my colleagues and friends Mark Wexler, Christophe Lalanne, Sylvain Charron, Ewan Dunbar, Cédric Foucault, Henri van den Driessche.

SOFTWARE INSTALLATION

We explain below how to install:

- a code editor: *Visual Studio Code*
- a version control system: *Git*
- the statistical programming language *R*
- the integrated development environment *RStudio*
- The programming language *Python* (**version 3.7** !)
- The experiment generator *Epyriment* (python module)

After the installation, you will need to check that everything works (see [Check your installation](#))

Warning:

- You will need to download several GB of software from the Internet. Therefore, make sure to have a decent connection.
- Make sure that you have at least 5GB of free space on your hard drive to unpack the various software.
- You will need to have administrator rights to install some of the software. If you are using a computer from an institution, check with the IT support team.

Contents

- *Software Installation*
 - *Instructions for native Windows (without WSL)*
 - *Instructions for Windows using the Windows Subsystem for Linux (WSL)*
 - *Instructions for MacOS X*
 - *Instructions for Ubuntu Linux*

2.1 Instructions for native Windows (without WSL)

Please follow these instructions carefully! If you do not, many things could go wrong and you might need to run the installers again.

2.1.1 R and Rstudio

R is a programming language specialized for statistical data analyses; Download and install it from <https://cran.rstudio.com/bin/windows/base/> (accepting all the default options proposed by the installer)

Rstudio is an *Integrated Development Environment* for R which greatly simplifies the use of RMarkdown. You can download and install the free version of RStudio Desktop from <https://posit.co/download/rstudio-desktop/> (accepting all the default options)

Launch RStudio, go to the “Tools” menu, select “install packages” and in the dialog window that opens, in the ‘packages’ box, type:

```
tidyverse lme4 ez
```

This will take a while to download packages from the internet and install them. Do not close rstudio until the process is finished (no more scrolling messages in the console). Meanwhile, you can start the installation of the other software below.

2.1.2 Code editor

A code editor is a program that allows you to edit pure text files such as Python programs, [Markdown](#) or [LaTeX](#) documents, etc.

If you do not already have a code editor that you like, download and install [Visual Studio Code](#) (accepting the default options proposed by the installer)

2.1.3 Python3 using the Anaconda distribution

You will need to install Python version 3.7 (because the expyriment module, to be installed later, requires this specific version)

There exists various Python distributions. Under Windows, we recommend the [Anaconda3 distribution](#) as it already contains many of the packages needed for cognitive science (but it is very large. If you lack disk space, you can install [miniconda](#), but later you will need to install many python packages manually)

1. Go to <https://www.anaconda.com/products/individual>, click on Download and select the 64-bit installer for Windows.
2. Execute the Anaconda3 installer. Pay special attention to the options:
 - To the question ‘Install for’, select **Just Me (recommended)**
 - Accept the default Destination folder and click on **Next**
 - **VERY IMPORTANT:** Check the boxes in front **Add Anaconda3 to my PATH** (ignore the warning that this is not recommended) and **Register Anaconda as my default Python** (all the other default options are fine). Click on **Install**.
 - upon completion, click on **Next**, then **Finish**

2.1.4 The Git version control system

Git is a version control tool for software development, an indispensable tool to do reproducible science.

IMPORTANT: you must wait for the installation of Anaconda to finish *before* trying to install Git.

Download the installer of [Git for Windows](#) and launch it.

1. When the GNU Licence is displayed, press **Next**;
2. Accept the default installation folder and press **Next**;
3. Accept all the Components selected by default and press **Next**
4. Accept the creation of a start menu folder named 'Git': press **Next**;
5. VERY IMPORTANT: When proposed a default editor, select 'Use the nano editor' (unless you want to learn Vim)
6. VERY IMPORTANT: When proposed to adjust the PATH environment variable, tick the box "Use Git and optional unix tools from the command line prompt".

You can accept all the other defaults.

Now, to finish the installation of git, launch `Git bash` (use the "Search box"), and on type:

```
conda init bash
```

then press 'Return'

If the computer replies with an error message of the type `conda: command not found`, you did not properly install Anaconda (you did not check the box that made sure it is added to the PATH). If you know how to do correct the PATH, correct it now, otherwise, reinstall anaconda.

Now, type:

```
echo "alias python='winpty python.exe'" >> ~/.bash_profile
```

and press 'Return'

Close the Git Bash Terminal, and reopen a new Git Bash.

Type:

```
which python
```

And then:

```
python
```

It should print a message 'Python 3.x.xx...' and a give a prompt '>>>'. You are talking to the python interpreter. Type:

```
2**100
```

This should display the 100 power of 2. To exit python, type:

```
quit()
```

If python does not start, there is something wrong. Most probably, you forgot to check the box **Add Anaconda to my PATH** during the installation of Anaconda. do it again. If it still does not work, ask for help.

Finally, you must configure Git: Within a Git Bash terminal, type the following commands (replacing `your_first_and_last_name_here` and `your_email_here` by relevant personal information)

```
git config --global user.name "your_first_and_last_names_here"  
git config --global user.email your_email_here  
git config --global core.editor nano
```

You can close Git Bash by typing the command *exit* or, faster, by pressing *Ctrl-D*, or by just closing its window.

2.1.5 The Pygame and Expyriment python modules

We will rely on the [Pygame module](#) to create stimuli and the [Expyriment Python Library](#) to program behavioral experiments.

Start **Git bash** and, in the Terminal that opens, type:

```
conda create -n expyriment python=3.7
```

And press ‘Return’ to accept the installation.

Then, type:

```
conda activate expyriment  
conda install ipython  
pip install expyriment[all]
```

To check the installation, type:

```
ipython
```

and then:

```
import expyriment
```

If a message *Experiment 0.10.0 ...* is displayed and no error message, the installation worked. Press *Ctrl-D* to quit ipython, and *Ctrl-D* again to quit Git Bash.

2.1.6 Checking everything

Now you should check if everything works, following the instructions in chapter [Check your installation](#).

2.2 Instructions for Windows using the Windows Subsystem for Linux (WSL)

As an alternative to installing Python et coll. as native Windows applications, you can install Ubuntu Linux under Windows, relying on Microsoft’s *Windows Subsystem for Linux* (aka WSL) and then install Python and coll. within Ubuntu.

The interest of having WSL is that it gives you the opportunity to learn to use Linux, which may come handy if one day you need to access remote computational facilities such as the Jean Zay supercomputer.

If you already have WSL Ubuntu installed on your Windows PC, you can just open an Ubuntu terminal and jump directly to the [linux](#) section.

If not, and want to try it, keep reading.

2.2.1 Installing the Microsoft's *Windows Subsystem for Linux*

If you want to install Linux under Windows using the WSL, follow the instructions at <https://docs.microsoft.com/en-us/windows/wsl/install> but be aware that the download is large (several GB) and the installation can be lengthy, depending on the power of your PC (30min-1h)

Here is an overview of the process:

- If you have Windows 11:
 1. Install the vGPU driver for your graphics card ([Intel](#), [AMD](#) or [Nvidia](#))² if it not already installed.
 2. Launch “Windows PowerShell” as administrator, and execute the command:

```
wsl --install -d ubuntu
```

- If you have Windows 10, follow [these instructions](#). (You may have to enable [Hyper-V](#))

This downloads the full Ubuntu Linux distribution which may take a while. Some versions of Windows may even ask you to reboot during the installation process. At some point during the install, a new Terminal window entitled “Ubuntu” will open and will require a new user name and password. You can type anything but it is crucial that you note down the password as it will be needed to install software under Ubuntu.

Note: If anything goes wrong during the installation check the [Troubleshooting WSL](#) section.

Now that the WSL installation is finished, jump to the [linux](#) section in order to install the required software.

2.3 Instructions for MacOS X

2.3.1 Code editor

A code editor is a program that allows you to edit pure text files such as Python programs, [Markdown](#) or [LaTeX](#) documents, etc.

Unless you already use a code editor that you are familiar and happy with, we recommend that you download and install [Visual Studio Code](#). Follow the instructions specific to MacOS.

2.3.2 The Git version control system

Download and install [Xcode](#), either from the App store, or by using the command line `xcode-select --install`. This will provide git.

To finish the installation of git, open a Terminal window¹ and type the following command lines, making sure to replace `your_first_and_last_name_here` and `your_email_here` by the relevant personal information:

² To determine which type of GPU (Intel, AMD, Nvidia) is installed on your computer, under Windows, open the *Task Manager*, e.g. with `Ctrl-Alt-Del`, and search for GPU under the *Performance* tab.

¹ To open a Terminal:

- Windows with the WSL: launch the “Ubuntu” app.
- Windows with Git for Windows: launch “Git bash”.
- Linux: Launch Terminal or press `Ctrl-Alt-T` (Gnome or Xfce) or `Win-Return` (i3).
- MacOS: Type `terminal` in the Spotlight search field. Alternatively, you can open a Finder window and select the *Application* folder, then the *Utilities* folder, then double-click on the *Terminal* icon..

```
git config --global user.name "your_first_and_last_names_here"
git config --global user.email your_email_here
git config --global core.editor nano
```

2.3.3 The R statistical software

Download and install the latest version of **R** from <https://cran.rstudio.com/bin/macosx/>

2.3.4 Rstudio Desktop

Download and install the latest version of **RStudio Desktop** from <https://posit.co/download/rstudio-desktop/>. Make sure to select the MacOS version!

2.3.5 Python

Download and install **Anaconda3 Python** from <https://www.anaconda.com/products/individual>

2.3.6 The Pygame and Expyriment python modules

1. Install **XQuartz**. Download the .dmg file from the official website and open it to install.
2. Open a Terminal and type `/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"` to install **Homebrew** (which is needed to install SDL).
 - If you see “Password: “, this means the Terminal is prompting you to type your computer’s administrator account password. Type your password and press the return key to continue.
 - If you see an error message such as “Error: /usr/local/Cellar is not writable. You should change the ownership and permissions of /usr/local/Cellar back to your user account: sudo chown -R \$(whoami) /usr/local/Cellar”, run in the Terminal the command that was suggested (here `sudo chown -R $(whoami) /usr/local/Cellar`), and then run the previous command to try installing Homebrew once again
3. In the same Terminal, type `brew install sdl2 sdl2_image sdl2_mixer sdl2_ttf pkg-config` to install SDL (which is needed to install expyiment). This may take a while.
4. In the same Terminal, type `pip install "expyriment[all]"` to install expyiment.
5. In the same Terminal, type `pip install -U pygame` to upgrade pygame (version `>= 2.0` is required on recent versions of macOS, but this must be done *after* installing expyiment otherwise the install of expyiment will fail).

Now you should check if everything works, following the instructions in chapter [Check your installation](#).

2.4 Instructions for Ubuntu Linux

These instructions were tested with Ubuntu 20.04

Open a Terminal^{Page 9, 1}.

Then, for each software section below, copy and paste in the terminal the lines that are in the boxes and press Enter to execute them.

2.4.1 Git

Git is a free distributed version control system.

```
sudo apt install git -y
```

Now, type the following command lines, making sure to replace `your_first_and_last_name_here` and `your_email_here` by the relevant personal information:

```
git config --global user.name "your_first_and_last_names_here"
git config --global user.email your_email_here
git config --global core.editor nano
```

2.4.2 Visual Studio Code editor

Install Visual Studio Code:

```
snap install code
```

2.4.3 R language for statistics

R is a free software environment for statistical computing and graphics.

```
sudo apt update -qq
```

```
sudo apt install --no-install-recommends software-properties-common dirmngr wget -qO- https://cloud.r-project.org/bin/linux/ubuntu/marutter_pubkey.asc | sudo tee -a /etc/apt/trusted.gpg.d/cran_ubuntu_key.asc
sudo add-apt-repository "deb https://cloud.r-project.org/bin/linux/ubuntu $(lsb_release -cs)-cran40/"
```

```
sudo apt install --no-install-recommends r-base sudo add-apt-repository ppa:c2d4u.team/c2d4u4.0+ sudo apt install --no-install-recommends r-cran-tidyverse
```

(in case of trouble, check the latest instructions at <https://cran.rstudio.com/bin/linux/ubuntu/>)

2.4.4 Rstudio Desktop

Rstudio is an *Integrated Development Environment* for R which greatly simplifies the use of RMarkdown. You can download and install the latest version of **RStudio Desktop** from <https://posit.co/download/rstudio-desktop/> Make sure to select the ubuntu version!

```
wget https://download1.rstudio.org/electron/jammy/amd64/rstudio-2023.06.2-561-amd64.deb
sudo apt install ./rstudio-2023.06.2-561-amd64.deb -y
```

2.4.5 Python3

Python is the main programming language used in these courses. The following commands install various modules that will be needed.

```
sudo apt install -y python3 ipython3 python3-dev python-is-python3 python3-future \
python3-opengl python3-pip python3-ipython python3-pygame python3-numpy \
python3-matplotlib python3-skimage python3-pandas python3-scipy \
python3-imageio python3-ipython
```

2.4.6 Expyriment

We now need to install the *Expyriment* module.

Try:

```
sudo apt-get install -y python3-dev libasound2-dev
sudo pip install simpleaudio

sudo pip install expyriment[all]
```

Check the installation by typing:

```
python
```

and then, after >>>:

```
import expyriment
```

If you see *No module named expyriment*, there was a problem (most probably due a version of pygame). Forcing the using of python version 3.7 should solve it.

First install *pyenv*, then:

```
pyenv install 3.7.6
pyenv virtualenv 3.7.6 expyriment
pyenv activate expyriment
pip install expyriment[all]
```

Note: Later, when you will need to run python scripts importing expyriment, you will need first to activate the virtual environment with:

```
pyenv activate expyriment
```

2.4.7 Psychtoolbox

(This is optional: we do not make use of the Psychtoolbox in this course)

Psychtoolbox-3 is a set Octave functions which is very popular in vision and neuroscience research. This installation is optional as the Psychtoolbox is **not used** in this book.

First, add the [Neurodebian](<https://neuro.debian.net/>) repository.

```
wget -O- http://neuro.debian.net/lists/focal.de-m.full | sudo tee /etc/apt/sources.list.  
↳d/neurodebian.sources.list
```

```
sudo apt-key adv --recv-keys --keyserver hhttps://keyserver.ubuntu.com 0xA5D32F012649A5A9
```

Then activate the sources and install the required packages:

```
sudo sed -Ei 's/^# deb-src /deb-src /' /etc/apt/sources.list  
sudo apt update  
  
sudo apt build-dep octave-psychtoolbox-3 -y  
sudo apt install subversion libdc1394-22-dev libfreenect* libgstreamer1.0-dev_␣  
↳libgstreamer-plugins-* -y
```

Download the psychtoolbox installation script:

```
wget https://raw.githubusercontent.com/Psychtoolbox-3/Psychtoolbox-3/master/Psychtoolbox/  
↳DownloadPsychtoolbox.m.zip  
unzip DownloadPsychtoolbox.m.zip  
  
mkdir ~/PTB3
```

Finally, start octave and, on Octave's command line, type:

```
DownloadPsychtoolbox('/home/neurostim/PTB3')  
PsychLinuxConfiguration()  
  
# test  
DrawingSpeedTest()
```

Now you should check if everything works, following the instructions in chapter *Check your installation*.

CHECK YOUR INSTALLATION

Contents

- *Check your installation*
 - *Check R and Rstudio*
 - *Check if Python can be executed in a Terminal*
 - *Check Git*
 - *Check Python*
 - *Check basic graphics*
 - *Check matplotlib*
 - *Check pygame*
 - *Check Expyriment*
 - *Appendices*
 - *Keep your local copy of the course material up to date*
 - *Basic surviving skill: how to enter command lines in a Terminal*

3.1 Check R and Rstudio

Launch Rstudio, and in the Console (left window), type:

```
example(density)
```

This should display a series of graphics (Press <Return> to advance).

Then type:

```
require(tidyverse)
```

If the package `tidyverse` is not found, you need to install it with:

```
install.packages('tidyverse')
```

Close RStudio. That will be all.

3.2 Check if Python can be executed in a Terminal

Open a Terminal (that is, *Git Bash* for Windows users) and type:

```
which python
```

This should print the location of the python interpreter, e.g. something like `/home/macron/anaconda3/python`

If no message is displayed, the `PATH` environment variable — which lists the directories where programs can be found — is messed up, most probably due not following closely the installation instructions (for Windows users, check the Anaconda and Git for Windows instructions)

Type `python` and press ‘Enter’.

You should see a prompt `>>>` followed by a blinking cursor. Python is waiting for your commands! Type:

```
2**100
```

This should display the 100 power of 2. Type `quit()` to leave Python and press *Ctrl-D* to close the Terminal. Basic Python is working.

3.3 Check Git

Download the course materials using Git by entering the following command line in a Terminal:

```
git clone https://github.com/chrplr/PCBS.git
```

You should see a message `Cloning into 'PCBS'...` and, if everything goes well, all the course materials (python scripts, data files, ...) should be downloaded in a new subdirectory called `PCBS`, within the current working directory. You can `cd` into it and list its content:

```
cd PCBS
pwd
ls
```

Your Terminal window should more or less look like this:

```

File Edit View Search Terminal Help
(base) cp983411@is154920 ~$ git clone https://github.com/chrplr/PCBS.git
Cloning into 'PCBS'...
remote: Enumerating objects: 4640, done.
remote: Counting objects: 100% (2335/2335), done.
remote: Compressing objects: 100% (1854/1854), done.
remote: Total 4640 (delta 998), reused 1780 (delta 462), pack-reused 2305
Receiving objects: 100% (4640/4640), 170.34 MiB | 26.86 MiB/s, done.
Resolving deltas: 100% (2111/2111), done.
(base) cp983411@is154920 ~$ cd PCBS/
(base) cp983411@is154920 ~/PCBS (master)$ pwd
/home/cp983411/PCBS
(base) cp983411@is154920 ~/PCBS (master)$ ls
automate_tasks  docs          pandoc.css      simulations      wip
coding-exercises  experiments  pdfs            slides
_config.yml      games        quiz            stats-and-data-analyses
CONTRIBUTORS.txt LICENSE       README.md       stimuli
databases        Makefile     requirements.txt week_exercises
(base) cp983411@is154920 ~/PCBS (master)$ 

```

Warning: If a folder named PCBS already exists in the current working directory, git will stop and will not download the content of the remote PCBS repository. In that case, you must delete or move the existing PCBS folder before running the `git clone` command above.

When you open a Terminal, the current working directory is your “home”, or “user”, directory, until you start navigating in the filesystem with the `cd` (change directory) command. If you are lost at this point, read [Navigating the file system](#).

3.4 Check Python

This tests if Python3 is installed and correctly configured.

```

cd ~/PCBS/games
python human-guess-a-number.py

```

```

is154920:/home/cp983411
File Edit View Search Terminal Help
(base) cp983411@is154920 ~/1_teaching/PCBS/games (master)$ python human-guess-a-number.py
I am thinking about a number between 1 and 100. Try to find it!
Your guess (1-100)? 50
Your guess is too high!

New guess? 25
Your guess is too low!
New guess? 40
Your guess is too high!

New guess? 35
Your guess is too high!

New guess? 30
You win! The number was indeed 30
(base) cp983411@is154920 ~/1_teaching/PCBS/games (master)$

```

Warning: If you receive an error message such as `bash: python: No such file or directory`, and you are sure that python is installed, the most likely reason is that the problem lies with the `PATH` environment variable) listing all the directories: the directory containing the python executable file may be missing from the list. This happens for example, if you run the Anaconda3 installer and did not check the relevant box.

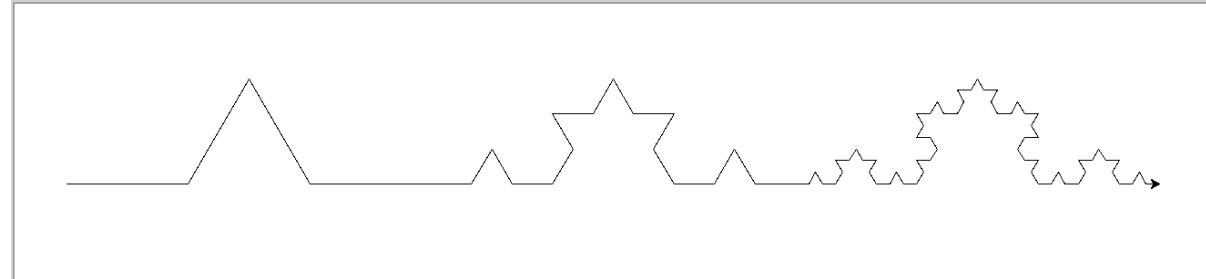
3.5 Check basic graphics

```

cd ~/PCBS/simulations/fractals
python koch0.py

```

Python Turtle Graphics



You can launch the Visual Code editor and open the python file `PCBS/simulations/fractals/koch0.py` to check the code.

3.6 Check matplotlib

matplotlib is a python library to create and display graphics. Here, we try to generate some graphical effects to check that this library has been correctly installed.

```

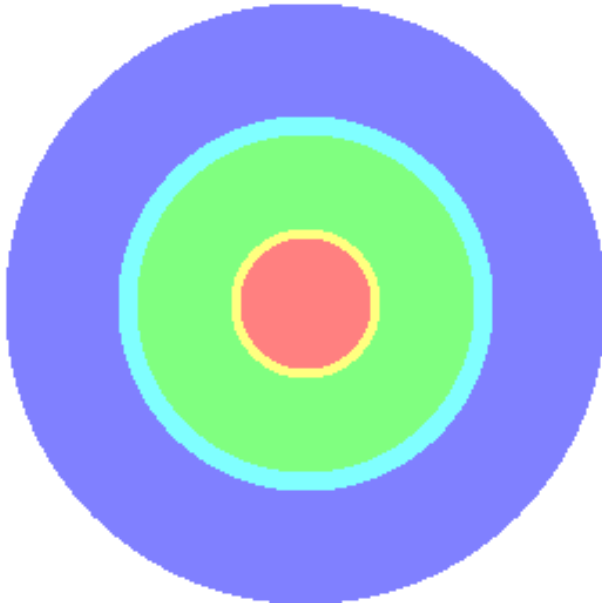
cd ~/PCBS/stimuli/visual

```

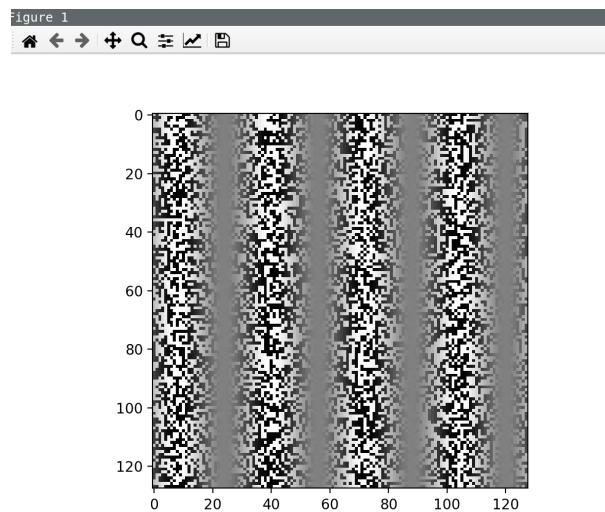
```

python bullseye.py

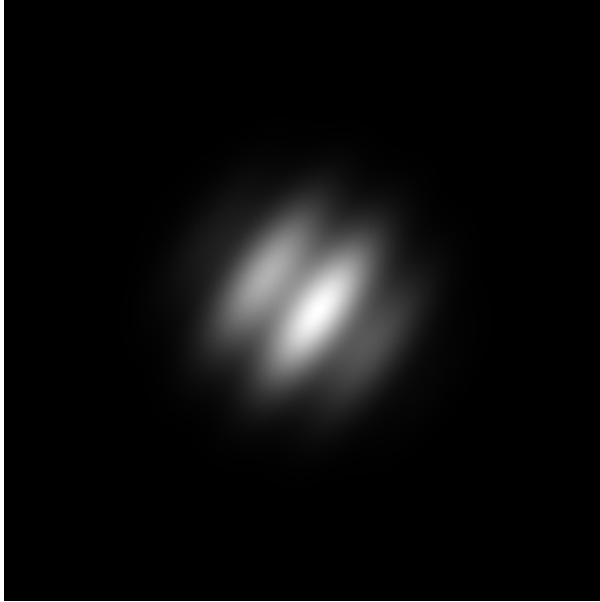
```

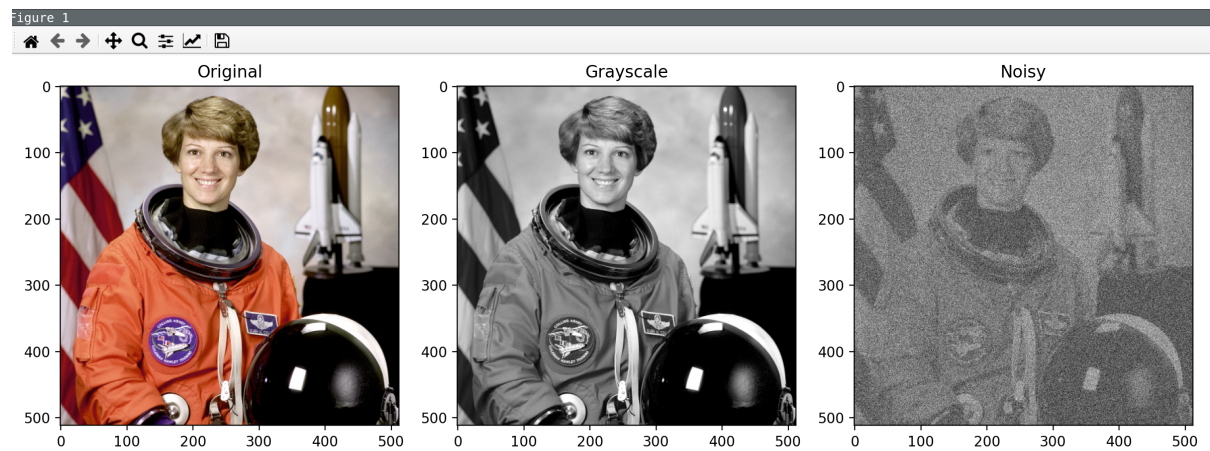
```
python contrast_modulated_grating.py
```



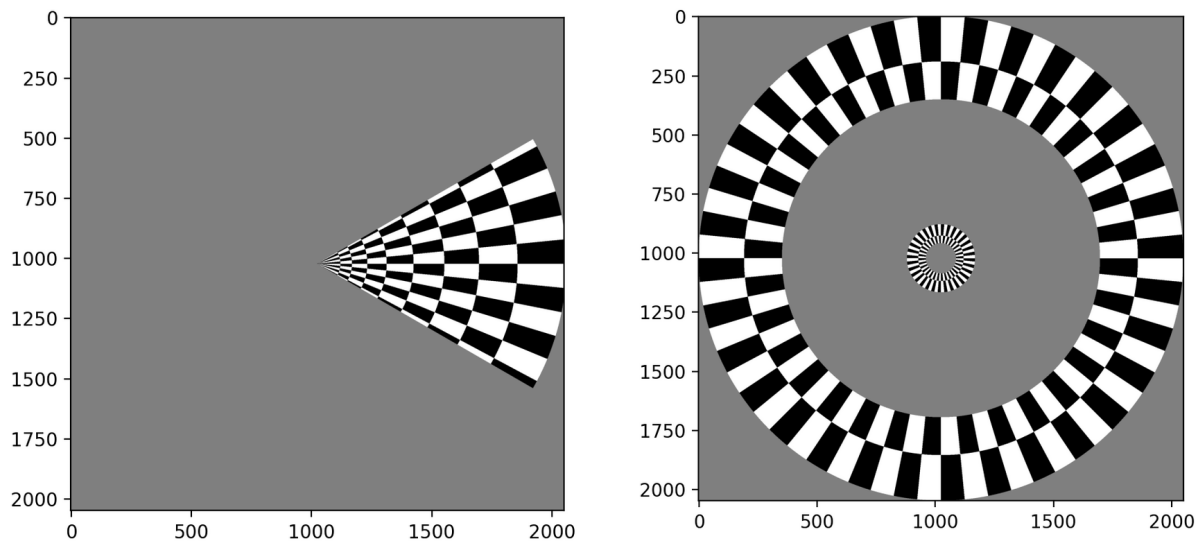
```
python gabor.py
```



```
python image-manipulation.py
```



```
python wedgering.py
```



3.7 Check pygame

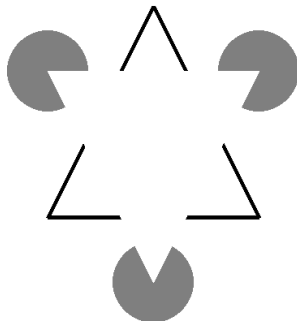
Pygame is a Python library to create simple audio visual games. It was installed along with `expyriment`. If you had to create a Python virtual environment when you installed `expyriment`, you need to activate it:

```
conda activate expyriment # if you use conda
pyenv activate expyriment # if you use standard python with pyenv
```

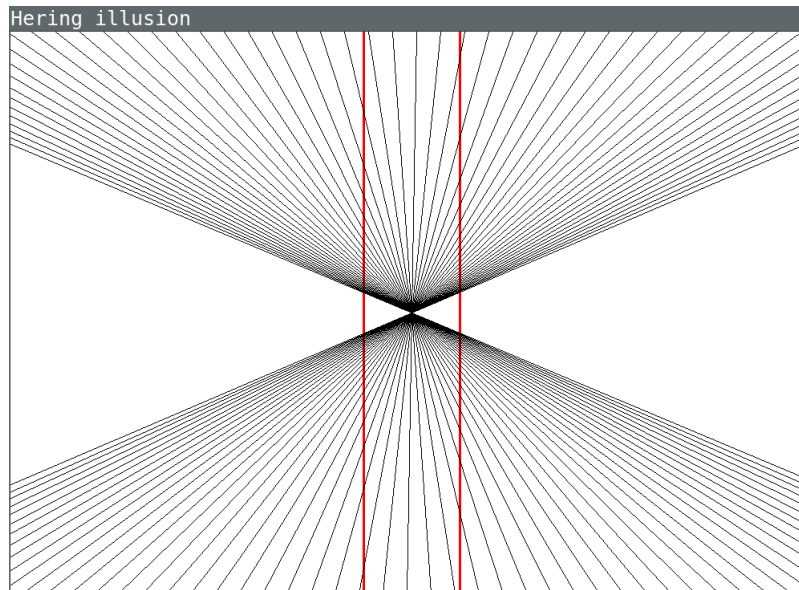
You can then check if `pygame` is installed by starting `python` on a command line and typing `import pygame` at the `>>>` prompt. A message ```Hello from the pygame community. should be displayed.`

```
cd ~/PCBS/stimuli/visual-illusions/
python kanizsa_triangle.py
```

```
triangle
```



```
python hering.py
```



```
python extinction-rotated.py
```

3.8 Check Expyriment

Expyriment is a Python library for designing and conducting behavioural and neuroimaging experiments.

If you had to create a Python virtual environment when you installed expyiment, you need to activate it (unless it is already activated in your current Terminal):

```
conda activate expyiment # if you use conda
pyenv activate expyiment # if you use standard python with pyenv
```

Try to run the following experiment (Note that the programs can be interrupted at any time by pressing the Esc key).

```
cd ~/PCBS/experiments/xpy_parity_decision
python parity_feedback.py
```

This should run a small experiment where the participant must check the parity (odd or even) of numbers.

That's all folks !

3.9 Appendices

3.10 Keep your local copy of the course material up to date

The course materials are often updated. To make sure you have the latest version, you can synchronize your local copy with the github repository <http://github.com/chrplr/PCBS>, with the commands:

```
cd ~/PCBS
git pull
```

Notes:

- if the PCBS directory is not in your home directory (-), you will need to use the appropriate path in the first `cd` command.
- do not manually modify or create new files in the PCBS folder. If you do so, git will notice it and might prevent an automatic upgrade and ask you to ‘resolve conflicts’. If you get such a message, the simplest course of action, for beginners, is to delete the PCBS folder (or move it if you want to keep a copy of your modifications) and reissue the `git clone` command above to reload the full folder.)

3.11 Basic surviving skill: how to enter command lines in a Terminal

Watch the video at <https://www.youtube.com/watch?v=2yhCWvBt7ZE&t> and try to perform the activities in it (the instructions also work for Mac or Linux: you just need to open a standard Terminal while in Windows you start ‘Git Bash’). Note: the game scripts mentioned in the video are available at <https://github.com/chrplr/PCBS/tree/master/games/games.zip>

For the moment, you mostly need to know the following three commands:

- `ls`: list the content of the current working directory
- `pwd`: path of current working directory
- `cd`: change directory

Read about them in http://linuxcommand.sourceforge.net/lc3_lts0020.php

Here are some resources to learn more about how to control your computer from a terminal:

- Learning the Shell http://linuxcommand.org/lc3_learning_the_shell.php
- OpenClassRoom : <https://openclassrooms.com/en/courses/43538-reprenez-le-controle-a-laide-de-linux/37813-la-console-ca-se-mange>

STARTING FROM SCRATCH

To begin our journey into programming, we will use [Scratch](https://scratch.mit.edu), a system developed the MIT media lab to teach how to program to kids.

A great advantage of [Scratch](https://scratch.mit.edu) is that programs are created using a graphical interface, preventing *syntactic errors*. Thus, you can learn the language without having to learn its grammar!

One can either work online at https://scratch.mit.edu/projects/editor/?tip_bar=home or offline, by downloading [Scratch](https://scratch.mit.edu) at

- https://scratch.mit.edu/scratch_1.4/ (version 1)
- <https://scratch.mit.edu/scratch2download/> (version 2)

We recommend the reader to run the tutorial “[Getting Started with Scratch](#)”

4.1 First steps

4.1.1 Program 001

In the motion group, take the instruction `turn 15 degrees` and drag it onto the Scripts panel.



Double-Click repeatedly on the block `turn 15 degrees`, you should see the cat (sprite 1) rotate.

In Scratch, when one double-clicks an instruction in the Scripts panel, the computer **executes** it.

4.1.2 Program 002

Drag the instruction `move 10 steps` from the motion group, and add it to the bottom of the instruction `turn 15 degrees`. Change the value 10 into 50.



You have just created a **block** of instructions, that is, your first **program** or **script**, Bravo!

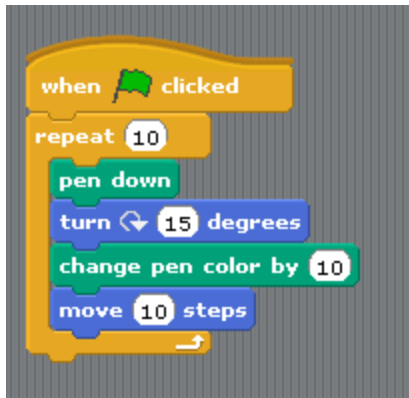
- Double-Click on the block and see the sprite moving.
- Note that inside a block, instructions are executed *sequentially*, one after the other. **Can you prove it?**
- Experiment with changing the **argument** of the instruction `move` (Tip: to clear the drawing area, move the instruction `pen/clear` to the script window and execute it)

4.1.3 Program 003

Click on the pen group, and add `pen down` at the top of the block.



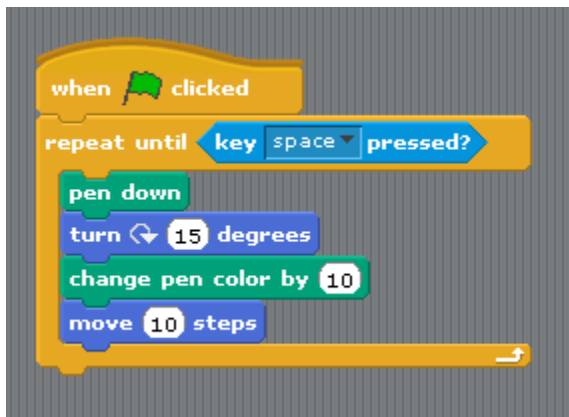
Run it.



- Clicking on the green flag will execute the block of instructions
- The Repeat instruction executes the inner block of instruction a number of times specified as an argument. This is called a **loop**
- Adjust the parameter of the Repeat instruction so that the sprite draws a full circle when you click once on the green flag.
- Replace the repeat instruction by **forever**.

4.3.2 Repeat until

Modify the script as follows:



Tip: the condition key space pressed? is in the Sensing group.

This illustrates a **repeat...until loop**: the inner block is executed until the **condition** is satisfied.

4.4 Two sprites

Add a new sprite, and duplicate the script from sprite1. Click on the green flag. You should see the two sprites running in circles.

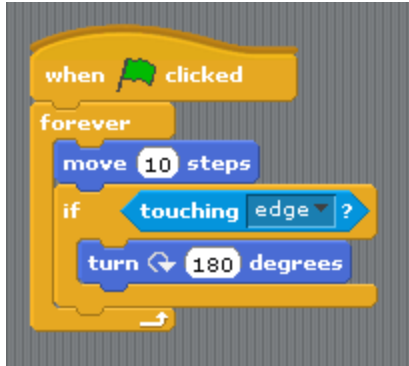


Remark that the scripts associated to the two sprites run in *parallel* (rather than sequentially).

4.5 Conditional execution or branching

Create a new scratch project, and change the costume of the sprite into a ball.

