
DÉVELOPPER LES VENTES EN MAGASIN

# SOFTWARE ENGINEERING FOR ML

*- From idea to product -*

## TELECOM PARISTECH MSBGD
## 2019 - 2020

**Nicolas Gallot**
Lead Data Engineer
@Armis_Tech
*nicolas.gallot@armis.tech*

2019-11-14

# Your Speakers

- Nicolas Gallot:
  - Arts et Métiers ParisTech engineer (2009)
  - 7 years of software development for Société Générale trading activities in Hong Kong: vba, C#, sql, MongoDb
  - MSBGD 2016-2017
  - Post master:
    - 1 year data engineer @ MFG Labs
    - 1+ year lead data engineer @ Armis
- Cyril Monti:
  - ESME Sudria (2017)
  - 1 year as Data engineer consultant at Capgemini
  - MSBGD 2018-2019
  - Post master : data engineer intern @ Armis

# THE COMPANY

# ARMIS

- ARMIS is a startup created in 2016, located in Paris (Madeleine)
- About 50 peoples in the company
- ARMIS helps clients to:
    - digitalize their circular
    - boost store visit
    - achieve better marketing campaigns performances
- SaaS platform
- Already two Fundraisings and a third in 2020 =>  Some opportunities in Data team
- Internationalization phase : American market for 2020

# Data at Armis

- 3 teams : Data Science, Data Engineer, Business Intelligence
- Data Science:
  - Make algorithms to help:
    - Optimizing spending on advertisement platforms
    - Create smarter media contents (nlp, image...)
- Data Engineering:
  - Builds a robust platform for interacting with DS algos
  - Handles bigger amounts of data
  - Trains DS team to write better and more scalable code

# Data at Armis

- Data Engineering team facts:
  - Did not exist 1 year ago
  - Refactored main algorithms written by DS team
  - Deployed 5 REST apis to interact with algorithms
  - Implemented Spark job for high volumes processing
  - Trained BI and DS teams to write better code

# INTRODUCTION

# Why this course?

- My personal feedback:
  - Most major Data Science courses (in France) are focusing mainly on maths / modeling ⇒ *very cool, if you want to do pure R&D*
  - You will be good at building models ⇒ *but you're not alone*
  - Recruiters (especially startups) are looking for "full stack" data scientists ⇒ *from designing model to deliver it in production*
  - Conclusion ⇒ *solid software engineering skills will make the difference*

