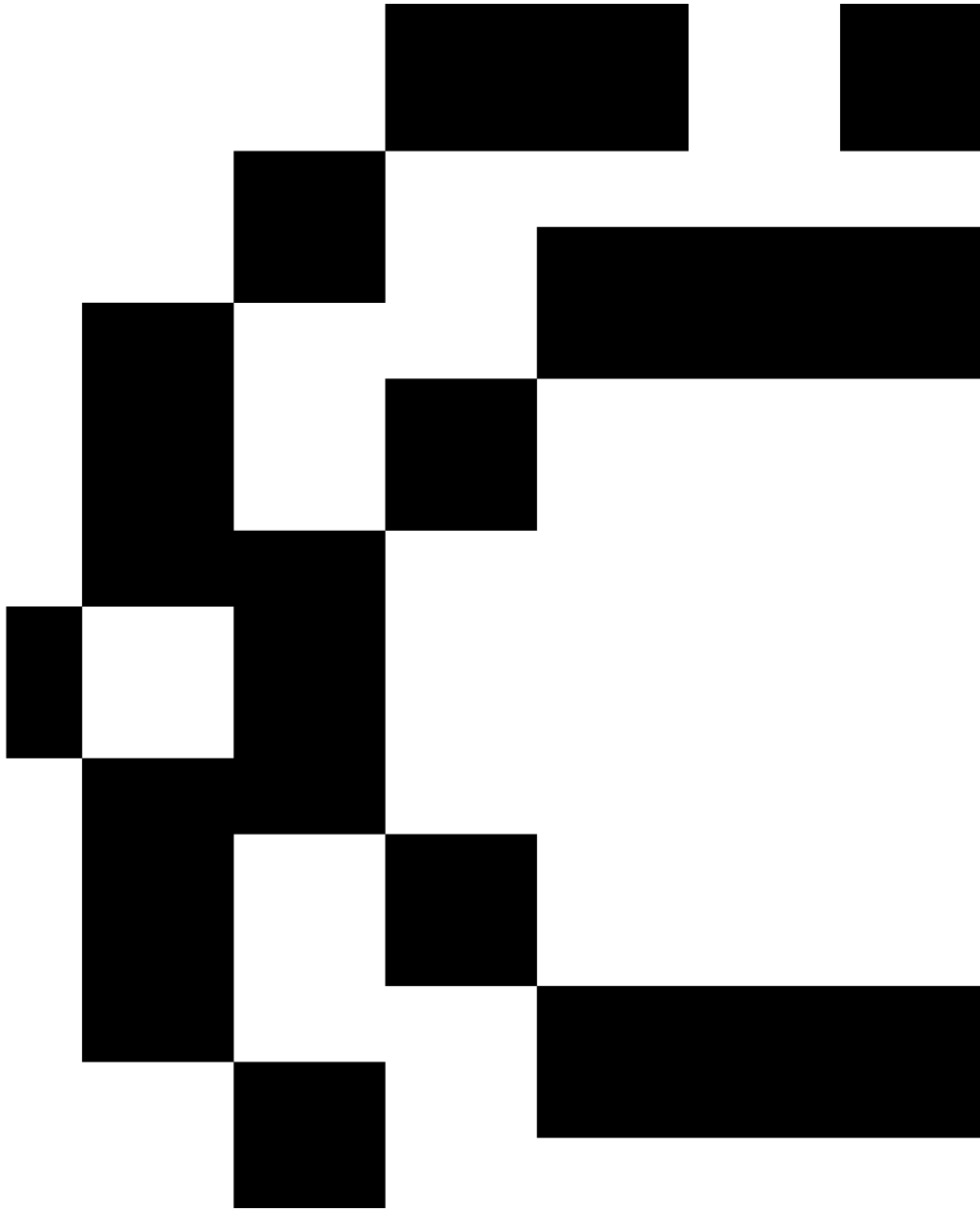
NOVA IMS INFORMATION MANAGEMENT SCHOOL

# Programming for Data Science

Lecture 1

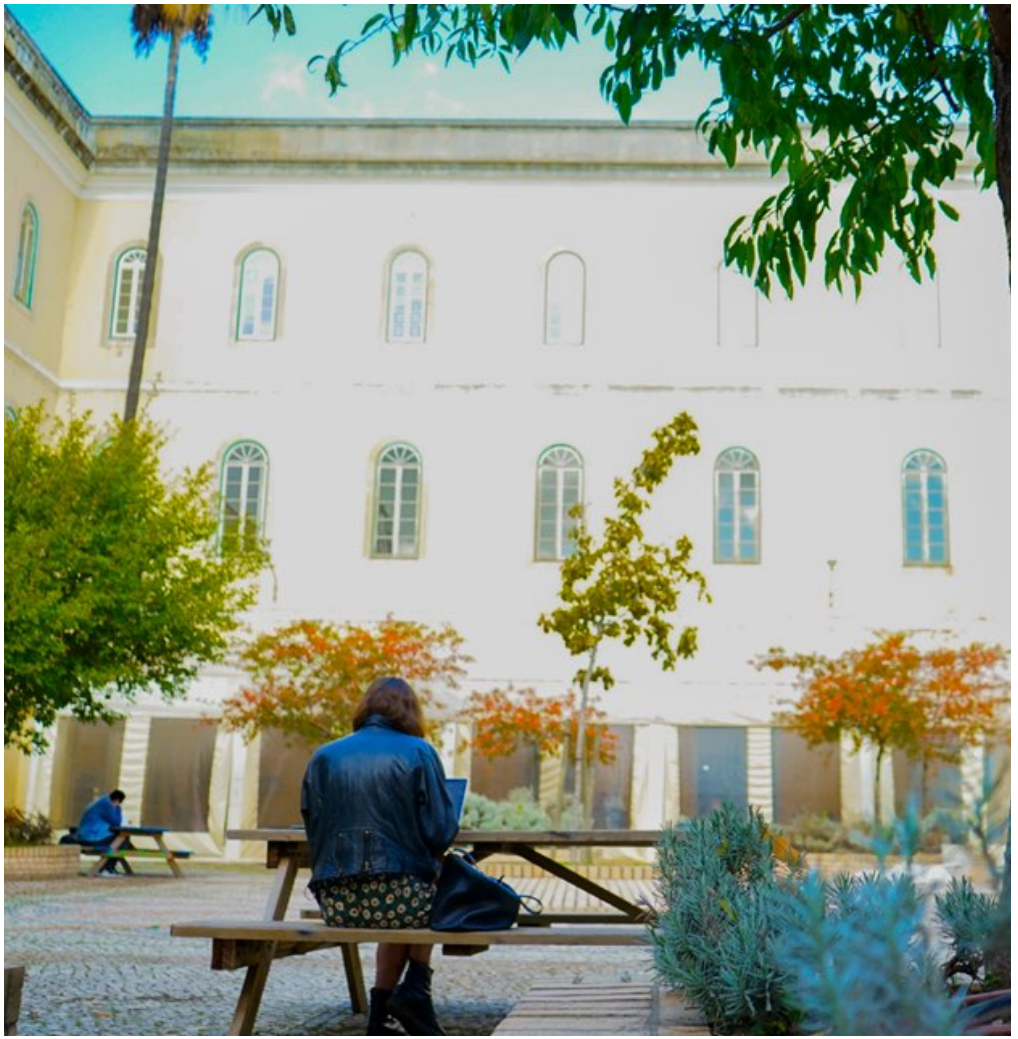
**Flávio L. Pinheiro**

fpinheiro@novaims.unl.pt | www.flaviolpp.com
www.linkedin.com/in/flaviolpp | X @flavio_lpp



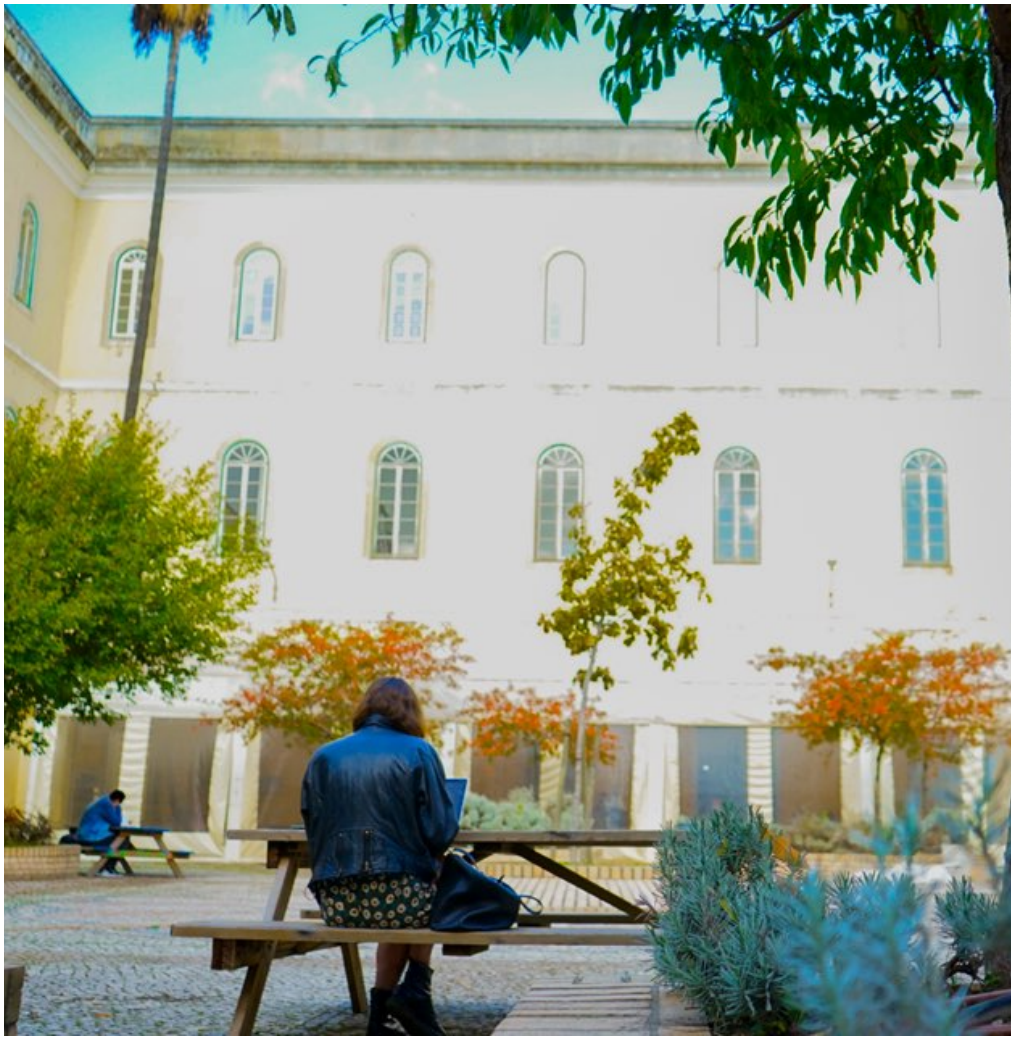
# Lecture 1

- Introduction and UC overview
- Why Python?
- of Computers and Computer Programs
- Ipython and Shell



# Lecture 1

- **Introduction and UC overview**
- Why Python?
- of Computers and Computer Programs
- Ipython and Shell




WELCOME!