Then write and execute the following script.



You should see the ball bounce on the edges.

4.6 First series of exercises

1. With Scratch, use the instructions “pen down” and “move” and “turn” to (a) make the cat draw a square (with sides measuring 100 steps) (b) draw an hexagon (c) draw a circle
2. Using the Control/Forever, make the cat turn continuously along a circle.
3. Bouncing ball
 - Delete the cat. Using new sprite/open, add a ball.
 - Make the ball move automatically horizontally from left to right and bounce when it touches an edge (tip: use Control/forever)
 - Make the ball follow the mouse.
 - Add a second ball that follows the first.
4. Create a script that asks for your name and then displays “Hello !”. Tip: use the instructions sensing/ask, looks/say and operator/join and the variable sensing/answer.

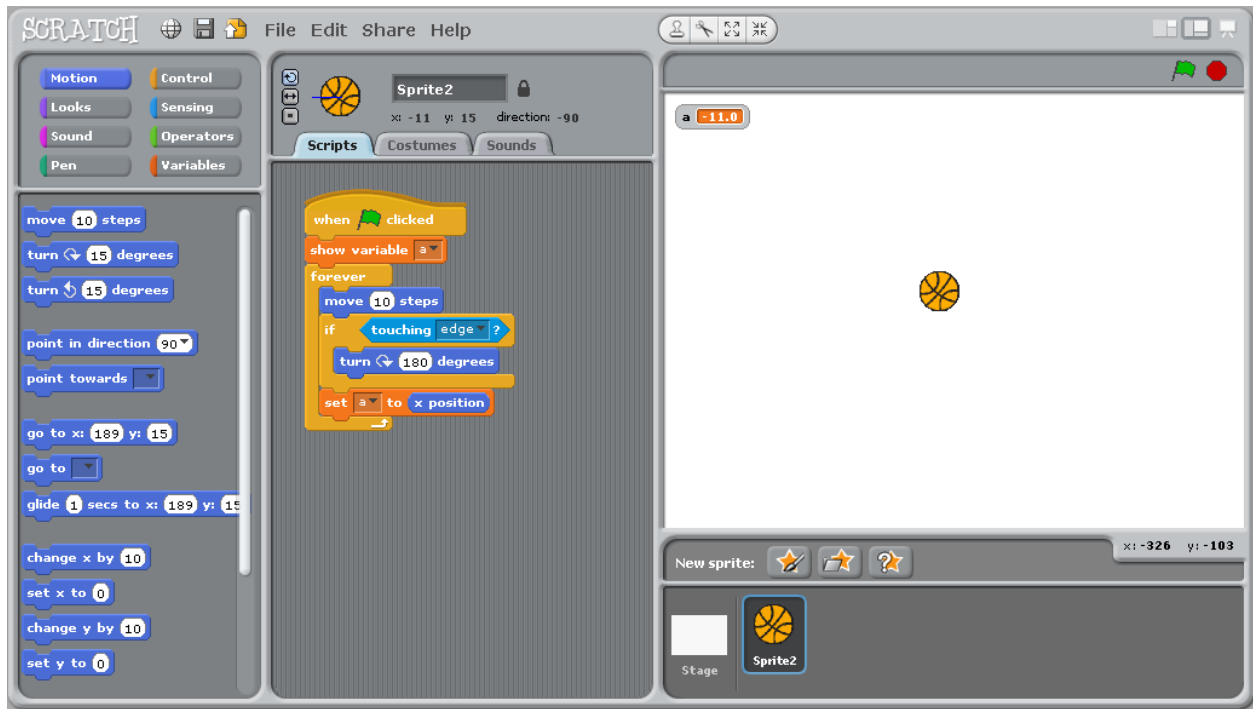
...



...

4.7 Variables

Using the group variable, we are going to create a **variable** *a* and make it display continuously the x-coordinate of the ball.



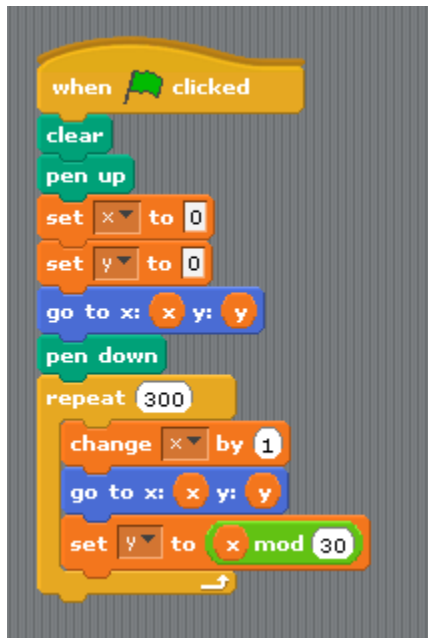
The concept of **variable** is very important. You can think of it as a name for a object that can change (here the object is a number).

Now study the following script:



The loop is executed 100 times. Each time, the value of the variable *a* is incremented by 1, and is used to compute new

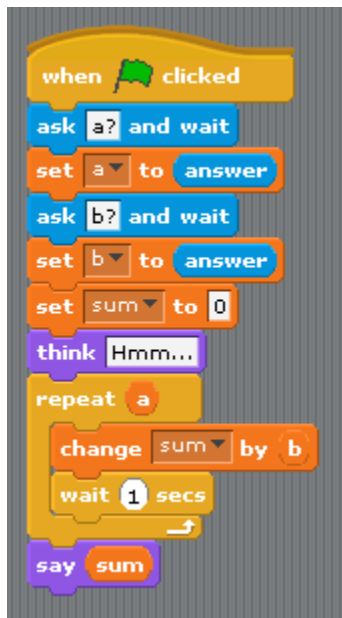
x and y coordinates where to sprite is instructed to moved to.



4.8 Second series of exercises

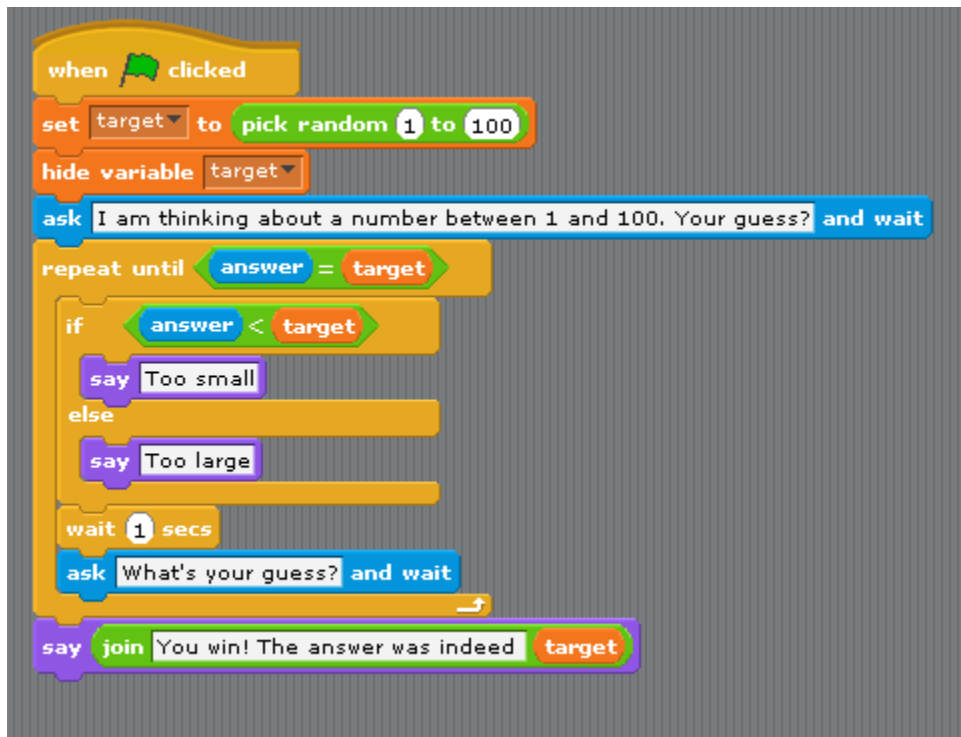
4.8.1 Multiplication

“Multiply by adding”: Write a program that reads in two integer numbers and displays their sum.



4.8.2 Guess a number

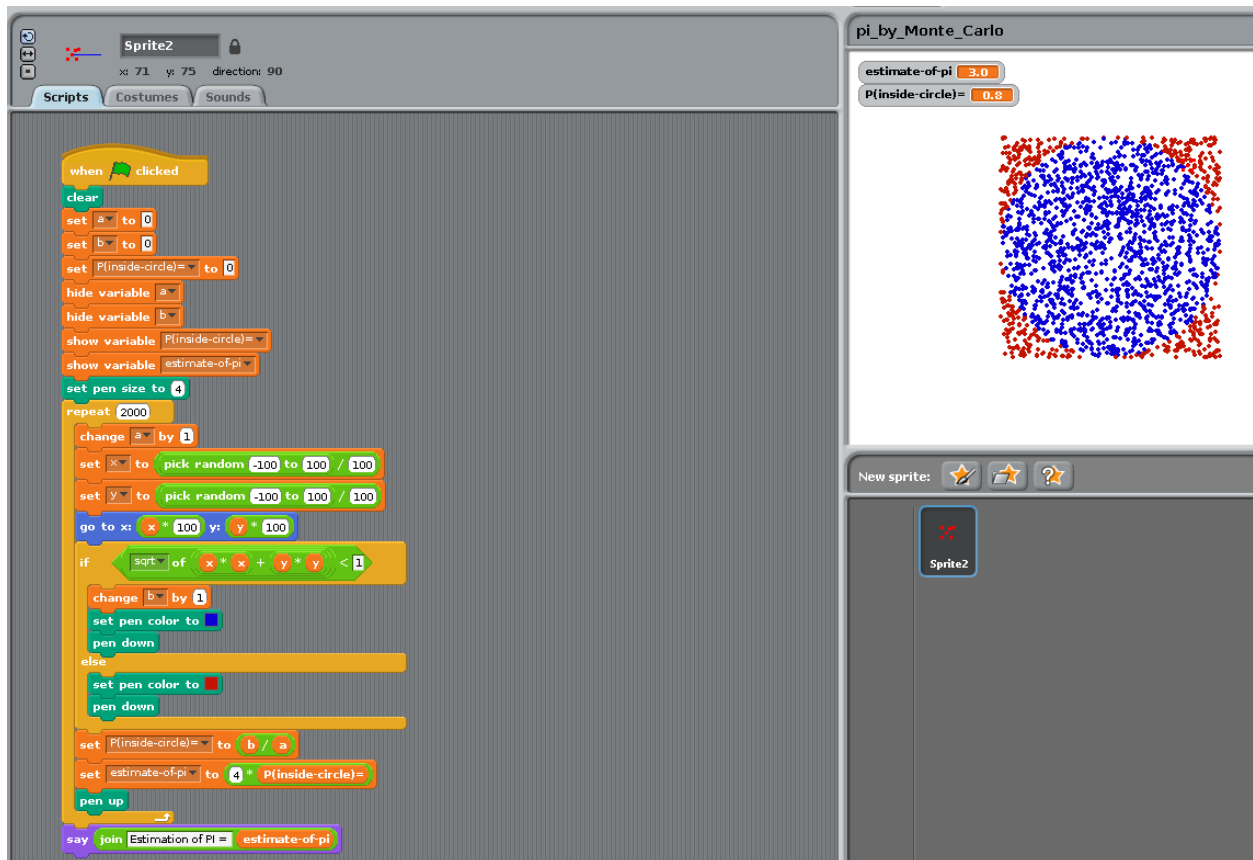
“Guess a number”. Make Scratch pick up a random number in the interval [1,100], and loop asking you for a guess and reply either too low, too high, or you win! depending on your answer.



4.8.3 Estimate PI

3. We are going to estimate the number PI by a Monte Carlo method.:

- Repeatedly (e.g. 2000 times) picks up two random numbers on the interval [-1, 1]. This corresponds to a dot inside a square of size 2x2.
- Count how many times the dot falls within the circle of radius 1 centered on the origin (Pythagore helps you here: the dot is within the circle iff $(x * x + y * y) < 1$.)
- The proportion of dots falling within the circle, multiplied by four (the area of the square), is an estimate of the area of the disk, that is, the number pi.



FUN PROGRAMMING LANGUAGE

If you liked Scratch, but found it too much child-oriented, you may enjoy:

- [Processing](#) a language for learning how to code within the context of the visual arts.
- [Sonic Pi](#) a code-based music creation and performance tool.

Give them a look: these are two great programming tools to create visual and audio effects and, being designed for non programmers, they are very good to learn the bases of programming without pain.

INTERACTING WITH A COMPUTER (IN A NUTSHELL)

Contents

- *Interacting with a computer (in a Nutshell)*
 - *The shell*
 - * *Why learn the command shell?*
 - *Disks, Directories and files*
 - *filenames, directory structure*
 - *Working directory. Absolute pathnames vs. relative pathnames (..)*
 - *What is the PATH?*
 - *What is a library (or module/package)?*

Nowadays, most interactions with the operating system (the “conductor of the orchestra” of the programs installed on your computer) are achieved by clicking on icons that represents programs or data files. When you click on a program, you start it; when you click on a document, you open it within its associated program.

The is convenient but limited. Imagine you could “talk” to your computer and ask it: “Slave ! find all the jpg graphics files in this folder, make copy that you reduce to size 512x512 pixels, and place them in a new folder”. This is actually possible, by typing a single line in a Terminal running the bash shell:

```
mkdir thumbnails; for f in *.jpg; do convert "$f" -resize 512x512 thumbnails/"${f%.jpg}-  
↪512x512.jpg"; done
```

6.1 The shell

Inside the terminal, you are interacting with a program which runs an infinite loop where it prints a ‘prompt’, wait for you to type some instructions and, when you press the Enter or Return Key, interprets the line and execute the instructions.

This program is called a **Shell**. Various Shells exist: bash, zsh, powershell. they speak slightly different languages.

There even is a shell that speaks python: it is called `ipython` (for interactive python).

One difficulty is that you need to know the language of the shell and, also, the available programs on your computer. By contrast with Graphical User Interfaces that display Windows/Icons/Menus, **Textual shells** have a poor ergonomoy in that there is no visual way of discovering the potential actions. Yet, because they are languages providing variables, loops, ... shells facilitate the automation of tasks.

For example, to create 20 directories in a single bash command under linux:

```
for f in 01 02 03 04 05 06 07 08 09 10; do mkdir -p subject_$f/data subject_$f/results;
done
```

The good news is: in these lectures, you will need to use only three shell commands: * `cd` (change directory) * `ls` (list) * `pwd` (path of working directory).

I recommend that you read the web page at http://linuxcommand.org/lc3_lts0020.php which provides very clear explanations.

You do not have to learn more about the shell for these lecture, but learning more about it may be good idea in the long run.

6.1.1 Why learn the command shell?

“To properly understand the role of a shell, it’s necessary to visualize what a computer does for you. Basically, a computer is a tool; in order to use that tool, you must tell it what to do—or give it “commands.” These commands take many forms, such as clicking with a mouse on certain parts of the screen. But that is only one form of command input.

By far the most versatile way to express what you want the computer to do is by using an abbreviated language called script. In script, instead of telling the computer, “list my files, please”, one writes a standard abbreviated command word—‘ls’. Typing ‘ls’ in a command shell is a script way of telling the computer to list your files.¹

The real flexibility of this approach is apparent only when you realize that there are many, many different ways to list files. Perhaps you want them sorted by name, sorted by date, in reverse order, or grouped by type. Most graphical browsers have simple ways to express this. But what about showing only a few files, or only files that meet a certain criteria? In very complex and specific situations, the request becomes too difficult to express using a mouse or pointing device. It is just these kinds of requests that are easily solved using a command shell.

For example, what if you want to list every Word file on your hard drive, larger than 100 kilobytes in size, and which hasn’t been looked at in over six months? That is a good candidate list for deletion, when you go to clean up your hard drive. But have you ever tried asking your computer for such a list? There is no way to do it! At least, not without using a command shell.

The role of a command shell is to give you more control over what your computer does for you. Not everyone needs this amount of control, and it does come at a cost: Learning the necessary script commands to express what you want done. A complicated query, such as the example above, takes time to learn. But if you find yourself using your computer frequently enough, it is more than worthwhile in the long run. Any tool you use often deserves the time spent learning to master it.”

(Extracted from Emacs’ eshell documentation)

To learn more about how to control the computer by interacting with the shell, I can only highly recommend two resources:

- Learning the Shell http://linuxcommand.org/lc3_learning_the_shell.php
- OpenClassRoom <https://openclassrooms.com/en/courses/43538-reprenez-le-controle-a-laide-de-linux/37813-la-console-ca-se-mange>

6.2 Disks, Directories and files

Most computers (not all) have two kinds of memories:

- volatile, fast, memory, which is cleared when the computer is switched off (processor's caches, RAM)
- 'permanent', slow, memory, which is not erased when the computer is switched off (DISKS, Flashdrives (=solid-state drives))

The unit of storage is the **file**.

Files are nothing but blobs of bits stored "sequentially" on disks.

A first file could be stored between location 234 and 256, a second file could be stored at location 456 on your disk.

6.3 filenames, directory structure

To access a file, one would need to know its location on the disk. To simplify human users' life, the OS provide a system of "pointers", that is **filenames**, organised in directories.

To help users further, the directories are organised in a hierarchical structure: a directory can contain filenames and other (sub)directories. The top-level directory is called the **root**.

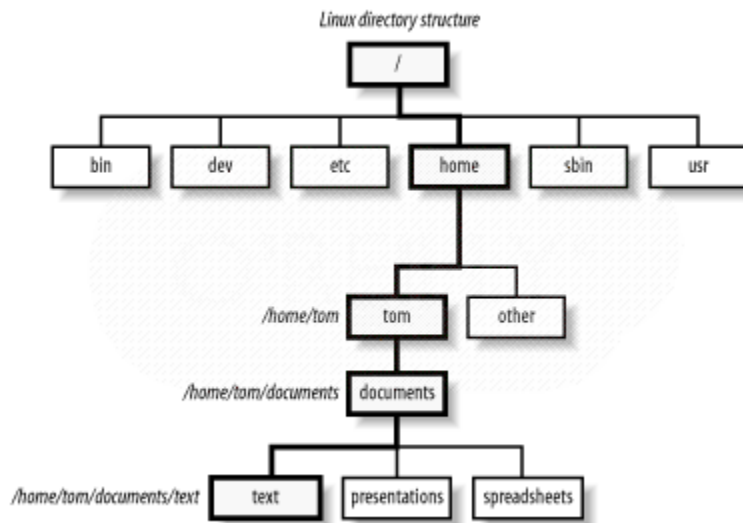


Fig. 1: Linux directory structure

To locate a file, you must know:

- its location in the directory structure
- its basename

See [absolute or relative pathnames](#)

Remark: a given file can have several names in the same or various directories (remember: a filename is nothing but a link between a human readable character string to a location on the disk)

6.4 Working directory. Absolute pathnames vs. relative pathnames (..)

It would be tedious to always have to specify the full path of a files (that is, the list of all subdirs from the root)

Here comes the notion of **working directory**: A running program has a working directory and filenames can be specified **relative** to this directory.

Suppose you want to access the file pointed to by `/users/pallier/documents/thesis.pdf`. If the current working directory is `/users/pallier`, you can just use `documents/thesis.pdf` (notice the absence of `/` at the beginning).

To determine the current working directory in a shell, list its content, and change it:

- under bash (or ipython):

```
pwd
ls
cd Documents
```

6.5 What is the PATH?

A command can simply consist of a program's name: typing it and pressing Enter will start the program.

The shell knows where to look for programs thanks to an **environment variable** called `PATH`.

The `PATH` variable lists all the directories that contains programs. Try the following commands:

```
echo $PATH
which ls
which python
```

Under bash, to add new directories to the `PATH`:

```
export PATH="newdirectory": "$PATH"
```

For example, under Git Bash for Windows, to be able to start sublime text from the command line, by just typing `subl`, you must add its folder to the `PATH`, as follows:

```
export PATH="/c/Program Files/SublimeText 3/": "$PATH"
```

To make this setting permanent, you must copy this line within a file `.bash_profile` in your home directory.

6.6 What is a library (or module/package)?

A library is a set of new functions that extend a language.

Libraries can be used simultaneously by several processes.

E.g. the function `@@sqrt@@` can be defined once, and called by several programs.

In Python, one uses `import` to be able to access functions from a library (a.k.a. module), for example:

```
import math
math.sqrt(2)
```

RUNNING PYTHON

Contents

- *Running Python*
 - *Running a python script from the command line*
 - *Testing a short piece of python code*
 - *Write code with a text editor (Edit-run cycle)*
 - *Using an Integrated Development Environment (IDE)*
 - *Perform an interactive data analysis with jupyter-notebook or jupyter-lab*
 - *Developing in Python with Emacs*

There are several ways to work with Python, each of which is best adapted to a certain type of task:

- To write reusable python scripts, use a code editor (e.g. `subl`` (Sublime Text)), or an Integrated Development Environment (e.g. `spyder`), save the script and run it from a shell command line inside a Terminal.
- To quickly test short pieces of code in an interactive manner, or to access the documentation of some functions or modules, open a terminal and run `ipython`
- To write a data analysis report use `jupyter notebook`

7.1 Running a python script from the command line

To run a python script, that is, a text file containing python code, you need to:

1. Open a Terminal
 - MacOSX: Search for `terminal` in Spotlight.
 - **Windows: Launch Git Bash (This assumes that you have installed Git for windows as described in *Instructions for software installation*)**
 - **Linux: Launch Terminal from your application menu or use `Ctrl-Alt-T` in Gnome, Xfce or `Win+Return` in i3.**
2. If the script is not located in your home directory, use the `cd` command to navigate to the directory that contains the script (to learn about this, read [Navigation in the shell](#)).
3. Type `python` followed by the script's filename, then press the `return/enter` key.

For example, let us suppose that you want to execute a script named `matches.py` located in a subdirectory `PCBS/games` of your home directory. Open a terminal and type:

```
cd PCBS/games
python matches.py
```

Remarks:

- When the script has run to completion, you will be back to interacting with [the shell](#).
- If you need to interrupt a running python script, you can press `Ctrl-C` in the Terminal.
- You can specify [absolute](#) or [relative pathnames](#) to specify the location of the script:

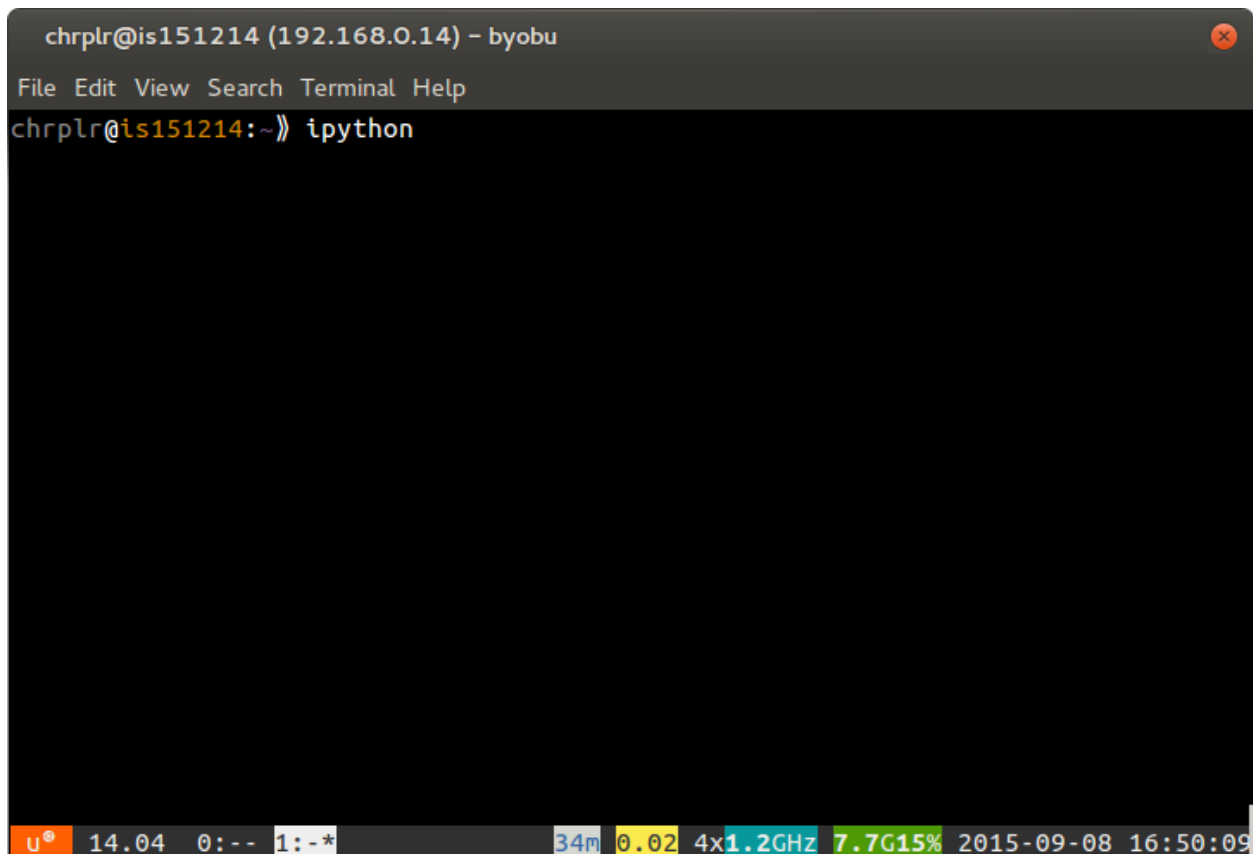
```
python ~/PCBS/games/matches.py
```

Note: The directory from which you started the script is the **current working directory**: it will be the root for any relative path names that may be used within the script.

- A good practice is to systematically take a glance to the code *before* running it. Depending on your system, you might be able to do this with the commands `cat filename.py` or `less filename.py` or `micro filename.py`.

7.2 Testing a short piece of python code

To quickly test some Python code, type `ipython` (“interactive Python”) on a command line.



Then press ‘Return’; you should obtain a should display like the following:

```
chrplr@is151214 (192.168.0.14) - byobu
File Edit View Search Terminal Help
chrplr@is151214:~» ipython
Python 2.7.6 (default, Jun 22 2015, 17:58:13)
Type "copyright", "credits" or "license" for more information.

IPython 1.2.1 -- An enhanced Interactive Python.
?      -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help    -> Python's own help system.
object? -> Details about 'object', use 'object??' for extra details.

In [1]:
```

A blinking cursor after [1]: indicates that ipython is ready and waiting for you to enter a Python statement that it will execute as soon as you press the “Return” key. For example, try:

```
2 ** 5
2 ** 64
```

```
import turtle
turtle.circle(50)
turtle.forward(100)
turtle.circle(50)

turtle.right(90)
turtle.forward(100)
turtle.right(90)
turtle.heading()
```

```
import matplotlib.pyplot as plt
import numpy as np
t = np.linspace(0, 30, num=3001)
plt.plot(t, np.sin(t))
```

A Window should open with a graphical representation of the sine function, You can press ‘q’ in this Window to close it.

It is possible to execute a python script from within ipython. While in ipython, try:

```
pwd
cd PCBS/games
%run matches.py
```

Finally, To quit `ipython`, type `quit()` or press `Ctrl-D`.

This approach is fine if you need to quickly test an idea. But as soon as you quit `ipython`, you lose all what you have done.

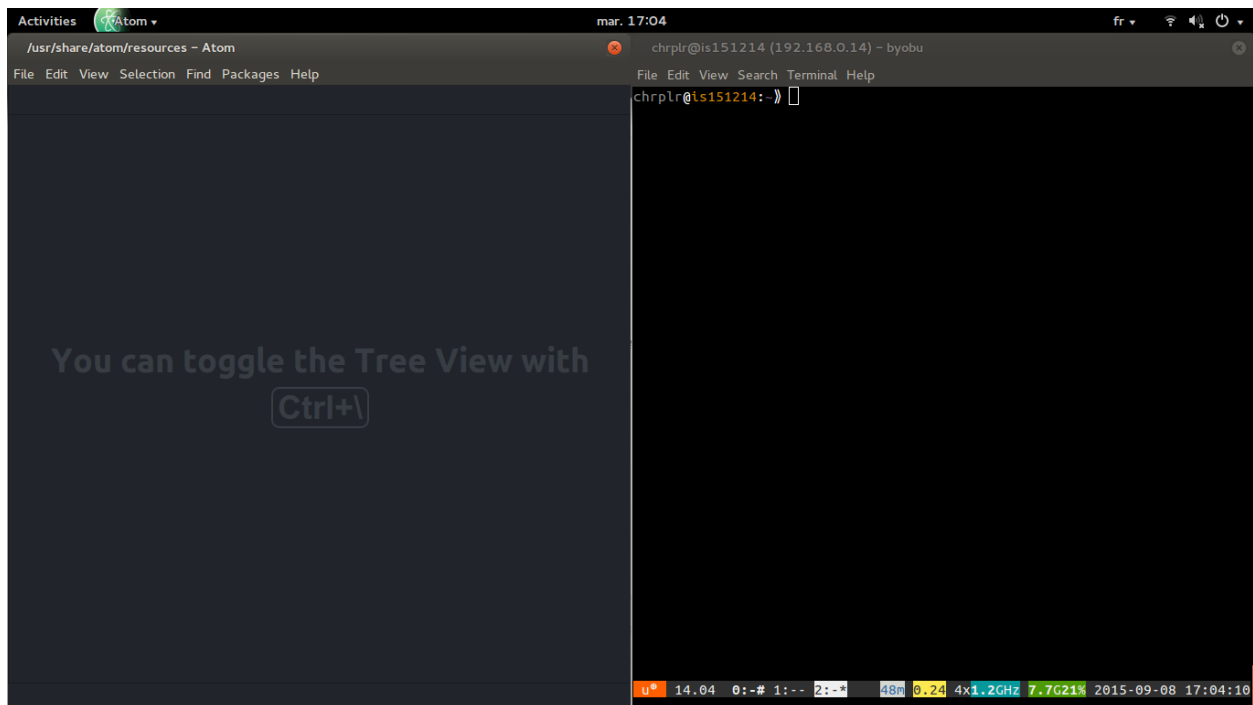
To keep track of your work, you need to use a code editor and the *Edit-Run* approach.

7.3 Write code with a text editor (Edit-run cycle)

A script is nothing but a pure text file, that is, a sequence of characters.

A Python script is written with a **text editor**, saved on the disk, and then executed.

1. Open a Text-Editor (e.g. Sublime Text) and a Terminal window side-by-side:



2. Create a New File in the Editor and enter the following text:

```
import turtle
turtle.forward(50)
turtle.left(120)
turtle.forward(100)
turtle.left(120)
turtle.forward(100)
turtle.left(120)
turtle.forward(50)
```

3. Using 'File/Save as', save the this text under the filename `myscript.py` in your personal (home) directory

- *run* with a python interpreter, by typing `python myscript.py` on a command line of the Terminal. Try it now.

Important: you must make sure that the *current working directory* of the terminal is the same directory where the file `myscript.py` has been saved. Otherwise, you will get an error message such as ‘No such file or directory’. To fix this problem, you must use the `cd` command to [navigate the directory structure](#).

Remarks:

- You can learn more about Turtle graphics by reading the documentation at <https://docs.python.org/2/library/turtle.html>
- WINDOWS Only: To be able to start ‘Sublime Text’ from the command line by just typing `subl`, copy the following command:

```
export PATH="/c/Program Files/SublimeText 3/":"$PATH"
```

in the file `$HOME/.bash_profile` (create it if necessary)

7.4 Using an Integrated Development Environment (IDE)

Some people like to work within a single application and avoid going back and forth from the text editor to the terminal. The Anaconda Python distribution comes with an integrated development environment (IDE), *Spyder*, which provides an environment somewhat similar to the MATLAB IDE. *PyCharm* and *Microsoft Visual Code* are two other popular (and more powerful) IDEs.

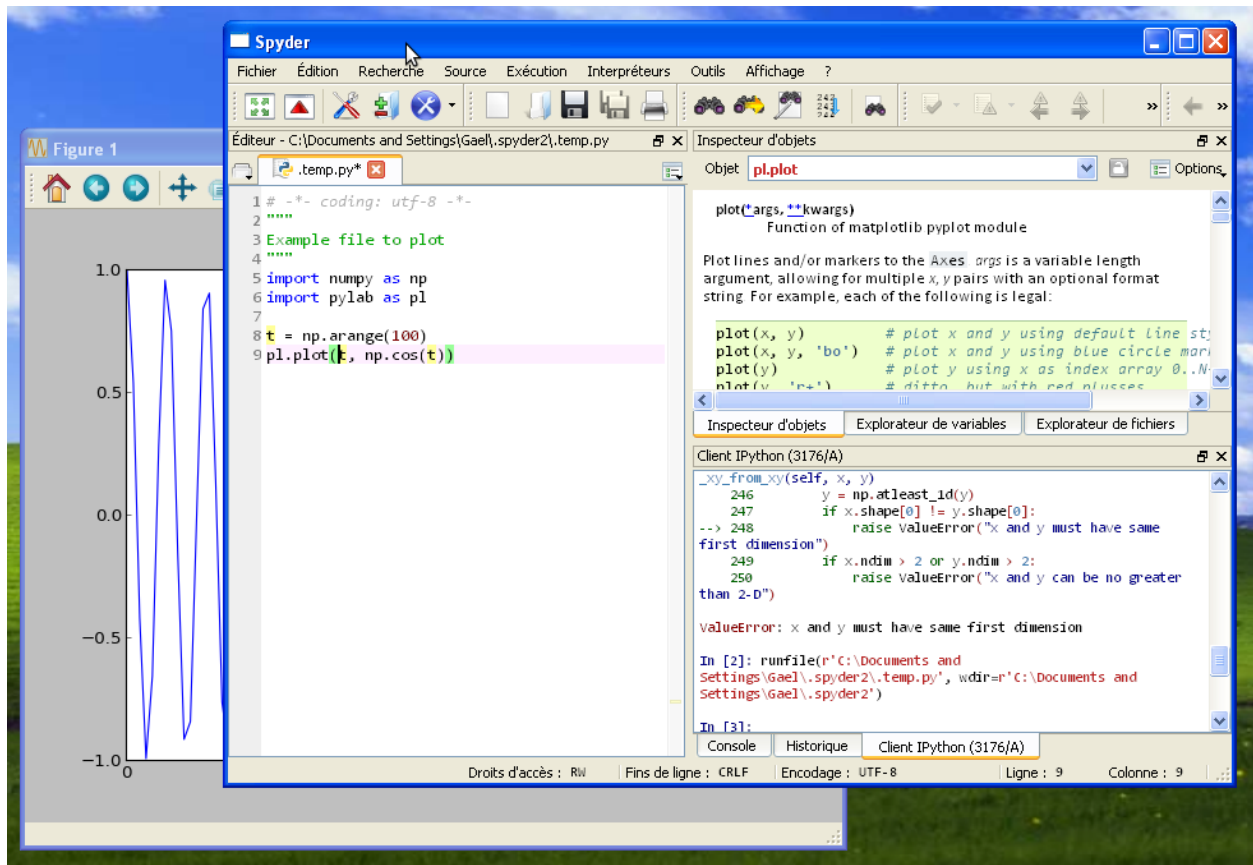


Fig. 1: The “Spyder” Integrated Development Environment

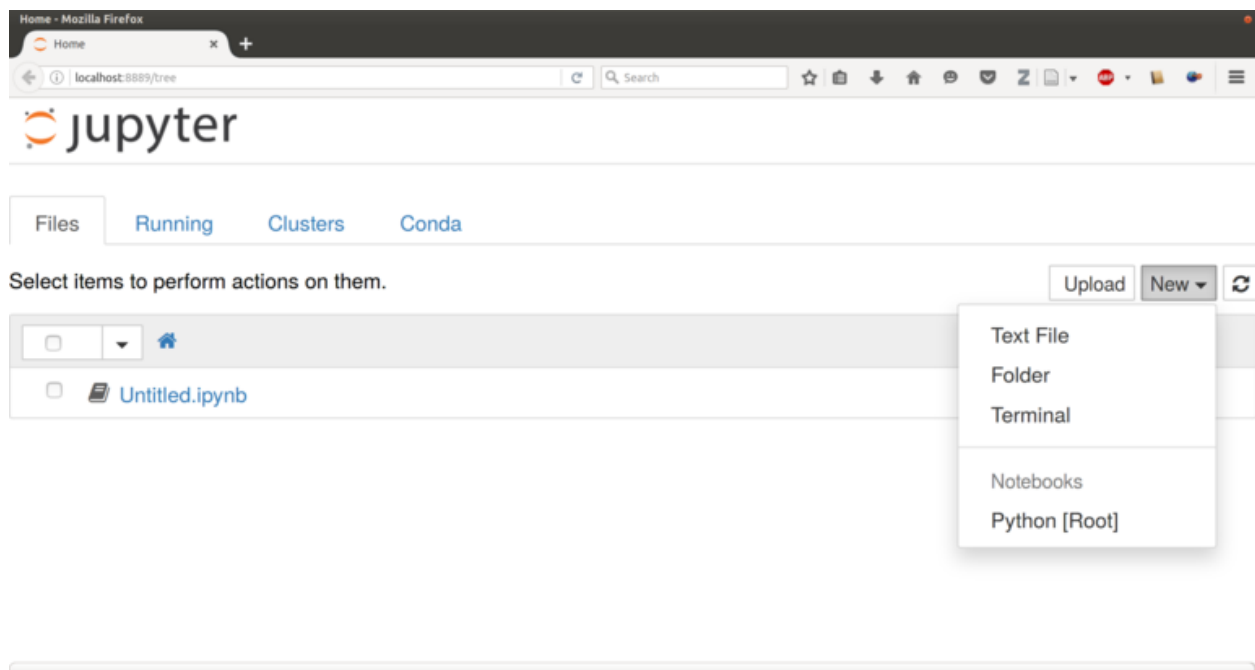
Visual Code, PyCharm, Spyder... are very nice IDEs but you should not use them to run python scripts that open new graphics windows (e.g. scripts using `tkinter`, `pygame`, ...) because, when such scripts crash, they can leave the IDE in an unstable state. It is always safer to run a script directly from the command line in a terminal windows.