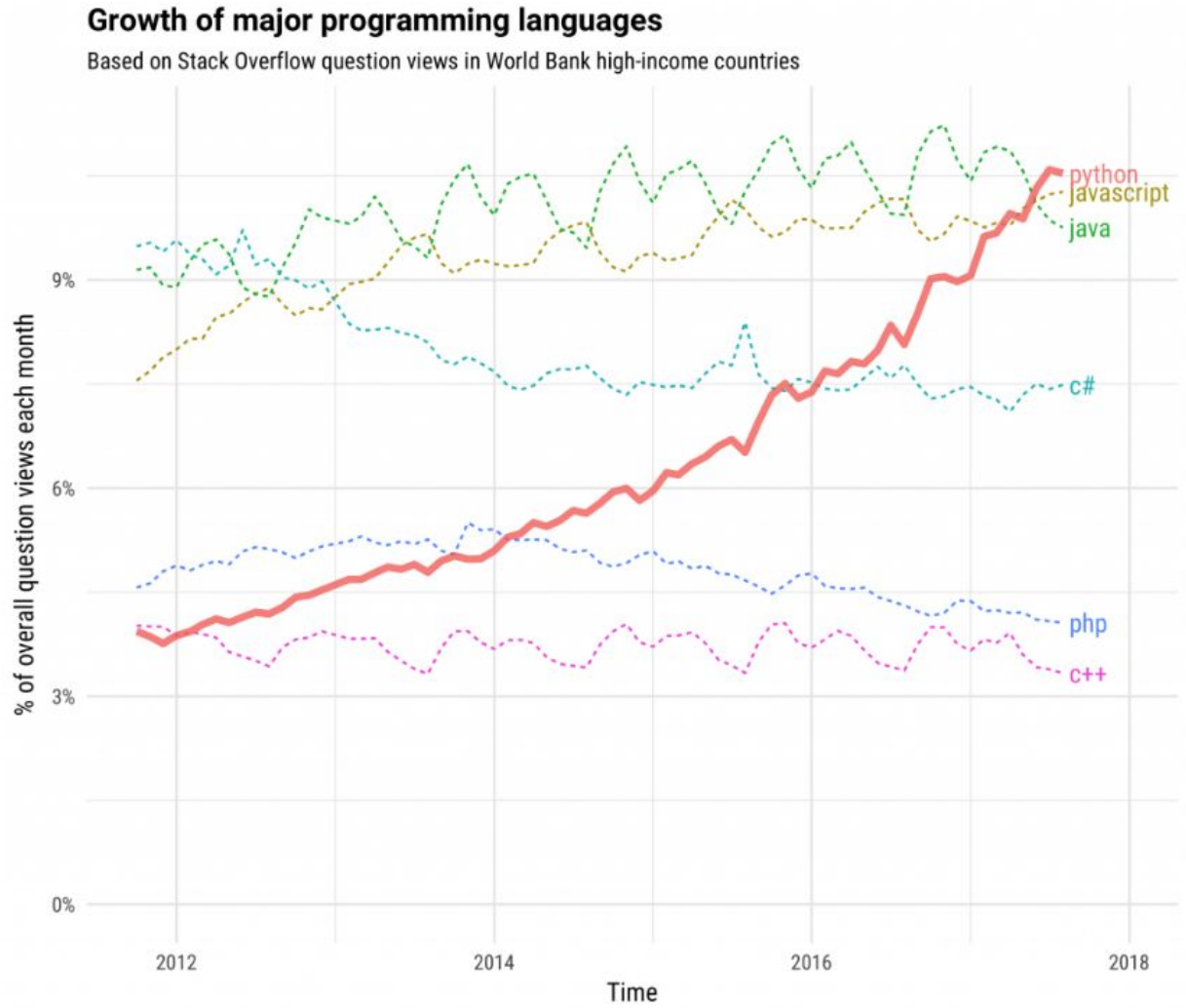
# — COURSE 1 —
# FROM WRITING CODE TO DEVELOPING SOFTWARE

# Course contents

1. Why Python?
    a. Some statistics, pros and cons of Python
    b. Few coding standards in Python
    c. Software developments tips and tricks
2. Software development golden rules
    a. Some obvious but useful techniques
    b. Git / GitHub crash course and methodology
    c. Introduction to semantic versioning
    d. The Python development setup: virtualenv, docker
3. Exposing Python code
    a. Making python packages
    b. REST api concepts
    c. Python FastApi framework

# WHY PYTHON?

# Some statistics




ARMIS

# Some Python facts

- One of the easiest languages to learn
  - Tons of open source libraries
- Heavily used in the data community
  - Pandas for data wrangling, analytics
  - Matplotlib / Plotly / Dash for data visualization
  - Python API for all “serious” ML libraries
- Used in production for very large projects in tech giants:
  - 21% of Facebook’s codebase
  - Google: first google search engine, 100% of Youtube Backend
  - Instagram: django app
  - Netflix / Spotify recommendation and analytics engine
  - ...