# Teaching Staff

Flávio Pinheiro
fpinheiro@novaims.unl.pt
Office 141

Maria Almeida
malmeida@novaims.unl.pt

Liah Rosenfeld
lrosenfeld@novaims.unl.pt

Niclas Sturm
nsturm@novaims.unl.pt

# Schedule & Communication

| Hours | | Tuesday | Tuesday | Tuesday | Wednesday |
|---|---|---|---|---|---|
| 08:00 | 08:30 | | | | |
| 08:30 | 09:00 | | | | |
| 09:00 | 09:30 | | | | |
| 09:30 | 10:00 | | | | |
| 10:00 | 10:30 | TP 1<br>Auditorium Annex 1 | | | |
| 10:30 | 11:00 | | | | |
| 11:00 | 11:30 | | | | |
| 11:30 | 12:00 | P2 \| Maria Almeida<br>Room 7 | TP 2<br>Auditorium A14 | | P7 \| Niclas Sturm<br>Room 7 |
| 12:00 | 12:30 | | | | |
| 12:30 | 13:00 | | | | |
| 13:00 | 13:30 | | | | |
| 13:30 | 14:00 | P4 \| Maria Almeida<br>Room 8 | P3 \| Liah Rosenfeld<br>Room 7 | P1 \| Niclas Sturm<br>Room 1 | |
| 14:00 | 14:30 | | | | |
| 14:30 | 15:00 | | | | |
| 15:00 | 15:30 | P6 \| Maria Almeida<br>Room 8 | P5 \| Liah Rosenfeld<br>Room 7 | | |
| 15:30 | 16:00 | | | | |
| 16:00 | 16:30 | | | | |
| 16:30 | 17:00 | | | | |
| 17:00 | 17:30 | | | | |
| 17:30 | 18:00 | | | | |
| 18:00 | 18:30 | | | | |

**Flávio**

Maria Almeida

Liah Rosenfeld

Niclas Sturm

Office Hours - schedule on demand by e-mail.

Use the Moodle Discussion Forum for questions related with the coursework, activities, or materials.

E-amil only for strictly private issues.

Do not switch Lab classes without prior consent from the Academic Services and the TA.

# Grading First Season

| Quizzes (20%) | Homework Assignments (30%) | Final Exam (50%) |
| --- | --- | --- |

# Grading First Season

**Quizzes**
(20%)

At the start of each Lecture (6 sessions).
Duration of up to 10 minutes.
The number of questions might vary from 5 to 10.
Covers material from previous Lecture/Week.
Only the 5 best scored count towards the final grade

# Grading First Season

**Homework Assignments** (30%)

Two homework assignments.
Duration of 48 hours.
Released and Delivery done through Moodle
"Pairs of Students" that cannot repeat
You will identify your pair at the moment of delivery
More details when assignments are released

# Grading First Season

**Second Season**
MC Exam
(50%)

– 40 multiple choice questions;
– Correct answers are worth 0.5 points, incorrect answers discount 0.2 points;
– Duration of 45 minutes;
– Cover Theoretical and Practical classes;

# Grading Second Season

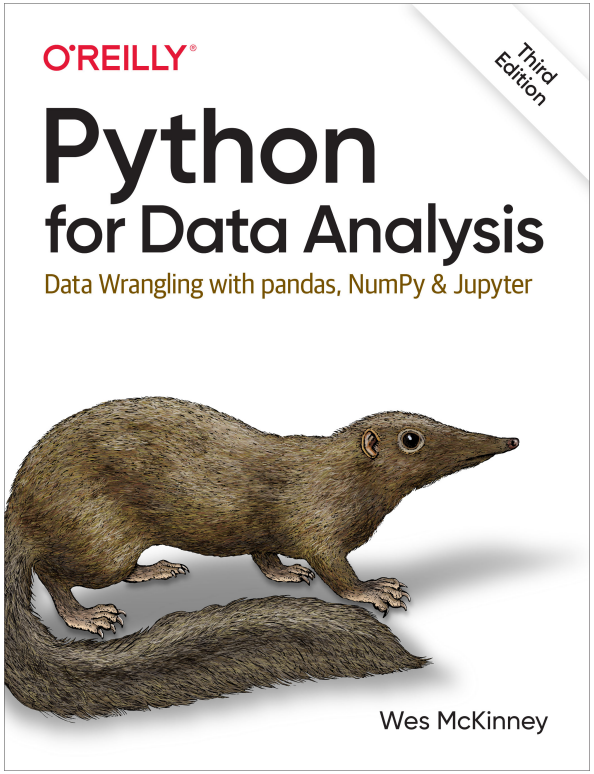
Exam Only!
(100%)

– 40 multiple choice questions;
– Correct answers are worth 0.5 points, incorrect answers discount 0.2 points;
– Duration of 45 minutes;
– Cover Theoretical and Practical classes;

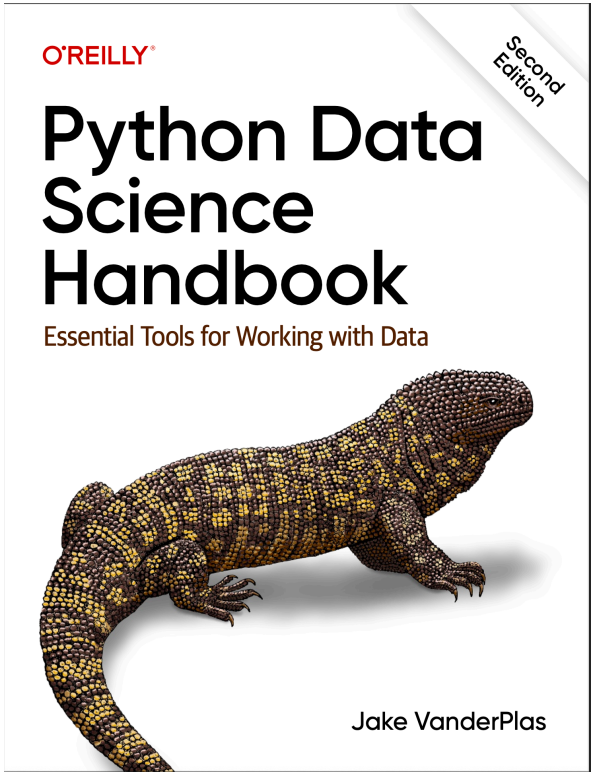
<u>Conditions to attend the 2nd season exam:</u>

– Second exam is optional if you pass the first season.
– If you failed the first season you have to enrol in the second grading season;
– If you pass the first season, you can enrol to improve your grade.

# Suggested Bibliography




https://wesmckinney.com/book/




link on Moodle

# Many Sources are Available

w3schools.com

codecademy

JournalDev

Data Flair

stackoverflow

datacamp

coursera

## Python 3.7.0 documentation

Welcome! This is the documentation for Python 3.7.0.

**Parts of the documentation:**

What's new in Python 3.7?
*or all "What's new" documents since 2.0*

Tutorial
*start here*

Library Reference
*keep this under your pillow*

Language Reference
*describes syntax and language elements*

Python Setup and Usage
*how to use Python on different platforms*

Python HOWTOs
*in-depth documents on specific topics*

Installing Python Modules
*installing from the Python Package Index & other sources*

Distributing Python Modules
*publishing modules for installation by others*

Extending and Embedding
*tutorial for C/C++ programmers*

Python/C API
*reference for C/C++ programmers*

FAQs
*frequently asked questions (with answers!)*

**Indices and tables:**

Global Module Index
*quick access to all modules*

General Index
*all functions, classes, terms*

Glossary
*the most important terms explained*

Search page
*search this documentation*

Complete Table of Contents
*lists all sections and subsections*

**Meta information:**

Reporting bugs

About the documentation

History and License of Python

Copyright

# Disclaimer

The PDS curricular unit is aimed at data analysis enthusiasts with no prior experience in programming.

If you already have experience in programming,
consider helping your colleagues in their learning path. You never know when you might need their help back!

# Final Word

PDS is in broad terms an easy and accessible curricular unit.
However, if you are learning programming/python for the first time
it will require dedication and hard work. Programming is not difficult
but as any other skill it requires a minimum number of hours committed to it
to develop mastery. That's what we are aiming for here! Accumulate hours
of programming experience, to save ourselves many more hours in the future.
Hope you enjoy, and remember we are here to help you!

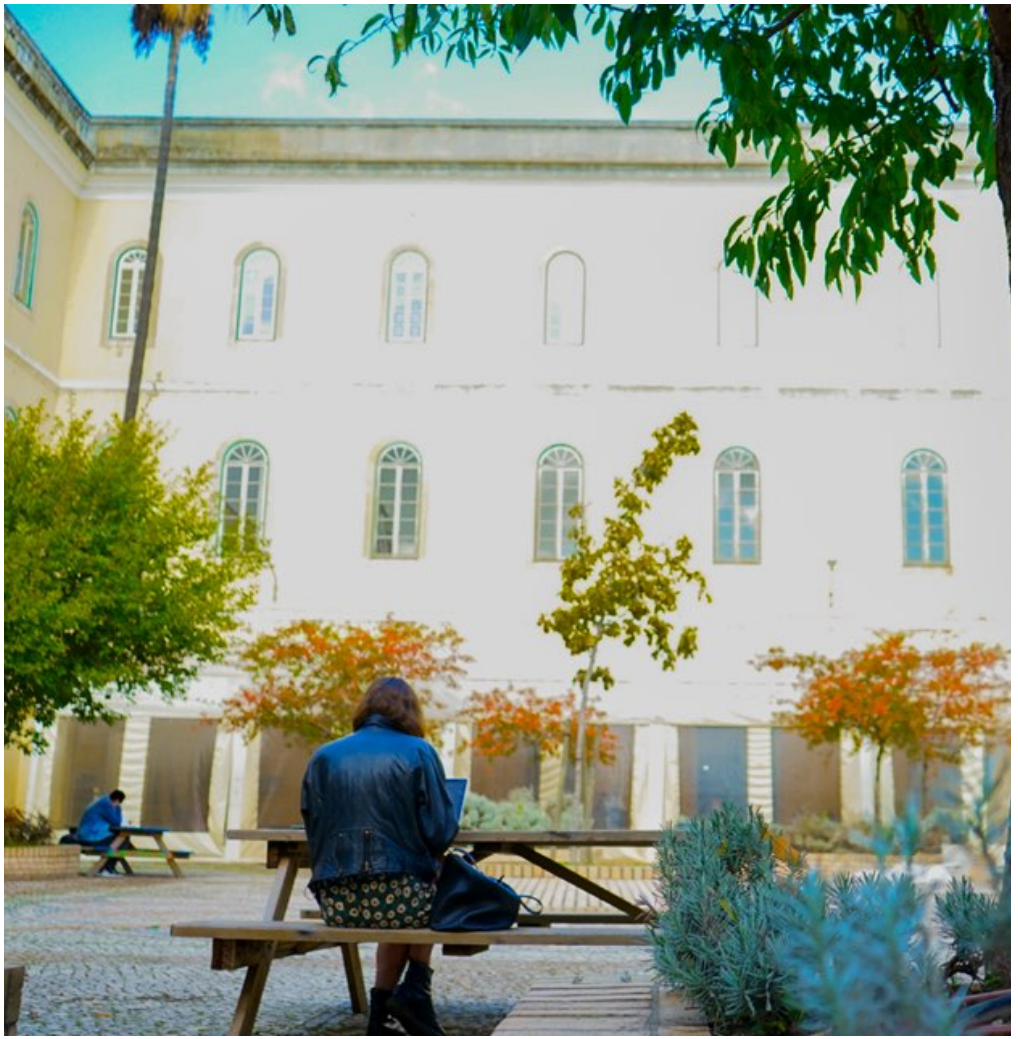

# I don’t care

The goal of this UC is to provide you with a space to learn; if GPT makes it easier for you, then Great. But you better prepare for the final exam.