One commendable approach is to use an IDE to edit python code, but use the command line to run the scripts.

7.5 Perform an interactive data analysis with jupyter-notebook or jupyter-lab

To perform data analyses and produce nicely formatted reports, one can use `jupyter-notebook` or `jupyter-lab` (see <https://jupyter.org/>).

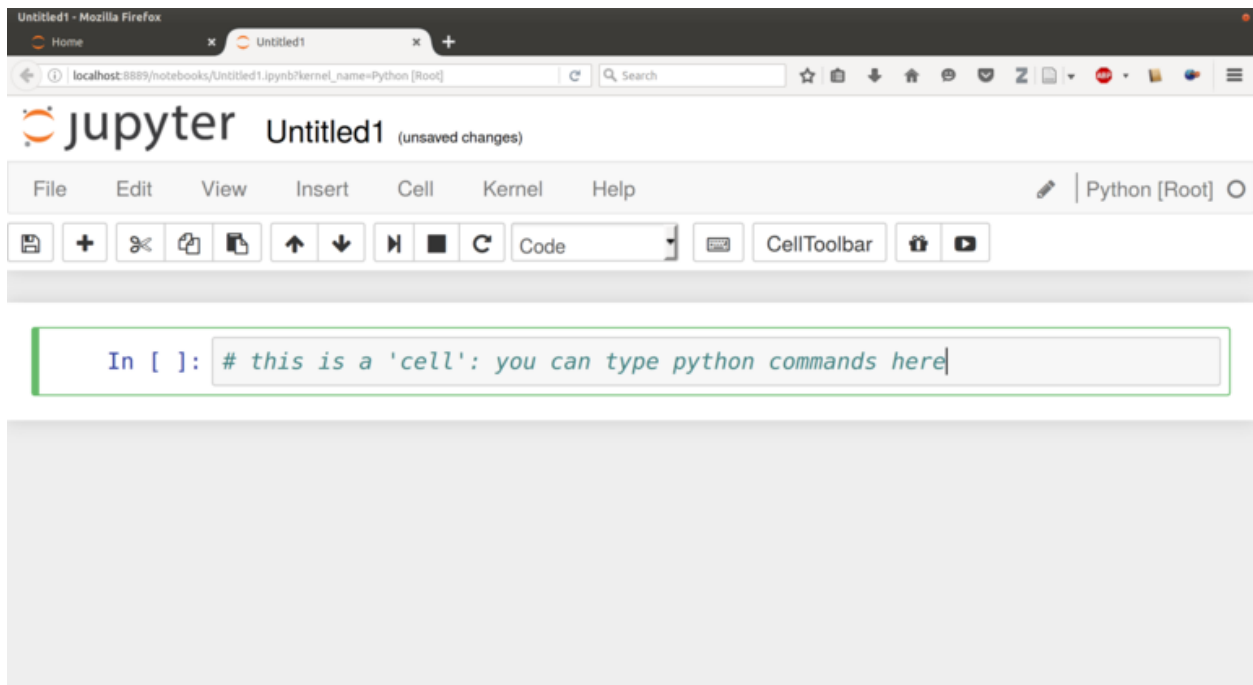
In practice, launch Jupyter Notebook from the Start Menu/Anaconda3 (in Windows) or type `jupyter notebook` in a terminal (Linux, MacOS). The “Jupyter homepage” should then open in your browser:



Clicking **New** and selecting **Python [root]** will open a new tab containing a notebook where you can enter python code inside so-called ‘cells’. To execute the code in a cell, just move the cursor there and press **Ctrl+Enter**

A nice feature of the Jupyter notebooks is persistence, i.e. they are saved automatically (in `.ipynb` files) and you can go on working on the same notebook when you reopen it. This is also very handy, for example, to send a data analysis report by email.

Jupyter’s documentation is available at <http://jupyter.readthedocs.io/en/latest/index.html>



7.6 Developping in Python with Emacs

Action	Shortcut	Function
Comment or Uncomment cleverly	M-;	comment-dwim
Indent	C->	python-indent-shift-right
Unindent	C-<	python-indent-shift-left
Navigate the function definitions	C-c C-j	imenu
Move backward to block	M-a	python-nav-backward-block
Move forward to block	M-e	python-nav-forward-block
Checking for errors, etc.		flymake
Checking for errors, etc.		flycheck
Reformat code to best practices		yapify
Launch the Python Debugger (Pdb)		pdb

Note: I am actually using [Spacemacs](#) with the [python](#) layer

See also <https://realpython.com/emacs-the-best-python-editor/>

CODING EXERCISES

Contents

- *Coding Exercises*
 - *Flow control*
 - * *Shining*
 - * *Multiplication tables*
 - * *Taxis*
 - * *Estimation of PI by a Monte-Carlo method*
 - * *Computer-guess-a-number*
 - *Lists*
 - * *Prime numbers*
 - * *Pascal triangle*
 - *Functions*
 - * *Convert temperatures from Fahrenheit to Celcius and vice-versa.*
 - * *Permutations*
 - * *Eight queens puzzle*
 - *Strings*
 - * *Pseudowords*
 - *Dictionaries*
 - * *unique*
 - * *word count*
 - * *Anagrams*
 - *File reading and writing*
 - * *head*
 - * *tail*
 - * *string-detector*
 - * *Kaprekar numbers*

- * *RPN Calculator*
- * *Rodrego-simulator*
- * *Cellular automata*
- * *Analysis of a Signal Detection Experiment*

(Note: Solutions to most exercises are available in <https://github.com/chrplr/PCBS/tree/master/coding-exercises>)

8.1 Flow control

You can read about [loops in Python](#), and more generally about [flow control](#).

8.1.1 Shining

Write a python script that prints 1000 times the line `All work and no play makes Jack a dull boy.`

Check a solution at `shining.py`

8.1.2 Multiplication tables

Write a script that displays the tables of multiplication from 1 to 10 as a table:

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100

Check out `multiplication_table.py` for a solution.

8.1.3 Taxis

Two taxi companies propose different pricing schemes:

- Company A charges 4.80€ plus 1.00€/km
- Company B charges 3.20€ plus 1.20€/km

Write code to find which company is the cheapest as a function of the distance to travel.

Check out `taxis.py`

8.1.4 Estimation of PI by a Monte-Carlo method

One way to estimate the value of π is to generate a large number of *random points* in the unit square and see how many fall within the unit circle; their proportion is an estimate of the area of the circle. See <https://academo.org/demos/estimating-pi-monte-carlo/>

Implement the proposed algorithm to estimate the value of π .

Check out `pi_monte_carlo.py`

8.1.5 Computer-guess-a-number

Read [chapter 3 of Invent your own games with Python](#) where the author presents a game where the computer chooses a random number that the user must guess. Study the code.

Now, your task is to write another program, where the roles are inverted: the computer tries to guess a number that the user has in mind. The computer proposes a number and the user answers with '+' (the number he has in mind is larger), '-' (if it is smaller), 'y' (if the guess is correct)

Check a solution at `computer-guess-a-number.py`

8.2 Lists

These exercises require list manipulations. If you do not know Lists in Python, you can read:

- [Python Lists](#)
- [List comprehensions](#)
- <https://automatetheboringstuff.com/2e/chapter4/>

Try to solve the following exercises:

- Given a list of numbers, print their sum
- Given a list of numbers, print their product
- Given a list of numbers, print the sum of their squares
- Given a list of numbers, print the largest one.
- Given a list of numbers, print the second largest one.

After you have tried to solve these problems, you can check `lists.py`

8.2.1 Prime numbers

Write a script that lists all prime numbers between 1 and 10000 (A prime number is a integer that has no divisors except 1 and itself). You can use the following function:

```
def is_factor(d, n):
    """ True if `d` is a divisor of `n` """
    return n % d == 0
```

Check a solution at `prime-numbers.py`

8.2.2 Pascal triangle

Write a program that prints the first N rows of Pascal's triangle (see <https://www.youtube.com/watch?v=XMriWTvPXHI>). For example:

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
1 8 28 56 70 56 28 8 1
1 9 36 84 126 126 84 36 9 1
```

To solve this problem, one solution is to store the values of the current line in a Python list, and write a function that given a list as an argument, calculates and returns the following line in a new list.

Proposed solutions:

- `Pascal-triangle_v1.py`
- `Pascal-triangle_v2.py`

8.3 Functions

Read about *functions* in Python: - https://www.w3schools.com/python/python_functions.asp - <https://automatetheboringstuff.com/2e/chapter3/>.

8.3.1 Convert temperatures from Fahrenheit to Celcius and vice-versa.

- Read <https://en.wikipedia.org/wiki/Fahrenheit> and write a function that converts a temperature from Fahrenheit to Celsius, and another one that converts from Celsius to Fahrenheit
- Add code that reads temperatures from the standard input and print the converted numbers.

A solution is available here: `Fahrenheit_celsius.py`.

8.3.2 Permutations

Generate all the permutations of a set, e.g. $(1..n)$.

Note: This is an advanced exercise, which requires mastery of recursive functions (functions that call themselves)

A solution is proposed at `generate_all_permutations.py`.

To run it:

```
python generate_all_permutations.py 4
```

8.3.3 Eight queens puzzle

The eight queens puzzle is the problem of placing eight chess queens on an 8×8 chessboard so that no two queens threaten each other; thus, a solution requires that no two queens share the same row, column, or diagonal. See https://en.wikipedia.org/wiki/Eight_queens_puzzle

As there can be only one queen per column and per row, a winning solution can be represented by a set of 8 numbers, one per line, which represent the column in which there is a queen. Because the columns must be different, the solutions are a subset of the permutations of 8 numbers. We just have to check that no two queens are in the same diagonal.

In Python, you can get all the permutations of a set, with the `permutations` function from the `itertools` module

```
from itertools import permutations
list(permutations(range(3)))
```

For a solution to the eight queens problem, check out <https://code.activestate.com/recipes/576647-eight-queens-six-lines/>

The code is available at `eight_queens.py`.

8.4 Strings

8.4.1 Pseudowords

- Search the internet to find out how to generate a random integer number in a interval in Python
- Read about strings in Python at <https://realpython.com/python-strings/>
- Write functions that generate pseudowords from words. The first function will delete a character in a random position from a string passed as argument. The second will insert a random character at a random position. The third will swap two characters at random location.
- If you know about file input/output (see <https://automatetheboringstuff.com/2e/chapter9/>), you can read a dictionary (e.g. <http://www.pallier.org/extra/liste.de.mots.francais.frgut.txt>) and use it to filter out any actual words.

8.5 Dictionaries

8.5.1 unique

Given a list of words, print how many different words are in that list (hint: use a dictionary or a set)

```
liste = ['bonjour', 'chat', 'chien', 'bonjour']

n = 0
d = dict()
for e in liste:
    if not e in d.keys():
        d[e] = 1
        n = n + 1
print(n)

print(len(set(liste))) # shortest solution using a set
```

8.5.2 word count

Given a list of words, count the number of times each word appears in the list. Eg. [Jim, Alan, Jim, Joe] -> Jim:2, Alan:1, Joe:1 (hint: use a dictionary)

```
liste = ['Jim', 'Alan', 'Jim', 'Joe']
counts = dict()
for word in liste:
    if word in counts.keys():
        counts[word] += 1
    else:
        counts[word] = 1
print(counts)
```

8.5.3 Anagrams

Two words are anagrams if they contain the same letters in different orders, e.g., *binary* and *brainy*.

- write a function that take two strings as arguments and returns True if they are anagrams.
- Given a list of words, print all subsets that form anagrams.

Check my solution at `anagrams.py`. Running:

```
python anagrams.py < liste.de.mots.francais.frgut.txt
```

will list *all* anagrams in French! (`liste.de.mots.francais.frgut.txt` contains a list of French words)

8.6 File reading and writing

Read the chapter about files reading and writing at <https://automatetheboringstuff.com/2e/chapter9/>

8.6.1 head

Write a script that prints the first 10 lines of a file (or the whole file is it is less than 10 lines long).

```
with open('aga.txt', 'r', encoding='utf-8') as f:
    for l in f.readlines()[:10]:
        print(l, end='')
```

8.6.2 tail

Write a script that prints the last 10 lines of a file (or the whole file is it is less than 10 lines long).

```
with open('aga.txt', 'r', encoding='utf-8') as f:
    all_lines = f.readlines()
    for l in all_lines[-10:]:
        print(l, end='')
```

8.6.3 string-detector

Read [Chap. 8 of Automate the boring stuff](#).

Write a script that opens and read a text file, and print all the lines that contain a given target word, say, `cogmaster`.

Check out `search-file.py`

8.6.4 Kaprekar numbers

A Kaprekar number is a number whose decimal representation of the square can be cut into a left and a right part (no nil) such that the sum of these two parts gives the number initial. For example:

- **703 is a number of Kaprekar in base 10 because $703^2 = 494\ 209$ and that $494 + 209 = 703$.**
- **4879 is a number of Kaprekar in base 10 because $4879^2 = 23\ 804\ 641$ and $04641 + 238 = 4879$**

Write a program that returns all Kaprekar numbers between 1 and N.

Solution: `Kaprekar-numbers.py`

8.6.5 RPN Calculator

Write a reverse Polish arithmetic expression evaluator (See https://en.wikipedia.org/wiki/Reverse_Polish_notation).

E.g. `3 4 * 5 -` - evaluate to 7.

Solution: `rpn-calculator.py`

8.6.6 Rodrego-simulator

Write a Python script that simulates a [RodRego machine](#) with 10 registers. The program is stored in a string or in file that is read and then executed. Your program must contain a function which, given the 10 initial values of the registers, and the program, returns the new register values when the END command is reached.

Check two possible solutions: - `rodrego_maxime_caute.py` - `rodrego_christophe_pallier.py`

8.6.7 Cellular automata

Implement a 1-dimension [elementay cellular automata](#). (Further reading: https://en.wikipedia.org/wiki/A_New_Kind_of_Science)

Solution: `1d-ca.py`

8.6.8 Analysis of a Signal Detection Experiment

In a signal detection experiment, a faint stimulus (e.g. a faint sound or a faint visual target) is presented or not at each trial and the participant must indicate whether he has perceived it or not. There are four possible outcomes for each trial:

- A *hit* occurs when the participant correctly detects the target.
- A *miss* occurs when the target was there but the participant did not detect it.
- A *false alarm* occurs when the participant reports the presence of the target when it was not actually there.
- A *correct rejection* occurs when the participant correctly reports that the target was not present.

One defines;

- The *hit rate*, equal to $\text{\#hits} / (\text{\#hits} + \text{\#misses})$
- The *false alarm rate*, equal to $\text{\#false alarms} / (\text{\#false alarms} + \text{\# correct rejections})$

Let us first suppose that the data from a participant is represented as a string. This string represents a series of trials, each trial being represented by two characters indicating the trial type (1=target present, 0=target absent) and the participant's response (Y=target perceived, N=No target perceived). For example:

```
data = "0Y,0N,1Y,1Y,0N,0N,0Y,1Y,1Y"
```

Exercise:

- Write a function which, given such a string, returns the Hit rate and the False rate.
- Now, the results from different participants are stored in different files `subj*.dat` (download the files from <https://github.com/chrplr/PCBS/tree/master/coding-exercises/subjdat.zip>) Write a script that computes the hit rates and false alarms for each subject, and displays the group averages and standard deviations.

Solution `sdt.py`

AUTOMATA AND COMPUTERS

Contents

- *Automata and Computers*
 - *The Computational Theory of Mind*
 - *What is computation anyway ?*
 - *The ancestors of the computer: the automata*
 - *Formal description of an automaton*
 - *Examples of transition diagrams*
 - *What is a Computer?*
 - *Register machines*
 - * *Exercice: implementation of a Register machine*
 - *The Seven secrets of computers revealed*
 - *Programmable computers*
 - *Compilation and interpretation*
 - *Operating systems*
 - *What is a Terminal?*

9.1 The Computational Theory of Mind

Cognitive science founding disciplines :

- Psychology
- Linguistics
- Philosophy of mind
- Neurosciences
- Computer science (Cybernetics + AI)

Could a machine think? Could the mind itself be a machine?

Computers were designed to simulate the mental operations realized by a human mathematician performing a... computation (see Alan Turing)

The *Computational Theory of Mind* has been defended and attacked many times.

More readings:

- Zylberberg, Ariel, Stanislas Dehaene, Pieter R. Roelfsema, and Mariano Sigman. 2011. “The Human Turing Machine: A Neural Framework for Mental Programs.” *Trends in Cognitive Sciences*
- Van Gelder, Tim. 1995. “What Might Cognition Be, If Not Computation?.” *Journal of Philosophy* 92 (7): 345–81.
- Jerry Fodor *The Mind does not work that way*
- Douglas Hofstadter *Gödel, Escher & Bach: an eternal golden braid* and *I am a strange loop*

9.2 What is computation anyway ?

One common answer is:

“Computation is what a Turing machine can do”

But what is a Turing machine?

9.3 The ancestors of the computer: the automata

An **automaton** is a device designed to automatically follow a predetermined sequence of operations.

(see Descartes’ *Les Animaux Machines* [Lettre au Marquis de Newcastle](#))

9.4 Formal description of an automaton

At a abstract level, an automaton can be formally described by:

- a set of internal **states**
- a **transition** table (or diagram) that describes the **events** that lead to changes from one state to the other state.

9.5 Examples of transition diagrams

A Finite State Automaton can be used to generate or recognize regular languages.

In Formal Language Theory, a language is a set of strings.

Examples:

- { a, aa, ab, ba, aab, bab, ... }
- { ha!, haha!, hahaha!, hahahaha!, ... }
- { ab, aabb, aaabbb, ... }
- { the set of grammatical English sentences }

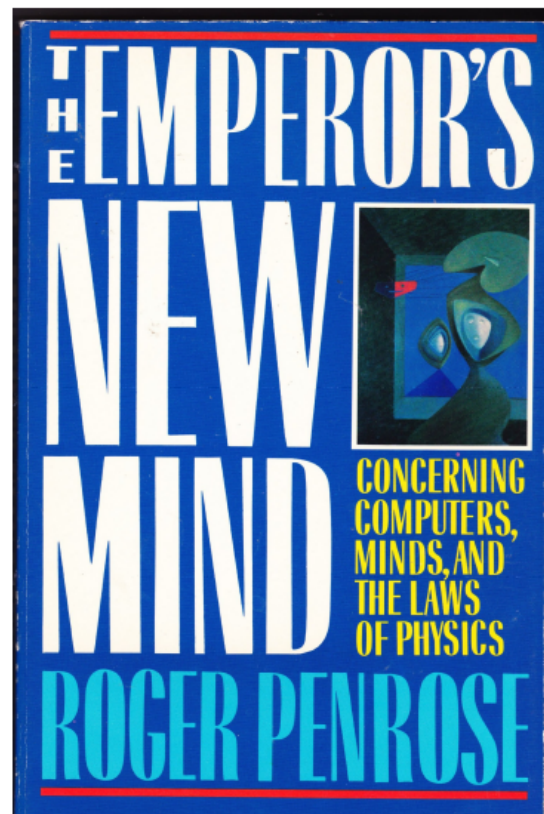
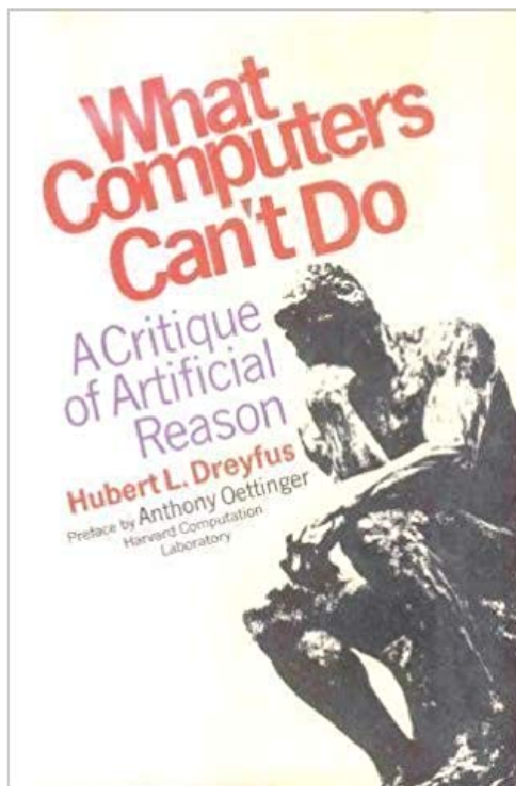
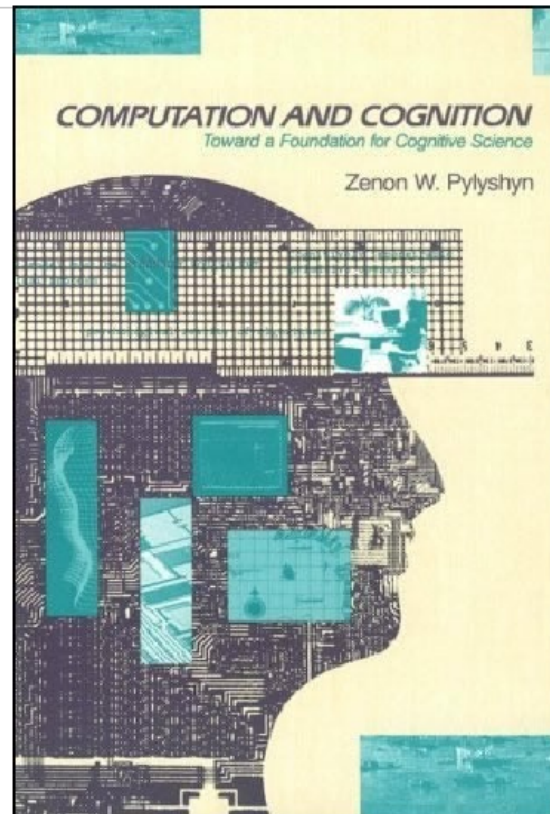
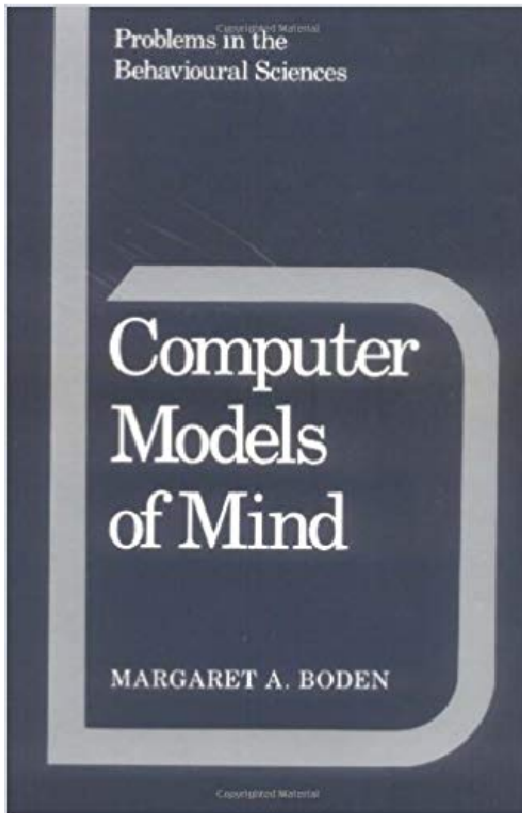




Fig. 1: Examples of Automata: A vending machine, A clock, Vaucanson's duck

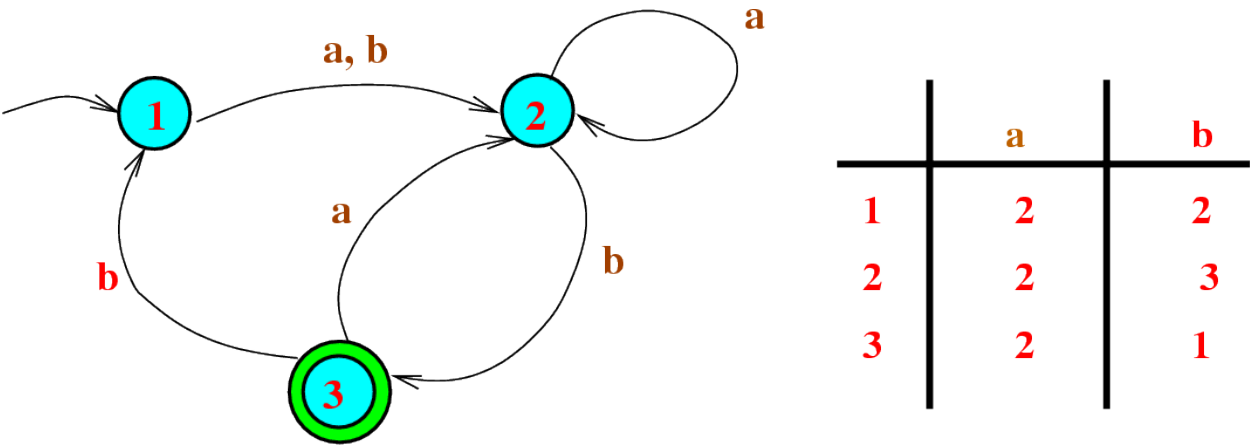


Fig. 2: Diagram and Tabular representation of a finite state automaton

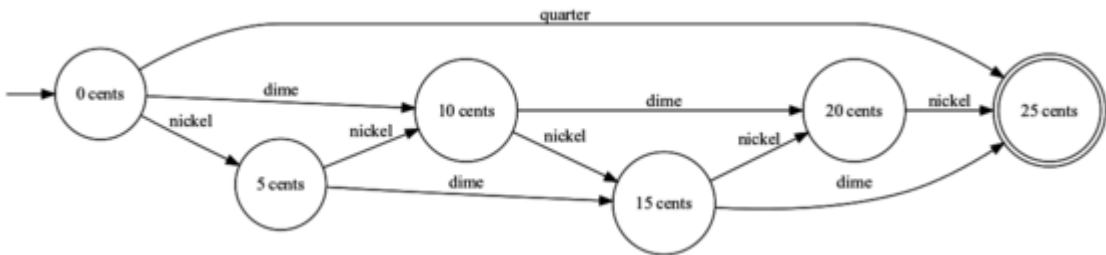
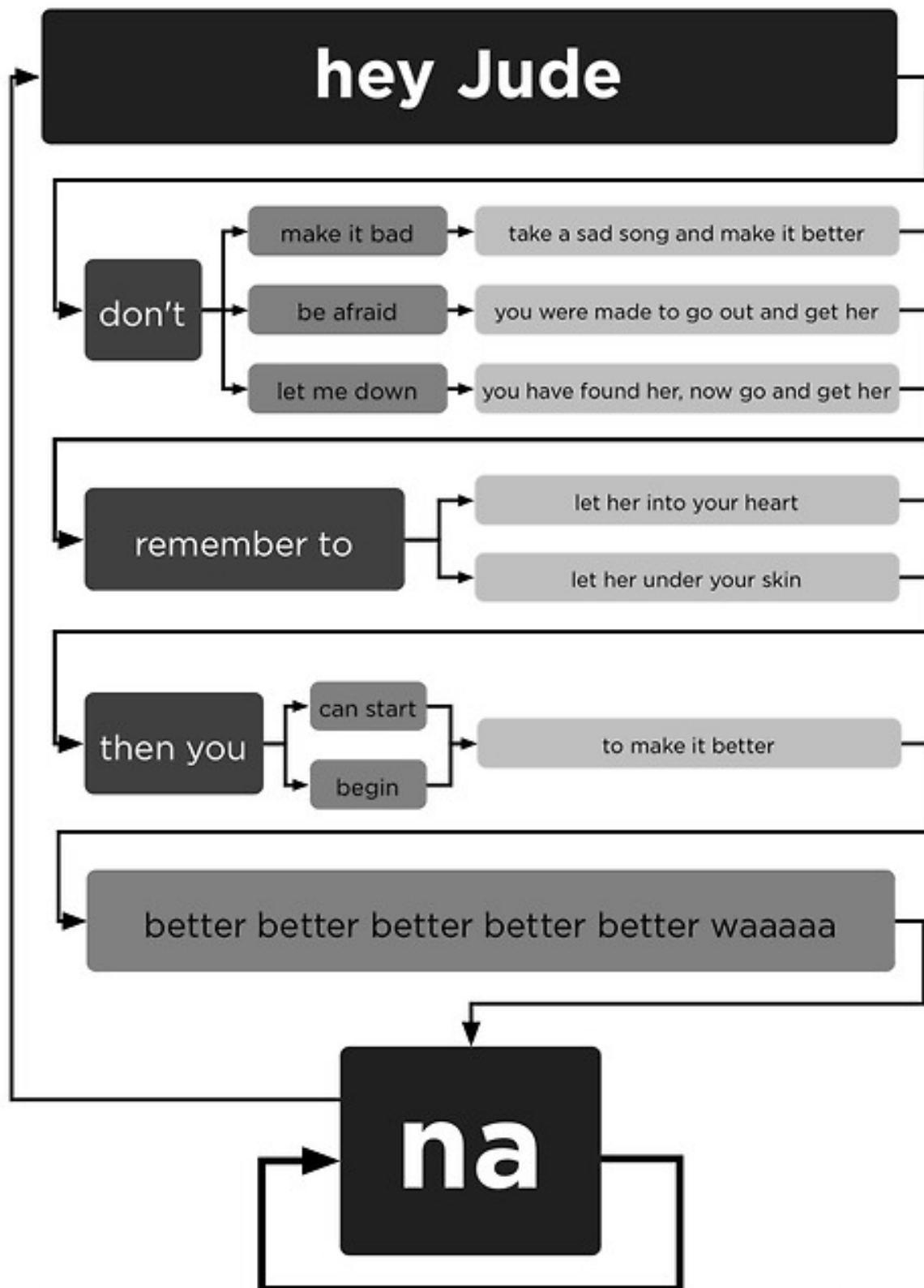


Fig. 3: The change counter of a vending machine



loveallthis.tumblr.com

lyrics © sony atv

Fig. 4: Transition Diagram for the lyrics of *Hey Jude*

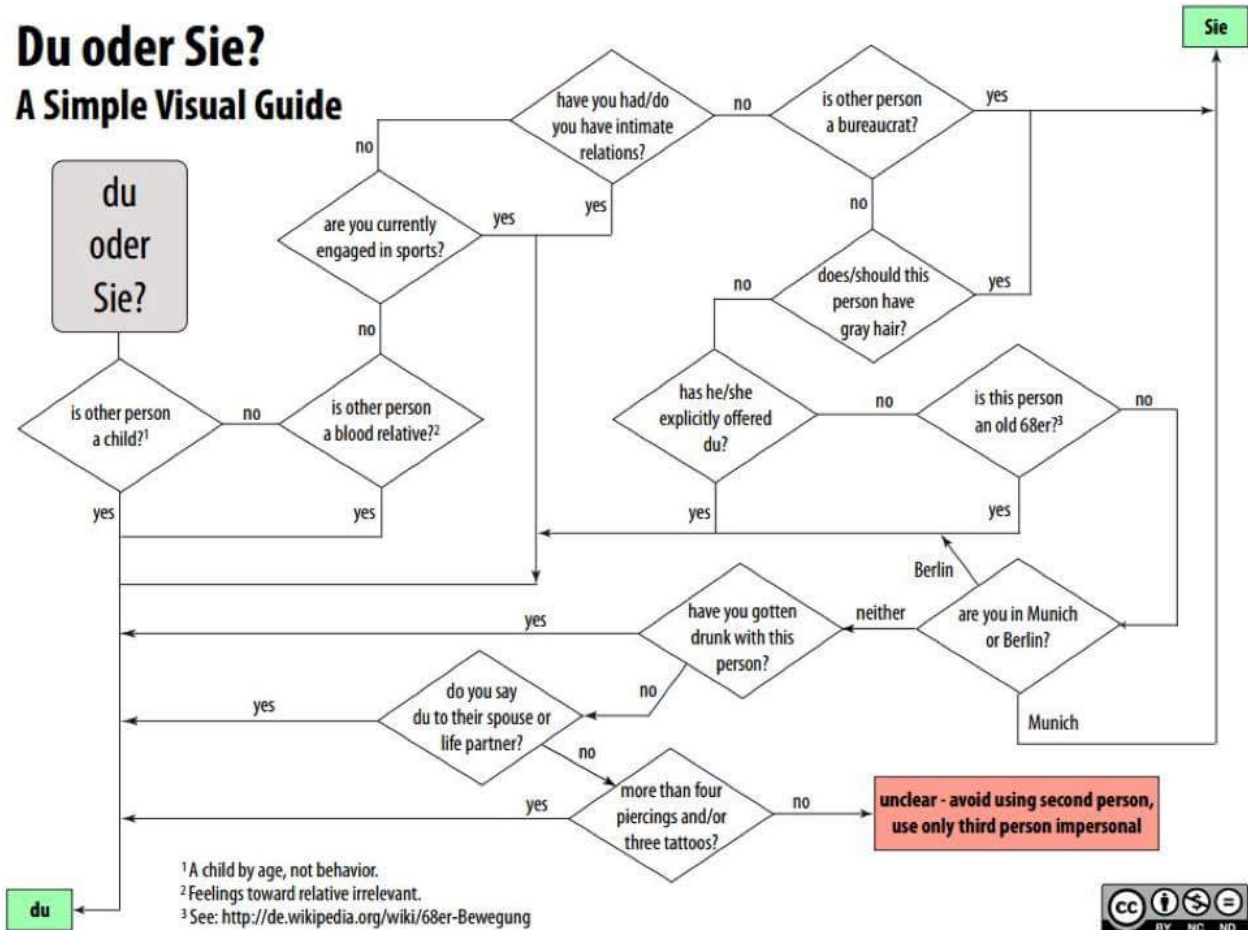


Fig. 5: Algorithm to decide if you must use “du” or “sie” in German

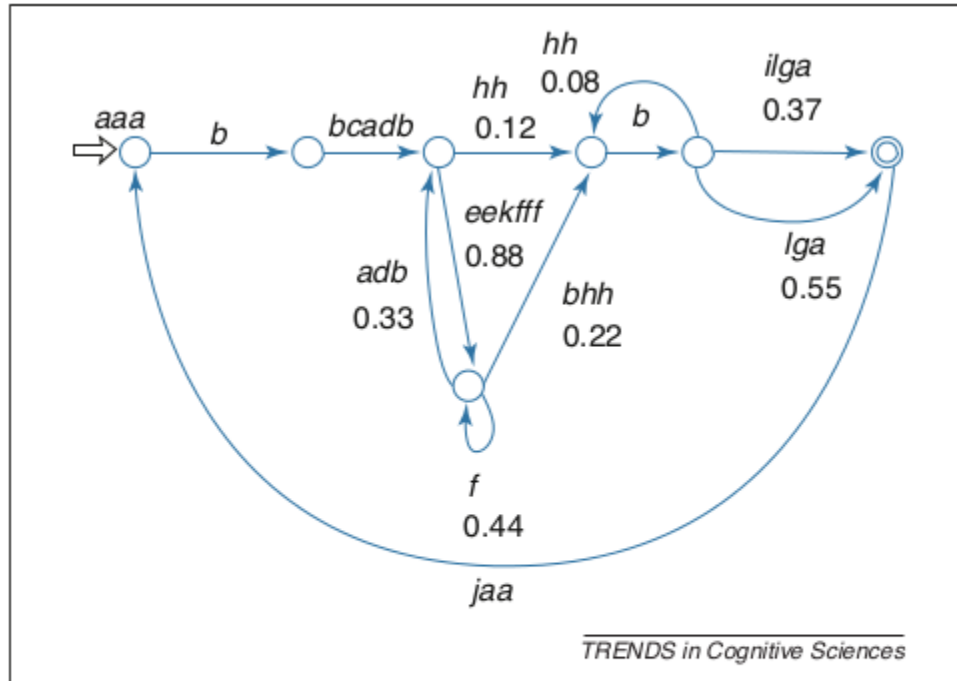


Figure 3. Probabilistic finite-state transition diagram of the song repertoire of a Bengalese finch. Directed transition links between states are labelled with note sequences along with the probability of moving along that particular link and producing the associated note sequence. The possibility of loops on either side of fixed note sequences such as *hh* or *lga* mean that this song is not strictly locally testable (see [Box 3](#) and main text). However, it is still *k*-reversible, and so easily learned from example songs [35]. Adapted, with permission, from [75].

Fig. 6: A (Probabilistic) Finite state diagram for Bengalese Finch songs (Berwick et al., 2011 *Trends in Cognitive Sciences*)

9.6 What is a Computer?

A computer is basically an automaton augmented with a *memory store*.

This is particularly clear in the case of the *Turing machine*, a mathematical model of computation (Turing offered the Turing machine as an analysis of the activity of an (idealised) human mathematician engaged in computing).

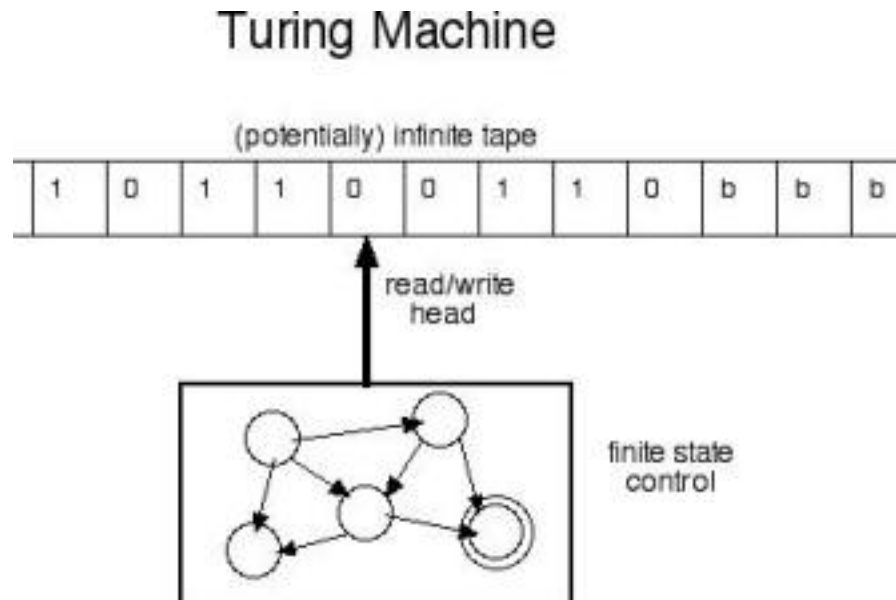


Fig. 7: A Turing machine = FSA + memory store

A Turing machine is a finite state machine augmented with a tape and a mechanism to read/write on it.

Read Roger Penrose's chapter's on Turing machines and https://en.wikipedia.org/wiki/Turing_machine. You may also read the Alan Turing's seminal paper.

Tape symbol	Current state A			Current state B			Current state C		
	Write symbol	Move tape	Next state	Write symbol	Move tape	Next state	Write symbol	Move tape	Next state
0	1	R	B	1	L	A	1	L	B
1	1	L	C	1	R	B	1	R	HALT

Fig. 8: A table describing a Turing machine: try to simulate it.

Other computing machines have been invented, yet:

“All attempts to give an exact analysis of the intuitive notion of an effectively calculable function have turned out to be equivalent, in the sense that each analysis offered has been proved to pick out the same class of functions, namely those that are computable by Turing machine.

The concept of effective calculability has turned out to be formalism-independent, in that all these different formalisms pick out exactly the same class of functions.” (B. Jack Copeland “The Church-Turing thesis” in *Stanford Encyclopedia of Philosophy Archive*)

Another computing model which is closer to actual computers, is the *register machine*.

9.7 Register machines

Read [The seven secrets of computer power revealed](#) (Chapter 24 from Daniel Dennett's *Intuition Pumps and other tools for thinking*)

The RogRego computer possesses:

- a bank of registers, or memory locations, each with a unique *address* (1, 2, 3, ...), and each able to have, as *content*, a single integer (0, 1, 2, ...)
- a processing unit can execute instructions in a stepwise, one-at-a-time fashion. The processor knows only 3 instructions:
 1. **End:** finishes the programs
 2. **Increment register with 2 arguments:**
 - a register number to increment by 1
 - a step (line) number to jump to when the increment is complete
 3. **Decrement register and Branch with 3 arguments:**
 - a register number to decrement by 1
 - a step number to jump if the register contains a non null value.
 - a step number to jump if the register contains 0

An online demo is available at <http://proto.atech.tufts.edu/RodRego/>

You can enter the following program “ADD[0,1]”, on a machine where Reg0 contains 4 and Reg1 contains 7. Try to explain what it is doing:

```
1 DEB 0 2 3
2 INC 1 1
3 END
```

...

This program adds the content of register 0 to register 1 (destroying the content of 0)

...

Exercise: write a program Program 2 “MOVE[4,5]” that moves the content of reg4 into reg5

...