# Python pros/cons

- Pros:
  - development speed
  - ease of use
  - ecosystem / community

- Cons:
  - maybe too easy ⇒ also easy to write bad code
  - not compiled ⇒ more difficult to track mistakes

ARMIS

# Python basic standards

- Few coding standards in Python
  - PEP8 rules!
  - Use type hints (variables and functions) as much as possible
  - Naming rules:
    - Always self explanatory!
    - Always in english!
    - snake_case for variables and file names, CamelCase for classes

- Some useful things:
  - Overriding the __str__ method in classes: this is how to print useful things
  - Use f-strings to use variables inside strings

# Python basic standards

```python
from typing import List


class Student:
    _first_name: str
    _last_name: str

    def __init__(self, first_name: str, last_name: str):
        self._first_name = first_name
        self._last_name = last_name

    @property
    def first_name(self) -> str:
        return self._first_name

    @property
    def last_name(self) -> str:
        return self._last_name

    @property
    def full_name(self) -> str:
        return f'{self._first_name} {self._last_name}'

    def __str__(self) -> str:
        return self.full_name


class TelecomMsbgdClass:
    _year: int
    _students: List[Student]

    def __init__(self, year: int, students: List[Student]):
        self._year = year
        self._students = students

    def __str__(self):
        students = '\n    '.join(map(str, self._students))
        return f'''
Telecom MSBGD:
- year:
    {self._year}
- students:
    {students}
        '''
```

```
In [3]: from typing import List

In [4]: axel_camara: Student = Student(first_name='Axel', last_name='Camara')

In [5]: chloe_vuillet: Student = Student(first_name='Chloé', last_name='Vuillet')

In [6]: msgbd_class = TelecomMsbgdClass(year=2019, students=[axel_camara, chloe_vuillet])

In [7]: print(msgbd_class)

Telecom MSBGD:
- year:
    2019
- students:
    Axel Camara
    Chloé Vuillet
```

ARMIS

# Software development tips and tricks

- RTFM! (aka Read The F... Manual): official documentation, GitHub readme, etc...
- Google is your friend: always in English!
- Stackoverflow is your second best friend (often where Google redirects you...)
- Think first, code after
- Don’t try to make everything generic at the beginning. Refactoring and shared code will come after. Implement only the feature you’ve been asked.
- Open source libraries choices:
  - Check the comments on stackoverflow
  - Check the number of stars / contributors / forks on GitHub
  - Is it still maintained? Under development?

# Software development tips and tricks

- Make your code understandable by others (variable names, type hints, doc).
- Handle properly exceptions with try/except blocks
- Use logging instead of print()
- Use database connections with care: you can make an application crash easily.

  Great ORM in Python: SQLAlchemy. Use connection pooling.
- When making http requests to query an api:
  - Check carefully the client api documentation
  - Handle responses status code
  - Parallelize multiple requests: use ThreadPool or asynchronous requests.