# Questions?

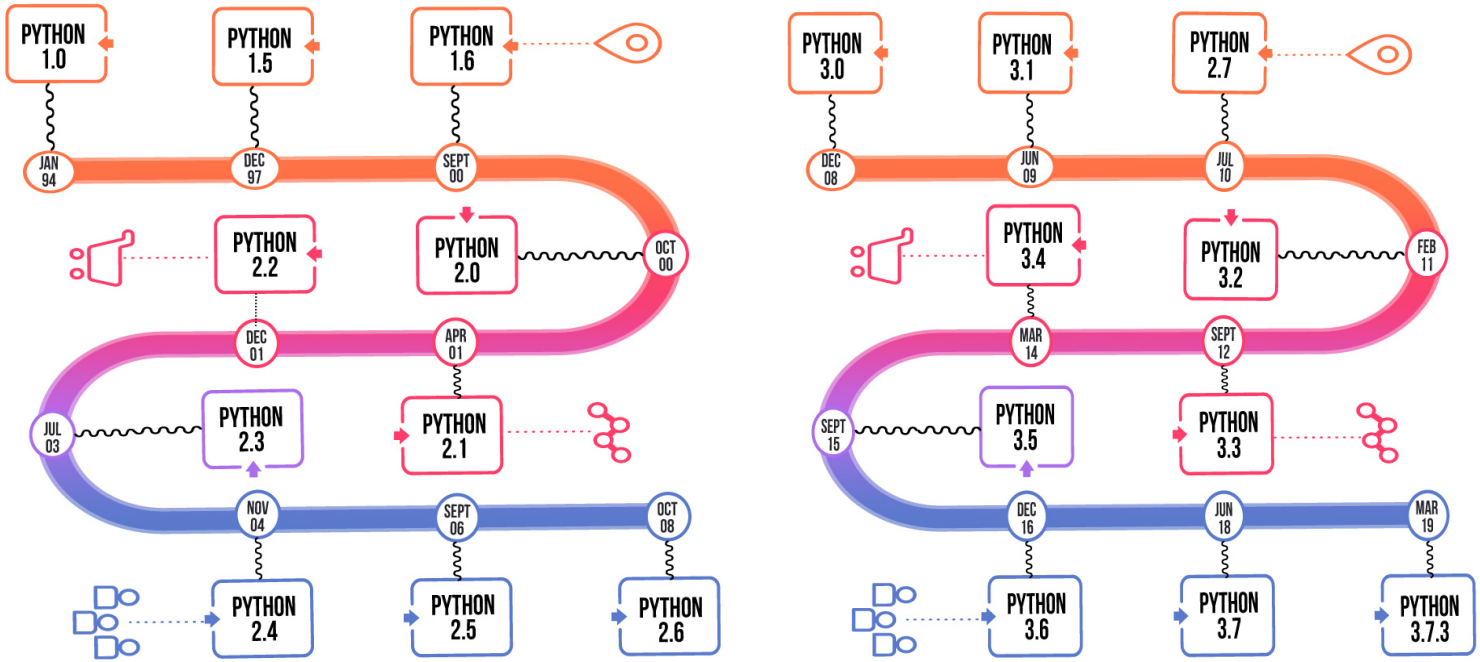
# Lecture 1

- ~~Introduction and UC overview~~
- **Why Python?**
- of Computers and Computer Programs
- Ipython and Shell



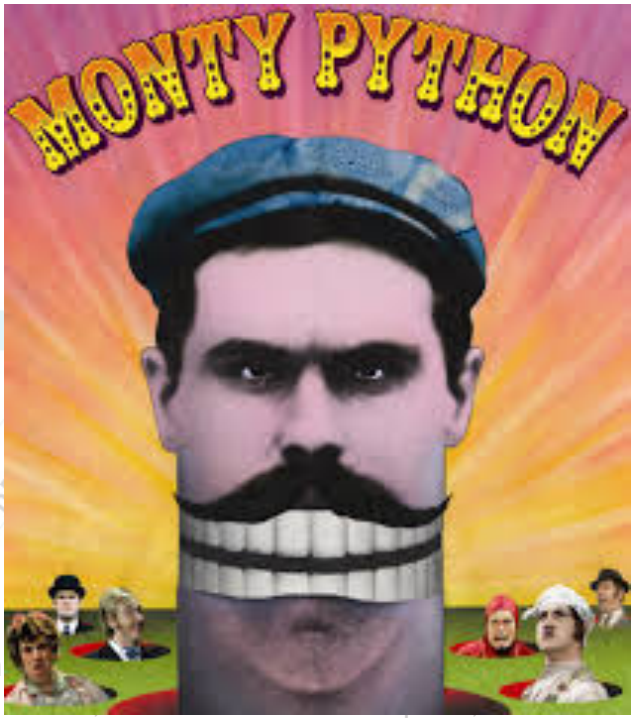
# A Brief History of Python

Guido van Rossum




...In December 1989, I was looking for a "hobby" programming project that would keep me occupied during the week around Christmas. My office ... would be closed, but I had a home computer, and not much else on my hands. I decided to write an interpreter for the new scripting language I had been thinking about lately: a descendant of ABC that would appeal to Unix/C hackers. I chose Python as a working title for the project, being in a slightly irreverent mood (and a big fan of *Monty Python's Flying Circus*).

# A Brief History of Python




Guido van Rossum

much else on my hands. I decided to write an interpreter for the new scripting language I had been thinking about lately: a descendant of ABC that would appeal to Unix/C hackers. **I chose Python as a working title for the project, being in a slightly irreverent mood (and a big fan of *Monty Python's Flying Circus*).**

# Why Python?

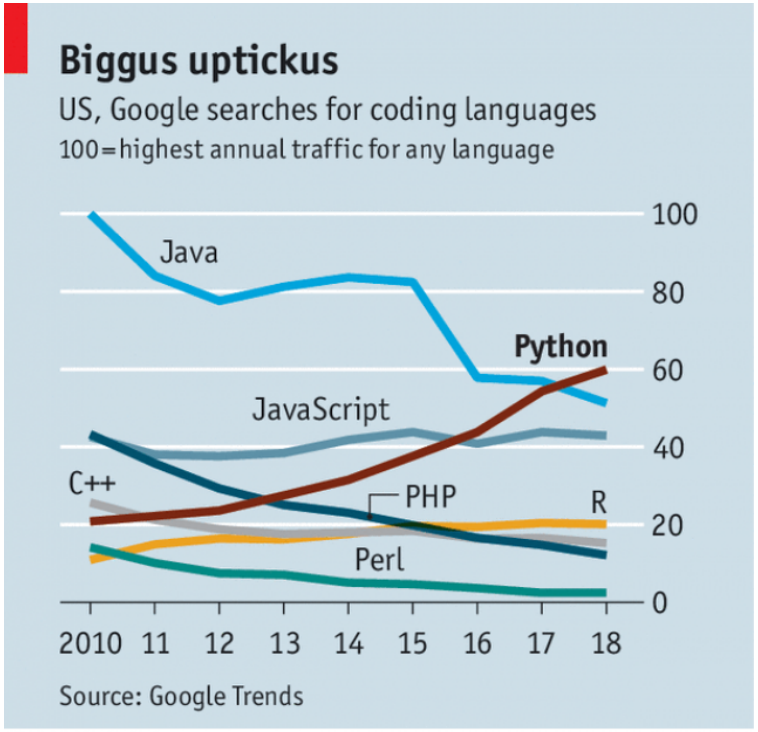
| Total | | | | | |
|---|---|---|---|---|---|
| | Energy | | Time | | Mb |
| (c) C | 1.00 | (c) C | 1.00 | (c) Pascal | 1.00 |
| (c) Rust | 1.03 | (c) Rust | 1.04 | (c) Go | 1.05 |
| (c) C++ | 1.34 | (c) C++ | 1.56 | (c) C | 1.17 |
| (c) Ada | 1.70 | (c) Ada | 1.85 | (c) Fortran | 1.24 |
| (v) Java | 1.98 | (v) Java | 1.89 | (c) C++ | 1.34 |
| (c) Pascal | 2.14 | (c) Chapel | 2.14 | (c) Ada | 1.47 |
| (c) Chapel | 2.18 | (c) Go | 2.83 | (c) Rust | 1.54 |
| (v) Lisp | 2.27 | (c) Pascal | 3.02 | (v) Lisp | 1.92 |
| (c) Ocaml | 2.40 | (c) Ocaml | 3.09 | (c) Haskell | 2.45 |
| (c) Fortran | 2.52 | (v) C# | 3.14 | (i) PHP | 2.57 |
| (c) Swift | 2.79 | (v) Lisp | 3.40 | (c) Swift | 2.71 |
| (c) Haskell | 3.10 | (c) Haskell | 3.55 | (i) Python | 2.80 |
| (v) C# | 3.14 | (c) Swift | 4.20 | (c) Ocaml | 2.82 |
| (c) Go | 3.23 | (c) Fortran | 4.20 | (v) C# | 2.85 |
| (i) Dart | 3.83 | (v) F# | 6.30 | (i) Hack | 3.34 |
| (v) F# | 4.13 | (i) JavaScript | 6.52 | (v) Racket | 3.52 |
| (i) JavaScript | 4.45 | (i) Dart | 6.67 | (i) Ruby | 3.97 |
| (v) Racket | 7.91 | (v) Racket | 11.27 | (c) Chapel | 4.00 |
| (i) TypeScript | 21.50 | (i) Hack | 26.99 | (v) F# | 4.25 |
| (i) Hack | 24.02 | (i) PHP | 27.64 | (i) JavaScript | 4.59 |
| (i) PHP | 29.30 | (v) Erlang | 36.71 | (i) TypeScript | 4.69 |
| (v) Erlang | 42.23 | (i) Jruby | 43.44 | (v) Java | 6.01 |
| (i) Lua | 45.98 | (i) TypeScript | 46.20 | (i) Perl | 6.62 |
| (i) Jruby | 46.54 | (i) Ruby | 59.34 | (i) Lua | 6.72 |
| (i) Ruby | 69.91 | (i) Perl | 65.79 | (v) Erlang | 7.20 |
| (i) Python | 75.88 | (i) Python | 71.90 | (i) Dart | 8.64 |
| (i) Perl | 79.58 | (i) Lua | 82.91 | (i) Jruby | 19.84 |