```
1 DEB 5 1 2
2 DEB 4 3 4
3 INC 5 2
4 END
```

...

Program 3 “COPY[1,3]” copies the content of reg1 into reg3, leaving reg1 unchanged:

```
1 DEB 3 1 2
2 DEB 4 2 3
3 DEB 1 4 6
4 INC 3 5
5 INC 4 3
```

(continues on next page)

(continued from previous page)

```
6 DEB 4 7 8
7 INC 1 6
8 END
```

Program 4 (NON DESTRUCTIVE ADD[1,2,3]):

```
1 DEB 3 1 2
2 DEB 4 2 3
3 DEB 1 4 6
4 INC 3 5
5 INC 4 3
6 DEB 4 7 8
7 INC 1 6
8 DEB 2 9 11
9 INC 3 10
10 INC 4 11
11 DEB 4 12 13
12 INC 2 11
13 END
```

...

Note that *conditional branching* is the key instruction that gives the power to the machine. Depending on the content of memory, the machine can do either (a) or (b).

9.7.1 Exercise: implementation of a Register machine

Write a Python script that simulates a [RodRego machine](#) with 10 registers. The program is stored in a string or in file that is read and then executed. Your program must contain a function which, given the 10 initial values of the registers, and the program, returns the new register values when the END command is reached.

Check two possible solutions: - `rodrego_maxime_caute.py` - `rodrego_christophe_pallier.py`

—

9.8 The Seven secrets of computers revealed

1. Competence without comprehension. A machine can do perfect arithmetic without having to comprehend what it is doing.
2. What a number in a register stands for depends on the program
3. The register machine can be designed to discriminate any pattern that can be encoded with numbers (e.g. figures, text, sensory inputs,...)
4. Programs can be encoded by numbers.
5. All programs can be given a unique number which can be treated as a list of instructions by a Universal Machine.
6. all improvements in computers over Turing machine (or Register machine), are simply ways of making them faster
7. There is no secret #7

9.9 Programmable computers

- The first computers were not programmable. They were hardwired!
- An important milestone was the invention of the *programmable* computer:
 - a program is a set of instructions stored in memory.
 - Loaded and executed by a processor.
 - Such programs are written in machine language (the language of the processor)

9.10 Compilation and interpretation

Programs written in higher-level languages (rather than Machine language) can be either:

= **compiled**, or = **interpreted**

In both cases, you write the program as text files called **source files**.

A **compiler** translates the program into an executable file in machine language. The executable file is standalone, that is, the source code is not needed.

An **interpreter** reads the file and execute the commands one by one. It is slower, but easier to interact with. Disadvantage: you need the interpreter to execute it.



Figure 1.1: An interpreter processes the program a little at a time, alternately reading lines and performing computations.

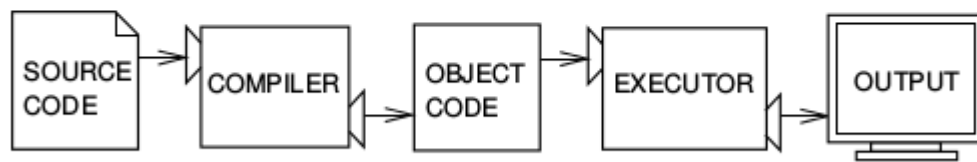


Figure 1.2: A compiler translates source code into object code, which is run by a hardware executor.

Fig. 9: Interpretation and compilation

9.11 Operating systems

In the first computers, there was only **one** program running. One would load the program into memory, then run it until it halted. Several Programs were ran in *batch mode*, in a sequence.

Then, it was realized that computers could *time-share* between programs, allowing several users (or programs) to share the computer.

This requires an **operating systems** (O.S.). The O.S. is the first program that loads into the computer during the boot. When running:

- The OS controls the hardware (screen/printer/disk/keyboard/mouse,...) (drivers)
- The OS manages all the other programs (processes/tasks/applications).
 - sharing memory
 - allocating processors and cores
 - allocating time

Check out *Task Manager* (Windows)/*System Monitor* (Linux)/ *Activity Monitor* (Mac)



Fig. 10: Three popular operating systems

Different OSES offer different “views” of the computer (e.g. 1 button mouse in Mac, 2 in Windows, 3 in Linux), so often programs are designed to work on one OS (bad!). Prefer multiplatform software (like Python).

Several OS can be installed in a given machine:

- choice at boot (multiboot)
- an OS can run inside a **virtual machine**, that is a program running in another (or the same) OS, and emulating a full computer.

9.12 What is a Terminal?

Terminal (or **console**): originally, a device comprising a keyboard and screen, allowing a human to *interact* with a computer.

Remarks:

Before keyboards and screens, there were punchcards and printers:

Historically, terminals used to be a dumb screen/keyboard connected to a central computer.



Fig. 11: Terminals

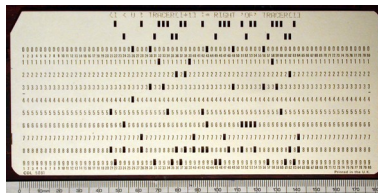
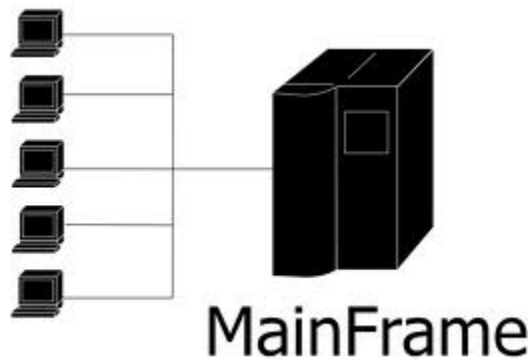


Fig. 12: Early computers had no keyboard, no screen. The input was done through punched cards and output would be printed out

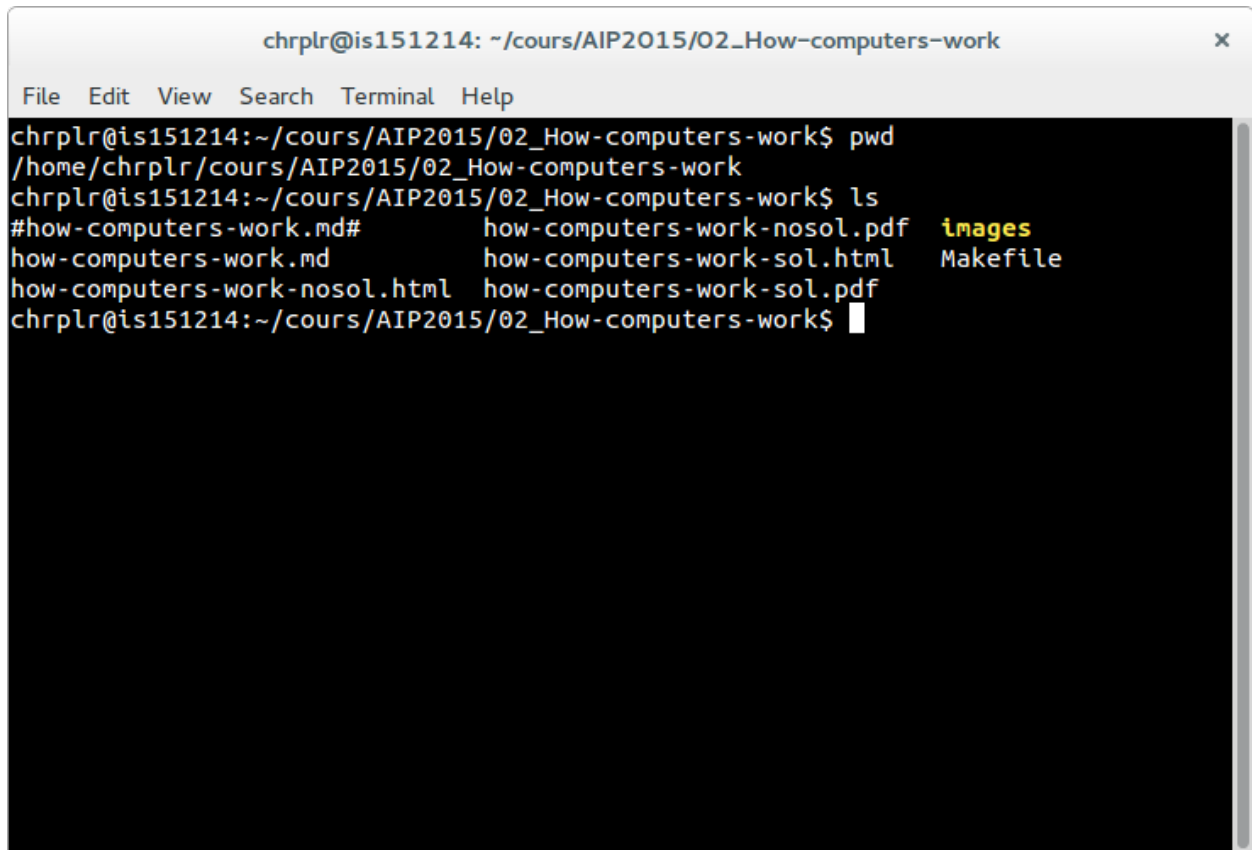


In the mainframe era, many terminals were connected to a single, powerful, computer. Everybody was sharing the same computer

With the advent of *Personal Computers*, the terminal and the computer became a single apparatus.

However, terminals can be *virtual*. A terminal is a program that let you run text programs. You interact by typing and displaying text. No graphical interface/no mouse.

When you open a terminal, a program called a *shell* is started that displays a prompt, waiting for you to enter commands with the keyboard.



```
chrplr@is151214: ~/cours/AIP2015/02_How-computers-work
File Edit View Search Terminal Help
chrplr@is151214:~/cours/AIP2015/02_How-computers-work$ pwd
/home/chrplr/cours/AIP2015/02_How-computers-work
chrplr@is151214:~/cours/AIP2015/02_How-computers-work$ ls
#how-computers-work.md#      how-computers-work-nosol.pdf  images
how-computers-work.md       how-computers-work-sol.html  Makefile
how-computers-work-nosol.html  how-computers-work-sol.pdf
chrplr@is151214:~/cours/AIP2015/02_How-computers-work$
```

Fig. 13: Picture of a ‘virtual’ terminal in Linux

REPRESENTATIONS OF NUMBERS, TEXT, IMAGES

Contents

- *Representations of numbers, text, images*
 - *Representation of integers*
 - * *From binary to decimal*
 - * *From decimal to binary*
 - *Representation of text*
 - * *Strings*
 - * *search/replace a substring within a string*
 - * *splitting a strings at delimiters*
 - * *Interactive input from the command line*
 - * *Reading and writing to text files*
 - * *Counting lines and words in a text file*
 - *Representation of images*
 - * *Grey level pictures*
 - * *Colored bitmaps*

10.1 Representation of integers

There are 10 kinds of people: those who count in binary and the others.

Computers represent everything as series of 0 and 1, also known as *bits* (for “binary digits”).

A number represented in basis ‘b’ by four digits ‘ $d_3d_2d_1d_0$ ’, has a value of: $d_3.b^3 + d_2.b^2 + d_1.b^1 + d_0.b^0$

- In binary, there are only two possibilities for the digits: {0, 1}
- In decimal, there are 10 possible characters
- In hexadecimal, 16 possible characters 0-9, A, B, C, D, E, F:: $D8F1 = 14*16^3 + 8*16^2 + 15*16^1 + 1*16^0$

Just like a number can be written in base 10, it can be written in base 2 (or in any other base):

```
12 = 10 + 2 = 1.(10^1) + 2.(10^0) => '12' in base 10
12 = 8 + 4 = 2^3 + 2^1 => '1010' in base 2

33 = 30 + 3 = 3.(10^1) + 3.(10^0) => '33' in base 10
33 = 32 + 1 = 2^5 + 2^0 => '100001' in base 2
```

Here are the binary representations of the first integers:

```
0 : 0
1 : 1
2 : 10
3 : 11
4 : 100
5 : 101
6 : 110
7 : 111
...
```

To learn more about how integer numbers are represented in binary format, you can check out <http://csunplugged.org/binary-numbers>

Exercise 1: Convert (manually) into decimal the following binary numbers:

- 101
- 1000
- 1011
- 1111111

...

Answers: 5, 8, 11, 255

10.1.1 From binary to decimal

Exercise 2: Let us write a function that, given the binary representation of a number as a string of 0 and 1, returns its value as a integer.

1. Let us first suppose that we want to convert a string containing exactly 8 binary digits (e.g. '01011010') into decimal. How would you do that?

...

```
def todec8bits(s):
    """ converts a 8 bits string (binary representation) into a integer """
    return int(s[0])*128 + int(s[1])*64 + int(s[2])*32 + \
           int(s[3])*16 + int(s[4])*8 + int(s[5])*4 + \
           int(s[6])*2 + int(s[7])

todec8bits("00001010")
todec8bits("01010101")
```

One issue with this code is that it handles only 8 bits strings

```
todec8bits("0101010")
todec8bits("010101010")
```

A better version of the fonction would be:

```
def todec8bits(s):
    """ converts a 8 bits string (binary representation) into a integer """
    assert len(s) == 8 # 's' should be exactly 8 bits long
    return int(s[0])*128 + int(s[1])*64 + int(s[2])*32 + \
           int(s[3])*16 + int(s[4])*8 + int(s[5])*4 + \
           int(s[6])*2 + int(s[7])
```

Remark: On your computers, integers are represented either as 32 or 64 bits, depending on your processor/operating system.

Why is this is relevant? Suppose you perform an EEG recording with 256 electrodes every milliseconds for one hour. How large is the data?

Beware: in some programming languages, the computer can make mistakes if you add too large numbers!

Here is another solution demonstrating several python features (list comprehensions, zip constructions, increment operator, ...):

```
def todec(s):
    """ converts a 8 bit strings into an integer """
    assert len(s) == 8 # 's' should be exactly 8 bits long
    pow2 = [2 ** n for n in range(7, -1, -1)]
    n = 0
    for b, p in zip(s, pow2):
        n += int(b) * p
    return n
```

...

Exercise: modify the function above to handle strings of any size as input.

Here is a code that works with strings of unlimited size:

```
def todec(s):
    """ convert a string of 0 and 1 representing a binary number into an integer """
    n = 0
    for b in s:
        n = n * 2 + int(b)
    return n

for i in ['101', '1000', '1011', '11111111']:
    print(todec(i))
```

Can you understand how/why it works ?

10.1.2 From decimal to binary

Now we will go in the other direction: Our aim is to write a program that, given a number (in decimal), computes its binary representation.

Exercise: If you have an idea how to program it, please proceed. Else, I propose that you follow the following steps:

Examine the script below and execute it for various values of the variable *num*. Note that the sign % stands for the *modulo division operation* which produces the remainder of an integer division. If *x* and *y* are integers, then the expression *x* % *y* yields the remainder when *x* is divided by *y*.

Do you understand the last line? Do you see a limitation of this program?

```
num = 143
d3 = int(num/1000) % 10 # thousands
d2 = int(num/100) % 10 # hundreds
d1 = int(num/10) % 10 # dec
d0 = num % 10
print(str(d3) + str(d2) + str(d1) + str(d0))
```

Adapt the above program to print the binary representation of *num*

...

```
num = 17
b0 = num % 2
b1 = int(num/2) % 2
b2 = int(num/4) % 2
b3 = int(num/8) % 2
b4 = int(num/16) % 2
b5 = int(num/32) % 2
b6 = int(num/64) % 2
b7 = int(num/128) % 2
b8 = int(num/256) % 2
print(str(b8) + str(b7) + str(b6) + str(b5) + str(b4) + str(b3) + str(b2) + str(b1) + \
      str(b0))
```

...

(6) Modify the above program to print the binary representations of all the integers between 0 and 255.

...

```
def tobin(num):
    """ Returns the binary representation (strings of bits) of a 0 <= num <= 255 """
    b7 = int(num/128) % 2
    b6 = int(num/64) % 2
    b5 = int(num/32) % 2
    b4 = int(num/16) % 2
    b3 = int(num/8) % 2
    b2 = int(num/4) % 2
    b1 = int(num/2) % 2
    b0 = num % 2
    return (str(b7) + str(b6) + str(b5) + str(b4) + \
           str(b3) + str(b2) + str(b1) + str(b0))
```

(continues on next page)

(continued from previous page)

```
for n in range(256):
    print(n, ': ', tobin(n))
```

...

(7) (Advanced) Write an improved version that uses a loop and does not have a limitation in size.

...

```
def binary(n):
    """ returns the binary representation of `n` """
    if n == 0:
        return '0'
    s = ''
    while n > 0:
        b = str(n % 2)
        s = b + s
        n = n // 2
    return s
```

...

(8) Study the following code. Do you understand why it works?

```
def binary(num):
    """ returns the binary representation of `num` """
    if num == 0:
        return '0'
    if num == 1:
        return '1'
    return(binary(int(num / 2)) + binary(num % 2))

print(binary(1234))
```

...

Answer: It is a recursive function which calls itself. See http://en.wikipedia.org/wiki/Recursion_%28computer_science%29

...

Remark: measures of memory size

- 1 byte = 8 bits
- 1 Kilobyte (KB) = 1024 bytes
- 1 Megabyte (MB) = 1024 kbytes = 1048576 bytes
- 1 Gigabytes (GB) = 1024 Mbytes
- Terabyte, Petabyte, Exabyte...

Exercise (advanced): Write a function that return the hexadecimal representation (base 16) of a number.

To go further:

- If you want to know how negative integer numbers are represented, see http://en.wikipedia.org/wiki/Two%27s_complement

- To understand how real numbers (a.k.a. “floats”) are encoded, read [What Every Programmer Should Know About Floating-Point Arithmetic](https://docs.python.org/2/tutorial/floatpoint.html#tut-fp-issues) and <https://docs.python.org/2/tutorial/floatpoint.html#tut-fp-issues>

10.2 Representation of text

A text file is nothing but a sequences of characters.

For a long time, characters were encoded using ASCII code.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENO	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	\$	%	&	'	()	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL

Fig. 1: ascii table

In Python, you can know the code of a character with the function `ord`:

```
print(ord('a'))
print(ord('@'))
```

The inverse of `ord` is `chr`.

- lookup the ASCII representation of your first name in the table and use the `chr` function of Python to print it.

...

For example, if you name is 'ZOE', you would type:

```
print(chr(90)+chr(79)+chr(69))
```

Remark: **ASCII** codes use one byte (=8bits) per character. This is fine for English, but cannot cover all the characters of all alphabets. It cannot even encode french accented letters.

Unicode was invented that associate a unique 2 bytes number to each character of any human script. It is possible to write text files using these number, but more economic to encode the most common letters with one byte, and keep the compatibility with ASCII (UTF-8).

```
print(''.join([chr(c) for c in range(20000, 21000)]))
```

10.2.1 Strings

In Python, text can be stored in objects called *strings*.

String constants are enclosed between single quotes:

```
'Bonjour le monde!'
```

Or double quotes:

```
"Bonjour le monde !"
```

Or “triple” quotes for multiline strings:

```
"""
Bonjour le monde!

Longtemps je me suis levé de bonne heure,
Les sanglots longs des violons,
...
"""
```

They have a type `str`:

```
type('bonjour')
```

To convert an object to a string representation:

```
str(10)
a = dict(("a",1), ("b",2))
str(a)
```

A string is nothing but a sequence of characters:

```
a = 'bonjour'
print(a[0])
print(a[1])
print(a[2])
print(a[2:4])
print(len(a))

for c in 'bonjour':
    print(c)
```

Operations on strings:

```
a = 'bonjour'
b = 'hello'
a + b
a + ' ' + b
```

A set of functions to manipulate strings is available in the module ‘string’:

```
str.upper(a)
str.lower('ENS')
```

10.2.2 search/replace a substring within a string

```
a = 'alain marie jean marc'
print(a.find('alain'))
print(a.find('marie'))
print(a.find('ma'))
print(a.find('marc'))
print(a.find('o'))

a.replace('marie', 'claire')
print(a)
```

10.2.3 splitting a strings at delimiters

```
a = 'alain marie jean marc'
a.split(" ")
```

Read <https://docs.python.org/3/library/stdtypes.html#string-methods> to learn about more string functions.

10.2.4 Interactive input from the command line

```
name = input('Comment vous appelez-vous ? ')
print("Bonjour " + name + '!')
```

10.2.5 Reading and writing to text files

1. With Atom, create a text file containing a few lines of arbitrary content, and save it under the filename 'test.txt'
2. with ipython running in the same directory where you saved test.txt

```
with open('test.txt', 'r') as f:
    o = f.read()
    print(o)
    lines = o.split("\n")
    print(lines)
```

10.2.6 Counting lines and words in a text file

Download [Alice in Wonderland](#)

```
with open('alice.txt') as f:
    o = f.read()
    print(o)
    lines = o.split("\n")
    print(lines)
```

Exercise: Write a program that counts the number of lines, and number of words in `alice.txt` (we suppose that words are separated by spaces).

...

```
with open('alice.txt') as f:
    o = f.read()
    print(o)
    lines = o.split("\n")

    nlines = len(lines)

    nw = 0
    for l in lines:
        nw += len(l.split(" "))

    print(nlines)
    print(nw)
```

(11) Write a program that detects if a text file contains the word 'NSA'

...

```
def spot_nsa(filename):
    """ detects if the text file pointed to by filename contains 'NSA' """
    with open(filename) as f:
        o = f.read()
        lines = o.split("\n")
        found = False
        for l in lines:
            if "NSA" in l.split(" "):
                found = True
                break
    return found
```

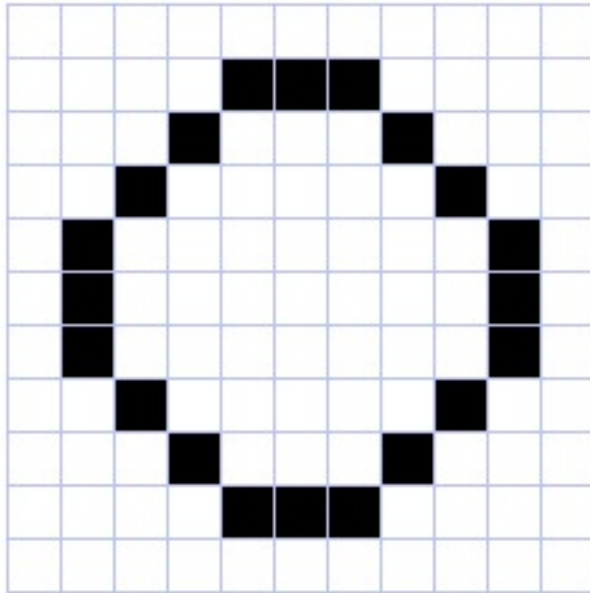
10.3 Representation of images

Images can be stored either:

- as bitmaps, that is a two dimensional arrays of dots (formats: bmp, png, gif, jpeg...)
- as vectorized formats, the image contain instruction for drawing objects (eps, pdf, svg, ...).

Here we are just going to manipulate bitmaps.

Each dot (pixel) is either '0' (black) or '1' (white).



(12) What is the size in kilobytes of a 1024x768pixels black and white image ?

...

Answer: $1024 * 768 / 8 / 1024 = 96$ KB

(13) Execute the following code in ipython:

```
import numpy as np
import matplotlib.pyplot as plt

a = np.array([[0, 0, 0, 0, 0, 0, 0],
              [0, 0, 1, 1, 1, 0, 0],
              [0, 0, 1, 1, 1, 0, 0],
              [0, 0, 1, 0, 1, 0, 0],
              [0, 0, 1, 1, 1, 0, 0],
              [0, 0, 1, 1, 1, 0, 0],
              [0, 0, 1, 1, 1, 0, 0],
              [0, 0, 0, 0, 0, 0, 0]])

plt.imshow(a, cmap=plt.cm.gray, interpolation='nearest')
plt.show()
```

Numpy's arrays are a new type of object. There are similar to lists, but optimised for mathematical computations. Notably, they can be multidimensional (i.e. you can use `a[i,j]` notation). You can learn more about arrays in the documents <https://scipy-lectures.org/> and <https://www.projectpro.io/data-science-in-python-tutorial/numpy-python-tutorial>

Here is another example:

...

```

a = np.zeros((200,200))
for i in range(200):
    a[i, i] = 1
plt.imshow(a, cmap=plt.cm.gray, interpolation='nearest')
plt.show()

a[0:200:2,] = 1
plt.imshow(a, cmap=plt.cm.gray, interpolation='nearest')
plt.show()

```

10.3.1 Grey level pictures

Each dot is now associated to an integer value, e.g. ranging from 0 to 255 for 8-bits codes, coding for a grey level (smaller=darker). Each dot needs one byte.

How large is the file for an image 1024x768 pixels with 256 grey levels?

The following code displays an image:

```

from skimage import data
from skimage.color import rgb2gray

original = data.astronaut()
grayscale = rgb2gray(original)
plt.imshow(grayscale, cmap=plt.cm.gray)
plt.show()

```

This code runs a low pass (averaging) filter on it:

```

import scipy.ndimage
bl = scipy.ndimage.gaussian_filter(grayscale, 3)
plt.imshow(bl, cmap=plt.cm.gray)
plt.show()

```

Edge detector It is easy to implement an edge detector with a neural network. See <https://courses.cit.cornell.edu/bionb2220/UnderstandingLateralInhibition.html>.

Using the `ndimage.convolve` function, apply the following filters to the image and display the results.

```

from skimage import data
from skimage.color import rgb2gray

original = data.astronaut()
grayscale = rgb2gray(original)

kernel1 = np.array([[ -1, -1, -1],
                    [ -1,  8, -1],
                    [ -1, -1, -1]])

bl = scipy.ndimage.convolve(grayscale, kernel1)
plt.imshow(bl, cmap=plt.cm.gray)
plt.show()

```

(continues on next page)

(continued from previous page)

```
kernel2 = np.array([[ -1, -1, -1, -1, -1],
                    [-1,  1,  2,  1, -1],
                    [-1,  2,  4,  2, -1],
                    [-1,  1,  2,  1, -1],
                    [-1, -1, -1, -1, -1]])
bl=scipy.ndimage.convolve( grayscale, kernel2)
plt.imshow(bl, cmap=plt.cm.gray)
plt.show()
```

More manipulations are available at http://scipy-lectures.github.io/advanced/image_processing/.

10.3.2 Colored bitmaps

Each dot is now associated to three bytes, representing the Red, Green and Blue intensities (see <http://www.colorpicker.com/>).

How large is the file for a 1024x768 RGB image?

Exercise: What are the RGB triplets for BLACK, WHITE, RED, YELLOW?

```
from skimage import data
plt.imshow(data.astronaut())
plt.show()
```


CREATING STIMULI

Contents

- *Creating stimuli*
 - *Static visual stimuli*
 - * *Displaying geometric shapes*
 - * *Troxler effect*
 - * *Kanizsa illusory contours*
 - * *Herman grid*
 - * *Extinction illusion*
 - * *Ebbinghaus-Titchener*
 - * *Fixation cross*
 - * *Hering illusion*
 - * *Random-dot stereograms*
 - * *Kitaoka visual illusions*
 - * *Stroop Effect*
 - *Dynamic visual stimuli*
 - * *Illusory line-motion*
 - * *Flash-lag illusion*
 - * *Dynamic version of the Ebbinghaus-Titchener*
 - * *Lilac Chaser*
 - *Creating and playing sounds*
 - * *Shepard tone*
 - * *Sound localisation from binaural dephasing*
 - * *Pulsation (Povel & Essen, 1985)*
 - *More illusions*

11.1 Static visual stimuli

To generate visual stimuli, we are going to rely on the `Pygame` module. You can check if it is installed on your system by typing `python` on a command line and, at the `>>>` prompt, `import pygame`. If all is well, you should get the following message:

```
>>> import pygame
pygame 2.0.1 (SDL 2.0.14, Python 3.8.5)
Hello from the pygame community. https://www.pygame.org/contribute.html
```

Warning: If, instead, you obtain a message ending in `ModuleNotFoundError: No module named 'pygame'`, one reason could be that you forgot to activate the virtual environment where you installed `expyriment`. Try `conda activate expyriment` or `pyenv activate expyriment` (use `conda env list` or `pyenv virtualenvs` to list all available environments)

If this does not work, you can install `pygame` with `pip install pygame`.

For more detailed instructions, see *Software Installation*.

11.1.1 Displaying geometric shapes

Here is a Python script that opens a window and displays a square:

```
""" Display a square.