ARMIS

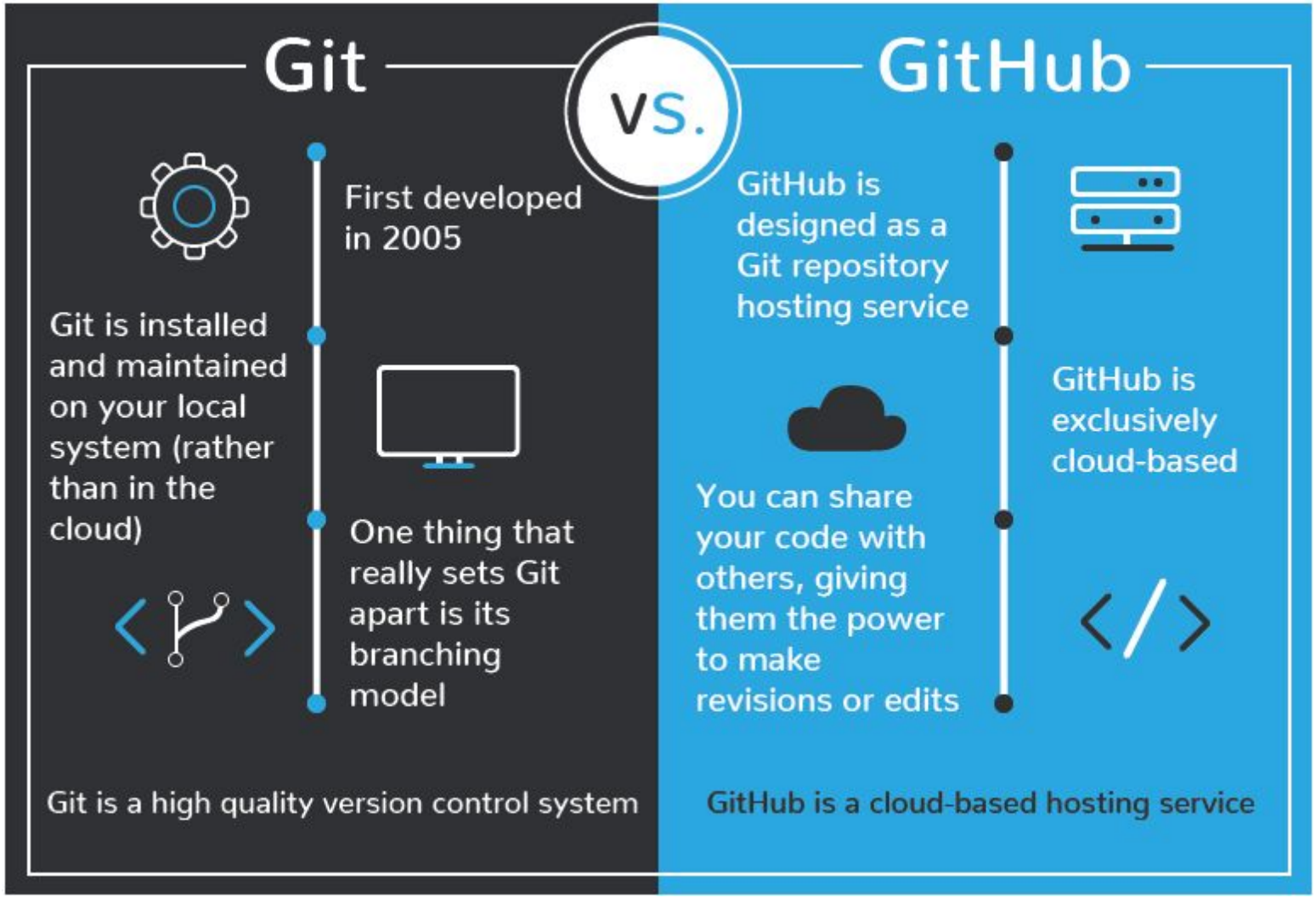
# Software development tips and tricks

- Use .ini files with configparser in Python to manage configuration files.
- Never EVER commit secrets / credentials. They should be kept away from the repo.
- Use environment variables to store credentials (database passwords, api secret key, etc...)
- Python's configparser can interpolate environment variables