**Python is one of the least Energy and Time efficient languages. What a piece of crap!**

Normalized global results for Energy, Time, and Memory

https://haslab.github.io/SAFER/scp21.pdf

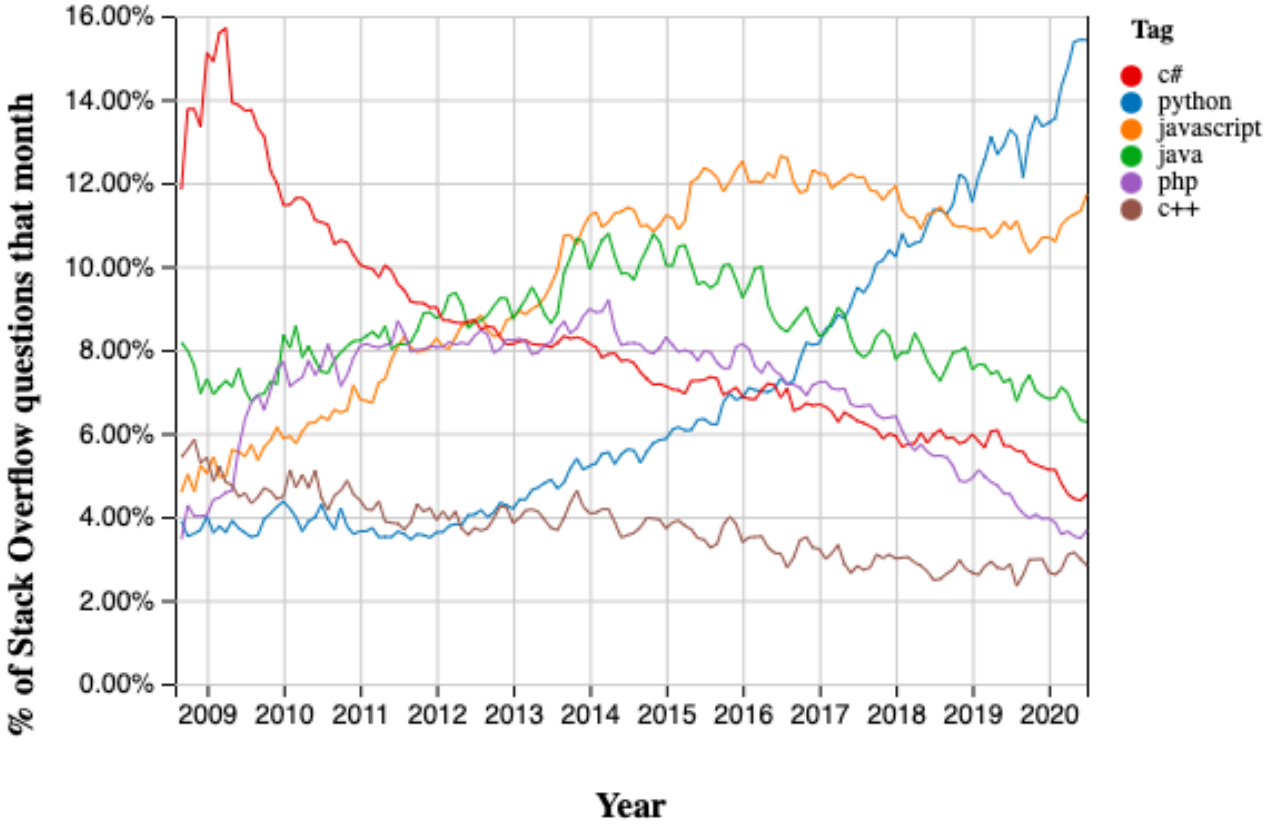
# Why Python?




"Python is not perfect. Other languages have more processing efficiency and specialized capabilities. C and C++ are "lower-level" options which give the user more control over what is happening within a computer's processor. Java is popular for building large, complex applications. JavaScript is the language of choice for applications accessed via a web browser. Countless others have evolved for various purposes. But Python's killer features—**simple syntax that makes its code easy to learn and share**, and its **huge array of third-party packages**—make it a good general-purpose language. Its versatility is shown by its range of users and uses. The Central Intelligence Agency has employed it for hacking, Pixar for producing films, Google for crawling web pages and Spotify for recommending songs."

https://www.economist.com/graphic-detail/2018/07/26/python-is-becoming-the-worlds-most-popular-coding-language

# Why Python?




**Popularity of Languages in Stack Overflow discussion forums**

Source: Stack Overflow

# Why Python?

python™

for data scientists

[Scipy, Numpy, Pandas, Matplotbib, Scrapy, StatsModels, NetworkX …]

# Disclaimer i my toolset

# Disclaimer ii there is life beyond python

# Classroom Stack

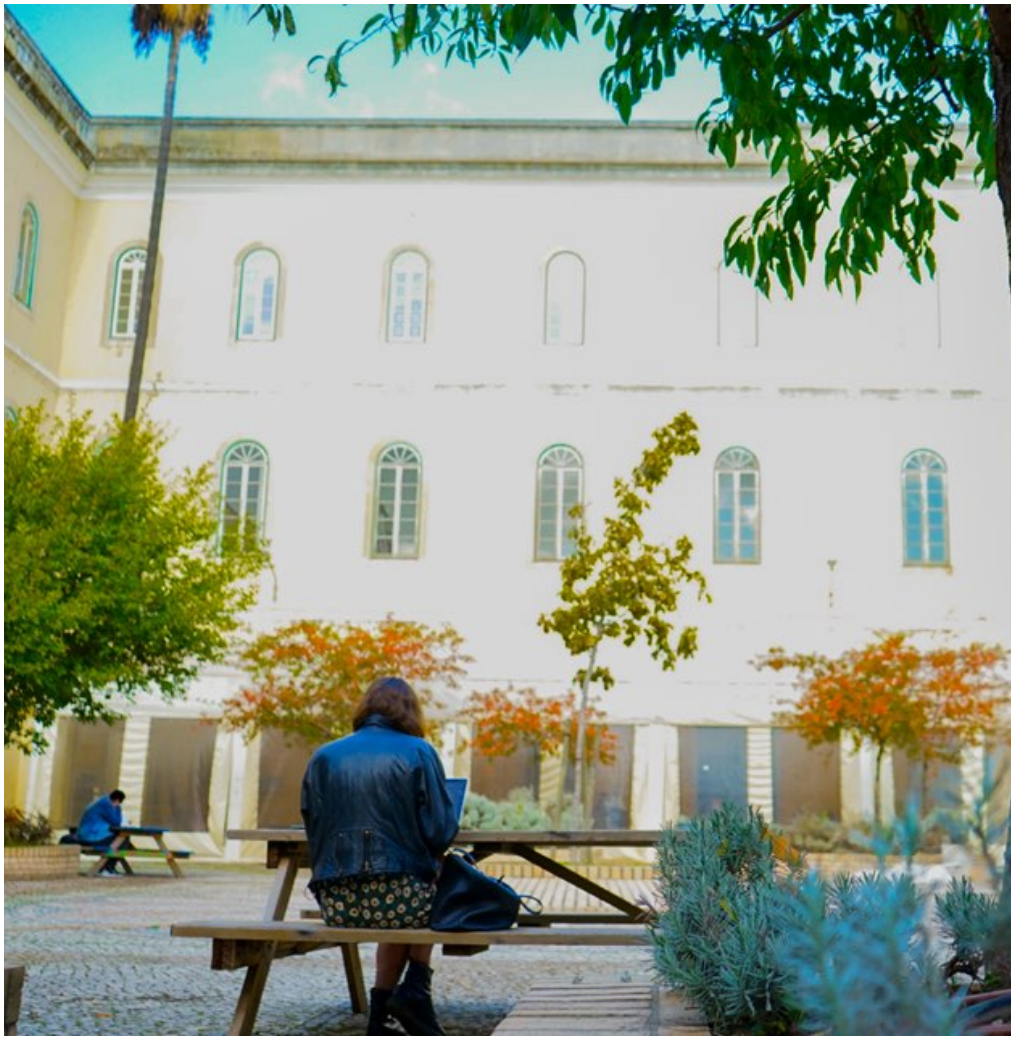

The Environment


The Interface


The Interpreter

# Lecture 1

- ~~Introduction and UC overview~~
- ~~Why Python?~~
- **of Computers and Computer Programs**
- Ipython and Shell



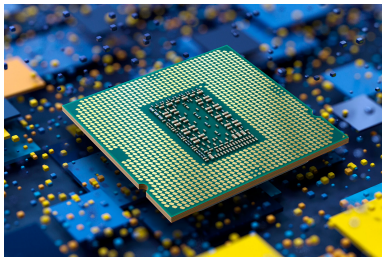
# of Computers

Storage

Working Storage (RAM)


Volatile
Fast

Processing /CPU

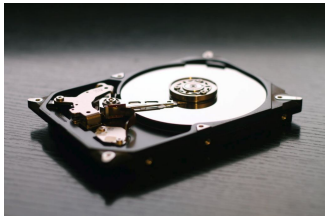


Writes

Reads

Computer Processing Unit

Lifetime Storage (Hard Drive)



HDD


SSD

Persistent
Slow
HDD cheaper
SSD costly

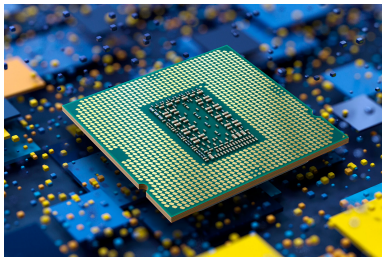
# of Computers

Storage

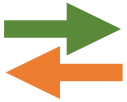
Working Storage (RAM)


Volatile
Fast

Processing /CPU



Writes



Reads

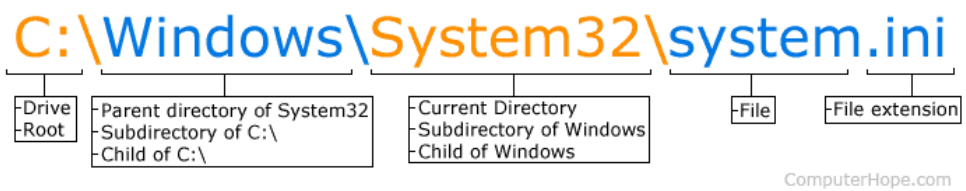
Computer Processing Unit

Lifetime Storage (Hard Drive)