    See https://sites.cs.ucsb.edu/~pconrad/cs5nm/topics/pygame/drawing/
"""

import pygame

# Colors are triplets containint RGB values
# (see <https://www.rapidtables.com/web/color/RGB_Color.html>)
BLACK = (0, 0, 0)
WHITE = (255, 255, 255)
GRAY = (127, 127, 127)
RED = (255, 0, 0)
GREEN = (0, 255, 0)
BLUE = (0, 0, 255)

# Parameters of the Graphics Window
W, H = 500, 500 # Size of the graphic window
# Note that (0,0) is at the *upper* left hand corner of the screen.
center_x = W // 2
center_y = H // 2

pygame.init()

# Create the Graphic Window (designated by a variable `screen`)
screen = pygame.display.set_mode((W, H), pygame.DOUBLEBUF)
pygame.display.set_caption('square')
```

(continues on next page)

(continued from previous page)

```

screen.fill(WHITE) # fill the window with white

# Draw a rectangle at the center of the window (in the backbuffer)
width, height = 200, 200 # dimensions of the rectangle in pixels
left_x = center_x - width // 2 # x coordinates of topleft corner
top_y = center_y - height // 2 # y coordinate of topleft corner
pygame.draw.rect(screen, BLUE, (left_x, top_y, width, height))

pygame.display.flip() # display the backbuffer on the screen
# Note: this function is synonymous with `pygame.display.update()`

# Save the image into a file
pygame.image.save(screen, "square-blue.png")

# Wait until the window is closed
quit_button_pressed = False
while not quit_button_pressed:
    pygame.time.wait(10)
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            quit_button_pressed = True

pygame.quit()

```

Download (square.py) and run it by typing:

```
python square.py
```

Have a look at the code.

Exercise (*): make a copy of the script and modify the copy to

- change the color of the rectangle to RED
- change the size of the rectangle to 100 x 300
- comment the line `pygame.display.flip()` and run the script. You should realize that merely drawing something to the display surface (*screen*) doesn't cause it to appear on the screen – you need to call `pygame.display.flip()` to move the surface from general memory to video memory. This will be useful when you want to make an animation, that is, draw a sequences of images.

Have a look at:

- [Pygame drawing basics](#)
- [Pygame tutorial](#)
- [Pygame's on-line documentation](#)

It is of course possible to draw other shapes. Check out for example the two scripts: - `circle.py` and - `triangle.py`

Exercise (*): modify `circle.py` to draw *two* circles, one red and one blue, side-by-side

(solution in `two_circles.py`)

Exercise (*): Note that the circles above are filled with the color (actually, they are disks). Browse [Pygame online documentation](#) to find how to color the circumference of the circle and keep its inner part white.

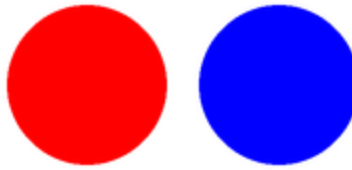


Fig. 1: Two Circles

11.1.2 Troxler effect

Fixate your gaze at the center of the picture below for 30 seconds



Fig. 2: Troxler effect

What happened after a few seconds? This is the *fill-in phenomenon* (See <https://en.wikipedia.org/wiki/Filling-in>).

Exercise (*): Program the Troxler stimulus (hint: use <https://www.google.com/search?q=color+picker> to find the RGB values for the disks)

For a solution, check out `troxler.py`

11.1.3 Kanizsa illusory contours

Created by Italian psychologist Gaetano Kanizsa in 1955, the *Kanizsa Triangle* is a classic example of illusory contours. In the image below, a white equilateral triangle can be clearly perceived even though there are no explicit lines or enclosed spaces to indicate such a triangle. (To find out more about this illusion, perform a Google search with the keywords *illusory contours*.)

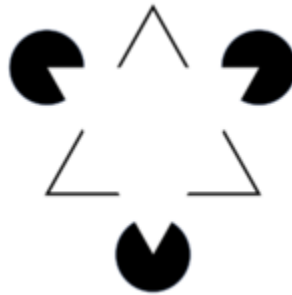


Fig. 3: Kanizsa triangle

There exists many variants, e.g. the Kanizsa squares:



Fig. 4: Kanizsa square

Exercise (**): Inspiring yourself from the code in `square.py` and `circle.py`, create a script that displays the (right) Kanizsa square .

A possible solution is proposed in `kanizsa-square.py`

11.1.4 Herman grid

Read about the [Herman grid illusion](#)

Exercise (**): Using `square.py` as a starting point, write a program to display the grid.

Hints:

- use paper and pencil to draw the figure
- find out the formulas to compute the left top of the square in the i th row and j th column

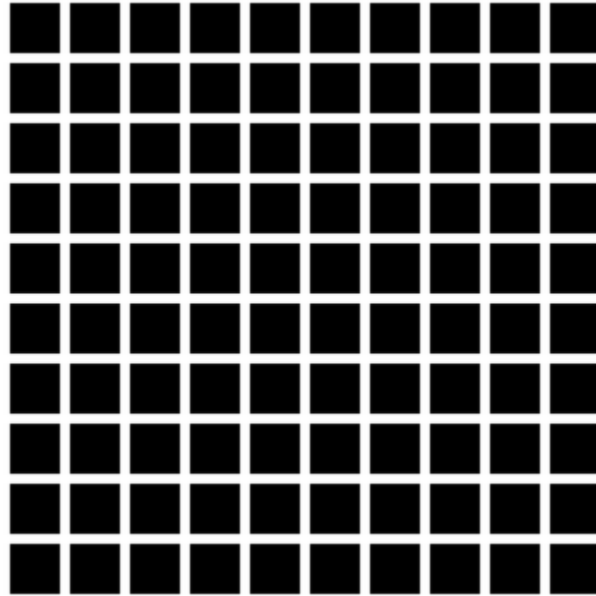


Fig. 5: Hermann Grid

- in your python script, use nested `for` loops over rows and columns to display each square one by one.

Check out `grid.py`.

Optional exercises:

- Play with the parameters ‘size of the squares’ and ‘space between the squares’ to see how they affect the illusion.
- Read <https://stackabuse.com/command-line-arguments-in-python/> to learn how to read arguments on the command line use the `sys.argv[]` list from the `sys` module. Create a version of the grid script that can get the number of columns, rows, the size of sides of squares, and the size of the space between squares. Play with those parameters to see if you can make the illusion come and go. (see `grid-args.py`)

Remark: there exists two powerful modules to help parse arguments on the command line: `argparse` or `docopt`

11.1.5 Extinction illusion

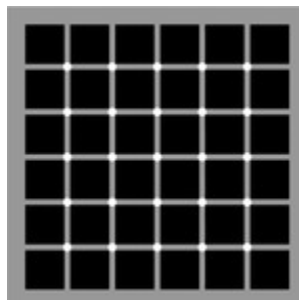


Fig. 6: McAnany-Levine extinction illusion (see McAnany, J. J. and Levine, M. W. (2004) The blanking phenomenon: a novel form of visual disappearance. *Vision Research*, 44, 993-1001.)

Exercise: Program the McAnany-Levine extinction stimulus, that is, a grid of black squares with white circles at the intersection.

Check out `extinction.py`

Remark: There exists variants of the extinction illusion:

- Niño's Extinction illusion

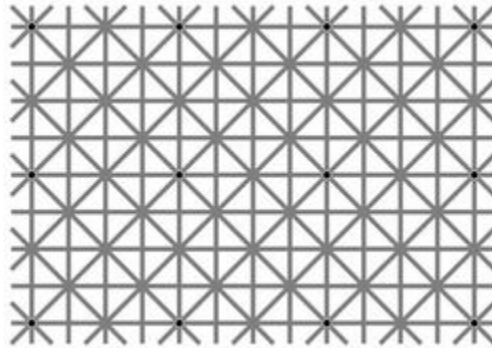


Fig. 7: Niño's Extinction illusion

- The **Honeycomb illusion**. You can read about it in Bertamini, Herzog, and Bruno (2016). A Python script to generate the stimulus is available on Bertamini's web site but it requires installing the module `PsychoPy` which can be challenging.

11.1.6 Ebbinghaus-Titchener

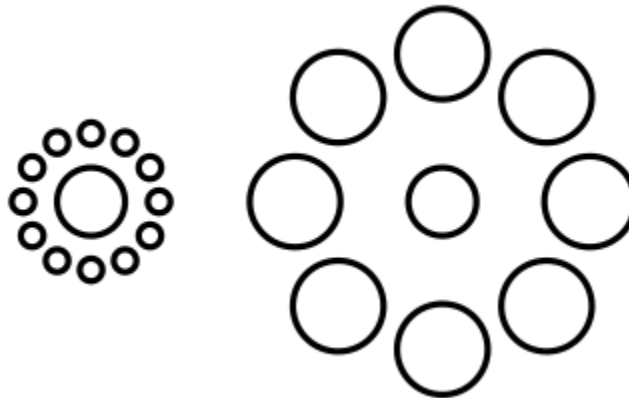


Fig. 8: Ebbinghaus illusion

Read about the [Ebbinghaus–Titchener stimulus](#).

Exercise (**): Using `circle.py` as a starting point, write a program to display a static stimulus (one central circle surrounded by a number of circles).

Hint: A little bit of [trigonometry](#) helps:

The coordinates of a location at and at distance R from the origin and an angle α from the left-right line are:

```
x = R * cos(alpha)
y = R * sin(alpha)
```

Consult <https://www.mathsisfun.com/polar-cartesian-coordinates.html> if you need to convince yourself about that.

Check out `ebbinghaus.py`

11.1.7 Fixation cross

Many visual experiments require participants to fixate a central fixation cross (in order to avoid eye movements).



Fig. 9: Fixation cross

Exercise (*): Using the function `pygame.draw.line()`, write a script that displays a cross at the center the screen. (Solution at `fixation-cross.py`)

11.1.8 Hering illusion

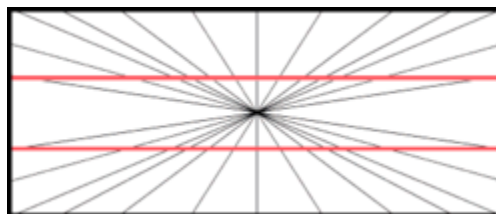


Fig. 10: Hering illusion

Exercise (**): Program the stimulus . Then, check a solution at `hering.py`

11.1.9 Random-dot stereograms



A random dot stereogram is a pair of images of random dots which, when viewed with the eyes focused on a point in front of or behind the images, produces a sensation of depth. To see how they can be generated, read the wikipedia entry on [random dot stereograms](#), to understand the phenomenon in details, read the one about [Stereopsis](#).

Exercise (***) Write a script that generates random-dot stereograms (warning: this requires a bit of knowledge of [Numpy](#) to represent the images as 2d arrays, and of [slicing](#))

Check out `random_dot_stereogram.py`

11.1.10 Kitaoka visual illusions

Professor Akiyoshi Kitaoka has produced many fascinating *visual illusions* <http://www.ritsumei.ac.jp/~akitaoka/index-e.html>. Notably:

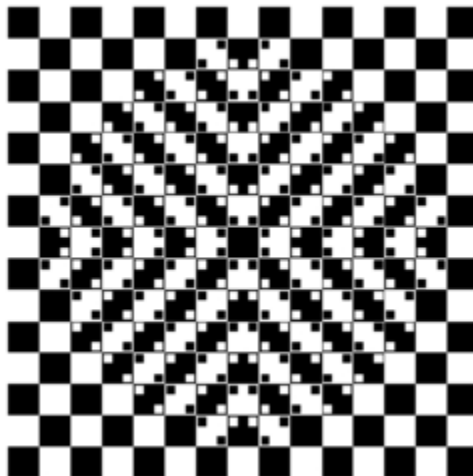


Fig. 11: The *Bulge*

Other notable stimuli are: the *Rotary extinction illusion*, *Unstable square*, *Rotating snakes*, *Rotating rays*, *Primrose's field*, *Rollers*, *Slippage*, *Gaku ga gakugaku*, *Spa*, *Expanding cushions*, *Convection*, *The music*, *Seaweed*, *Joro-gumo*,

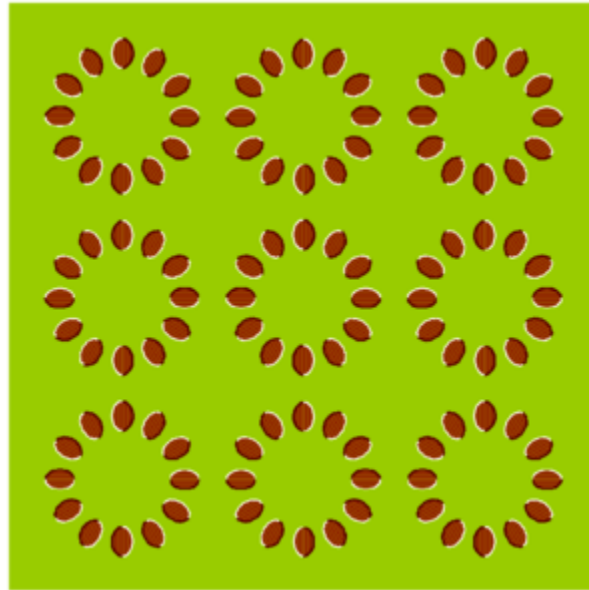


Fig. 12: The *Dongururin*

Packed cherries, Earthquake, Wedding in Japan, Sausages, Raspberries, A curtain, Pyramids of donguri, Dongurakokko (The donguri wave), Brownian motion, Waterways, A flow of the ecological flooring, Computer worms.

They are available on the following pages:

<http://www.ritsumei.ac.jp/~akitaoka/index-e.html>

<http://www.psy.ritsumei.ac.jp/~akitaoka/ol1saishe.html>

<http://www.psy.ritsumei.ac.jp/~akitaoka/kieru2e.html>

<http://www.psy.ritsumei.ac.jp/~akitaoka/saishin2e.html>

<http://www.psy.ritsumei.ac.jp/~akitaoka/saishin3e.html> <http://www.psy.ritsumei.ac.jp/~akitaoka/saishin4e.html>

Note: there are no exercise in this section. But, if you want to code some of the stimuli, feel free to do it, and please, share your code with us!

11.1.11 Stroop Effect

In the Stroop Task, participants are presented with a cards on which words are written in various colors. The task is to name as quickly as possible the colors of the printed words.

It is difficult to name the color of a color word if they do not match. This phenomenon, known as the **Stroop Effect**, demonstrates the automaticity of reading. Write a python script to create 4x4 cards for the task, as image files, avoiding repetitions of colors in neighboring cells.

You will need to read about how to generate images containing text, for example, in the tutorial [How to display text with pygame](#)

Then, check a solution at `create_stroop_cards.py`



Fig. 13: Stroop card

11.2 Dynamic visual stimuli

Animated movies are just a succession of still pictures. If the rate of presentation is fast enough, the brain creates an illusion of continuity.

With pygame, programming an animation will follow the following temporal logic:

```
#draw picture1 in the backbuffer
#flip the backbuffer to screen

#draw picture2 in the backbuffer
#wait for some time
#flip the backbuffer to screen

#draw picture3 in the backbuffer
#wait for some time
#flip the backbuffer to screen

...

```

We take advantage of the double buffering mode (set by the option `DOUBLEBUF` in the call to `pygame.display.set_mode()`) to draw the next image in memory while the current one is displayed on the screen. It is only when we call `pygame.display.flip()` that the image in memory is displayed, replacing the current one on the screen.

11.2.1 Illusory line-motion

Illusory line motion (ILM) refers to a situation in which flashing a light at one end of a bar prior to the bar's instantaneous presentation results in the percept of motion.

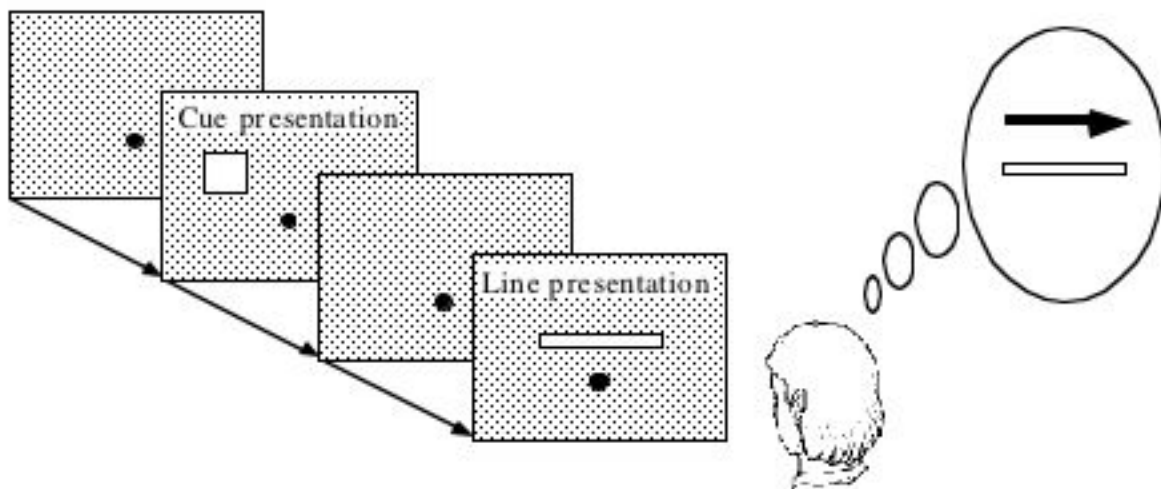


Fig. 14: Illusory line-motion

Exercise (*): Program the stimulus, that is, first draw a square, wait for a few milliseconds using the function `pygame.time.wait()`, then draw a rectangle overlapping with the initial square.

Check out `visual-illusions/line-motion.py`

11.2.2 Flash-lag illusion

- Download `visual-illusions/flash-lag.py` and run it. Do not look at the code yet.
- Do you feel that the moving square's x position coincides with the flashing square or not? If you want to read about the [Flash-lag illusion](#).

Exercise:

1. Create a movie of a square moving horizontally, back and forth. The principle is simple: you just need to create a loop where you display a square at coordinates x , y , wait a few milliseconds, then clear the screen, and increment or decrement the x coordinate by a fixed amount. This strategy is explained in details at http://programarcadegames.com/index.php?lang=en&chapter=introduction_to_animation

Check out out version `visual-illusions/moving_square.py`

2. Add the presentation of a flashing square then the moving square passes the middle line, to generate the flash-lag illusion.

Now, you can look at the code in `visual-illusions/flash-lag.py`

11.2.3 Dynamic version of the Ebbinghaus-Titchener

- Watch [this video](#).
- Program a version where the outer circles (inducers) grow and shrink in size.
- Check out `visual-illusions/ebbinghaus-dynamic.py`

11.2.4 Lilac Chaser

The [Lilac Chaser](#) is a dynamic version of the Troxler fill-in illusion.

Exercise (**): Program the Lilac Chaser stimulus, with 12 rose disks (you can use full disks without any blurring). Try different colors.

For a possible solution, check out `visual-illusions/lilac_chaser.py`

(Optional exercise for advanced students: add blurring to the disks to make a stimulus similar to that of the wikipedia page [Lilac Chaser](#). Then, for a solution, check out `visual-illusions/lilac_chaser_blurred.py`)

11.3 Creating and playing sounds

Install the *simpleaudio* module:

```
pip install simpleaudio
```

Then run the quick check with `ipython`:

```
import simpleaudio.functionchecks as fc
fc.LeftRightCheck.run()
```

Check out [simpleaudio's tutorials](#)

The module `sound_synth.py` provides several functions to load, create, and play sounds.

Exercise (**): Using functions from the `sound_synth` module, write a script that loads the file `cymbal.wav` and plays it 10 times, at a rhythm of one per second. (Warning: a basic knowledge of numpy arrays is necessary to concatenate the samples).

Check a solution at `cycle.py`

11.3.1 Shepard tone

Watch [this video <https://www.youtube.com/watch?v=LVWTQcZbLgY>](https://www.youtube.com/watch?v=LVWTQcZbLgY) about *Shepard tones*.

Exercise (***): Program a Shepard tone.

11.3.2 Sound localisation from binaural dephasing

Exercise (**): Take the channel of a mono sound and create a stereo sound. Then dephase the two channels by various delays, and listen to the results.

Hints: load the sound file into a one dimensional numpy array, make a copy of the array and shift it, assemble the two arrays in a bidimensional array (matrix) and save it as a stereo file

If you know nothing about [Numpy](#), you may find useful tutorials on the web, e.g. at https://github.com/paris-saclay-cds/data-science-workshop-2019/blob/b370d46044719281932337ca4154e1b0b443ad97/Day_1_Scientific_Python/numpys/numpy_intro.ipynb

11.3.3 Pulsation (Povel & Essen, 1985)

Exercise (***): Create rhythmic stimuli such as the ones described in [Povel and Essen \(1985\) Perception of Temporal Patterns](#)

11.4 More illusions

You can train your Python skills by programming some of the illusions at <https://www.illusionsindex.org/>

EXPERIMENTS

Contents

- *Experiments*
 - *Simple reaction times*
 - *Decision times*
 - *Numerical distance effect*
 - *Posner's attentional cueing task*
 - *Stroop Effect*
 - *A general audio visual stimulus presentation script*
 - *Sound-picture matching using a touchscreen*
 - *More examples using Epyriment*

12.1 Simple reaction times

Many psychology experiments measure *reaction-times* or *decision-times*.

The script `simple-detection-visual-pygame.py` is a simple detection experiment programmed with pygame. The task is simple: the participant must press a key as quickly as possible when a cross appears at the center of the screen.

Download it and run it with:

```
python simple-detection-visual-pygame.py
```

The results are saved in `reaction_times.csv` which you can inspect with any text editor.

If you are an R aficionado, you can open it and type:

```
data = read.csv('reaction_times.csv')
summary(data)
attach(data)
plot(RT)
dev.new()
plot(RT ~ Wait)
```

Here are my results:

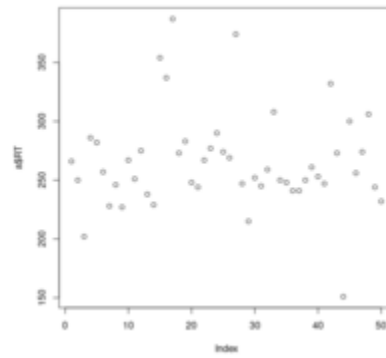


Fig. 1: Simple Reaction Times as a function of trial

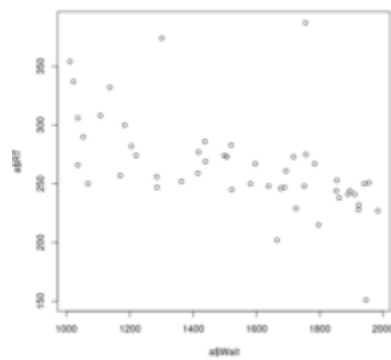


Fig. 2: Relationship between wait time and reaction time

Browse the code of `simple-detection-visual-pygame.py`

It is pretty technical! This is because `Pygame` is meant to program simple video games, not psychology experiments.

A more adequate library for this task is `Expyriment` (another one is `Psychopy`).

From here, we are going to use it to generate experiments.

Make sure you have installed `Expyriment`:

```
$ python
>>> import expyiment
```

If an error message `moduleNotFoundError: No module named 'expyriment'` appears, check [Software Installation](#).

Let us start by downloading `simple-detection-visual-expyriment.py` and run it with:

```
python simple-detection-visual-expyriment.py
```

Then, in the subfolder `data`, locate a file with a name starting with `simple-detection...` and the extension `.xpd`. This is a text file containing the reactions times. To analyse them, download `analyse_rt.py` and run:

```
python analyse_rt.py data/simple-detection-visual-expyriment_*.xpd
```

Compare the codes of `simple-detection-visual-expyriment.py` and `simple-detection-visual-pygame.py`. This should convince you that using `expyriment` will make your life simpler if you need to program a psychology experiment.

The documentation of `expyriment` is available at <http://docs.expyriment.org/>. Have a quick look at it, especially <http://docs.expyriment.org/expyriment.stimuli.html>

The basic principles of the `expyriment` module are introduced in <https://docs.expyriment.org/Tutorial.html>. I provide a minimal template at `expyriment_minimal_template.py` that one can use to start writing a `expyriment` script.

Exercises:

1. Modify `simple-detection-visual-expyriment.py` to display a white disk instead of a cross.
2. Modify `simple-detection-visual-expyriment.py` to display a white disk on half of the trials and a gray disk on the other half of the trials (thesis experimental conditions should be shuffled randomly). Then modify it to display disks with four levels of gray. Thus you can assess the effect of luminosity on detection time. (see `xpy_simple_reaction_times/grey-levels.py` for a solution using `Expyriment`'s `design.Block` and `design.Trial` objects).
3. Modify `simple-detection-visual-expyriment.py` to play a short sound (`click.wav`) in lieu of displaying a visual stimulus (hint: use `stimuli.Audio()`). Thus, you have created a simple audio detection experiment.
4. Download and run `simple-detection-audiovisual.py`:

```
python simple-detection-audiovisual.py
```

There are three blocks of trials: a first one in which the target is always visual, a second one in which it is always a sound, and a third one in which the stimulus is, randomly, visual or auditory. Are we slowed down in the latter condition? Use `analyse_audiovisual_rt.py` to analyse the reaction times.

Exercise: add python code to `simple-detection-audiovisual.py` to display instructions at the start of the experiment.

12.2 Decision times

In the previous example, the user just had to react to a stimulus. This involved a very simple type of decision (“is a target present or not?”)

Other tasks involves taking a decision about some property of the stimulus.

Exercise: | - Modify `simple-detection-visual-expyriment.py` to display, rather than a cross, a random integer between 0 and 9 (hint: Use `stimuli.TextLine()`). Now, the task is to decide if the figure is odd or even, by pressing one of two keys.

Here is a solution: `parity.py`

Comparing the average decision time to the time to react to a simple cross provides a (rough) estimate of the time to decide about the parity of a number. By the way, one can wonder what happens for multiple digits numbers: are we influenced by the flanking digits?

- Add feedback; when the subjects presses the wrong key, play the sound `wrong-answer.ogg`.

Here is a solution: `parity_feedback.py`

12.3 Numerical distance effect

Exercise: Create a script to present, at each trial, a random number between 1 and 99, and ask the subject to decide wether the presented number is smaller or larger than 55. Plot the reactions times as a function of the number.

Do you replicate the distance effect reported by Dehaene, S., Dupoux, E., & Mehler, J. (1990) in “Is numerical comparison digital? Analogical and symbolic effects in two-digit number comparison.” *Journal of Experimental Psychology: Human Perception and Performance*, 16, 626–641.?

12.4 Posner’s attentional cueing task

Exercise (**): Read about [Posner’s attentional cueing task](#) and program the experiment.

See a solution in `Posner-attention/posner_task.py` (you will need `Posner-attention/right-arrow.png`, `Posner-attention/star.png` and `Posner-attention/left-arrow.png`)

12.5 Stroop Effect

The Stroop effect (Stroop, John Ridley (1935). “Studies of interference in serial verbal reactions”. *Journal of Experimental Psychology*. 18 (6): 643–662. doi:10.1037/h0054651) may be the most well known psychology experiment. Naming the color of the ink is difficult when there is a conflict with the word itself. This is interpreted as a proof that reading is automatic, i.e. cannot be inhibited.

In the previous chapter, we created Stroop cards with Pygame.



Stroop card

(see `create_stroop_cards.py`)

Download `stroop.zip`. Extract the files and run:

```
python stroop_task.py
```

The times are in the subfolder `data`. Compute the average reading times as a function of the language (using R or python).

Exercise: Program a Stroop task with a single colored word displayed at each trial. To record actual naming times, you will need to record (!) the subject's vocal response. A simple solution is to run a audio recording application while the script is running. You script should play a brief sound each time you present a target. Then, with a audio editor (e.g. [Audacity](#)), you can locate the times of presentation of stimuli and the onsets of vocal responses. Check out the program "CheckVocal" at <https://github.com/0avasns/CheckVocal> which does just that!

12.6 A general audio visual stimulus presentation script

In some experiments, we know in advance the precise timing of all stimuli (the program flow does not depend on external events). A script that reads the timing of audiovisual stimuli in a csv file and presents them at the expected times is available at <https://www.github.com/chrplr/audiovis>

12.7 Sound-picture matching using a touchscreen

The `sentence-picture-matching.py` script presents a sound, followed by a picture and waits for the participant to press a button.

Exercise: Modify the previous script to present *two* pictures and use `expyriment`'s `TouchScreenButtonBox` to record the subject's response, using the example from `expyriment/touchscreen_test/touchscreen-test.py`

12.8 More examples using Expyriment

Besides the examples from this course, you can find more `expyriment` scripts at

- <https://mbroedl.github.io/cognitive-tasks-for-expyriment/>
- <https://github.com/expyriment/expyriment-stash>

PROGRAMMING A LEXICAL DECISION TASK

In a lexical decision experiment, a string of characters is flashed at the center of the screen and the participant has to decide if it is a actual word or not, indicating his/her decision by pressing a left or right button. Reaction time is measured from the word onset, providing an estimate of the speed of word recognition.

Let us program such a task.

13.1 Step 1: stimuli in constants

Modify the `parity task` script to display either a word or a pseudoword at each trial (in a random order). the task of the subject is to press 'F' when the displayed stimulus is a word, 'J' if it is a pseudowords.

For testing purposes, let us assume that:

```
words = ['bonjour', 'chien', 'président']  
pseudos = ['lopadol', 'mirance', 'clapour' ]
```

Run the script and check the results in `/data`.

Compare your script with the solution proposed `lexdec_v1.py`

13.2 Step 2: read stimuli from a csv file

Modify the lexical decision script so that it reads the stimuli from a comma-separated text file (*stimuli.csv*) with two columns. Here is the content of `stimuli.csv`:

```
item,category  
bonjour,W  
chien,W  
président,W  
lopadol,P  
mirance,P  
clapour,P
```

(hint: To read a csv file, you can use `pandas.read_csv()`)

A solution is proposed in `lexdec_v2.py`

Note: You can use a file comparator, e.g. [meld](#), to compare the two versions:

```
meld lexdec_v1.py lexdec_v2.py
```

Optional;

13.3 Select words in a lexical dabatase

1. Go to <http://www.lexique.org>

Click on “Recherche en Ligne” and play with the interface:

- enter 5...5 in the nblettres field
 - enter ^b.t\$ in the field Word field (see http://www.lexique.org/?page_id=101 for more examples of patterns that can be used)
2. how many words of grammatical category (cgram) ‘NOM’, and of length 5 (nblettres), of lexical frequency (freqfilms2) comprised between 10 and 100 per millions are there in this database? (answer=367). Save these words (i.e. the content of the field Words) into a words.csv file (you may have to clean manually, ie. remove unwanted columns, using Excel or Libreoffice Calc).

13.4 Automatising database searches with R and Python

To select words, rather than using the interface at <http://www.lexique.org>, one can write scripts in R or Python. This promotes reproducible science.

1. Open <https://github.com/chrplr/openlexicon/tree/master/documents/Interroger-Lexique-avec-R> and follow the instructions in the document `interroger-lexique-avec-R.pdf`
2. Read <https://github.com/chrplr/openlexicon/tree/master/scripts>

To select 100 five letters long nouns for our lexical decision, execute:

```
import pandas
lex = pandas.read_csv("http://www.lexique.org/databases/Lexique382/Lexique382.tsv", sep=
    ↪'\t')
subset = lex.loc[(lex.nblettres == 5) & (lex.cgram == "NOM") & (lex.freqfilms2 > 10) &
    ↪(lex.nombre == 's')]
samp = subset.sample(100)
samp2 = samp.rename(columns = {'ortho':'item'})
samp2.item.to_csv('words.csv', index=False)
```

This creates words.csv.

13.5 Generate nonwords

1. Write a function that returns a nonword (a string containing random characters)

```
def pseudo(length):
    """ returns a nonword of length `length` """
```

Solution at `create_nonwords.py`

2. Use this function to create a list of 100 nonwords and save it in a file "pseudowords.csv" (one pseudoword per line) (see <https://www.pythontutorial.net/python-basics/python-write-text-file/>)

13.6 Create a stimuli file

Merge `words.csv` and `pseudowords.csv` into a single `stimuli2.csv` file:

```
import pandas
w = pandas.read_csv('words.csv')
w['category'] = 'W'
p = pandas.read_csv('pseudowords.csv')
p['category'] = 'P'
allstims = pandas.concat([w, p])
allstims.to_csv('stimuli2.csv', index=False)
```

13.7 Use `sys.argv` to pass the name of the file containing the list of stimuli

Modify `lexdec_v2.py` to be able to pass the name of the stimuli file as an argument on the command line:

```
python lexdec_v3.py stimuli2.csv
```

(hint: use `sys.argv[]`; see <https://www.geeksforgeeks.org/how-to-use-sys-argv-in-python/>)

Solution at `lexdec_v3.py`

13.8 Improving the pseudowords

1. Check out the [Unipseudo](#) pseudoword generator.
2. Generate a new list of pseudowords and add them to a new `stimuli3.csv` file

13.9 Data analysis

After running:

```
python lexdec_v3.py stimuli2.csv
```

the subject's responses are stored in the subfolder `data/` contains a file `lexdec...xpd`

You can download this `xpd` file as an example.

1. Use `pandas.read_csv(..., comment='#')` to read the responses into a pandas dataframe.
2. Compute the average reaction times for words and for pseudo-words.
3. Plot the distribution of reactions times using `seaborn.boxplot()`
4. Use `scipy.stats.ttest_ind()` to perform a Student t-test comparing the RTs of Words and Non-Words.

Check a solution `analyze_RT.py`

13.10 Auditory Lexical Decision

Transform `lexdec_v1.py` into an auditory lexical decision script using the sound files from the *lexical decision* folder `<../experiments/xpy_lexical_decision/>`:

```
bonjour.wav  
chien.wav  
président.wav  
clapour.wav  
lopadol.wav  
mirance.wav
```

Solution at `lexdec_audio.py`

13.11 Finally

Check out the example of a ‘real’ lexical decision experiment at <https://chrplr.github.io/PCBS-LexicalDecision/>)

DATA ANALYSES

Contents

- *Data Analyses*
 - *Permutation tests*
 - *Bootstrap*
 - *Basic Data Analysis with R*
 - *Comparing means using Easy ANOVA (Analysis of Variance)*
 - *Frequency Analysis*
- *Lexical Statistics*
 - *Zipf law*
 - *Benford's law*
 - *Neuroimaging*

14.1 Permutation tests

- Read about the principle of [permutation tests](#)
- Implement a python script that uses a permutation test to compare two samples.
- Check out the solution I propose: `permutation_test/permutation_test.py`.

14.2 Bootstrap

- Implement the [bootstrap](#) to obtain confidence intervals on the means of a sample.

14.3 Basic Data Analysis with R

See <http://www.pallier.org/examples-of-basic-data-analyses-with-r.html#examples-of-basic-data-analyses-with-r>

14.4 Comparing means using Easy ANOVA (Analysis of Variance)

See <http://www.pallier.org/easy-anova-with-r.html#easy-anova-with-r>

Note: Check out David Lakens's github and, in particular, https://lakens.github.io/statistical_inferences/repository

14.5 Frequency Analysis

- See `short-intro-fourier`

LEXICAL STATISTICS

15.1 Zipf law

- The script `Zipf/word_count.py` computes the distribution of frequencies of occurrences in a list of words. Use it to compute the distribution of word frequencies in *Alice in Wonderland*.

Note: To remove the punctuation, you can use the following function:

```
import string
def remove_punctuation(text):
    punct = string.punctuation + chr(10)
    return text.translate(str.maketrans(punct, " " * len(punct)))
```

- Zipf law states that the product rank X frequency is roughly constant. This ‘law’ was discovered by Estoup and popularized by Zipf. See http://en.wikipedia.org/wiki/Zipf%27s_law. Create the Zipf plot for the text of *Alice in Wonderland* showing, on the y axis, the log of the frequency and on the x axis the word rank (sorting words from the most frequent to the least frequent).
- Display the relationship between word length and word frequencies from the data in `lexical-decision/lexique382-reduced.txt`
- Generate random text (each letter from a-z being equiprobable, and the space character being 8 times more probable) of 1 million characters. Compute the frequencies of each ‘pseudowords’ and plot the rank/frequency diagram.
- To know more about lexical frequencies:
 - Read Harald Baayen (2001) *Word Frequency Distributions* Kluwer Academic Publishers.
 - Read Michel, Jean-Baptiste, Yuan Kui Shen, Aviva P. Aiden, Adrian Veres, Matthew K. Gray, The Google Books Team, Joseph P. Pickett, et al. 2010. “Quantitative Analysis of Culture Using Millions of Digitized Books.” *Science*, December. <https://doi.org/10.1126/science.1199644>. (use scholar.google.com to find a pdf copy). Check out **google ngrams** at <https://books.google.com/ngrams>. (Note that at the bottom of the page, there is a message “Raw data is available for download here”).

15.2 Benford's law

Learn about [Benford's law](#). Write a Python script that displays the distribution of the most significant digit in a set of numbers. Apply it to the variables in [Benford-law/countries.xlsx](#).

A solution: [Benford-law/Benford.py](#)

15.3 Neuroimaging

- Check out [nilearn](#) and [nistats](#) and [MNE-python](#)
- See [stats-and-data-analyses/Example of a single subject-single run fMRI analysis with nistats.ipynb](#)

USING HTML FOR WEBPAGES

16.1 HTML Basics

Browser display (including with jsPsych) relies on HTML, most often HTML5. Let us see how it works.

To go further The point of this lecture is to raise your understanding of HTML, CSS, JS & JsPsych enough to code your own experiments. For those who may want to go further / better understand the magic at hand, I will leave notes as the present one in the form of quotes. They are not necessary, but may answer questions you may have.

16.1.1 HTML document structure overview

An HTML file can be as simple as this: open your notepad, copy-paste the following lines, and save the file as `test.html` (or whatever name you want, but keep the `.html` extension).

```
<!DOCTYPE html>
<html>
  <head></head>
  <body></body>
</html>
```

Now try to open this file (if your computer asks, use a web browser). Congratulations, you just created your first HTML page ! Granted, it's a bit bland; we'll populate this later. Let us quickly go over what we have here.

Our HTML file contains: - A declaration that we are using HTML5;

```
<!DOCTYPE html>
```

- A head where most webpage-level elements (e.g. title) will be defined,

```
<head></head>
```

- A body which will contain the main elements of the page, especially what will be displayed to the user.

```
<body></body>
```

What happens if you remove DOCTYPE? - A note on browser compatibility Most browsers will not care and behave similarly. However, you have no guarantee that it will do so: in fact, Internet Explorer will just treat your file as plain text! In the case of an online experiment, this means you may lose some participants/data.

Tags in HTML Here, we write `head` and `body` between chevrons `<>`. This is known as a *tag* in HTML. Tags tell information to your browser, here that this is the head of the document.

Notice that we have two tags here, one after another. Think of them as brackets: everything that we want to define in the head will be enclosed between the tags. Accordingly, the first tag is called the *opening tag* and the other one the *closing tag*. Closing tags specifically have an additional / slash at the start).

Note that a few tags do not enclose content and only use a single tag, with the slash / at the end (e.g. line breaks `
`).

16.1.2 Setting up text and title

So far, our document is empty: it does only display a blank page and the title is basically the files path. Can you change the title to My HTML file and display the following text in your page?

Hello, world!

You can find a solution in the following file: `hello_world.html`. But before looking at it...

Some questions for you! - Did you end up having My HTML file Hello, world! written on your page?

This is expected and means that you probably miss a tag: how can your browser know that you want this to be a title? Placing it inside the head is not sufficient: you need to use the `<title></title>` tag!

- Did you specify the title within the head, and the text within the body?

If so, good, but would it work otherwise? Try, and see that it could indeed. But we generally prefer to do this kind of stuff this way. More onto this in the lecture if I have time.

- What happens if you try to write `Enchanté, cher monde !` instead of `Hello world!`?

There is a chance that some of you will have a weird character at the end of `Enchanté`, while others will have no issues. This is because your browser does not necessarily know how to interpret those weird diacritics that non-English speakers use. To fix this issue, you can specify to your computer that you want to use utf-8 encoding, inserting `<meta charset="utf-8">` in the head of your file.

16.2 Simple shapes

Let us now create more interesting pages than text alone! Just like we did with python, we start with shapes.

16.2.1 Square

Creating divs

Rectangles are very simple shapes to draw in HTML. Although there are a variety of way, we will start by using colored 'divs'. Here is the code for what we will first try to achieve: `square-div.html`.

Divs Divs are precisely HTML element with a rectangle shape. They are most often used as generic containers, but this won't interest us right now.

You may use the following body for your HTML code.