ARMIS

# SOFTWARE DEVELOPMENT GOLDEN RULES

# Git / GitHub crash course



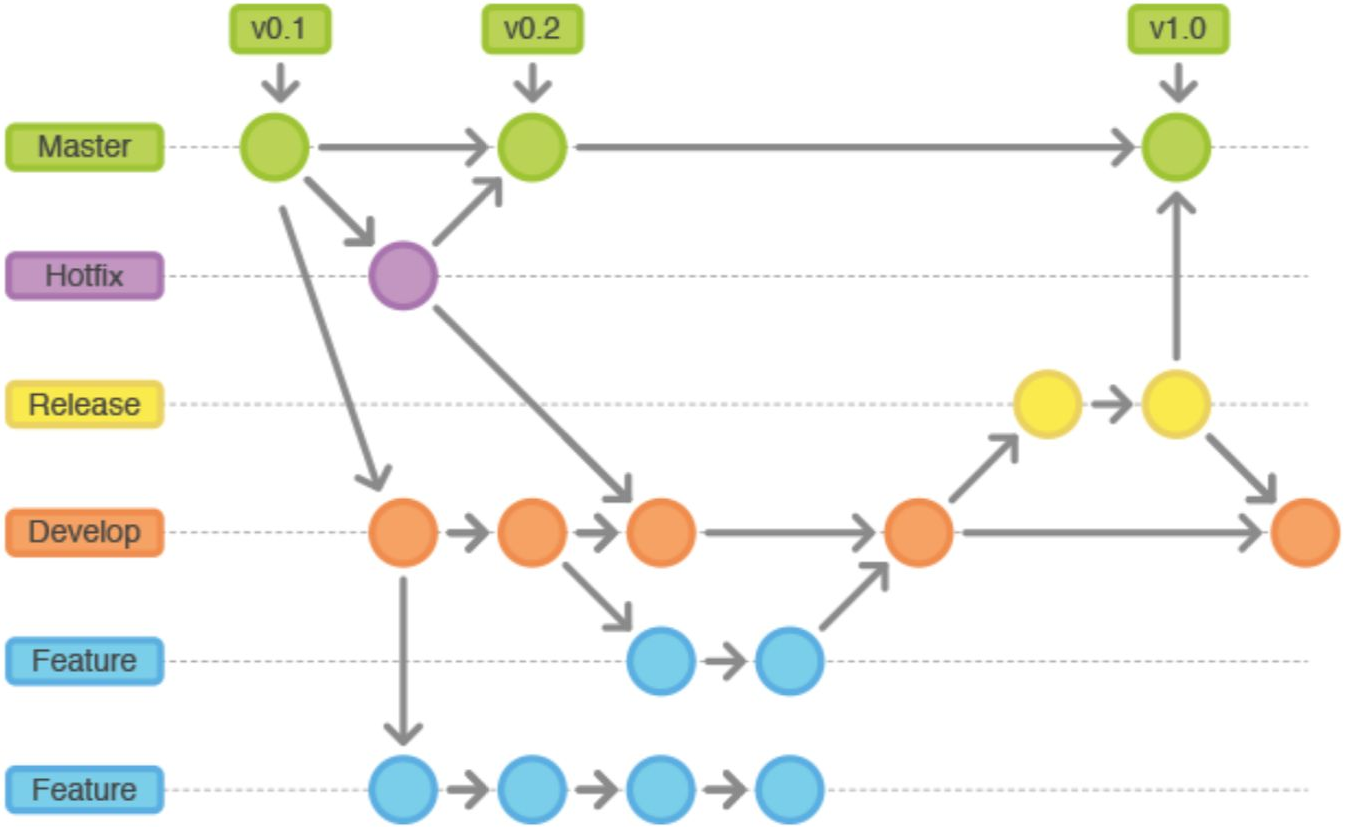

ARMIS

# Git / GitHub crash course

- Git Flow: a standard way to develop new features on an existing code base.
  - Each new feature is developed on a dedicated branch
  - Each *deployed* environment has its own branch
  - Steps to do when developing a new feature (called *awesome_feature*):
    - git checkout *master (or develop / staging)*
    - git pull
    - git checkout -b *feature/awesome_feature*
    - …. doing my code changes
    - git diff
    - git add *my_changed_file.py*
    - git commit -m *'The message I want to show to other developers'*
    - git push (if pushing for the first time: git push *--set-upstream origin master*)
    - ⇒ Then I go to GitHub, and open a Pull Request for my change
    - Code review with another member from the team
    - Merge the pull request and eventually delete the branch.