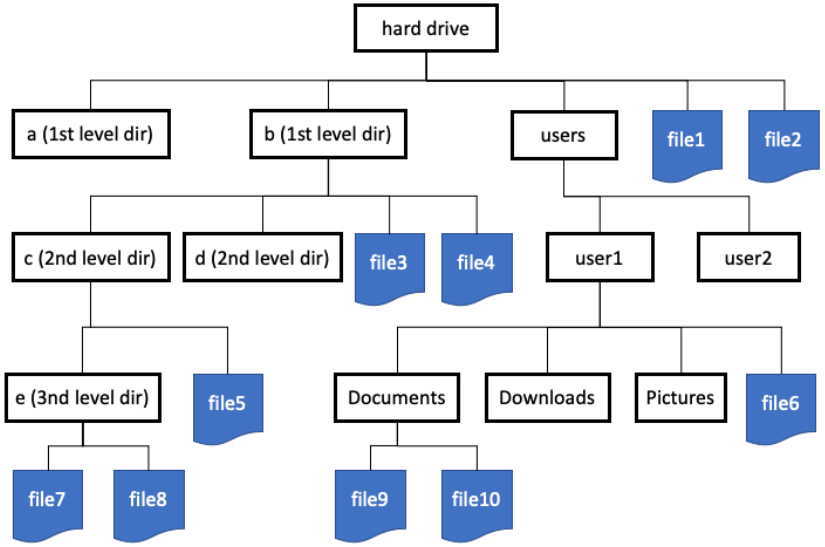

Persistent
Slow
HDD cheaper
SSD costly

| Speed Statistic | HDD (Hard Disk Drive) | SSD (Solid State Drive) | NVMe m.2 (Nonvolatile Memory Express) |
|---|---|---|---|
| Read Speed | 80 MB/s | 200MB/s | 5000 to 7300 MB/s |
| Write Speed | 160 MB/s | 550 MB/s | 5000 to 6350 MB/s |
| Capacity Available | From 250GB to 14TB | 250GB to 4TB | 500GB to 4TB |

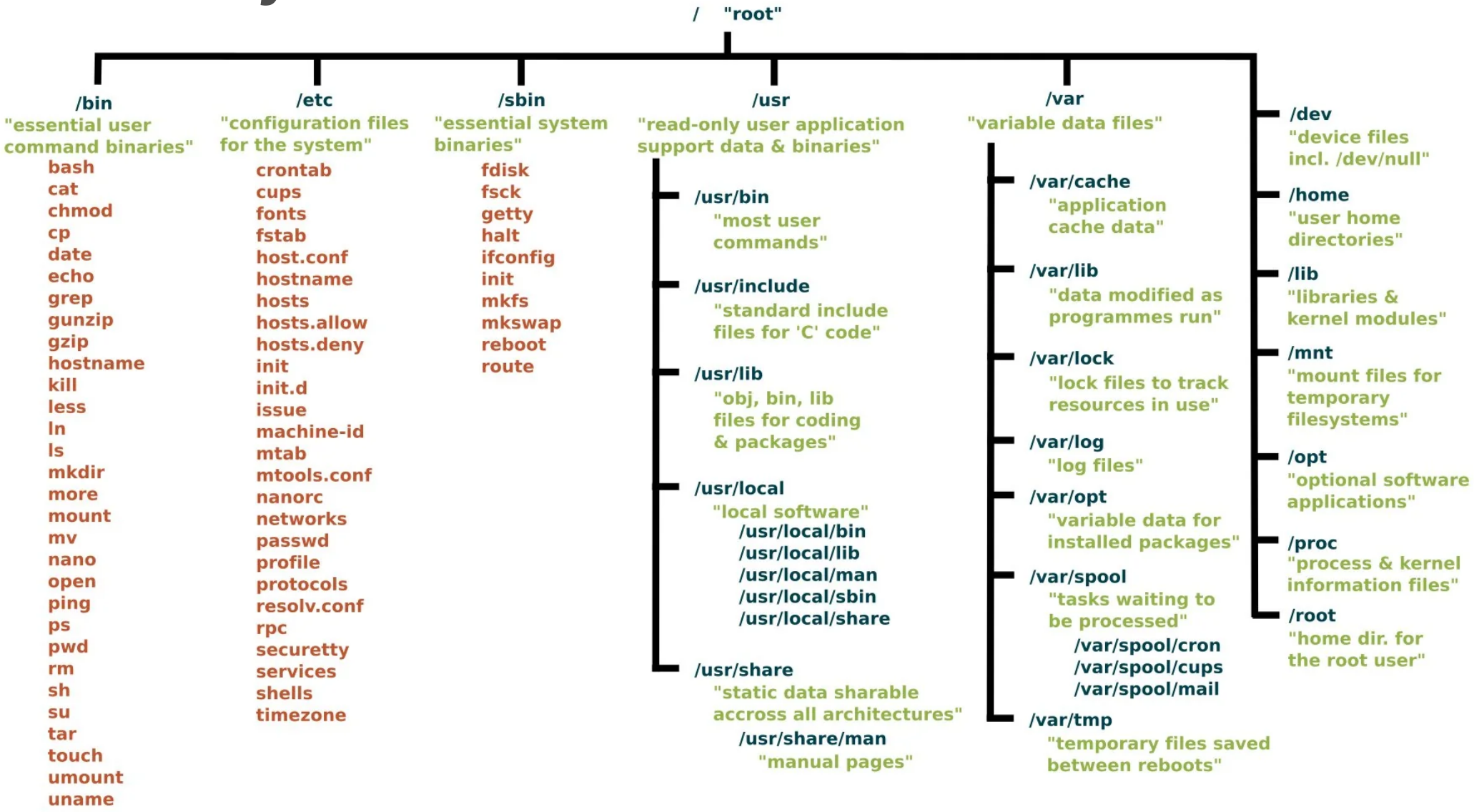
# filepath




# Tree-like Structure



# Linux File System Tree



# Computer Program

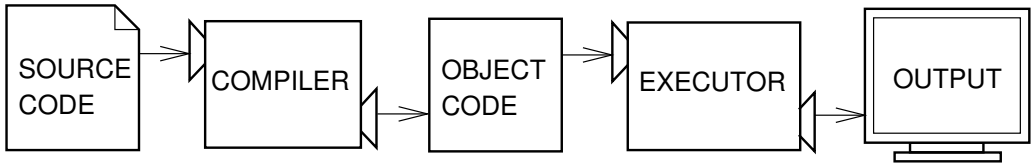
A **program** is a sequence of instructions that specifies how to perform a computation

| | |
|---|---|
| **input** | Get data from the keyboard, a file, or some other device. |
| **output** | Display data on the screen or send data to a file or other device. |
| **math** | Perform basic mathematical operations like addition and multiplication |
| **conditional execution** | Check for certain conditions and execute the appropriate sequence of statements |
| **repetition** | Perform some action repeatedly, usually with some variation |

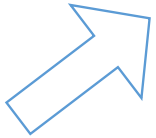
# Programming Languages

## There are two kinds ...

### Compiled Languages




human readable

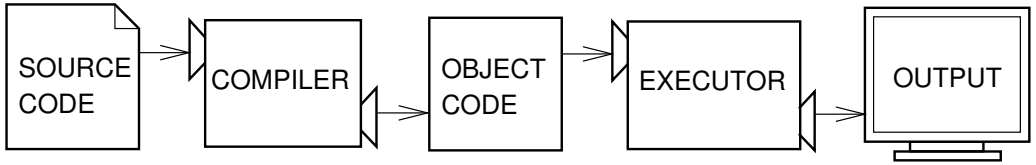


machine readable

# Programming Languages

## There are two kinds ...

### Compiled Languages




### Interpreted Languages

# What is the simplest program we can write in Python?

The “Hello, World!” saga …

# What is the simplest program we can write in Python?

The “Hello, World!” saga …

“Hello World” in python versus C

**Python**

```
print(“Hello, World!”)
```

```
#include <stdio.h>

int main(){
    printf(“Hello, World!\n”);
    return 0;
}
```

* needs to be compiled

# Python print() function

The print() function prints the specified message to the screen, or other standard output device.
The message can be a string, or any other object, the object will be converted into a string before written to the screen.

Syntax `print(*object(s)*, separator=*separator*, end=*end*, file=*file*, flush=*flush*)`

| **Parameter** | **Description** |
|---|---|
| *object(s)* | Any object, and as many as you like. Will be converted to string before printed |
| sep='*separator*' | Optional. Specify how to separate the objects, if there is more than one. Default is '' |
| end='*end*' | Optional. Specify what to print at the end. Default is '\n' (line feed) |
| *file* | Optional. An object with a write method. Default is sys.stdout |
| *flush* | Optional. A Boolean, specifying if the output is flushed (True) or buffered (False). Default is False |

# kill me that bug please

Programming is a complex process, and because it is done by human beings,
it often (always) leads to errors.

| | |
|---|---|
| **Runtime Errors** | Lots of warning, and cryptic error messages.<br>Your program will fail to run. |
| **Semantic Errors** | Your program runs without runtime errors.<br>But it doesn't output what you expected. |
| **Experimental debugging** | It is an art, and of the most important skill to acquire.<br>Comes with 50% practice and 50% problem solving skills.<br>If the bug killing force is strong in you, then you are a hacker. |

# kill me that bug please

**Desired Output**

Untitled by

Machi Tawara

```
print("Cherry, cherry cherry trees begin to bloom,\n\
and bloom is over -- \n\
In the park where nothing (it seems) ever happened.")
```

```
Cherry, cherry cherry trees begin to bloom,
and bloom is over --
In the park where nothing (it seems) ever happened.
```

**Runtime Error**

Why?

```
print("Cherry, cherry cherry trees begin to bloom,\n
      and bloom is over -- \n
      In the park where nothing (it seems) ever happened.")
```

```
  File "<ipython-input-2-70d8c5a00b0a>", line 1
    print("Cherry, cherry cherry trees begin to bloom,\n
                                                       ^
SyntaxError: EOL while scanning string literal
```