```
<body>
  <div></div>
</body>
```

As you may notice, the page is still blank. Press f12 to understand why.

Inspector On Windows and Linux, f12 opens your browser's inspector, which allows you to see the HTML code of the web page you are currently browsing. It can be a little less straightforward on MacOS depending on the distribution; you may find some guidance [here](#).

Open the body tags and hover over the <div> element. It should show you the element on the webpage, and give you its dimensions. Notice the issue? It is simply of width 0, so of course you won't see it.

TODO IMAGE

Setting size

Let's specify a width for our <div>. To do so, we will add specifications to our tag, so that the browser knows how to deal with the element it marks. Here, we will use the `style` keyword to specify a style that forces a 200px width and a 200px height.

The result is as follows:

```
<body>
<div style = "width: 200px; height: 200px"></div>
</body>
```

Notice that the style specification has a precise syntax: `keyword: value`, with successive entries being separated by semicolons `;`. The style won't be applied if you omit semicolons, or use equal sign instead of colon `:`! Similarly, the value part must have a unit. Here we use pixels (`px`), but there are many others!

Setting size with style in HTML Here we use `style` to specify the width and height of the element. There are other ways, with specific `width` and `height` tags. However, these specifications may behave unexpectedly at times, which is why we will use `style` in this lecture.

Size units in HTML To set the size of an element, we have many useful units that can adapt to each screen. Here we used pixels (`px`) which are the base unit of computer screens. Since pixel size may vary between computers, we could also use centimeters (`cm`) to get a constant value. Conversely, we could want to adapt our display to the size of the window, and use viewport height (`vh`) and width (`vw`). If we want more specifically to adapt to a given container, we can use percents (`%`).

Setting background color

If you update the page, you'll see that you in fact still don't see the div. Check again with f12; it should highlight an actual square this time. The reason why you don't see it is that, by default, elements take the background color of their parent, here <body>. So you are looking at a white square on a white background, which is a good reason not to see it!

To specify the color (actually background color of the square), you may use another specification in the style:

```
<body>
<div style = "...; background-color: red"></div>
</body>
```

Names with spaces Names with spaces are always annoying when programming, since they should actually be taken as a whole by the language. To prevent this, several alternatives exist (such as CamelCase or snake_case), with each language having its usually preferred alternative. In HTML/CSS, we replace spaces with dashes `-`.

Changing background color of the body Like with any other elements, you can change the style of the body. Try setting it to gray with the `background-color` specification!

Centering

At this point, you should finally have a square ! However, it lies sad and alone in the corner of the screen. We'll see more on the placing of elements, but for now we will stick to simple solutions.

First, we can specify the position of the left corner on the square in the style. This works similar to setting the dimensions of the square.

```
<div style = "...; top: 100px; left: 200px"></div>
```

Although we are moving the square, it is still not centered on the screen. It is pointless to use trial-and-error here, as it won't be centered anymore if you resize your browser window. To get a unit relative to the size of the window, we will use viewport height (vh) and width (vw). 1vh correspond to 1% of the *height* of the window. 1vw is 1% of the *width* of the window. Do not confuse them!

As such, we can (somewhat) center the square using the following style:

```
<div style = "...; top: 50vh; left: 50vw"></div>
```

Notice that we are still slightly off, since we actually centered the top left corner of the square. To correct this we will apply a simple translation, of half the square dimensions.

```
<div style = "...;
  top: 50vh; left: 50vw;
  transform: translate(-50%, -50%)"></div>
```

Percent unit The percent unit % refers to the dimension of the parent container. E.g., for our `div` within the body, setting `top` and `left` to 50% would put our top left corner to the center of the body. Here, with the call to `translate`, it becomes as if self centered, and the translation is thus of 50% of the *square* size.

ID

We can specify the id of an element using `id = "my-id"`.

IDs are not necessary, but they come in handy for several reasons. The main reason for us now is to be able to identify component in the inspector view. It also helps identification of the element by other elements, which helps for applying a specific style (more later) or retrieving the element in JavaScript (more even later, see next session).

And voilà, we have a neat centered square!

16.2.2 Circle

Rounded divs

Let's move on to the next shape: a circle! We will create it in two ways. The first way will use `divs`, as we just saw: `script`.

As said above, `<div>s` are rectangle elements, but they may also be slightly modified. As an example, their corners can be rounded, a property which we will make use of to make circles. For that we will use a `border-radius` specification within our style.

```
<div style = "...; border-radius: 50%"></div>
```


You may try and change the value of this `border-radius`, to better understand the behavior we're making use of. Notice how much we start definitely resorting to tricks here, which may (and will) be insufficient at some point. HTML proposes alternatives that are more suited to drawing shapes, such as *Scalable Vector Graphics* (SVG). The adapted code can be found [here](#).

SVGs

In HTML, SVGs are elements like `divs`, but which are designed to contain shapes. Here we will use the `<circle>` shape element. We will specify its properties (radius, center, color) with tags directly linked to the element.

```
<svg>
  <circle cx="100" cy="100" r="100" fill="red"/>
</svg>
```

Notice that we are at the same level as style **TODO** Also notice that here we space things with spaces and not semicolons. some attributes are specific to `<circle>`

What is going wrong here? Well, `fl2` can enlighten us here again. As you may see, the circle is cut by the border of the container. In other words, our 150x300 pixels containers does not have the right shape to display the whole shape. We thus have to specify the size of the container, with the usual `style` attribute.

```
<svg style = "height: 200px; width: 200px">
  <circle cx="100" cy="100" r="100" fill="red"/>
</svg>
```

16.2.3 Triangle

A good reason to learn about SVGs is that you can't draw triangles with `divs` (or rather, you will have an extremely hard time doing so). With SVGs, doing so is much easier, as you can draw any polygon using the `<polygon/>` tag. `<polygon/>` takes a specific attribute named `points` which takes a list of integers corresponding to the coordinates of the polygon's vertices. Integers in the list will be paired to create the x and y coordinates of each point.

```
<svg>
  <polygon points="0 200, 200 200, 100 0" fill="red" />
</svg>
```

You may separate integers with spaces `` `` or commas , alike. In the code for an isocetes triangle above (full file), I use a mixture of both: spaces separate x and y coordinates, while commas separates vertices.

16.2.4 Style usage

Regardless of whether you used `divs` or `svgs` above, you most likely used the same `style` attribute to center the shape, over and over. To avoid tiresome repetitions, HTML provides a convenient way to deal with this: providing a stylesheet. A stylesheet essentially defines keywords, which can be later used to apply the desired style to an element. You may find an example for our square [here](#).

Definition in the head

The simplest way to define a style is to do so in the head of your document. You can also do it in a separate file; more on that later.

```
<head>
  <style>
    <!-- Put the style here -->
  </style>
</head>
```

Comments in HTML The `<!--` and `-->` serve as opening and closing markers for comments in HTML. This is made so that you'll (hopefully) never need them for any other purpose, since HTML is designed to display all kinds of texts.

We can now define our stylesheet. First, let us make all divs have a red background by default.

```
<style>
  div {
    background-color: red;
  }
</style>
```

This property can now be removed from the style of the `<div>` elements of the body. Try it!

We now want to deal with the centering elements. Since we don't want to center everything, we'll manually flag elements that should be centered using the `class` attribute. To define a style for a class named `my-class`, we reuse the same syntax as before, but replace the element name (`div`) with the class name `my-class` preceded by a dot `.`. The dot indicates that this style applies to a class.

```
<style>
  .centered {
    position: absolute;
    top: 50vh; left: 50vw;
    transform: translate(-50%, -50%);
  }
</style>
```

Cascading Style Sheets Style sheets can apply at several levels: to all elements of the document, to all elements of a kind (e.g. `divs`), to all elements of a special class (defined with the `class` attribute), or elements with a given `id`... These levels apply one after another, with most specific style sheets applying over the more generic ones; they are, in a sense, cascading. This precisely gave this 'style' language its name: *Cascading Style Sheets*, or *CSS* for short.

To apply this style to our divs, we have to specify that this class applies such as in the following example.

```
<body>
  <div class = "centered">
  </div>
</body>
```

Multiple classes You may apply several classes to a single element, simply by listing them with a space in between different classes: e.g. `class = "centered circle"` if you also happen to have a `.circle` style.

Definition in a separate file

Of course, redefining it at the beginning of each sheet can be very tedious, which is why style sheets are often defined in their own .css file. Move everything we previously defined within `<style>` into a file named `shapes.css`. You may now load the style in your HTML file, using the following code in the `<head>` section. Here is how it looks like in our `square` file, using a separate spreadsheet. Notice how the code is much simpler to read!

```
<head>
  <link rel="stylesheet" href = "../shapes.css">
</link>
</head>
```

Be careful, if you move the file from the current folder you will have to update the `href` attribute with the new path!

16.3 Exercises

It is now your turn.

16.3.1 Recreate the shapes

Could you rewrite the code for the circle (a solution [here](#)) and the triangle ([here](#))? Bonus points if you manage to use a single stylesheet for both!

16.3.2 Illusions

Could you recreate the complex stimuli seen in [this lecture](#), this time in html? 1. The two circles (a solution [here](#)) 2. The troxler effect ([here](#)) 3. Kanisza's square ([here](#))

And anything else your heart may wish for! Remember that programming makes perfect (in programming, at least).

USING JAVASCRIPT

17.1 Combining shapes with JS

[The previous chapter] should have given you the basics to recreate, using HTML, the following illusions from [previous lectures on Python](#): 1. The two circles (a solution [here](#)) 2. The troxler effect ([here](#)) 3. Kanisza's square ([here](#))

If you did Kanisza (or peeked at the solution), you may have notice that we didn't actually draw circle slices, but rather hid the undesired parts of the circle with a square. This is because there is no simple way to do it with the tools we have now.

The issue of the present design Since the result is visually satisfying, one may think it is not a big deal to leave it as such. However, remember that the whole point of the Kanisza illusion is to trigger a form *that does not exist in the first place!* You do not always control what happens on the screen, and as such this may introduce some terrible noise in your data. As an example, since HTML elements are actually displayed one after another, old computers might show the square with a delay that could be a confounding factor to the effect you want to show!

In the next section, we will learn how to draw these slices using canvas. These are some sort of 'drawing boards' that have to be drawn upon using JavaScript.

17.1.1 Plugging JavaScript into HTML

You can plug a JavaScript script in HTML using the `<script>` tag. Note that everything within this tag will be interpreted as JavaScript.

For our first script, we will display a simple text on the console. To this end, we may use the line code `console.log(myText)`.

```
<body>
  <script type="javascript">
    // All that is written here is JavaScript!
    console.log("Bonjour le monde !");
  </script>
</body>
```

TODO Here we use the method `log` from the object `console`. This relationship is embodied by the `.` between the two.

Do not expect to see anything on your HTML page! The text is printed in the console, which you can access alongside the inspector. This can be very useful for debugging!

17.1.2 Basic syntax of JavaScript

The following code shows you the basics of the JavaScript syntax. You may try it in your browser console, which is accessible in the same window as the inspector.

```
// We define a variable x with value 0.
let a = 0;

// We define a function that prints a number
function printNumber(x){
  console.log(x);
}

// We define a function that add 1
function add1(x){
  return x + 1;
}

printNumber(a);
a = 1
printNumber(a);
```

The semicolon ; is facultative if you use line breaks.

17.1.3 Loops

JavaScript uses two kind of loops you may be familiar with: `for` loops, and `while` loops

For loops

`For` loops are used to execute a piece of code a *given* number of times. As an example, below we want to print 5 integers, from 0 to 4.

```
for (let i = 0; i < 5; i++){
  console.log(i);
}
```

The way to understand the code is as follows: “starting from `i=0`, do `i++` (increase `i` by 1) as long as `i < 5`”.

JavaScript also allows you to loop through collections such as lists. You can define a list between square brackets `[]` using commas , as separators, like this : `[1, 2, 3]`.

There are two ways to loop through a list: `for (let x in l)` and `for (let x of l)`. Try this in your browsers. Can you spot the difference?

```
for (let x in [1,2,3]){
  console.log(x);
}

console.log("---");

for (let x of [1, 2, 3]){
  console.log(x);
}
```

17.2 Modifying elements with innerHTML

The main interest of javascript is that it can interact with HTML. As an example, you can directly modify the HTML code of a document using `document.body.innerHTML`. Here is an example.

```
document.body.innerHTML +=
  "<div style = 'background-color:red; height: 200px; width: 200px'></div>"
```

Multiline strings in JS It is done by adding a backslash \ continuation at the end of each line.

```
"This is \
a \
multiline string".
```

Be careful not putting any space after the continuation!

17.3 Modifying elements with pure JS

Of course, this innerHTML technique will not bring us very far, since we are basically rewriting the HTML code with an additional JavaScript layer... Hopefully, we can also create everything directly using JavaScript. We can create an element using the `document.createElement` method. We simply have to specify what kind of element we want to have, e.g., a div, by passing it as an argument: `let element = document.createElement("div")`.

Then, we can modify its attributes, among which its style, using the following code snippets: `element.id = "my-id"` or `element.style.height = "200px"`. You can find here a code for the red square, with the JavaScript part detailed below.

```
let square = document.createElement('div');
square.id = "my-square";
square.style.background = "red";
square.style.position = "absolute";
square.style.width = "200px";
square.style.height = "200px";
square.style.top = "50vh";
square.style.left = "50vw";
square.style.transform = "translate(-50%, -50%)";

// Do not forget to add your element to the document!
document.body.appendChild(square)
```

Chains of properties. Note the weird `square.style.X` syntax. This is because JavaScript works with objects: `square` is an object, that is a kind of box that stores several variables and functions. One of these variables is the `id`, which we access with `square.id`. Another is the `style`, which we access with `square.style`. The `style` itself is an object, that has many variables like the `background`, the `position`... all of which are character strings, or strings for short.

17.4 Drawing on canvas.

Back to our problem of Kanisza's square, we can introduce canvas, which are exactly that: canvas on which we will paint. We can create the canvas using the same method as above. You can find the resulting file [here](#).

```
let canvas = document.getElementById("tutorial");
```

To paint on it, we will want to access its 'context'.

```
let ctx = canvas.getContext("2d");
```

Here is some code to draw a 50x50 rectangle, at the position (10, 10) –of the canvas!

```
// Specify the color
ctx.fillStyle = "rgb(200, 0, 0)";
// Fill the rectangle
ctx.fillRect(10, 10, 50, 50);
```

Drawing a circle is slightly more complicated, because there is no proper built-in function. Instead, we will draw the path of our brush.

```
ctx.beginPath();
// Draw a full circle (from 0 to 2pi radians) at position (100, 75) with radius 50px
ctx.arc(100, 75, 50, 0, 2 * Math.PI);
// Draw the path
ctx.stroke();

// Or we could fill it!
// ctx.fill();
```

**** TODO** a note about style vs height & width

Think that you'll fill based on your starting point! You can move it use *ctx.moveTo(x,y)*.

17.4.1 Back to Kanisza

You can now recreate a cleaner version of Kanisza's square: [here](#)

17.5 Using JsPsych

JsPsych is a library that allows you to easily create experiments from premade plugins. First, download the library in version 7.3.0 from the [following link](#), and unzip it in your code folder. The following codes assume that the folder is named `jspsych-7.3.0`.

We will start by creating a very simple 'experiment' that greets the participant and registers any key they press : `jspsych-hello-world-example.html`.

17.5.1 Loading JsPsych

The library itself consists in the `jspsych.js` JavaScript file, which we will load in our experiment. To load an external script in HTML, one can simply use the `src` attribute of the `<script>` tag, with the path to the script file as a value.

```
<!DOCTYPE html>
<head>
  <title>A simple jsPsych experiment</title>
</head>
<body>
  <script src="./jspsych-7.3.0/jspsych.js">
  </script>
</body>
```

Here, you only loaded all the helper functions of JsPsych. You will now create an instance of the plugin using `initJsPsych`, which will handle all your JsPsych-related instructions.

```
const jsPsych = initJsPsych();
```

Constants Notice that here we use a `const` instead of a `let` or `var` declaration. This means that the value of this variable can not be changed. This is convenient to prevent undesired bugs from redeclaring a variable.

17.5.2 Timeline and trials

As said in the introduction of the JsPsych lecture series, JsPsych revolves around successive trials forming what is called a *timeline*. This timeline is implemented as an array containing all the trials. Arrays in JavaScript are defined using square brackets `[]`. We will first start with an empty timeline, which we'll gradually fill.

```
let timeline = [];
```

Initializing non-empty arrays Arrays may be implemented with items already in them, by simply putting the items within the square brackets `[]` and separating them with commas `,`. As an example, if you already have two trials `trial1` and `trial2`, you may create an array containing both (in this order) with `[trial1, trial2]`.

We now want to create trials to fill our timeline with. You can think of trials as a parametrized task, with the task being effectively encoded as a JsPsych plugin.

For now, we will stick to simple decision tasks. Stimuli will be displayed from simple HTML code similar to what we used previously. The dedicated plugin is (logically) called `jsPsychHtmlKeyboardResponse`.

We can thus instantiate a trial with this plugin, using an object structure. Long story short, an object structure is defined using brackets `{}`; it holds properties, defined with `name: value`, and separated by commas `,`. Below is the instantiation of a `jsPsychHtmlKeyboardResponse` trial.

```
let trial = {
  type: jsPsychHtmlKeyboardResponse,
};
```

Trailing commas You may notice I left a comma `,` after the `type` property, although I did not specify any other property. This is not a typo: it is what we call a *trailing comma*. JavaScript licenses them as it makes it easy to add new elements.

You may now add the trial to the timeline using the `push` method of arrays, which adds an element at the end of it.

```
timeline.push(trial);
```

In-place modifications TODO

And we can finally run the experiment with our 1-trial timeline, using the jsPsych instance we previously created.

```
jsPsych.run(timeline);
```

Your final code should look like this:

```
// We initialize JsPsych
const jsPsych = initJsPsych();

// We create an empty timeline
let timeline = [];

// We create a basic decision trial
let trial = {
  type: jsPsychHtmlKeyboardResponse,
};

// We add this trial to the timeline
// /\ Do not forget this essential step /\
timeline.push(trial);

// We run the timeline with JsPsych
jsPsych.run(timeline);
```

You may now run it by opening your HTML page. Press a key and see what happens .

If nothing happens (and this should be the case!), just do as you should always do in this situation: open the console. It should display you the following error message in red: “You must specify a value for the stimulus parameter in the html-keyboard-response plugin.”. Such errors are fatal and prevent the script from proceeding any further.

The issue here is that, although we did specify the type of our trial, we did not give it the necessary parameters for it to run properly. As the message tells us, we actually didn’t specify what stimulus this decision task was about. In fact, the plugin displays “unspecified” as the top of the page.

Let us first specify a simple text prompting to press any key as our stimulus. We can do it as follows.

```
let trial = {
  type: jsPsychHtmlKeyboardResponse,
  stimulus: "Bonjour! Please press any key."
}
```

Now, loading the page should prompt you with the text you entered. If you press any key, it disappears: the experiment is actually finished.

We could also use `jsPsychImageKeyboardResponse` if we want to pre-generate our stimuli as images and display them directly. More precisions [here](#).

17.5.3 Using the console interactively: accessing experiment data

Before going any further, let us test that the experiment worked as intended. If so, the data in our trial should have been registered. You can access JsPsych's saved data using `jsPsych.data.get()`

If we break down this line, here we access the property `data` of our `jsPsych` instance. But `data` actually saves many meta-informations which are not of interest to us. Luckily; this `data` object has a convenient function (or method) `get()` that allows us to precisely access test data.

Although you could use it in your script to access it at any given time (and, e.g. print it), you can also use the console to access it whenever you want. Just type the line into it!

It should print you something of the form `Object { trials: (1) [...] }`, which you can unfold: `trials` precisely contain the data about each trial. Right now, it should only contain one single trial, as an object with `rt`, `stimulus`, and `response` properties.

17.5.4 Response keys

In your trial's data, `response` may contain any single key, since all are allowed by default. However, decision tasks will require them to press one of two chosen keys. We can specify the valid keys using (yet another) parameter: `choices`. As a value, we will pass it an array of valid keys in the forms of strings, here `'f'` and `'j'`

```
let trial = {
  type: jsPsychHtmlKeyboardResponse,
  stimulus: "Bonjour! Please press any key."
  choices: ['f', 'j'].
}
```

17.6 Practice: color-detection task

You should now be able to program a simple experiment. Say we want to test if shapes interfere with color detection: subjects will have to flag the color of successive shapes. They will have to press `'f'` for red shapes and `'j'` for blue shapes. The design should be 3 shapes (rectangle, triangle, circle) by 2 colors (red and blue), with 6 trials in total. The order will be fixed, and you are in charge of choosing it!

Beware of priming effects!

You can find a solution [here](#).

Difference between viewport width (``vw``) and height (``vh``) and percents (``%``) If you used percents, you may notice that the figures are slightly off. JsPsych uses a content wrapper, so `%` refers to it size.

17.6.1 Randomizing order

Of course, an experiment with trials in a fixed order is not interesting, because any effect we find may be restricted to this specific order.

JsPsych provides use with a function to shuffle an array, i.e. order its element randomly: `jsPsych.randomization.shuffleNoRepeats`. To randomize the timeline, use:

```
timeline = jsPsych.randomization.shuffleNoRepeats(timeline);
```

Here, we create a random array from the timeline. The `...NoRepeat`s part specifies that equal elements are not in successive order. Since we only have a single occurrence of each trial, no item in our timeline is equal, and it thus does not have any effect here.

However, it allows more to do more than prevent repetition of identical trials: we can also specifically define what it means to be equal. To do so, we simply pass an additional argument: a function that returns whether two trials are equals. Here, we want to define equal trials as those which have the same shape.

First, let's add a shape property to our trial object. If you coded cleanly, creating a trial should be done using parameters (in a `for` loop or even better a function) including a `shape` variable. Adding it to the trial should thus be fairly straightforward.

```
trial = {
  ...
  color: color;
}
```

Additional properties to the trial In JsPsych, a trial is a javascript object that uses some mandatory and/or optional properties. It will only ever look up those, but that doesn't mean you can not add other properties.

You may check with the console that properties added this way will not be added in the data! The next session will develop how to do it.

In the mean time, we can now define our equality function:

```
timeline = jsPsych.randomization.shuffleNoRepeat(timeline,
  function(trial1, trial2){return trial1.shape == trial2.shape});
```

Factorial design We used here a 3 by 2 factorial design, which was simple enough to generate with a `for` loop. For more complicated factorial design, you may want to look up the `jsPsych.randomization.factorial` function <<https://www.jspsych.org/7.0/reference/jspsych-randomization/#jspsychrandomizationfactorial>>`__.

17.6.2 Adding data to be saved

Although we could theoretically retrieve the color and property from the HTML string, it would be rather uneasy. We can rather save directly `color` and `shape` values in our data, using the `data` property of our trial. `data` will be an object that contains, as properties, everything we might want to plug into our data.

```
let trial = {
  ...
  data: {color: "red", shape: "blue"},
}
```

As a small exercise: how can we update our equality test function?

17.6.3 Saving answer

If you go through the trials and try to analyse your data, you may notice that `response` only contains the pressed keys, and not the color responded by the participant. While you could theoretically reconstruct it during your data analysis, this approach is error-prone (in particular when you randomly assign responses keys).

Random response keys It is advised to randomly assign response keys to your participants, since there are some known interactions between response side and task performance (see, e.g., [the SNARC effect](#)). To implement such a random choice, you may want to have a look at the `Math.random` function https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/random from native JavaScript.

However, since the response is not known *a priori*, there is not much you can do as you create the trial (but you should register response side for safekeeping!). JsPsych provides us with a neat workaround with the `on_finish` property of trials. `on_finish` has to be a function that takes the trial's data as an argument; it is not expected to return anything.

We can thus use `on_finish` to modify the response encoded in our data:

```
let trial = {
  ...,
  on_finish: function(data){
    // We first save the response key in a more adequate variable
    data.responseKey = data.response;

    // We then save the actual responded color as the response
    if(data.responseKey == "f"){
      data.response = "red";
    } else {
      data.response = "blue";
    }
  }
};
```

Ternary operators The if-else construction here is rather cumbersome. Most languages (including JavaScript) offer a ternary operator `?:` that allow to replace it: `condition ? a : b` is `a` when `condition` is true, and `b` otherwise. Try it!

This design is however **very** error-prone: if the `[F]` key is not literally encoded as the character `"f"` (or whichever you use here), it may assign the wrong color to the response key! You also have to adapt everything each time you want to change the keys or the color.

We'll only focus on the first issue of key encoding here, since you should be able to have a code that is more robust to keys/color changes on your own. JsPsych provides us with a way to compare the encoding of a key to a representation such as `"f"`: `jsPsych.pluginAPI.compareKeys`.

```
let trial = {
  ...,
  on_finish: function(data){
    // We first save the response key in a more adequate variable
    data.responseKey = data.response;

    // We then save the actual responded color as the response
    if(jsPsych.pluginAPI.compareKeys(data.responseKey, "f")){
      data.response = "red";
    } else {
      data.response = "blue";
    }
  }
};
```

(continues on next page)

(continued from previous page)

```

    }
  }
};

```

17.6.4 Audio feedback

Here are two .wav sounds: `correct.wav` and `incorrect.wav`. We want to play them at the end of the trial to give audio feedback to our participants.

To play audio in JavaScript, you first have to create `Audio` objects containing the audio file you want to play.

```
let audio = new Audio(pathToFile);
```

You can now play the audio using the `play` function of this audio object:

```
audio.play()
```

As small exercise, you should now be able to play a valid auditory feedback at the end of every trial. Hint below!

Hint

You should use the ``on_finish`` property we saw above!

17.6.5 Saving the data

The experiment is almost ready! What we want to do now is to save our data. It can be saved locally (on the machine that took the experiment), or, more interestingly, on a distant server.

In this course, we will only use local save, which is still useful for debugging and/or piloting. Our `data` object possesses a `localSave` method that precisely saves the experiment's data as a .csv file:

```
jsPsych.data.get().localSave('csv', "data.csv");
```

Where (i.e. when) to should this instruction be executed? At the end of the experiment! Similarly to trials, our `JsPsych` instance can be created with an additional `on_finish` method. Note that unlike for trials, this one does not take a `data` argument.

```
let jsPsych = initJsPsych({
  on_finish: function(){
    jsPsych.data.get().localSave('csv', "data.csv");
  }
})
```

You may be surprised that we make a reference to the variable `jsPsych` within its actual creation. This is possible because JavaScript will not evaluate functions before actually calling them. In other words, when `on_finish` is called at the end of the trial, the function will then (and only then) look at whatever variable labeled `jsPsych` it can find. By then, we will have created the variable already and so it will work. I personally dislike this design which is error-prone (what if some code changes the value of `jsPsych`?); however, this is what is officially used in [JsPsych's documentation](#). One protection I can propose is to make `jsPsych` a constant with the `const` keyword. In JS like in most languages, constants have a name in capital letters and spaces `""` are replaced by underscores `_`: `JS_PSYCH`.

Of course, you want to go further than just storing the data on the participant's computer. We want to retrieve it on our laboratory server! Since the code will be very tributary of how said server is set up, you should see details with your lab's referent (where can you store the code, what protections...). You may find some documentation [here](#)

17.6.6 Random ID

You may notice that we haven't done anything about participant IDs. Assigning each participant a *random* ID is of course mandatory in psychology experiments. JsPsych provides use with a convenient way to generate random IDs of a given length: `jsPsych.randomization.randomID`.

We can create a 10-character long ID for our participant with the following line. We use a constant here because it should never be modified.

```
const ID = jsPsych.randomization.randomID(10);
```

We can now add this ID info to all our trials. To this end, you can modify each trial individually using the `data` property as above. Another way is to add a common property to the whole data, as describe in the [documentation](#).

As a final note, you will most likely want to use this ID for the data file you save at the end of the experiment: if all participants' files have the same name, they will overwrite one another!

```
jsPsych.data.get().localSave('csv', "data-"+ID+".csv");
```

String formatting To get a cleaner script, you may use string formatting to plugging code output into a string. Formatted string use this quote ``and have codes marked between brackets, the opening bracket being preceded by a dollar sign\$. An exemple: `Bonjour! My name is ${my_name}!`.

17.6.7 Final code

You can find a solution for the final code [here](#). Make sure to try out to code it first! Practice makes perfect.

I did not do it in this example, but you should leave an end message to your participants, thanking them for their time. You can create a `jsPsychHtmlKeyboardResponse` trial with no possible response by giving the `choices` property the `"NO_KEYS"` value.

REGULAR EXPRESSIONS

18.1 Tutorial

1. Watch the following introduction video to see how useful regular expressions can be.

Use of regular expressions to extract and transform information from text.

[Watch on youtube](#)

2. Read the first two sections in Chapter 7 of Al Sweigart's book "Automate the boring stuff", page 161 to 166.

This will show you how regular expressions can be used in Python.

The book is [available on Schoology](#)

3. Do all the lessons of the interactive tutorial at regexone.com

This will teach you, through practice, the syntax of regular expressions.

4. Continue onto the practice problems at regexone.com

To practice putting regular expressions to use in real-world problems.

18.2 Example in Cognitive Science research

Regular expressions have many applications, especially in natural language processing. Check out, for example, the chapter [Regular Expressions](#), [Text Normalization](#), [Edit Distance](#) from Dan Jurafsky and Alex Martin's book [Speech and Language Processing](#).

Regular expressions were used to locate syllable boundaries in the phonetic representation of words by [Pallier \(1999\)](#). [Syllabation des représentations phonétiques de brulex et de lexique](#). (see <https://github.com/chrplr/openlexicon/tree/master/scripts/french-syllabation> for scripts and more).

18.3 More on the Theory

A regular regression describes a *formal language*. A formal language is a set of strings over a finite alphabet. For example, the set $L = \{ a^n b^n \mid n > 0 \}$ is a formal language.

A formal language is often defined by means of a *formal grammar*: a start symbol and a set of *production rules* that, when applied, generate all the strings of the language but no more. For example, the grammar with start symbol 'S' and production rules $S \rightarrow aSb$, $S \rightarrow ab$ generates L.

The problem of *matching* or *recognizing* a language is to decide whether a given strings belongs to the language (as when deciding whether a string matches a regular expression). Depending on the language, this can be more or less

difficult to solve. For example, L cannot be recognized by a finite-state automaton, but can be recognized by a type of automaton equipped with a stack (known as a pushdown automaton). One such automaton would push onto its stack each ‘a’ encountered starting from the beginning of the string, then pop one ‘a’ from the stack for each ‘b’ encountered, and would decide that the string matches iff the stack is empty when reaching the end of the string.

Chomsky’s hierarchy establishes a direct correspondance between certain classes of languages or grammars and certain types of machines/automata that can recognize them. See [the wikipedia article](#)

Grammar	Languages	Automaton	Production rules (constraints)*	Examples ^[3]
Type-0	Recursively enumerable	Turing machine	$\gamma \rightarrow \alpha$ (no constraints)	$L = \{w w \text{ describes a terminating Turing machine}\}$
Type-1	Context-sensitive	Linear-bounded non-deterministic Turing machine	$\alpha A \beta \rightarrow \alpha \gamma \beta$	$L = \{a^n b^n c^n n > 0\}$
Type-2	Context-free	Non-deterministic pushdown automaton	$A \rightarrow \alpha$	$L = \{a^n b^n n > 0\}$
Type-3	Regular	Finite state automaton	$A \rightarrow a$ and $A \rightarrow aB$	$L = \{a^n n \geq 0\}$

Fig. 1: Chomsky’s hierarchy (copied from wikipedia)

Regular expressions are a class of languages that can be recognized very efficiently while still being expressive enough for many applications, which is why they are such a practical tool for everyday practice. (Note: The “regular expressions” provided in modern computer software are a bit more expressive than the narrower “regular language” class as defined in theory, because they include additional mechanisms such as lookahead).

To learn more about theory, check out this reference book (chapter 3 for regular expressions and automata):

Alfred V. Aho, Ravi Sethi & Jeffrey D. Ullman. “Compilers: principles, techniques, & Tools”. Addison-Wesley, 1986.

SIMULATIONS

Contents

- *Simulations*
 - *Monte Carlo Estimation*
 - *Fractals*
 - *Formal systems*
 - * *MU Puzzle*
 - * *Cellular Automata*
 - * *Natural Language Parsing*
 - *Artificial Neural networks*
 - * *Hopfield network*
 - * *Perceptron*
 - * *Backpropagation*
 - * *Using tensorflow and keras to build neural nets*

19.1 Monte Carlo Estimation

- Read about [Monte Carlo estimation of PI](#)
- Write a script that estimates the number 'pi' using this method.
- Compare it to my solution: `simulations/estimate_PI_by_MonteCarlo.py`)

19.2 Fractals

Fractals are figures that are self-similar at several scales.

- Write a script that displays the [Koch snowflake](#)

Hints:

- use the turtle module
- use recursion

My solution: `koch0.py`

19.3 Formal systems

19.3.1 MU Puzzle

A famous seminal book in Cognitive Science is *Gödel Escher Bach: An Eternal Golden* by Douglas Hofstadter. Its main topic is recursion and self-reference (see also *I am strange loop* by the same author).

According to Hofstadter, the formal system that underlies all mental activity transcends the system that supports it. If life can grow out of the formal chemical substrate of the cell, if consciousness can emerge out of a formal system of firing neurons, then so too will computers attain human intelligence. Gödel, Escher, Bach is a wonderful exploration of fascinating ideas at the heart of cognitive science: meaning, reduction, recursion, and much more (from <https://medium.com/@alibedirhan.d/mu-puzzle-f651ef3957c5>)

The book is filled with puzzles, including Hofstadter's famous **MU puzzle**. The MU puzzle involves a simple formal system called **MIU**.

A starting string, **MI**, is given. Four rules for changing the string of characters into a new one are provided (see below). At each step, the current string can be transformed into a new string by the application of one of the four rules. Note that rules are one-way! In case there are several applicable rules, there is nothing that will dictate which rule you should use, it's up to you! Here are the rules:

1. If you possess a string whose last letter is **I**, you can add on a **U** at the end. For example **MIUI** can be rewritten **MIUII**. This rule can be written $xI \rightarrow xIU$ where x represents any string
2. Suppose you have **Mx**. Then you may rewrite it **Mxx**. For example, from **MIU**, you may get **MIUIU** ($x = IU$ therefore; $Mxx = MIUIU$; From **MUM**, you may get **MUMUM**, From **MU**, you may get **MUU**, ...
3. **If III occurs in one of the strings, you may make a new string with U in place of III. For example,**
From **UMIIMU**, you could make **UMUMU**; From **MIIII**, you could make **MIU** (also **MUI**). From **IIMII**, you can't get anywhere using this rule because the three **I**'s have to be consecutive.
4. If **UU** occurs inside one of your strings, you can drop it. From **UUU**, you get **U**. From **MUUUIII**, get **MUIII**.

The **Mu Puzzle** asks whether starting from the string **MI**, there exists a *derivation*, that is a sequence of applications of the rules, that can yield the string **MU**.

Exercise: Write a Python script that explores the set of strings generated by this formal system: study the length of the string produced after 'n' steps (run this process several times and compute an histogram).

Then you may read :

- https://en.wikipedia.org/wiki/MU_puzzle
- Ernest Nagel and James Newman's book [Gödel's Theorem](#) (translated into French : [Le théorème de Gödel](#))

19.3.2 Cellular Automata

Learn about Conway's *Game of Life*. Watch [this](#) and [that](#) videos.

- Implement an *Elementary cellular automaton*. The aim is to reproduce the graphics shown at the bottom on the previous page. you can take inspiration from the excellent *Think Complexity* by Allen B. Downey. My solution is at [cellular-automata/1d-ca.py](#).
- Implement the Game of Life in 2D.
- Going further: If you enjoy Cellular Automata, you can read Stephen Wolfram's *A New Kind of Science*. A more general book about Complexity is Melanie Mitchell's *Complexity: a guided tour*.

19.3.3 Natural Language Parsing

Parsing refers to building the syntactic structure of a sentence from the linear sequence of words that compose it.

- Read Chapter 12 (Constituency Grammars) and 13 (Constituency Parsing) of Dan Jurafsky and James H. Martin's *Speech and Language Processing*
- Explore the various parsing algorithms using the *Natural Language Toolkit*.

19.4 Artificial Neural networks

19.4.1 Hopfield network

Read <https://towardsdatascience.com/hopfield-networks-are-useless-heres-why-you-should-learn-them-f0930ebeadcd> and do not look at the jupyter notebook implementation provided by the author. Try to program a hopefield network and teach it a few patterns. Only then, check the author's solution.

To go further you can read:

- Ramsauer, Hubert, Bernhard Schäfl, Johannes Lehner, Philipp Seidl, Michael Widrich, Thomas Adler, Lukas Gruber, et al. 2020. "Hopfield Networks Is All You Need." ArXiv:2008.02217 [Cs, Stat], December. <http://arxiv.org/abs/2008.02217>.

19.4.2 Perceptron

1. Read about the Perceptron at <https://medium.com/@thomascourtz/perceptrons-in-neural-networks-dc41f3e4c1b9>
2. Implement a Perceptron in Python

For a solution, check : <https://blog.dbrgn.ch/2013/3/26/perceptrons-in-python/>

19.4.3 Backpropagation

To understand the basics of artificial neural networks, I recommend that you first read <https://victorzhou.com/blog/intro-to-neural-networks/> and then watch the four excellent videos at https://www.youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi . The last two of them focus on the backpropagation algorithm that allows one to train network to learn mappings.

Next, you can read and try to understand this [implementation](#) of the backpropagation algorithm.

Then, see a modern and efficient implementation of neural networks: https://pytorch.org/tutorials/beginner/deep_learning_nlp_tutorial.html

More readings:

- [The Unreasonable Effectiveness of Recurrent Neural Networks](#) on Andrej Karpathy's blog.
- [understanding LSTM Networks](#)
- [Pattern recognition and machine learning](#) by Christopher M. Bishop

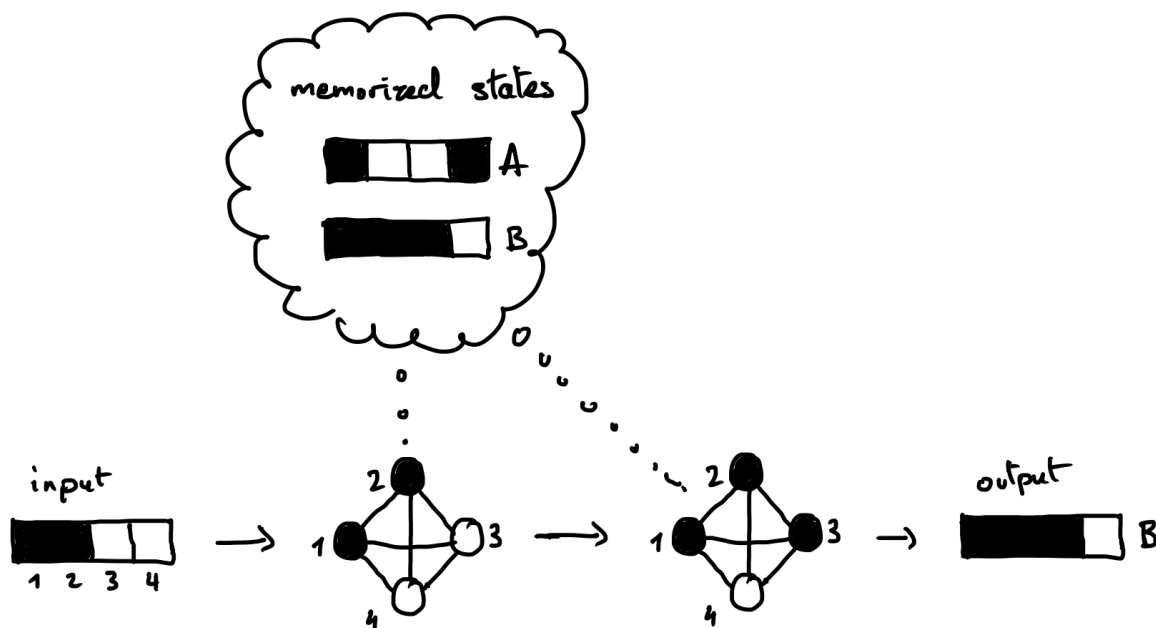
19.4.4 Using tensorflow and keras to build neural nets

The following book is a gentle introduction: <https://www.goodreads.com/book/show/53483757-ai-and-machine-learning-for-coders>

HOPFIELD NETWORKS

Hopfield networks are a type of artificial neural network that can implement an associative memory.

A Hopfield network can be wired to memorize certain states: when given an incomplete or corrupted version of one of the memorized states, it will restore the original state. [Demo](#).



20.1 Model

20.1.1 Structure

- A Hopfield network consists of N neurons, each of which can have two states, $+1$ or -1 . The state of the whole network can be written as a binary vector of size N . We will use a 1-dimensional array to code that state, e.g. `state = np.array([+1, -1, -1, -1])`.
- The weights of the connections between the neurons can be written as a matrix of size $N \times N$. We will use a 2-dimensional array to code that matrix, e.g. `weight = np.array([[0, -1, -2, +1], [-1, 0, +3, -1], [-2, +3, 0, -1], [+1, -1, -1, 0]])`.

- In Hopfield networks, the connections are symmetric and there are no self-recurrent connections. Thus, the weight matrix is symmetric (`weight[i, j] == weight[j, i]`) and its diagonal is zero (`weight[i, i] == 0`).

20.1.2 Updating

- The state of a neuron is updated according to the sign of the input it receives in the following way:
- Here we will use the synchronous update rule, where all of the neurons are updated at the same time from one time step to the next.

20.1.3 Exercise 1.

- Write a function to compute the next state of a Hopfield network given the previous state and the weight matrix.
Tip: You can use a [matrix multiplication](#) (to do it in NumPy, [see here](#)).
- Write a function to compute the final state reached by a network given an initial state and its weight matrix. This will either be a *stable state*, starting from which the next state remains the same, or the state after a max. number of iterations that you will set in case no stable state is reached.
- Test your function with the below initial state and weight matrix. The final state should be `[+1, -1, -1, +1]`. Is it a stable state?