**Semantic Error**

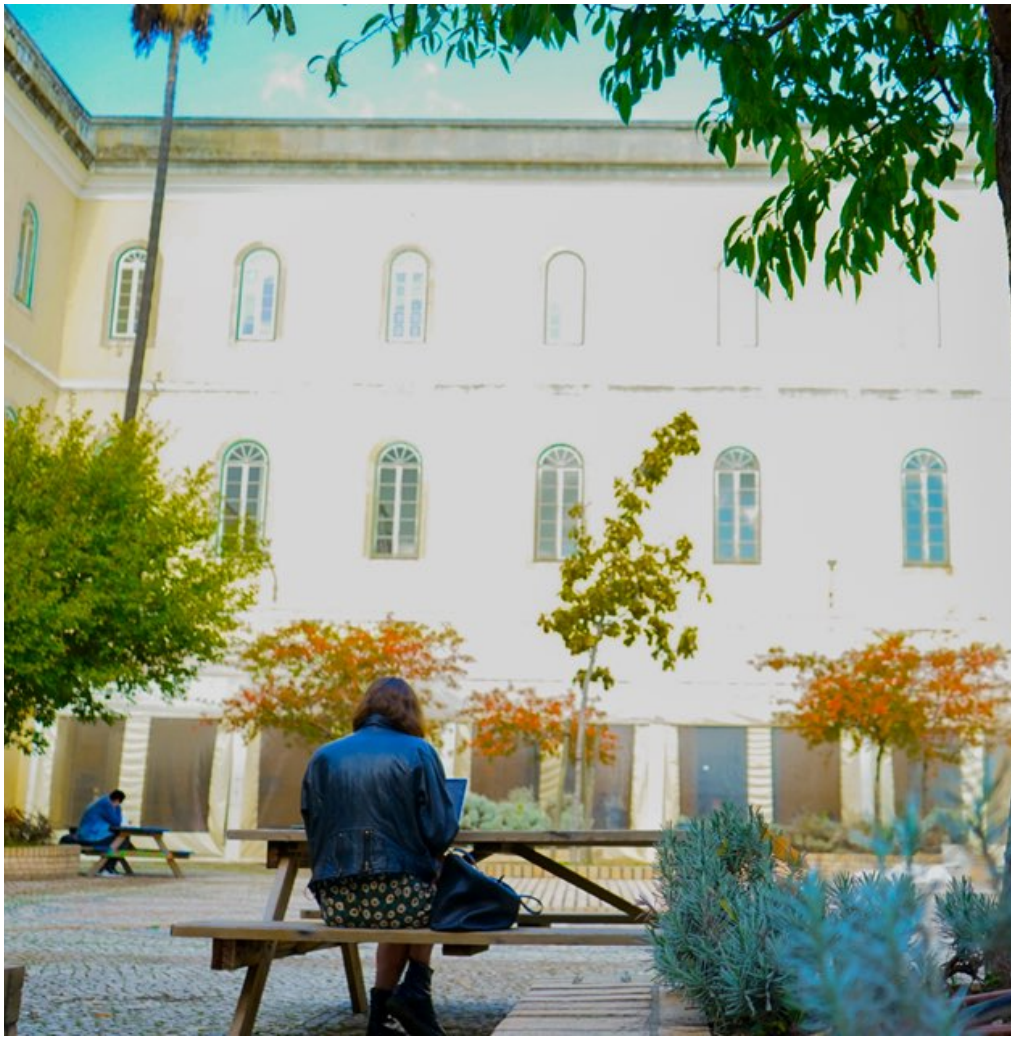
Why?

```
print("Cherry, cherry cherry trees begin to bloom,\
and bloom is over -- \
In the park where nothing (it seems) ever happened.")
```

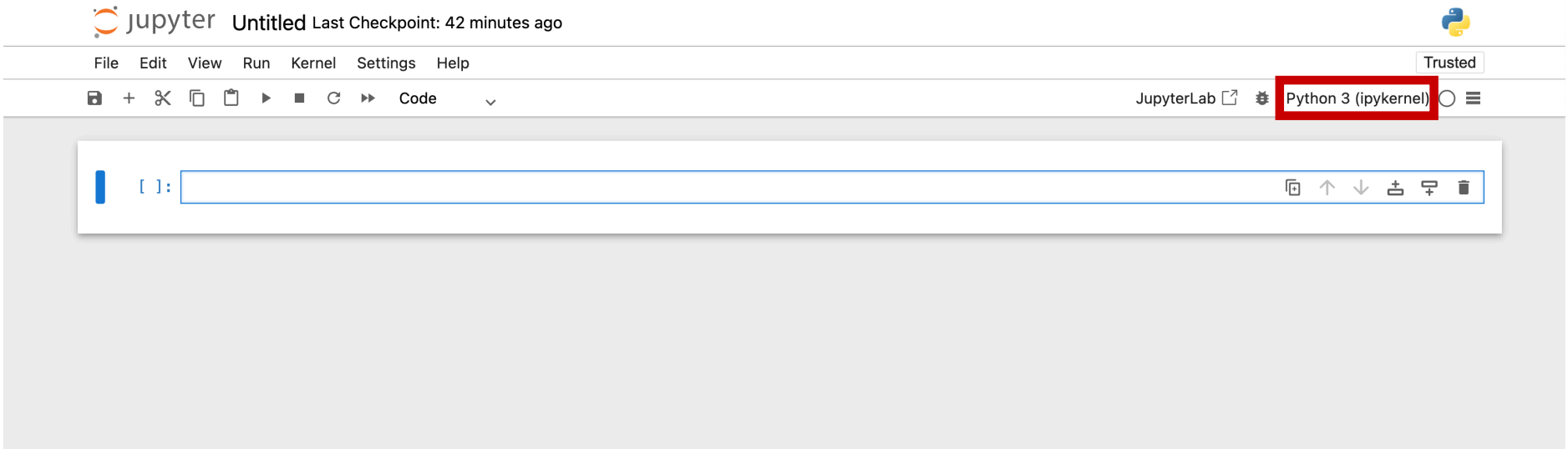
```
Cherry, cherry cherry trees begin to bloom,and bloom is over --
```

# Lecture 1

- ~~Introduction and UC overview~~
- ~~Why Python?~~
- ~~of Computers and Computer Programs~~
- **Ipython and Shell**



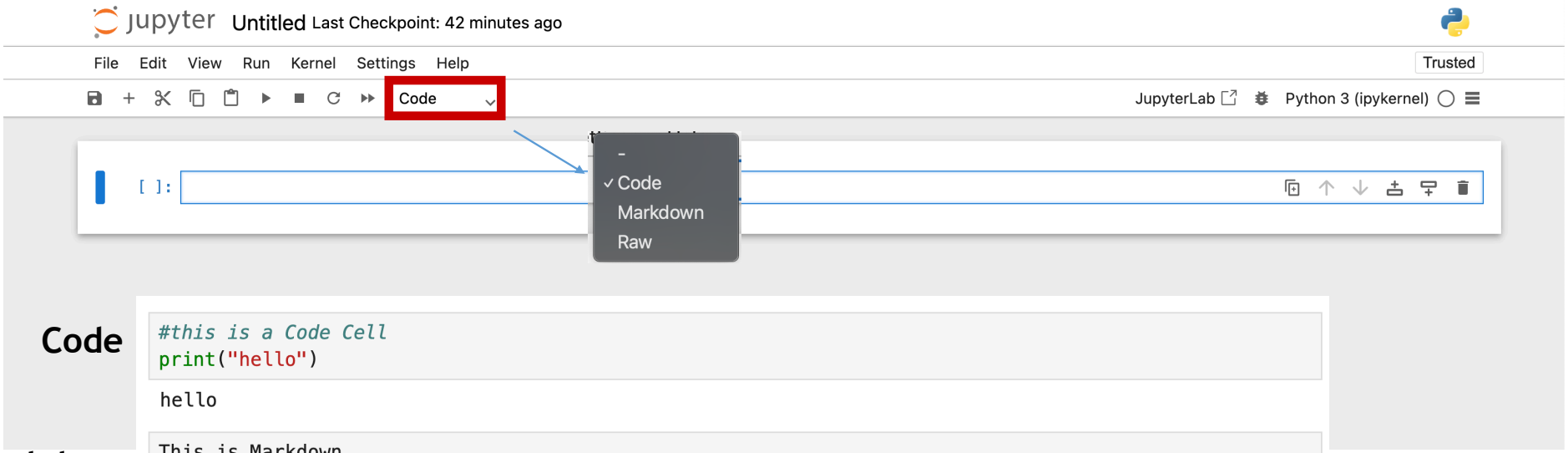
# Jupyter Notebook




The **Jupyter Notebook** is a browser-based graphical interface to the **IPython shell**, and builds on it a rich set of dynamic display capabilities. As well as executing Python/IPython statements, notebooks allow the user to include formatted text, static and dynamic visualizations, mathematical equations, JavaScript widgets, and much more.

**IPython** is an interactive command-line terminal for Python. It was created by Fernando Perez in 2001. IPython offers an enhanced read-eval-print loop (REPL) environment particularly well adapted to scientific computing. In other words, IPython is a powerful *interface* to the Python language. But it is certainly not the only one. Besides IPython, the most common way to use Python is to write *scripts*, files with the .py extension.

# Jupyter Notebook




**Code**

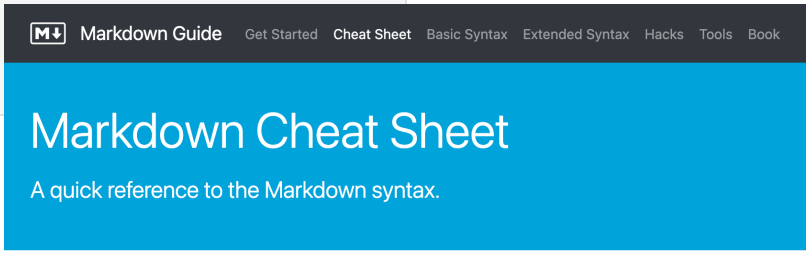
```
#this is a Code Cell
print("hello")
```

```
hello
```

**Markdown**

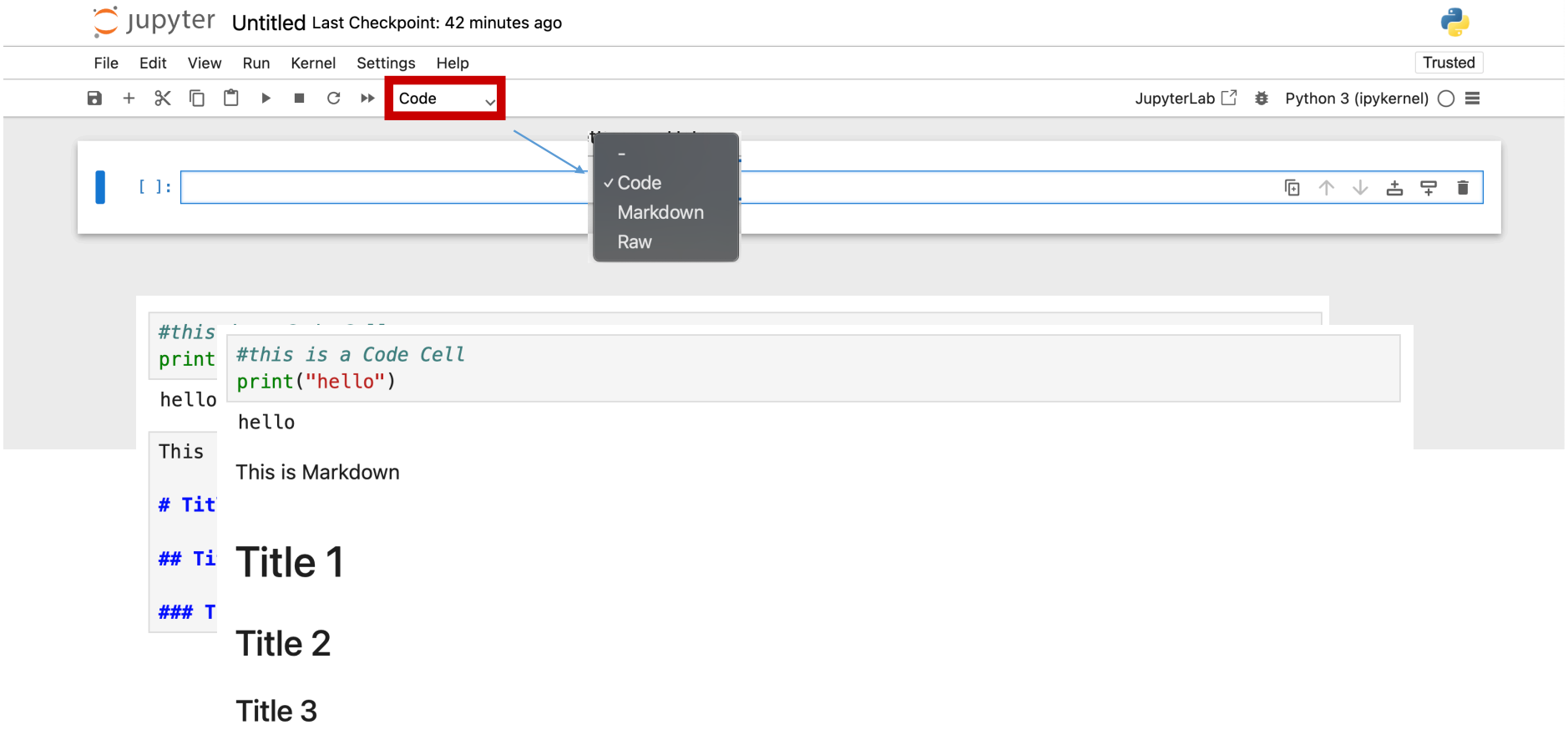
```
This is Markdown

# Title 1

## Title 2

### Title 3
```

**html, latex, markdown**

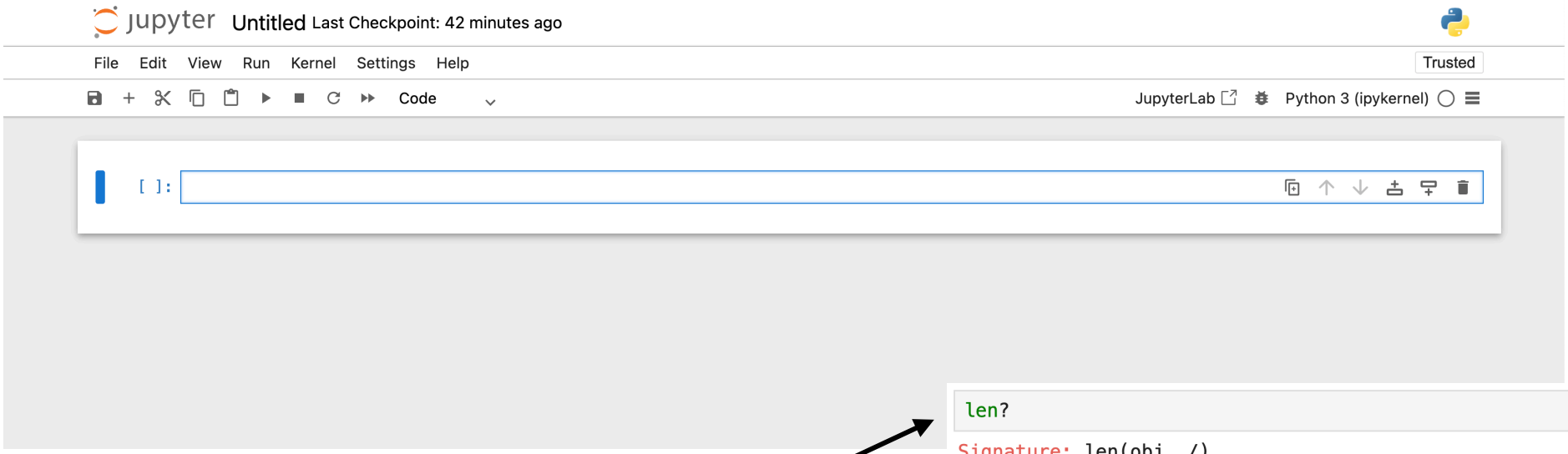



https://www.markdownguide.org/

# Jupyter Notebook



# Jupyter Notebook




useful things to keep in mind:

- Tab completion for objects and modules
- **use ? to access the documentation**
- **use ?? to acccess the source code (only if written in python)**
- wildcard matching
- check Keyboard shortcuts (Help menu)

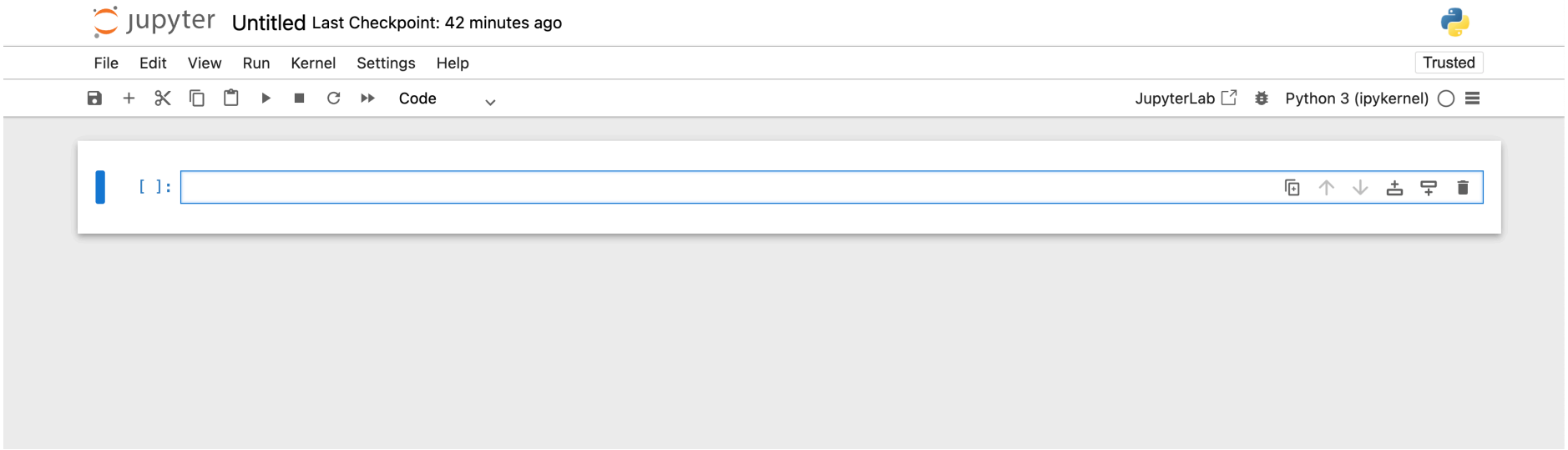
```
len?
```

```
Signature: len(obj, /)
Docstring: Return the number of items in a container.
Type:      builtin_function_or_method
```

```
square??
```

```
Signature: square(a)
Source:
def square(a):
    '''Return the square of a.'''
    return a**2
File:      /var/folders/_9/nnlnjrjd2yx_hp0l0rwlzqx00000gn/T/ipykern
03058.py
Type:      function
```

# Jupyter Notebook




useful things to keep in mind:

- Tab completion for objects and modules
- use ? to access the documentation
- use ?? to acccess the source code (only if written in python)
- **wildcard matching**
- check Keyboard shortcuts (Help menu)

```
str.*find*?
```

```
str.find
str.rfind
```

Notice that the * character matches any string, including the empty string.

# Magic Commands

are designed to solve various common problems in standard data analysis. Magic commands come in two flavors: line magics, which are denoted by a single **% prefix** and operate on a **single line** of input, and cell magics, which are denoted by a double **%% prefix** and operate on **multiple lines of input**.

**%timeit** - determines the execution time of a single-line statement. Performs multiple runs to achieve robust results.

```
%timeit L = [n**2 for n in range(1000)]
```

```
29.4 µs ± 234 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

**%%timeit** - same as %timeit but for the entire cell

```
%%timeit
L = []
for n in range(1000):
    L.append(n**2)
```

compreehensions are faster :(

```
35.7 µs ± 203 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

**%time** - determines the execution time of a single-line statement. Performs a single run!

```
%time L = [n**2 for n in range(1000)]
```

```
CPU times: user 158 µs, sys: 7 µs, total: 165 µs
Wall time: 169 µs
```

**%%time** - same as %time but for the entire cell

```
%%time
L = []
for n in range(1000):
    L.append(n**2)
```

```
CPU times: user 193 µs, sys: 9 µs, total: 202 µs
Wall time: 206 µs
```

# Magic Commands

are designed to solve various common problems in standard data analysis. Magic commands come in two flavors: line magics, which are denoted by a single **% prefix** and operate on a **single line** of input, and cell magics, which are denoted by a double **%% prefix** and operate on **multiple lines of input**.

**%run** - Run the named file inside IPython as a program

```
%run myscript.py
```

```
1 squared is 1
2 squared is 4
3 squared is 9
```

```
!cat myscript.py
```

```
# file: myscript.py

def square(x):
        """square a number"""
        return x ** 2

for N in range(1, 4):
        print(f"{N} squared is {square(N)}")
```

# Magic Commands

are designed to solve various common problems in standard data analysis. Magic commands come in two flavors: line magics, which are denoted by a single **% prefix** and operate on a **single line** of input, and cell magics, which are denoted by a double **%% prefix** and operate on **multiple lines of input**.

**%history** – displays the command history. Use %history -n to display last n-commands with line numbers.

**%recall <line_no>** – re-executes command at line_no. You can also specify range of line numbers.

**%who** – Shows list of all variables defined within the current notebook

**%lsmagic** – Shows a list of magic commands

**%magic** – Quick and simple list of all available magic functions with detailed descriptions

**%quickref** – List of common magic commands and their descriptions

# Magic Commands

are designed to solve various common problems in standard data analysis. Magic commands come in two flavors: line magics, which are denoted by a single **% prefix** and operate on a **single line** of input, and cell magics, which are denoted by a double **%% prefix** and operate on **multiple lines of input**.