```
initial_state = np.array([+1, -1, -1, -1])
weight = np.array([
    [0, -1, -1, +1],
    [-1, 0, +1, -1],
    [-1, +1, 0, -1],
    [+1, -1, -1, 0]])
```

A solution: [hopfield_1.py](#).

20.2 Computing the weight matrix

20.2.1 To memorize a single state

Suppose you want to memorize the state `[s1, s2, s3, s4]`. To do so, you will set the matrix to:

```
weight = np.array([
    [0, s1*s2, s1*s3, s1*s4],
    [s2*s1, 0, s2*s3, s2*s4],
    [s3*s1, s3*s2, 0, s3*s4],
    [s4*s1, s4*s2, s4*s3, 0]])
```


20.2.2 Exercise 2a.

- Try it using your own example state. Replace the values in the above weight matrix with the values of the state you want to memorize, then test that it is stable state using the function you wrote earlier.
- Also try an initial state that is a corrupted version of the memorized state where one neuron has been flipped. Is the state effectively restored?

20.2.3 To memorize multiple states

Now, suppose that you want to memorize multiple states. To do so, you simply need to sum the weight matrices corresponding to each state according to the previous formula.

20.2.4 Exercise 2b.

- Write a function to compute the weight matrix from a set of states.
- If you have time: Implement this function using the outer product (if you haven't done so already). It gives a neat and efficient way to perform this computation.

Tip: Formally, the individual weight matrix to memorize one state is the *outer product* of this state with itself. The [wikipedia page](#) tells you how to compute this outer product with a matrix multiplication. You can then think of how to extend this operation for multiple states such that it gives you the sum of the outer products of individual states.

- Test your function with two memorized states of size 4. Verify that they are stable states, and that you can recover them starting from a corrupted state with one neuron flipped.

A solution: [hopfield_2.py](#).

20.3 Encoding images

- Let's make all of this more visual and fun! We will encode 5×5 binary images into states. To define these images, we will use strings such as this one:

```
"""
11111
1....
11111
....1
11111
"""
```

20.3.1 Exercise 3.

- Write a function to convert such a string into 2d matrix which represents the image. Test your function and display the image using `matplotlib.pyplot.imshow()`.
- Write a function to convert such a 2d image matrix into a 1d state vector that can be encoded by a Hopfield network, and another function to convert the state into a 2d image. [Here is a tip](#). Test your functions.

A solution: [hopfield_3.py](#).

20.4 Simulations

20.4.1 Exercise 4

- Make a network to memorize the [following images](#).
- For each of the [following initial states](#), compute the final state that the network reaches and plot the 2d images corresponding to the initial state and the final state side-by-side.
- What do you observe? Does the network successfully restore the memorized images? Does it always converge to one of these images? Are there any unexpected behaviors?

A solution: [hopfield_4.py](#).

20.4.2 Exercise 5

- Now make another network to memorize the [following images](#)
- Test the network with those same states as initial states.
- What do you observe?

A solution: [hopfield_5.py](#).

20.5 Discussion

What do you think are the strengths and weaknesses of such networks as a model of the brain?

WEB SCRAPING

(activity developed by [Ewan Dunbar](#))

- *!!Read this first!!*
- *Overview*
- *Web scraping*
 - *General instructions and a partly-worked example exercise*
 - * *Notes on idiomatic Python*
 - * *Notes on Python style conventions*
 - * *Notes on how to do this project*
 - * *Notes on opening files*
 - * *Notes on testing your code*
 - * *Hint*
 - * *Unicode*
 - *Exercise 1: Command line arguments*
 - * *Recommended approach*
 - *Exercise 2: Processing HTML*
 - * *Structure*
 - * *Reminder*
 - * *Documentation strings*
 - * *Docstrings versus inline comments*
 - * *What functions should do*
 - * *Note*
 - * *Scope*
 - *Exercise 3: Accessing web pages*
 - * *BE CAREFUL*
 - * *This is a bigger task*
 - *Exercise 4: Refactoring your code*
 - * *Importing functions*

- * *Scripts and modules*
- * *Testing your code*
- *Exercise 5: Crawling websites*
- *Exercise 6: Cleaning up text*
- *Exercise 7: Transition probabilities*
- *Exercise 8: Generating text from a bigram language model*
- *Resources*

!!Read this first!!

This is an optional project aimed at those who have a fair bit of experience programming and want to learn quickly how to do some useful things in Python, and/or feel like they know how to mess around with code but don't quite feel like "programmers," or would just like to become better programmers. You may not feel like a programmer when you are done either, but, if you follow the instructions you will certainly feel much **more** like a programmer. In that spirit, it is very much a "self-guided tour." You will get instructions on what to do, but you will need to make heavy use of online resources to figure out how to do these things.

This project consists of a series of exercises. It is incremental. You need to start at the beginning, because each part relies on the previous, but you can stop at any point.

This project will not be marked. But, if you take this project on, we would like to:

- See you finish it by the end of the course
- Check in on you each week
- Spend a small amount of time at the end of the course going over your project to give qualitative comments

We will also of course help you by answering any questions you may have by email or in person. For this reason, if you are going to do this project, we ask that you sign up: send an e-mail to both Info 2 instructors (Christophe Pallier and Ewan Dunbar) saying that you're going to do the project (not to the Google Group). Keeping in touch will also help us to fix any potential bugs in the project instructions as quickly as possible.

You are encouraged, but not required, to find a partner, and work on this project in pairs. If you choose to do this, it is up to you to divide the labour. It is recommended that you use the opportunity to read each other's code carefully, check it over, and discuss it.

In either case, again, **if you are going to do this project**, we ask that you sign up: please **send an e-mail to us saying that you're going to do the project** (not to the Google Group). If you are going to work in pairs, one email (cc'ed to both partners), will suffice.

Overview

This project will introduce you to a programming task that will be useful if you do projects with language: harvesting textual data from the web and cleaning it up. The last exercise will also get you thinking about what you can do with textual data.

On the way, you will be required to learn certain key Python skills which are far more generally applicable, including:

- Using the Python documentation
- Reading from a file

- Writing to a file
- Downloading web pages
- Command line arguments
- Common Python idioms
- Imports
- Basic text processing

Thus, perhaps the more pertinent goal of this project is to give a self-guided tour of Python to those who have programmed before in other languages, and/or to give those who have programmed before, either in Python or not, a good sense for some basic good programming practices, including:

- Style and variable naming
- Documentation
- Use of functions
- Organizing your code
- Attacking complex problems by sub-dividing them

The point of the last is to help you build up the skills to create working code. The point of the first four is to prevent you from creating code that is “write-only”: code which works, but which cannot be revisited or modified (or maybe even finished) by you or anyone else, because it cannot be read or understood. The larger goal of these last four is to remind you that computer **source code** is exclusively for human beings, not for computers; compiled computer **machine code** (which you may never touch) is for computers. Source code is a way for you to sort out a human-comprehensible explanation of what the machine code to solve your problem should do, save it, share it with other human beings who would like to understand it too, and use it again in the future for other purposes (it is, incidentally, also the easiest way to generate machine code, but it is not the only way). The instructions for this project will give you some very general guidance on how to do these five things.

The instructions for this project will **not** give you much guidance on how to integrate two other essential programming practices into your work, the use of **unit tests** and the use of the **interactive debugger**, because the above is already a lot, but, if you’d like a bit more practice, you may also find this to be a good exercise in developing these practices. You can find useful information on unit tests and the interactive debugger below under **Resources**.

By the end of the project, your instructions will become less detailed. You will have to make more and more design and style decisions for yourself. To guide you, should try and find as many examples of good Python code as possible. [This page is a good resource](#).

Web scraping

You may often want to download a large set of web pages, or specific content from those web pages, from an entire web site or set of web sites. A program which does this is called a **web crawler**. A web crawler that looks for very specific information and tries to extract it automatically is called a **web scraper**. Web scraping is sometimes used by companies to collect up-to-date information on prices or other quickly-changing information, and may not be well looked upon by the target sites, for various reasons, but mainly that any web crawler has the potential to create a huge amount of traffic and overload the site. However, if done in a respectful way, web crawlers can be very useful, and, indeed, are essential to your daily life: Google relies on a massive web crawler to find new or updated web pages, so that it can then index them so that you can find them when you search.

Your job is to write Python code that automatically downloads a portion of Wikipedia. It is not strictly necessary to crawl Wikipedia, because the entire contents are freely available to download in large compressed files. If you want to work with Wikipedia in the future, it is recommended that you make use of these files instead of crawling. However, Wikipedia permits light crawling of articles, and, as Wikipedia is a very useful collection of natural language texts, it serves as a useful example.

General instructions and a partly-worked example exercise

This is a warmup exercise that is mostly done for you. This section is intended to establish some coding conventions that you should use in this project.

All of your code should run on the command line (rather than from inside a Python notebook or iPython). If you're not familiar with how to run Python scripts programs from the command line, [read this document first](#) (if your path is set up correctly on Windows, which it already should be, then you should not need to type the full path name to the Python interpreter, `python.exe`, contrary to what is suggested in the document).

You should create a folder just for this project.

Go in your browser to [the English Wikipedia page for the stipple-throated antwren](#). Copy and paste all of the text from this page and save it as a text file in your project folder. Let's start by writing a Python program that reads this file and prints it to the screen.

Notes on idiomatic Python

In each programming language, there are conventions for writing programs or for doing certain programming tasks in a particular way that are generally adhered to, or at least very easily understood, by the community, but may not be obvious a priori (programming language “idioms”). Programmers in that language adhere to these conventions mainly because doing so helps make programs in that language easier to read and understand (for you, not just for others), and also sometimes because they may be useful ways to avoid errors.

The idiomatic Python way to write a program that runs on the command line is to structure your file like this:

```
# Imports, definition of functions, ...

if __name__ == "__main__":
    # The code that runs when the program is launched...
```

coming at the **end** of the file. The reason for this convention is explained on [this Stack Overflow question](#).

You should adhere to this convention for the rest of this project.

For example, in this exercise - which doesn't really require you to write any new functions, but which might require you to import the `sys` module, depending on how you solve it - you might get a short script that looks something like this:

```
import sys

if __name__ == "__main__":
    # All your code...
```

You can get more tips on writing idiomatic Python [at this site](#).

Notes on Python style conventions

Similar to idioms (but more to do with low-level things like formatting) is “style.” This includes naming variables and constants, the number of spaces with which you indent, how many spaces you put around parentheses, and so on. When working on large collaborative projects you will almost always be asked to adhere to a set of style guidelines (and if you are working in a pair, you should do that here too). At the very least, you need to be consistent internally. Many text editors have a “Format” function which can apply many of your personal style conventions automatically, and most will do automatic indentation and allow you to set the number of spaces.

In Python, the standard, and highly recommended, style guidelines are called [PEP 8](#), and are accessible [here](#).

The specification for this exercise asks you to read **one** file. The simplest way to do this (and therefore the ideal way, following the [Zen of Python](#)) is to hard-code the filename as a constant (a variable that doesn’t change). In Python, unlike in other languages, constants have no special status - they are variables like any other, and it is up to you not to change them. The style convention for showing that something is a constant (nearly universally adhered to in all programming languages) is to put them in ALL_CAPS_SEPARATED_BY_UNDERSCORES. Thus, your program will look something like this:

```
# Imports if necessary

if __name__ == "__main__":
    INPUT_FILE = "stipple_throated_antwren.txt"
    # Code to read and print the contents of that file...
```

Notes on how to do this project

This project is a self-guided tour. That means that the [Python documentation](#) and [Stack Exchange](#) are your new best friends - well, right after [Google](#).

Here, for example, is the [sub-page from the Python tutorial on modules and imports](#).

Notes on opening files

There are many ways to open a file and you may find various pieces of advice, but there is an idiom. The idiom has changed over the years, so we point out the idiomatic way to do this today, which is using `with`, as described [here](#). This page will give you almost the whole solution to this exercise (but not quite).

Notes on testing your code

You should always verify that your code is correct (i.e., that it gives the right answer on some important cases for which you know the right answer).

In this case, the way to do that is to put the output of your program in a new file and then compare that one with the original file. You can do this by redirecting the output that is printed to the screen into a text file, such as `output.txt` (see [instructions here](#) for Unix-type systems, a subset of which should also work on Windows).

On Unix-type systems (such as Linux and OS X), as well as Windows 10 (if you follow [these instructions](#)), you can then compare the two files byte-by-byte using the `diff` program. If you don’t have access to `diff`, there are many tools online for comparing two files.

The best practice is to have a program that automatically runs tests on your code, so that when you change it, you know that it’s still doing what it used to do correctly. An important tool for doing this is to write

unit tests for each of your functions. As discussed above, developing the habit of unit testing is beyond the scope of this project. However, it is a good idea, and if you wish to start now, you can [start by reading this document about unit testing in Python](#).

Hint

Yes, the output should be exactly the same. If there are extra spaces or blank lines in your output, even at the end of the file, get it so that your script's output matches exactly before moving on.

Unicode

Sooner or later, you will run into error messages that mention Unicode, which have to do with special (non-ASCII) characters. These errors are awful. Fixing problems with Unicode was a major motivation behind Python 3. We, however, are using Python 2. As soon as you start getting these errors, see the **Unicode** section under **Resources** below, and learn how to work them out as quickly as you can.

Exercise 1: Command line arguments

Finish the example exercise if you haven't already, and save it as `exercise_warmup.py`. Make a copy called `exercise_1.py` and modify it so that it reads the input filename as the first and only command-line argument rather than storing it as a constant, and gives an appropriate error if no arguments are given. Continue to save the rest of the exercises as separate scripts (for example, with the next exercise in a new file called `exercise_2.py`).

Recommended approach

After a bit of Googling, you will be able to work out the basics of how command-line arguments work in Python (you'll know you've arrived when you start playing around with something called `sys.argv`). You may also discover modules called `getopt` and `argparse`, which are more general solutions for reading command-line arguments. It is very rarely a good idea to use a general solution when there is a simple one unless you really need it, but `argparse` is so useful that there is no reason not to use it all the time. Learn to use `argparse`. It will serve you well for the rest of the project, and for the rest of your career, to keep a template handy for all your new Python scripts that might look very roughly like this:

```
import sys
import argparse

if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    # ... Set up argparse ...
    args = parser.parse_args(sys.argv[1:])
    # ... Rest of your program starts here ...
```


Exercise 2: Processing HTML

In a web browser, save the raw HTML of the Wikipedia article on the stipple-throated antwren (rather than just copying and pasting the text). Write a function called `extract_wikipedia_contents` that takes HTML source and returns just the text of the article. Your script should take a single command-line argument which is the name of the HTML file, call this function on the contents of the HTML file, and print the text of the article to the screen (to be more precise, to *standard output*). It's up to you to determine what exactly "the text of the article" means, except that it shouldn't contain HTML codes or JavaScript, and it should correspond, basically, to the English text that a human being would read if they went to read the Wikipedia article. It doesn't need to be the exact text that you copied and pasted above (and in fact, it probably shouldn't, because that likely contained Wikipedia navigation links which aren't part of the article text).

Structure

This is where the idiomatic Python script structure starts to become non-trivial. Put your function definition(s) **above** `if __name__ == "__main__":`, not inside it. This will make your script look something like this:

```
# ... Imports ...

def extract_wikipedia_contents():
    # ... Docstring ...
    # ...

if __name__ == "__main__":
    # ... Get command line arguments, read file ...
    article_contents = extract_wikipedia_contents(article_html_source)
    # ... Print article contents ...
```

Reminder

Your function should apply to the contents of the HTML file, not to the filename, and it should return the text, not print it.

Documentation strings

All of your functions must be documented with block comments called documentation strings, or simply **docstrings**. The purpose of a docstring is to explain what the arguments to your function are, what it returns, as well as a very concise, one-phrase summary of what it does, with perhaps a short paragraph elaborating some relevant details. To get you started, here is a partial docstring for the `extract_wikipedia_contents`. (Notice that `html_source` is the name of the argument to the function.)

```
"""Extract the text of a Wikipedia article from HTML

Here, it would be useful to describe a bit more about how you've chosen to
format the text that you're returning, and what exactly you mean by the
"article text."

Args:
```

(continues on next page)

(continued from previous page)

```
html_source (str): HTML source of a Wikipedia article

Returns:
    str: The text of the Wikipedia article
    """
```

A good practice (one which you have to force yourself to do, but which will help you write your programs faster) is to write your docstrings **before** you write your functions. They force you to state exactly what you intend the function to do. If you have that sorted out, writing the function becomes much easier. (For example, now it should be absolutely clear that the input and output to this function are supposed to be **strings**, and not lists or anything else.)

In the **Resources** below, you will find guidelines for writing docstrings in Python. Be consistent in your style and remember that you're communicating with the rest of the world.

Docstrings versus inline comments

Docstrings are not the same as inline comments. Inline comments (comments interspersed in your code) should be used sparingly, and only where necessary, unlike docstrings. Inline comments explain the logic of your code, if it isn't obvious. They should not be used to explain what variable or function names mean, or to explain enigmatic constants or clever ways to do things in only one line. The way to clarify unclear variable names or mysterious constants is not to use unclear variable names or mysterious constants. The way to clarify the clever thing you did is to never do clever things. Source code is a human-comprehensible explanation of how some machine code works. You have the power to make fun puzzles for the reader, but you shouldn't.

Here is a good summary of [appropriate uses of inline comments](#).

What functions should do

Another useful result of writing your docstrings before you write your functions is that you will find out whether or not your function is trying to do too much. A general rule is that if you can't state precisely in a few words what your function does, it's probably trying to do too much.

Note

You may, of course, write as many functions as you want for this exercise, as long as you write `extract_wikipedia_contents`.

Scope

The goal of this exercise is not to learn to parse HTML. That's a pain. Doing it well demands a whole course of its own. Find a Python module that parses the HTML for you, then get the text out. Find the simplest one possible.

Exercise 3: Accessing web pages

Write a modified version of your previous script which accesses Wikipedia online. It will still take a single, text input file, specified on the command line, but that file will now contain a list of URLs to Wikipedia articles, each on one line. The script will download each of these pages, and then print the contents of all of them to standard output, in sequence. No clear separation between articles is necessary, nor do they need to be in any particular order. You should be able to find Python modules that will download the HTML contents of web pages for you and return them as a string, so you should be able to re-use your `extract_wikipedia_contents` function (copy and paste it into your new script).

BE CAREFUL

You are writing a script that accesses web pages. People who build websites make web pages for people, not for scripts. You should not push your luck, or you may be blocked from accessing the website. Wikipedia is fine with you using scripts to access it, as long as you respect some rules:

- Don't go too fast. Put a short delay (one second at least) between requests for pages.
- Make sure that your script reads and respects the `robots.txt` file. Specifically, that file (which is just a text file), specifies what URLs you are allowed to access and what files you are not. Don't access any URLs you're not allowed to.
- Respect [Wikipedia's policy on User-Agents](#).

This is a bigger task

This exercise is more complex. It may take some time, and it should take more than one function to accomplish. You need to read the `robots.txt` file, retrieve the contents of each of the URLs (being sure to filter appropriately to respect `robots.txt`), and then apply `extract_wikipedia_contents`. You need to make quite a few design decisions, and you need to document them in your docstrings. (Where do you apply `extract_wikipedia_contents`? Where do you filter the URLs? Where do you read the `robots.txt`? Where do you pause between requests?) Furthermore, before you do any of that, you need to do some digging to figure out just how to do each of those things. (How do you access web pages? Where is the `robots.txt` file? What happens if there's a problem accessing a web page?)

Exercise 4: Refactoring your code

Your code from Exercise 3 has two logical components, corresponding to Exercise 2 (cleaning up Wikipedia page HTML) and a new part corresponding to Exercise 3 (accessing websites). Up to now, you've been told to write separate scripts, one for each exercise, and you were told to copy and paste your function from Exercise 2 into your Exercise 3 script. In this exercise, you will learn how to share Python code across different files in the same directory, each of which collects together a set of useful, related functions.

Make a copy of your Exercise 2 script called `wikipedia.py`, and a copy of your Exercise 3 script called `web.py`. In your `exercise_4.py` script, you will treat those scripts as modules, and import the functions that you need from them. Your Exercise 4 script should behave exactly like your original Exercise 3 script from the outside (that is, this is an exercise in [refactoring your code](#)). However, you should ensure that all the functions in your new `web.py` script should work for any website, not just Wikipedia. You may simplify your interpretation of complex `robots.txt` files, so long as you err on the side of caution (never do anything that you're not allowed to do).

Importing functions

There are two conventional ways of importing functions (from your own code or from other modules) that are recommended. One is to import the individual functions you need, as follows:

```
from wikipedia import extract_wikipedia_contents
```

Another is to import the entire file/module as a separate namespace.

```
import wikipedia
```

This requires that you then make explicit reference to the file/module that you imported when calling functions from it:

```
article_contents = wikipedia.extract_wikipedia_contents(article_html_source)
```

This can get quite verbose, so there is a way of abbreviating the names of the files/modules you import (or, rather, of changing the names of the associated namespaces):

```
import wikipedia as wp

# ...

article_contents = wp.extract_wikipedia_contents(article_html_source)
```

There is one method that is often cautioned against, because it may fill up your namespace unexpectedly with function or variable names that you do not need and do not want, and that is this:

```
from wikipedia import *
```

In this case, it probably won't do anything different (unless you defined additional functions in `wikipedia.py`). But the above alternatives represent more predictable and understandable code.

Scripts and modules

You have just learned to import functions not from external modules, but from your own code! You may be wondering how this is possible. It is possible because a Python module is simply any Python script that defines functions, variables, or anything else ([see here](#)). Yet the files you've written weren't originally intended for that. They were intended to be used as scripts to be run on the command line. This is not a bug - it is a feature. There's no need to work against it by removing the `if __name__ == "__main__":` section of your newly created `wikipedia.py` and `web.py` files. This section of your code now serves as an example of a standard way to use the functions you've defined.

Testing your code

Your Exercise 4 script should behave exactly like your original Exercise 3 script, and you should ensure that all the functions in your new `web.py` script should work for any website, not just Wikipedia (including respecting the `robots.txt` file, under whatever conservative interpretation you have decided to take). As always, you should test your code and ensure that this is true. Now that you know how to import functions, it is all the easier to start unit testing, i.e., writing your tests as separate functions that you put into a separate testing script.

Exercise 5: Crawling websites

Instead of reading a list of URLs from a file, you will now crawl Wikipedia, starting from one article (or perhaps a set of articles) and following links from those articles to find more articles to download. The article or set of articles should be specified as titles (not as URLs) and you should also specify which language's Wikipedia contains the target article. You should decide whether it makes more sense to start from one article or from more than one, and how to pass the article titles and the language in to your program. Your crawler will download a maximum of 100 articles by following the **first** article link contained in each article. (Your job is **not** to crawl all of Wikipedia - that will take forever and it may get you blocked - and you should not need to build a tree of any kind.)

Instead of writing to standard output, you should now write the text of each article to a separate text file. Each text file should have a filename that uniquely identifies that article in that language (it needn't be the title and language directly, but it could be).

As in Exercise 4, structure your code using modules and imports. You are free to create as many new module files as you feel are appropriate, and add new functions to your old modules. Modules can import code from other modules if necessary, but try and organize them so that they don't, so that they can be used independently of each other, as much as possible. Don't reinvent the wheel. If you can reuse your old code, do so. As always, document and test everything.

Exercise 6: Cleaning up text

Exercise 6 will be intended to work on the basis of the output of Exercise 5. Your new script will take as command line arguments an output directory, and a list of file names (corresponding to individual Wikipedia articles). For each file name, you will save a new version of that file in the specified output directory, which has been cleaned up so as to have:

- One sentence per line
- The words in each sentence separated from each other by exactly one space
- Case (upper-case/lower-case) should be removed (normalize to either upper or lower case)

Choose to define "sentence" and "word" in some convenient (not necessarily careful) way. Decide how to organize your new code, and document and test it.

Exercise 7: Transition probabilities

Write a script that takes a collection of text formatted as in the output of Exercise 6, specified on the command line as a set of corpus files and estimates, over the whole corpus (i.e., all the files, taken together), the **transition probability** between words. That is, the probability of observing word **:presentation: B** right after word **:presentation: A** in the corpus. For example, in the following three sentences -

```
The fox jumped
The dog kicked the fox
A fox jumped
```

- an easy transition probability to estimate is the probability of *dog* given *the*. There are three instances of *the*, and one of them is followed by *dog*, so a reasonable estimate of the transition probability of *dog* following *the* would be 1/3. The word here is "estimate" because your corpus is only a finite sample, and simply counting would lead to some surprising results. For example, the transition probability of *dog* following *a* would come out as zero. Look up and implement **back-off** and **smoothing**, and implement simple versions of these. You will also need to calculate the probability of starting or finishing a sentence with a particular word.

A useful resource for understanding these concepts is [Jurafsky and Martin's textbook](#), and in particular Chapter 4 on n-grams.

Do not rely on NLTK or any external module for this.

Print the transition probabilities to a text file, in a format of your choosing. Decide how to organize your new code, and document and test it.

Exercise 8: Generating text from a bigram language model

Write a script that takes the output of Exercise 7 and generates random text that follows the transition probabilities.

Resources

Idioms and style

- *On the* `if __name__ == "__main__":` convention <<http://stackoverflow.com/questions/419163/what-does-if-name-main-do>>`__
- *On writing idiomatic* Python <<http://python.net/~goodger/projects/pycon/2007/idiomatic/handout.html>>`__
- *PEP 8 style* guidelines <<https://www.python.org/dev/peps/pep-0008/>>`__.
- *The* `with` *idiom for opening* files <<http://stackoverflow.com/questions/11555468/how-should-i-read-a-file-line-by-line-in-python>>`__

Docstrings and comments - [Wikipedia on Python docstrings](#) - [Google Python docstring style](#) - [PEP 257 docstring guidelines](#) - [What inline comments are for](#)

Command line arguments - `argparse` <<https://docs.python.org/3/library/argparse.html>>`__

Useful facts - [Standard input, output, and error](#) - [Namespaces](#) - [Python modules](#)

Examples [Example Python code](#)

Unicode - [Stack Overflow explanation of Unicode print problems](#) - [More on Unicode print problems](#) - [A more technical list of Unicode frustrations in Python](#) - [Reading Unicode](#) - [Python Unicode HOWTO](#)

Unit testing - *Unit testing and the* `unittest` *module* <<http://pymbook.readthedocs.io/en/latest/testing.html>>`__

Python debugger - `pdb`

NLP textbook - [Jurafsky and Martin draft](#)

TOOLS TO DO REPRODUCIBLE SCIENCE

You should strive to make your experiments, simulations and data analyses reproducible, that is, anyone should be able to check what you did, and reproduce it. *You* should be able to understand what you did, even years later.

This means that you should:

- keep track of exactly how you selected your materials (not only the end result)
- keep track of the precise recipes for the data analyses

This is very difficult to achieve! Here are some possible strategies:

1. keep a detailed lab notebook (few people manage to do it properly)
2. write computer scripts that automate the processing pipelines.
3. give up, hope you have not made mistakes, and will not need to check or rerun the experiment.

Although 3 is still the most widespread strategy among scientists, I cannot recommend it!

22.1 Two tools: literate programming and version control

Literate programming mixes code and text to produce a human readable document. It goes way beyond simply documenting one's code with comments. For example:

- Rmarkdown:
 - comparing two treatments with R (source: <https://github.com/chrplr/PCBS/blob/master/data-analysis/comparing-two-means-dependent-groups.Rmd>)
 - see more at https://github.com/chrplr/statistics_with_R
 - <http://chrplr.github.io/PCBS/databases/lexique/interroger-lexique-avec-R.nb.html>
- jupyter notebook:
 - tutorial on Fourier analysis
 - tutorial on fMRI data analysis
 - see more at <https://github.com/jupyter/jupyter/wiki/A-gallery-of-interesting-Jupyter-Notebooks>

Remarks:

1. There are problems with tools such as Excel, E-Prime, ...
 - they are impossible to check thoroughly.
 - the files are saved in binary format (not human readable)
 - the compatibility is not assured between successive versions.

2. In literate programming tools, text is often written in the markup language **Markdown**. You should definitely learn it! (see <https://www.markdownguide.org/getting-started>). It is good enough to produce scientific reports (LaTeX not needed!).

Version control systems (e.g. `git`, `svn`, `mercurial`,...) are useful for two reasons:

1. They allow you keep track of the complete history of a project. While your scripts or documents improve, it keeps *cached copies* of the previous versions. This is much, much better than M using a numbering scheme and multiple copies of the same file or directory (`myfile001.doc`, `myfiles002.doc`...).
2. They facilitate collaborating on projects (documents and programs) with other people.
 - **Bad**: use email's attachments to exchange successive versions of a file.
 - **Good**: collaborate on a project by using a shared `git` repository

To go further;

- “Software Carpentry” : a site dedicated to teaching basic computing skills to researchers. See <https://software-carpentry.org/lessons>
- INRIA's MOOC: “Recherche reproductible: principes méthodologiques pour une science transparente”
- Simon Tatham's “How to Report Bugs Effectively”

22.2 Lesson 101: how to compare files or directories

The most basic task is to compare two files. Your text editor may already have such a function built in — for example, in Emacs, it is accessible through the command `ediff` or the menu *Tools/Compare*).

You can also compare two files on the command-line with the command `diff`:

```
diff file1 file2
```

But I recommend another command, `meld` (which can be downloaded from <http://meldmerge.org>), as it has a more user-friendly output:

```
meld file1 file2
```

On Mac, `meld` can be difficult to install, you can rather use `opendiff` or `filemerge` (See <https://eclecticlight.co/2018/04/14/comparing-files-filemerge-opendiff-and-bbedit/>).

These tools also work on directories. To quickly check if there is any difference between two directories:

```
diff -r -q dir1 dir2
```

or:

```
meld dir1 dir2
```

Note: If you compare text files that contain natural language, I recommend `wdiff` which ignores changes in whitespaces (line breaks, etc.). For latex files, `latexdiff` produces a formatted output that clearly shows the textual differences.

22.3 Introducing git

A version control system keeps track of the history of changes to a project, allows one to explore ideas by maintaining several parallel versions of the code.

In these lectures, we use Git. Although Git is a complex beast, you will only need to know a few commands (`git clone`, `git pull`, `git init`, `git add`, `git status`, `git commit` and `git push`)

22.3.1 Creating a local repository

A *local repository* is simply a folder where you have ran the command `git init`:

```
mkdir project
cd project
git init
Initialized empty Git repository in /home/pallier/cours/Python/version_control/git-test/.
↪git/
```

Often, you will work on a local copy of a *remote repository*, imported either from another directory or from the Internet using `git clone`:

```
git clone https://github.com/chrplr/pyepl_examples
```

Note: If, when you create a repository you already know that you want to share it on the internet, I recommend to first create a repository on <http://github.com> or <http://bitbucket.org>, and then *clone* it on your local hard drive. In this way, the internet location will be automatically added to the list of remote repositories under the name *origin*.

Importantly, with git, you can still do version control locally, and only transfer your changes to the remote repository whenever you want, or never, because git is a *decentralized* version control system and all repositories are equal.

22.3.2 Adding files to the local repository

While working on the project directory, you can tag files to *track* using the `git add` command:

```
echo 'essail' > readme.txt # create a file "readme.txt"; you can also use an editor.
↪like atom
git add readme.txt
```

To check which files are currently being *tracked* (or *staged* in git's terminology), use the command ``git status``:

```
git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file:   readme.txt
#
```

Note that you can add entire directories, for example:

```
git add .
```

It is possible to prevent certain files to be tracked (see <https://help.github.com/articles/ignoring-files>)).

22.3.3 Creating a commit (a.k.a. committing)

Once you are satisfied with the files in your working directory, you can take a *snapshot*, that is make a permanent copy of all the tracked files. This operation is also called *committing* your changes:

```
git commit -m 'my first attempt'
[master (root-commit) a7a3a47] First commit
1 file changed, 1 insertion(+)
create mode 100644 readme.txt
```

This saves a snapshot (or *commit*) of the staged files in the hidden directory `.git` at the root of your project. Unless you delete this directory, this version of your files is saved there forever and will always be accessible.

Note: Before committing, it is always useful to check which files are tracked and which are not, with `git status`.

22.3.4 Modifying the project

Let us now modify the file `readme.txt` in the working directory:

```
echo 'line2' >> readme.txt
```

The command `git status` allows us to check the state of the files in the working directory:

```
git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   readme.txt
#
no changes added to commit (use "git add" and/or "git commit -a")

git add readme.txt
git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   readme.txt
#
```

Let us create a new file, `readme2.txt`:

```
echo 'trial2' >readme2.txt
ls
readme2.txt  readme.txt
```

(continues on next page)

(continued from previous page)

```
git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   readme.txt
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       readme2.txt
```

We now add `readme2.txt` to the repository:

```
git add readme2.txt
git commit
[master a7e25a1] First revision; added readme2.txt
2 files changed, 2 insertions(+), 1 deletion(-)
create mode 100644 readme2.txt
```

Let us consult the history of the project:

```
git log
commit a7e25a158ce52a75c62381420f7dc375de631b1b
Author: Christophe Pallier <christophe@pallier.org>
Date:   Mon Aug 27 10:49:54 2012 +0200

First revision; added readme2.txt

commit a7a3a47edfae9d7c720356b691000a81ded73906
Author: Christophe Pallier <christophe@pallier.org>
Date:   Mon Aug 27 10:47:32 2012 +0200

First commit

git status
# On branch master
nothing to commit (working directory clean)
```

22.3.5 Renaming a file

To rename a tracked file, you should use `git mv` rather than just ``mv``:

```
git mv file.ori file.new
```

22.3.6 Recovering a file deleted by accident

Let us delete readme2.txt “by accident”:

```
rm readme2.txt # oops
ls
readme.txt
git status
# On branch master
# Changes not staged for commit:
#   (use "git add/rm <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       deleted:    readme2.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

To recover it:

```
git checkout -- readme2.txt
ls
readme2.txt  readme.txt
cat readme2.txt
trial2
```

22.3.7 Checking for changes

Let us now modify readme2.txt and then compare the file in the current directory from the ones in the last commit:

```
echo 'line2 of 2' > readme2.txt
git diff
diff --git a/readme2.txt b/readme2.txt
index 33d1e15..e361691 100644
--- a/readme2.txt
+++ b/readme2.txt
@@ -1,1 @@
-trial2
+line2 of 2
git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   readme2.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

You prefer meld, you can use

```
git difftool -t meld
```

Compare the working version of a file with the one in the last commit

```
git diff HEAD
```

22.3.8 Inspecting the history of the project

```
git log
```

For a graphical view of the history of the project:

```
gitk
```

22.3.9 Branches

One interest of git is that it is possible to create several branches to make independent developments and merge them later.

To create a new branch:

```
git checkout -b [new_branch_name]
```

To list all branches:

```
git branch -a
```

To switch to an existing branch:

```
git checkout [branch_name]
```

To compare two branches

```
git difftool -d branch1..branch2
```

To compare a specific file:

```
git difftool branch1..branch2 -- filename
```

To merge a branch to the master branch:

```
git checkout master  
git merge [branch_name]
```

22.4 Working with remotes

To add a remote repository

```
git remote add -f nameforremote path/to/repo_b.git  
git remote update
```

To list the remotes

```
git remote -v
```

To compare the current branch with one in a remote

```
git diff master remotes/b/master
```

To see branches on remotes

```
git branch -r
```

(To see local branches: `git branch -l`, all branches, `git branch -a`)

22.4.1 Downloading the most recent changes from the distant repository

If you imported your repository from the internet with `git clone`, you can import the recent changes with:

```
git fetch
git merge
```

22.4.2 Comparing the local working directory with a remote

If you want to compare the current working directory with the distant remote `origin/master`.