*Table 2-2. Some frequently used IPython magic commands*

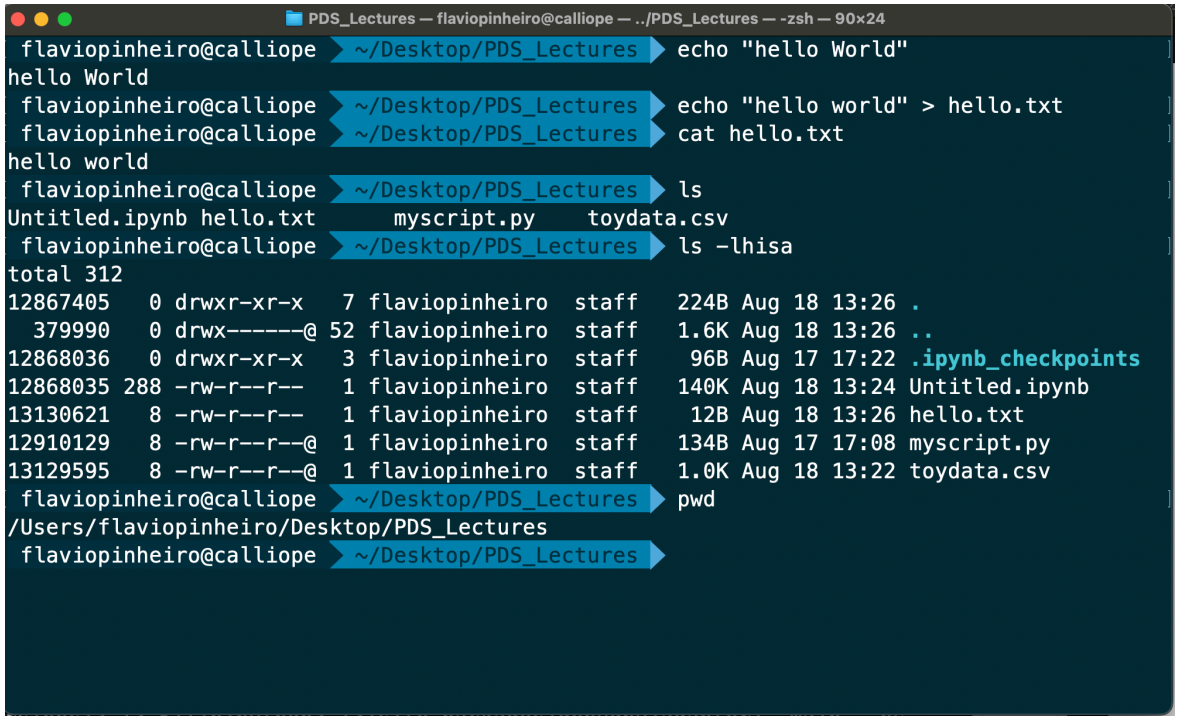
| Command | Description |
|---|---|
| `%quickref` | Display the IPython Quick Reference Card |
| `%magic` | Display detailed documentation for all of the available magic commands |
| `%debug` | Enter the interactive debugger at the bottom of the last exception traceback |
| `%hist` | Print command input (and optionally output) history |
| `%pdb` | Automatically enter debugger after any exception |
| `%paste` | Execute preformatted Python code from clipboard |
| `%cpaste` | Open a special prompt for manually pasting Python code to be executed |
| `%reset` | Delete all variables/names defined in interactive namespace |
| `%page` *`OBJECT`* | Pretty-print the object and display it through a pager |
| `%run` *`script.py`* | Run a Python script inside IPython |
| `%prun` *`statement`* | Execute *`statement`* with `cProfile` and report the profiler output |
| `%time` *`statement`* | Report the execution time of a single statement |
| `%timeit` *`statement`* | Run a statement multiple times to compute an ensemble average execution time; useful for timing code with very short execution time |
| `%who, %who_ls, %whos` | Display variables defined in interactive namespace, with varying levels of information/ verbosity |
| `%xdel` *`variable`* | Delete a variable and attempt to clear any references to the object in the IPython internals |

# Shell Commands

Shell commands are instructions you type into a computer's command-line interface, often called a "terminal" or "shell," to tell the computer what to do. Think of it like giving your computer short, text-based orders to perform tasks, such as opening files, moving folders, or running programs.

## Example of Shell Commands in Terminal

```
osx:~ $ echo "hello world"            # echo is like Python's print function
hello world

osx:~ $ pwd                           # pwd = print working directory
/home/jake                            # This is the "path" that we're sitting in

osx:~ $ ls                            # ls = list working directory contents
notebooks  projects

osx:~ $ cd projects/                  # cd = change directory

osx:projects $ pwd
/home/jake/projects

osx:projects $ ls
datasci_book   mpld3   myproject.txt

osx:projects $ mkdir myproject        # mkdir = make new directory

osx:projects $ cd myproject/

osx:myproject $ mv ../myproject.txt ./  # mv = move file. Here we're moving the
                                        # file myproject.txt from one directory
                                        # up (../) to the current directory (./).
osx:myproject $ ls
myproject.txt
```



# Shell Commands

Shell commands are instructions you type into a computer's command-line interface, often called a "terminal" or "shell," to tell the computer what to do. Think of it like giving your computer short, text-based orders to perform tasks, such as opening files, moving folders, or running programs. **Any standard shell command can be used directly in IPython by prefixing it with the ! character.**

## Example of Shell Commands in IPython

```
In [1]: !ls
myproject.txt

In [2]: !pwd
/home/jake/projects/myproject

In [3]: !echo "printing from the shell"
printing from the shell
```

some of these commands also have magic counterparts:
ls maps to %ls and the pwd to %pwd

you can save the outputs into variables in python

```
path = !pwd
```

```
print(path)
```

```
['/Users/flaviopinheiro/Desktop/PDS_Lectures']
```

```
type(path)
```

```
IPython.utils.text.SList
```

just note that they will have special types

# Shell Commands

Shell commands are instructions you type into a computer's command-line interface, often called a "terminal" or "shell," to tell the computer what to do. Think of it like giving your computer short, text-based orders to perform tasks, such as opening files, moving folders, or running programs. **Any standard shell command can be used directly in IPython by prefixing it with the ! character.**

## Example of Shell Commands in IPython

```
In [1]: !ls
myproject.txt

In [2]: !pwd
/home/jake/projects/myproject

In [3]: !echo "printing from the shell"
printing from the shell
```

some of these commands also have magic counterparts:
ls maps to %ls and the pwd to %pwd

some particularly useful commands for file inspection

```
!cat myscript.py
```

cat prints the content of a file

```
# file: myscript.py

def square(x):
        """square a number"""
        return x ** 2

for N in range(1, 4):
        print(f"{N} squared is {square(N)}")
```

```
!head toydata.csv
```

head prints the top ten lines

```
Name,Address,Age,Gender
Alice Johnson,"123 Maple St, Springfield",34,Female
Bob Smith,"456 Oak Ave, Rivertown",28,Male
Clara Davis,"789 Pine Rd, Hillview",45,Female
David Brown,"101 Elm St, Lakeside",52,Male
Emma Wilson,"234 Birch Ln, Sunnyvale",19,Female
Frank Miller,"567 Cedar Dr, Greentown",37,Male
Grace Lee,"890 Walnut St, Brookfield",26,Female
Henry Taylor,"111 Spruce Way, Meadowville" 41 Male
Isabella Clark,"222 Chestnut Rd, Fairviev
```

quite useful to inspect the format of a file without opening it!

# Shell Commands

Shell commands are instructions you type into a computer's command-line interface, often called a "terminal" or "shell," to tell the computer what to do. Think of it like giving your computer short, text-based orders to perform tasks, such as opening files, moving folders, or running programs. **Any standard shell command can be used directly in IPython by prefixing it with the ! character.**

## Example of Shell Commands in IPython

```
In [1]: !ls
myproject.txt

In [2]: !pwd
/home/jake/projects/myproject

In [3]: !echo "printing from the shell"
printing from the shell
```

some of these commands also have magic counterparts:
ls maps to %ls and the pwd to %pwd

some particularly useful commands for file inspection

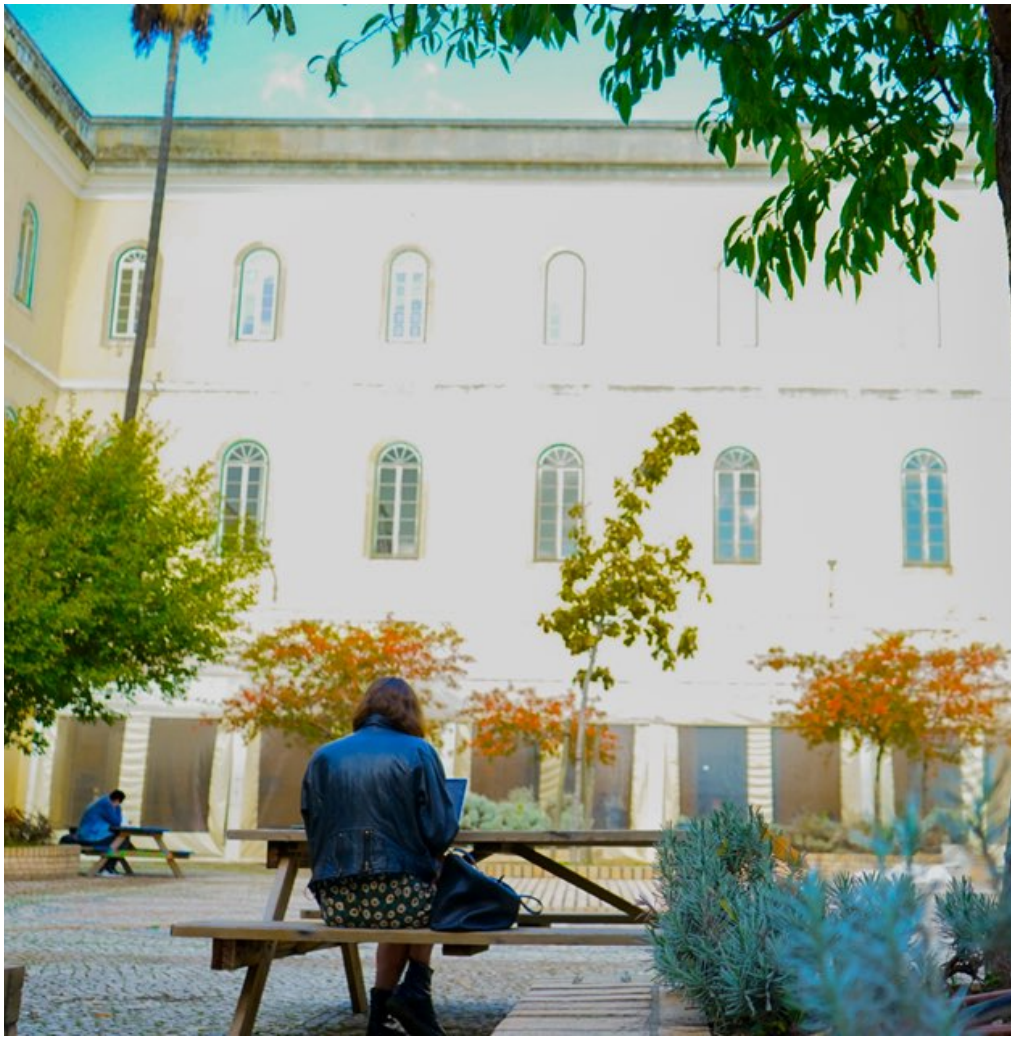
```
!cat toydata.csv | wc -l
```

```
20
```

cat prints the content of a file
wc is the word count command
the option -l prints only the number of lines
note that the symbol | is a pipe

# Next Episode!

- Language Semantics
- Python Data Types and Structures
- Operators, Comprehensions, Slices, Typecast
- Flow Control



# Quizzes

https://www.socrative.com

Login as a student

Room Name: **PDS2025**

Student ID: **Student Number**

or

https://api.socrative.com/rc/5NpKRX

Student ID: **Student Number**

**Please Join Now!**