```
git fetch origin master
git diff --summary FETCH_HEAD
git diff --stat FETCH_HEAD
```

22.4.3 Pushing your changes to the distant repository

You can send your modified repository (after committing) to the original remote internet repository:

```
git push
```

22.4.4 Handling very large files (e.g. data)

git-annex allows you to leave large files in some of the repositories and keep only links in others.

See <https://writequit.org/articles/getting-started-with-git-annex.html> and <https://git-annex.branchable.com/walkthrough/>

22.4.5 Resources to learn more about Git

To learn more about git, check out:

- Openclassrooms' MOOC [Manage your code with Git and Github](#)
- <https://product.hubspot.com/blog/git-and-github-tutorial-for-beginners>
- <https://git-scm.com/book/en/v2/Getting-Started-Git-Basics>
- [The Git Book](#)
- [My own git cheat page](#)
- <https://backlogtool.com/git-guide/en/>
- <https://www.atlassian.com/git/tutorials>

To understand the inner workings of git, the following documents are useful:

- [The Git Parable](#)
- [Git from the bottom up](#)

Finally, the comprehensive documentation is the [Git Book](#)

RESOURCES TO LEARN GIT

To understand why you need to learn git, see *Tools to do Reproducible Science*

- Openclassrooms' MOOC [Manage your code with Git and Github](#)
- <https://product.hubspot.com/blog/git-and-github-tutorial-for-beginners>
- <https://git-scm.com/book/en/v2/Getting-Started-Git-Basics>
- The [Git Book](#)
- My own [git cheat page](#)

HOW TO SOLVE PROBLEMS

“An expert is a man who has made all the mistakes which can be made, in a narrow field.” – Niels Bohr

Contents

- *How to solve problems*
 - *How to think like a programmer: lessons in problem solving*
 - * *Understand*
 - * *Plan*
 - * *Divide*
 - * *Stuck?*
 - * *Practice*
 - * *Conclusion*
 - *How to report bugs effectively*
 - *How to solve it by Polya*
 - *Computer Science Distilled*

24.1 How to think like a programmer: lessons in problem solving

The following sections are copied from <https://medium.freecodecamp.org/how-to-think-like-a-programmer-lessons-in-problem-solving-> by Richard Reis.

So, what should you do when you encounter a new problem?

Here are the steps:

24.1.1 Understand

Know exactly what is being asked. Most hard problems are hard because you don't understand them (hence why this is the first step).

How to know when you understand a problem? When you can explain it in plain English.

Do you remember being stuck on a problem, you start explaining it, and you instantly see holes in the logic you didn't see before?

Most programmers know this feeling.

This is why you should write down your problem, doodle a diagram, or tell someone else about it (or thing... some people use a rubber duck).

"If you can't explain something in simple terms, you don't understand it." –Richard Feynman

24.1.2 Plan

Don't dive right into solving without a plan (and somehow hope you can muddle your way through). Plan your solution!

Nothing can help you if you can't write down the exact steps.

In programming, this means don't start hacking straight away. Give your brain time to analyze the problem and process the information.

To get a good plan, answer this question:

"Given input X, what are the steps necessary to return output Y?"

Sidenote: Programmers have a great tool to help them with this... Comments!

24.1.3 Divide

Pay attention. This is the most important step of all.

Do not try to solve one big problem. You will cry.

Instead, break it into sub-problems. These sub-problems are much easier to solve.

Then, solve each sub-problem one by one. Begin with the simplest. Simplest means you know the answer (or are closer to that answer).

After that, simplest means this sub-problem being solved doesn't depend on others being solved.

Once you solved every sub-problem, connect the dots.

Connecting all your "sub-solutions" will give you the solution to the original problem. Congratulations!

This technique is a cornerstone of problem-solving. Remember it (read this step again, if you must).

"If I could teach every beginning programmer one problem-solving skill, it would be the 'reduce the problem technique.'

For example, suppose you're a new programmer and you're asked to write a program that reads ten numbers and figures out which number is the third highest. For a brand-new programmer, that can be a tough assignment, even though it only requires basic programming syntax.

If you're stuck, you should reduce the problem to something simpler. Instead of the third-highest number, what about finding the highest overall? Still too tough? What about finding the largest of just three numbers? Or the larger of two?

Reduce the problem to the point where you know how to solve it and write the solution. Then expand the problem slightly and rewrite the solution to match, and keep going until you are back where you started.”

–V. Anton Spraul

24.1.4 Stuck?

By now, you’re probably sitting there thinking “Hey Richard... That’s cool and all, but what if I’m stuck and can’t even solve a sub-problem??”

First off, take a deep breath. Second, that’s fair.

Don’t worry though, friend. This happens to everyone!

The difference is the best programmers/problem-solvers are more curious about bugs/errors than irritated.

In fact, here are three things to try when facing a whammy:

Debug: Go step by step through your solution trying to find where you went wrong. Programmers call this debugging (in fact, this is all a debugger does).

“The art of debugging is figuring out what you really told your program to do rather than what you thought you told it to do.” –Andrew Singer

Reassess: Take a step back. Look at the problem from another perspective. Is there anything that can be abstracted to a more general approach?

“Sometimes we get so lost in the details of a problem that we overlook general principles that would solve the problem at a more general level. [...]

The classic example of this, of course, is the summation of a long list of consecutive integers, $1 + 2 + 3 + \dots + n$, which a very young Gauss quickly recognized was simply $n(n+1)/2$, thus avoiding the effort of having to do the addition.” –C. Jordan Ball

Sidenote: Another way of reassessing is starting anew. Delete everything and begin again with fresh eyes. I’m serious. You’ll be dumbfounded at how effective this is.

Research: Ahh, good ol’ Google. You read that right. No matter what problem you have, someone has probably solved it. Find that person/ solution. In fact, do this even if you solved the problem! (You can learn a lot from other people’s solutions).

Caveat: Don’t look for a solution to the big problem. Only look for solutions to sub-problems. Why? Because unless you struggle (even a little bit), you won’t learn anything. If you don’t learn anything, you wasted your time.

24.1.5 Practice

Don’t expect to be great after just one week. If you want to be a good problem-solver, solve a lot of problems!

Practice. Practice. Practice. It’ll only be a matter of time before you recognize that “this problem could easily be solved with .”

How to practice? There are options out the wazoo!

Chess puzzles, math problems, Sudoku, Go, Monopoly, video-games, cryptokitties, bla... bla... bla...

In fact, a common pattern amongst successful people is their habit of practicing “micro problem-solving.” For example, Peter Thiel plays chess, and Elon Musk plays video-games.

Fast-forward to today. Elon [Musk], Reid [Hoffman], Mark Zuckerberg and many others say that games have been foundational to their success in building their companies.”–Mary Meeker (2017 internet trends report)

Does this mean you should just play video-games? Not at all.

But what are video-games all about? That's right, problem-solving!

So, what you should do is find an outlet to practice. Something that allows you to solve many micro-problems (ideally, something you enjoy).

For example, I enjoy coding challenges. Every day, I try to solve at least one challenge (usually on Coderbyte).

Like I said, all problems share similar patterns.

24.1.6 Conclusion

That's all folks!

Now, you know better what it means to “think like a programmer.”

You also know that problem-solving is an incredible skill to cultivate (the meta-skill).

As if that wasn't enough, notice how you also know what to do to practice your problem-solving skills!

Phew... Pretty cool right?

Finally, I wish you encounter many problems.

You read that right. At least now you know how to solve them! (also, you'll learn that with every solution, you improve).

“Just when you think you've successfully navigated one obstacle, another emerges. But that's what keeps life interesting.[...]

Life is a process of breaking through these impediments—a series of fortified lines that we must break through.

Each time, you'll learn something.

Each time, you'll develop strength, wisdom, and perspective.

Each time, a little more of the competition falls away. Until all that is left is you: the best version of you.”

– Ryan Holiday (The Obstacle is the Way)

Now, go solve some problems!

And best of luck!

24.2 How to report bugs effectively

A crucial skill is to be able to report problems so that other people can actually help you. Also by formulating a precise question, chances are that you will see the solution by yourself.

On this topic, read [How to Report Bugs Effectively](#)

24.3 *How to solve it* by Polya

A classic book giving advice on how to solve problem: [How to solve it](#) by George Polya.

You can get the gist of it at https://en.wikipedia.org/wiki/How_to_Solve_It

24.4 Computer Science Distilled

A relevant book: [Computer Science Distilled: Learn the Art of Solving Computational Problems](#)

WRITING CLEAN CODE

Contents

- *Writing clean code*
 - *Names*
 - *Functions*
 - *Successive refinement*
 - *General*
 - *Comments*
 - *Testing*

Most of these recommendations are copied verbatim from the book [Clean Code](#) by Robert C. Martin.

25.1 Names

- A function name should describes what the function does. Don't be afraid to make a name long. A long descriptive name is better than a short enigmatic name. A long descriptive name is better than a long descriptive comment.
- Don't be afraid to spend time choosing a name. Indeed, you should try several different names and read the code with each in place.
- Be consistent in your names. Use the same phrases, nouns, and verbs in the function or variables names.

25.2 Functions

- The first rule of functions is that they should be small. The second rule of functions is that they should be smaller than that. Between 1 and 10 lines is usually good.
- A function must do one thing, and do it well. This is because we want each function to be transparently obvious. For example, passing a boolean into a function is a truly terrible practice. It loudly proclaims that this function does more than one thing: It does one thing if the flag is true and another if the flag is false!
- A function should have no side effects. If it does, your function promises to do one thing, but it also does other hidden things. Sometimes it will make unexpected changes to the variables passed as parameters or to system

globals. In either case they are devious and damaging mistruths that often result in strange temporal couplings and order dependencies.

- Functions should either do something or answer something, but not both. Either your function should change the state of an object, or it should return some information about that object. Doing both often leads to confusion.

25.3 Successive refinement

- Writing software is like any other kind of writing. When you write a paper or an article, you get your thoughts down first, then you massage it until it reads well. The first draft might be clumsy and disorganized, so you wordsmith it and restructure it and refine it until it reads the way you want it to read.

- In R. Martin's words:

“When I write functions, they come out long and complicated. They have lots of indenting and nested loops. They have long argument lists. The names are arbitrary, and there is duplicated code. But I also have a suite of unit tests that cover every one of those clumsy lines of code.

So then I massage and refine that code, splitting out functions, changing names, eliminating duplication.

In the end, I wind up with functions that follow the rules I've laid down. I don't write them that way to start. I don't think anyone could.”

25.4 General

- We want the code to read like a top-down narrative. Each function must tell a story. And each function should led you to the next in a compelling order.
- Don't Repeat Yourself. Duplicating the same code at, say four locations in your program, is a problem because it bloats the code and will require four-fold modification should the algorithm ever have to change. It is also a four-fold opportunity for an error of omission. Structured programming and Object-oriented programming were invented in part as strategies to avoid duplication.

25.5 Comments

- Comments are, at best, a necessary evil. If our programming languages were expressive enough, we would not need comments very much—perhaps not at all.
- Comments should say things that the code cannot say for itself.
- A comment worth writing is worth writing well. If you are going to write a comment, take the time to make sure it is the best comment you can write. Choose your words carefully. Use correct grammar and punctuation. Don't ramble. Don't state the obvious. Be brief.
- Learn about [docstrings in Python](#)
- Learn about automatic generation of documentation with sphinx and readthedocs at <https://www.pythonforthelab.com/blog/documenting-with-sphinx-and-readthedocs/>

25.6 Testing

It is an excellent idea to write tests that check your that your modules and function keep working as they should. It can be as simple as having scripts that just run your code on some examples and check that it does not crash after an upgrade, or following a unit testing methodology (see [Getting Started With Testing in Python](#))

BUILDING ABSTRACTIONS WITH RECURSIVE FUNCTIONS AND HIGHER-ORDER FUNCTIONS

In programming, we control the intellectual complexity of our programs by **building abstractions** that hide details when appropriate. You already know of one such means to build abstractions in Python: the ability to **define your own functions**. This allows you to chunk compound operations as conceptual units, give them a name and manipulate them. In this module, you will explore two other means to build abstractions with functions: **recursive functions** and **higher-order functions**.

26.1 Recursive functions

1. Watch the introduction video and read the following summary paragraph.

Summary. Recursive definitions allow us to describe many complex concepts in a simple and elegant way. Likewise, *recursive functions* in Python allow us to define methods for solving problems in a simple and elegant way. When doing so, we think in terms of *base cases*, for which there is an immediate solution, and *recursive cases*, for which we can define the solution to our problem in terms of solutions to smaller versions of the same problem. Once the solution has been defined in such a way, translating it into Python code is a snap. This frees us from without having to worry about a whole series of questions that would arise if we were to solve the problem iteratively (how to traverse sub-problems, how to keep track of intermediate results, what the iteration variable is, what the stop conditions are...). In other words, we have abstracted the solution from the nitty-gritty details of control flow, memory, etc.

2. Read section 16.1 to 16.6 of the interactive book thinkcs.py at: <https://runestone.academy/runestone/books/published/thinkcs.py/IntroRecursion/toctree.html>

Sections 16.5. and 16.6 use the `turtle` graphics module which you may not be familiar with, but you should still be able to grasp the code easily. You can skip them if you want, but visualization is a powerful way to learn about recursion.

3. Do some of the following practice exercises.

You can move to the next section whenever you feel ready to learn more, but make sure to do at least two exercises (like any other programming concepts/techniques, recursive functions cannot be understood without practice).

Time to complete. [*]: short, [**]: medium, [***]: long.

Exercise 1. [*] Write a recursive function to reverse a list. (Hint: to concatenate two lists, use the '+' operator).

Exercise 2. [*] Write a recursive function to reverse a list.

Exercise 3. [**] Write a script that displays the Koch snowflake ([see on wikipedia](#)) using a recursive function. You can use the `turtle` graphics module to draw on screen (as shown in [section 16.5 of thinkcs.py](#)).

A solution: [link to code](#).

Exercise 4. **[**]** Write a script that returns the pathnames of all the files ending in `.txt` contained inside a directory (at any depth of the hierarchy). You will need to use `os.listdir()` and `os.path.isdir()`.

Exercise 5. **[**]** Write a recursive function to generate all permutations of a list of values. (This could be useful, e.g., to do a permutation test for statistical analyses.)

Exercise 6. **[***]** Write a recursive function to evaluate an arithmetic expression given in Polish notation (i.e. prefix notation, [see on wikipedia](#))

Exercise 7. **[***]** Produce strings from the MIU formal grammar and investigate the MU puzzle. See [detailed explanations on the MU Puzzle](#).

Side remark: in some cases, recursive functions can take a prohibitive amount of time and/or memory. This happens when the same computations are performed many times. This issue can be addressed using *memoization*. If you are curious, you can read the article [Memoization in Python](#)

26.2 Higher-order functions

1. Watch the introduction video.

2. Read section 1.6 Higher-Order Functions of the online textbook at: <https://wizardforcel.gitbooks.io/sicp-in-python/content/6.html>

This textbook is an adaptation for Python of the textbook cited at the bottom of the page. (Author's note: I haven't checked out the rest of this adaptation. I recommend to read the original book if you want to learn more.)

Stop and think: In the section of the book you have read, the functions `summation` and `iter_improve` were implemented iteratively, using a `while` statement. Can you think of how to implement them recursively?

3. Do some of the following practice exercises.

Time to complete. **[*]**: short, **[**]**: medium, **[***]**: long.

Exercise 1. **[*]** Write a `product` function to compute the product of the values of a function at points over a given range (the function should be an argument of `product`). Then, define a function to compute the factorial in terms of `product`. Finally, use `product` to compute an approximation to π using the Wallis product formula ([see on wikipedia](#)).

Exercise 2. **[**]** The `summation` function (from the textbook section you read) and the `product` function (defined above) can be defined as special cases of an even more general `reduce` function which has two more arguments: `binary_operator` (which specifies what operation should be applied to reduce all the values to a single one) and `neutral_element` (which specifies what base value should be used when the range is empty). Write such a `reduce` function, and then define `summation` and `product` functions in terms of `reduce`. Make sure to test your functions.

Exercise 3. **[**]** Write a function that finds a fixed point of a function ([see on wikipedia](#)) starting from an initial guess value, which could be defined as: `find_fixed_point(f, initial_value, error_margin)`. The returned value, `x`, should be such that $\text{abs}(f(x) - x) < \text{error_margin}$. Then, define a function to calculate the square root of a positive number, using the property that the square root of x is a fixed point for the function $y \rightarrow 1/2 (y + x/y)$. Make sure to test your square root function.

Exercise 4. **[**]** Write a function to numerically compute any statistic of an arbitrary random variable. The statistic and the random variable should both be functions which are given as argument.

(Hints. A statistic can be defined as a function of a collection of samples, e.g. sample mean, sample variance. A random variable can be defined as a function that, when it is called, generates one sample.)

26.3 Reference

This module was inspired by: Abelson, Harold, and Gerald Jay Sussman. *Structure and interpretation of computer programs*. The MIT Press, 1996.

It is an excellent computer science textbook. If you are curious, go check it out, it is freely available online [as pdf](#) and [as a web document](#).

SENDING TTL TRIGGERS

Many pieces of equipment synchronize by exchanging triggers (0/1 bits) on a [TTL interface](#).

A TTL signal is defined as “low” when it is between 0 V and 0.8 V with respect to the ground terminal, and “high” when it is between 2 V and VCC (5 V).

For example, you may need to send a trigger to an EEG or MEG recording machine, or wait for a MRI scanner signaling that it is starting to acquire brain images.

To send or read TTL signals with your computer, you need some extra hardware (unless you have a Raspberry Pi which has a GPIO port).

27.1 Parallel Port

A parallel port has become a rarity these days. It provides a number of Digital (TTL) Input/Output lines. The `ParallelPort` object in Expyriment <<https://docs.expyriment.org/expyriment.io.ParallelPort.html>>__ makes it extremely easy to access them.

For example, to switch on and off every second the data lines of the first parallel port:

```
from time import sleep
from expyriment.io import ParallelPort

pp = ParallelPort('/dev/parport0')

while True:
    pp.set_data(255)
    sleep(1)
    pp.set_data(0)
    sleep(1)
```

Or to detect any change on any input pin of the parallel ports:

```
from expyriment import io

pports = [io.ParallelPort('/dev/' + pp)
           for pp in io.ParallelPort.get_available_ports()]

prev_state = []
while True:
    state = str([bin(p.poll()) for p in pports])
    if state != prev_state:
```

(continues on next page)

(continued from previous page)

```
print(state)
prev_state = state
```

(under Linux, you may need to run <http://www.vdwalie.com/Norte/ppdiag.txt> to set the adequate permissions)

27.2 DLP-IO8-G

The **DLP-IO8-G** is a USB-TTL device that provides 8 digital lines on which you can read or write. Plugged in USB port, it is seen by the computer as a serial device. Thus, to communicate with it, you simply need to open the relevant serial port and then write to or read from it.



Under Linux, the `dlp-IO8-G` driver is already present in the kernel. Once plugged, to determine the serial port it is attached to, type the command `dmesg` in a Terminal. You should get something like:

```
[ 5128.109725] usbcore: registered new interface driver usbserial_generic
[ 5128.109730] usbserial: USB Serial support registered for generic
[ 5128.112142] usbcore: registered new interface driver ftdi_sio
[ 5128.112148] usbserial: USB Serial support registered for FTDI USB Serial Device
[ 5128.112175] ftdi_sio 1-1:1.0: FTDI USB Serial Device converter detected
[ 5128.112190] usb 1-1: Detected FT232RL
[ 5128.113130] usb 1-1: FTDI USB Serial Device converter now attached to ttyUSB0
```

The last line tells you that the device is at `/dev/ttyUSB0`.

27.3 Python

To use the device under Python, you need the `pyserial` module

27.3.1 sending triggers

```
from serial import Serial

dlp = Serial(port='/dev/ttyUSB0', baudrate=115200) # open serial port

dlp.write(b'QWERTYUI') # set all lines to '0'
dlp.write(b'12345678') # set all lines to '1'

#!/usr/bin/env python3

""" Generate a square wave on pin1 of DLP-I08-G """

from time import perf_counter
from serial import Serial

dlp = Serial(port='/dev/ttyUSB0', baudrate=115200) # open serial port
# byte codes to control line 1:
ON1 = b'1'
OFF1 = b'Q'

# number of periods
NPERIODS = 1000

# Timing of the square wave
TIME_HIGH = 0.010 # 10ms pulse
TIME_LOW = 0.090 # send every 100ms
PERIOD = TIME_HIGH + TIME_LOW

onset_times = [ (PERIOD * i) for i in range(NPERIODS) ]

i = 0
while i < NPERIODS:
    if i == 0:
        t0 = perf_counter()

        # wait until the start of the next period
        while perf_counter() - t0 < onset_times[i]:
            None

        dlp.write(ON1)

        # busy wait for 'TIME_HIGH' seconds. This should be more accurate than time.
        ↪ sleep(TIME_HIGH)
        t1 = perf_counter()
        while perf_counter() - t1 < (TIME_HIGH):
            None
```

(continues on next page)

(continued from previous page)

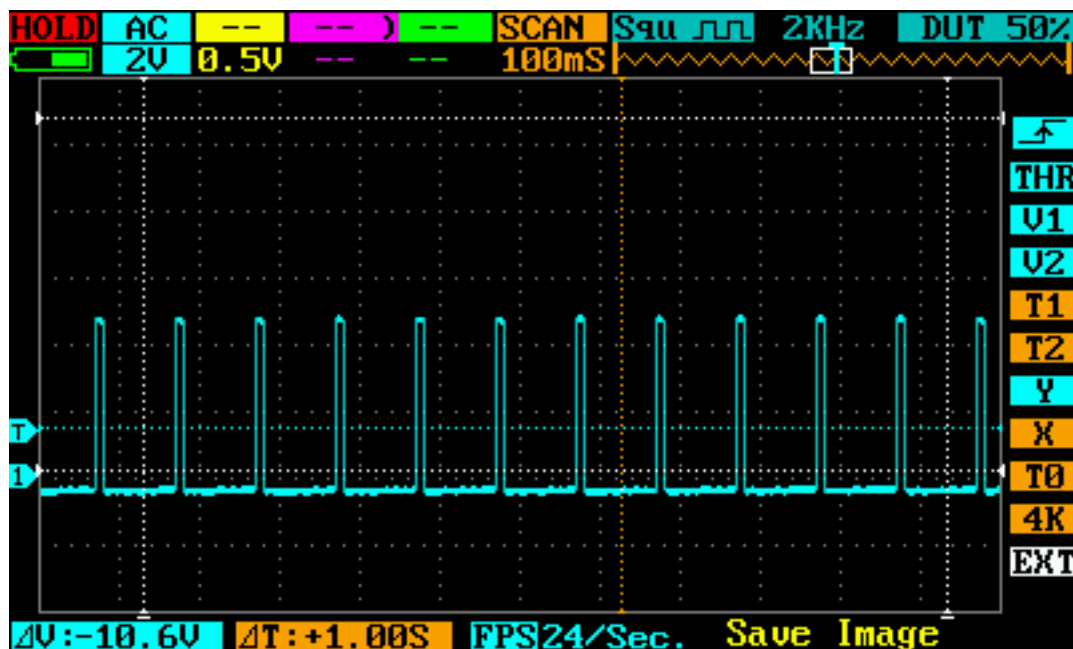
```

dlp.write(OFF1)
i = i + 1
print(f"\r{i:4d}", end='')

time.sleep(TIME_LOW)
print()
print(f'{NPERIODS} periods of {PERIOD} seconds')
print('Total time-elapsed: ' + str(perf_counter() - t0))
dlp.close()      # close the port

```

Here is the result on an oscilloscope:



27.4 Reading an input line

```

import time
import serial
import numpy as np
import matplotlib.pyplot as plt

dlp = serial.Serial(port='/dev/ttyUSB0', baudrate=115200) # open serial port
print(dlp.name)      # check which port was really used
dlp.write(b'`)')     # switch to ascii mode

N = 1000
o = np.zeros(N)      # will store timestamps when the input line is HIGH

```

(continues on next page)

(continued from previous page)

```

i = 0
while i < N:
    dlp.write(b'A') # request to read
    x = dlp.read(3).decode('utf-8')
    if x[0] == '1': # the line is HIGH
        o[i] = time.perf_counter()
        i += 1

plt.hist(np.diff(o) * 1000.0) # plot the deltas between timestamps

```

27.4.1 Latencies and reliability to measure a time interval

TODO

27.5 Arduino

If you do not have a DLP-IO8-G, another approach is to use an [Arduino](#) and program it to send a signal to your PC when it received a trigger. The [Leonardo version](#) is recommended as it can be seen as an HID device and it is trivial to program it to send a key press to your computer upon receiving a trigger. Thus, your stimulation program just has to wait for a keypress and does not even need to open a serial port.

27.6 Raspberry Pi

You can use [gpizero](#) or [RPi.GPIO](#)

The [RPi.GPIO](#) web page warns that “this module is unsuitable for real-time or timing critical applications. This is because you can not predict when Python will be busy garbage collecting. It also runs under the Linux kernel which is not suitable for real time applications - it is multitasking O/S and another process may be given priority over the CPU, causing jitter in your program. If you are after true real-time performance and predictability, buy yourself an Arduino”

This is true, but nevertheless the Raspberry PI may be sufficient for an application that does not overload the PC and just need to read or send some sparse triggers. The only way to know is to check for latencies using an external equipment.

COGMASTER LECTURES

Ateliers de Programmation pour les (Neuro)Sciences Cognitives / Programming for Cognitive and Brain Sciences

This year (2023-2024) the lecture is split in two:

1. PROG101 Introduction to Programming with Python (first semester)

This series of lectures targets students with little or no prior knowledge of programming. It will introduce the fundamentals concepts of coding : variables, expressions, functions, control structures, input/output. The course will make use of <https://pythontutor.com/>

2. PCBS Programming Experiments for Cognitive and Brain Sciences (second semester)

The purpose of this module is to train students to program cognitive psychology experiments. During the first 7 lectures, the course starts with an introduction and is followed by hands-on exercises. In the second part of the semester, Each student chooses an experiment to implement. The project's development should be public on github.

Instructors

- Henri van den Driessche <henri.vandendriessche@ens.fr> (PROG101)
- Christophe Pallier <christophe@pallier.org> (PROG201)
- Maxime Cauté <maxime.caute@ens-rennes.fr> (PROG201)

Moodle

<https://moodle.u-paris.fr/course/view.php?id=7378>

Slides

<http://github.com/chrplr/PCBS/tree/master/slides>

28.1 Course description

28.1.1 Objectives

The purpose of the PCBS course is to make students able to write clean code in order to solve the tasks that are typically encountered in cognitive or neurosciences (data manipulation and analysis, creation of stimuli, programming of real-time experiments, simulations...). The first half (6 weeks) of the course consists of lectures with hands-on exercises, then, during the last 6 weeks, students have to realize a project publicly available on <http://github.com>

28.1.2 Learning outcomes

On successful completion of this course, students should be able to write readable, well- documented, Python programs, and use system such as git that promote reproducible science.

28.1.3 Pedagogy, class organization and homework

The first classes are lectures with hands-on exercises. The remaining classes, I and the teaching assistant are present for individual support to help the students accomplish their project. I also give weekly assignments to be done *before* the next lecture.

28.1.4 Assessment

The projects will be graded on a 20 points scale. The main criterion is *clarity* (see [Projects](#) for more details).

28.1.5 Textbook and readings

All the materials are available on the course's web site at <http://github.com/chrplr/PCBS>.

28.1.6 Course policies

Laptops: Students must bring there own laptop (preferably fully charged!) with the specified software preinstalled (see <https://pcbs.readthedocs.io/en/latest/software-installation.html>)

Participation: You are strongly encouraged to participate in lectures and on the discord discussion forum. The more advanced students are expected to help the beginners.

28.2 Prerequisites

They should acquainted with basic programming concepts: instructions, variables, tests (if..then..else), loops (while and for).

Students are expected to know how to open a terminal and [navigate in the file system](#) and to know how to view and edit text files with a text editor such as [Sublime Text](#).

28.3 Projects

The project should try to replicate an experiment related to cognitive (neuro-) science.

An example of the kind of things that we expect is provided at <http://chrplr.github.io/PCBS-LexicalDecision>

- All documents (scripts, stimuli, documentation, data files) related to the project should be on a github.com repository with a name starts with PCBS- followed by a label that gives an idea of what the project is about.
- **The main page of the repository (README.md) should :**
 - describe the aim of the project (explain the experiment or the simulation
 - explain how to install and run the experiment on one's computer (which command line, which options if any).

- if the project involves analyses, the README.md should point to a documents (html, pdf, not Word!!!) containing the report.
- It is highly recommended to use the ‘Pages’ system of github to generate a nice looking page (see <https://docs.github.com/en/github/working-with-github-pages/configuring-a-publishing-source-for-your-github-pages-site>).
- It should be possible to clone your repository and execute your code without modifying anything. This implies that you should absolutely avoid absolute pathnames. Also your code must be portable and run on MacOS, Windows and Linux.
- Send us a link to your github project as soon as possible so that we can check your progress.
- The readability of the code, and of the main page are of the main criteria of evaluation.
- Do not be overambitious: a well written project that does a simple thing but well will receive a better score than one that has an unreadable code that does complicated things.
- You can work in binomes to read, check and criticize each other code regularly. it is very useful to have someone else check that the documentation and code that you write is readable.
- use the discord channel to ask questions
- **At the end of the *README.md* file, you must include a section detailing:**
 - your previous coding experience
 - what you have learned since then, by following the lecture, coding the project or working by yourself
 - what you missed in this course.

PROJECTS

The project should try to replicate an experiment or some simulations related to cognitive (neuro-) science.

An example of the kind of things that we expect is provided at <http://chrplr.github.io/PCBS-LexicalDecision>

- All documents (scripts, stimuli, documentation, data files) related to the project should be on a github.com repository with a name starts with PCBS- followed by a label that gives an idea of what the project is about.
- **The main page of the repository (README.md) should :**
 - describe the aim of the project (explain the experiment or the simulation
 - explain how to install and run the experiment on one's computer (which command line, which options if any).
 - if the project involves analyses, the README.md should point to a documents (html, pdf, not Word!!!) containing the report.
- It is highly recommended to use the 'Pages' system of github to generate a nice looking page (see <https://docs.github.com/en/github/working-with-github-pages/configuring-a-publishing-source-for-your-github-pages-site>).
- It should be possible to clone your repository and execute your code without modifying anything. This implies that you should absolutely avoid absolute pathnames. Also your code must be portable and run on MacOS, Windows and Linux.
- Send us a link to your github project as soon as possible so that we can check your progress.
- The readability of the code, and of the main page are of the main criteria of evaluation.
- Do not be overambitious: a well written project that does a simple thing but well will receive a better score than one that has an unreadable code that does complicated things.
- You can work in binomes to read, check and criticize each other code regularly. it is very useful to have someone else check that the documentation and code that you write is readable.
- use the slack forum <https://cogmaster-pcbs.slack.com/> to ask questions
- **At the end of the README.md file, you must include a section detaillling:**
 - your previous coding experience
 - what you have learned since then, by following the lecture, coding the project or working by yourself
 - what you missed in this course.

I have uploaded some experimental papers on the Schoology web size (in Materials: *papers-for-projects.zip*) that can be used as the basis of your project.

RESOURCES

30.1 Learning Python3

- Check Python in a Nutshell for a quick tour of Python.
- Books:
 - Automate the boring stuff with Python (highly recommended!)
 - Apprendre à Programmer avec Python3
 - Think Python
- MOOCs:
 - Udemy’s Python programming for absolute beginners
 - Code Academy’s Learn Python module
 - Openclassrooms’ Apprendre à programmer en Python
 - Python 3 : des fondamentaux aux concepts avancés du langage

30.2 Programming skills

- Software Carpentry provides nice lessons about writing software for science and do reproducible science.

30.2.1 Resources to learn the command shell

- The Linux Command Line by Williams Shotts.
- Openclassrooms MOOC

Remarks:

- Under Windows, if you have installed Git for Windows, you have access to `Git bash` which provides a terminal with the bash shell and emulates many Unix commands.
- Under Windows 10, Microsoft has recently made available the “Windows Subsystem for Linux”, which provides a virtual Linux system running inside Windows. (See <https://itsfoss.com/install-bash-on-windows/>, and <https://itsfoss.com/windows-linux-kernel-wsl-2/>).
- Under MacOSX, when you open a terminal, you may be interacting with the bash shell or the zsh shell (to know which, type `echo $SHELL`).

30.2.2 Resources to learn Git

To understand why you need to learn git, see *Tools to do Reproducible Science*

- Openclassrooms' MOOC Manage your code with Git and Github
- <https://product.hubspot.com/blog/git-and-github-tutorial-for-beginners>
- <https://git-scm.com/book/en/v2/Getting-Started-Git-Basics>
- The Git Book
- My own git cheat page

30.3 Books relevant to Cognitive and Brain Sciences Programming

- Programming Visual Illusions for Everyone by Marco Bertamini:
- *Neural Data Science: A Primer with MATLAB and Python* by von Erik Lee Nylen and Pascal Wallisch
- *Matlab for Brain and Cognitive Scientists* and *Analyzing neural time series data* by Mike X Cohen
- Python in Neuroscience
- *Modeling Psychophysical Data in R* by Kenneth Knoblauch & Laurence T. Maloney

30.4 Stimulus/Experiment generation modules

- <http://www.expyriment.org> (See Get started with Expyriment)
- <http://psychopy.org> (See Programming with PsychoPy)
- <http://psychtoolbox.org/> (See Psychtoolbox demos)
- <https://www.jspsych.org/> (See intro at <https://blog.s-m.ac/using-jspsych/>)
- <https://dialoguetoolkit.github.io/chattool/>

30.5 Data analyses, Statistics in Python

- Modules: numpy, scipy, pandas, seaborn, statsmodel, sklearn
 - Data manipulation:
 - * <http://pandas.pydata.org/pandas-docs/stable/tutorials.html>
 - Plotting:
 - * http://matplotlib.org/users/pyplot_tutorial.html
 - * <https://seaborn.pydata.org/tutorial.html>
- *Scipy Lecture Notes*: <http://www.scipy-lectures.org/>
- *Think Stats* by Allen B. Downey: <http://greenteapress.com/thinkstats2/>
- *Python Data Science Handbook* by Jake VanderPlas: <https://jakevdp.github.io/PythonDataScienceHandbook>
- *Introduction to Data Science in Python*: notebook from a 2 day workshop organized by the Paris-Saclay Center for Data Science: <https://github.com/paris-saclay-cds/data-science-workshop-2019>

- *Machine Learning with scikit-learn* MOOC: <https://www.fun-mooc.fr/en/courses/machine-learning-python-scikit-learn/> (on github: <https://inria.github.io/scikit-learn-mooc/>)

30.6 Simulations

- Think Complexity by Allen B. Downey
- The Brian spiking neural network simulator
- Deep Learning for Natural Language Processing with Pytorch