# Git / GitHub crash course

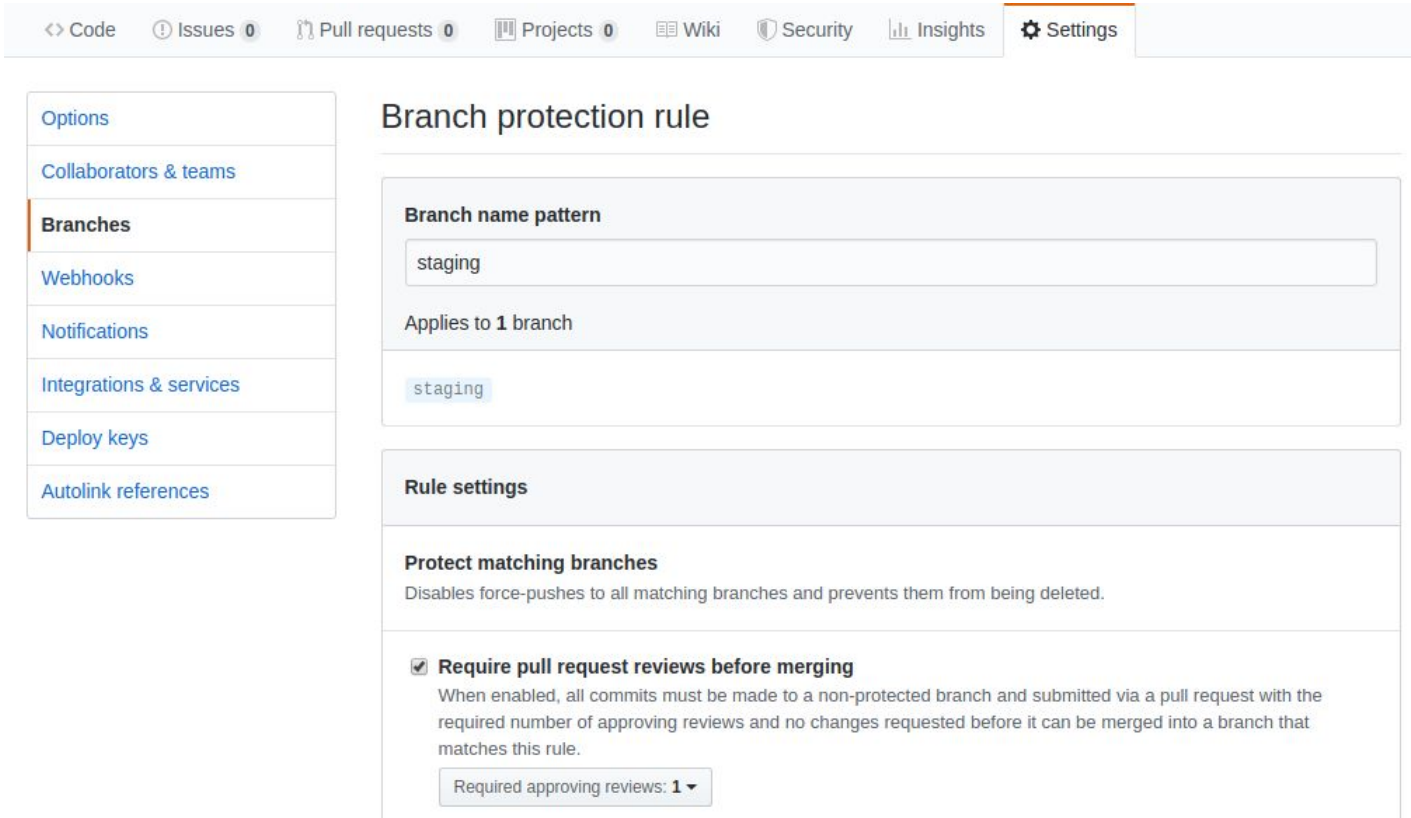
- Git Flow: a standard way to develop new features on an existing code base.




ARMIS

# Git / GitHub crash course

- ALWAYS create a GitHub repo to host your code
- OFTEN push your code changes to it to keep it synchronized
- PROTECT your code:
  - Set proper user rights
  - Use branch protection rules
    - Enforce pull requests
    - CI / CD to run unit tests
- ALWAYS add a .gitignore file




ARMIS

# Introduction to semantic versionning

- Version names follow standards called semantic versioning (semver)
- Where to put version information in Python?
  - setup.py
  - __init__.py
- There's even a python package to manage that!

```
>>> ver = semver.parse_version_info("3.4.5")
>>> ver.bump_major()
VersionInfo(major=4, minor=0, patch=0, prerelease=None, build=None)
```

ARMIS

# The Python development setup: virtualenv

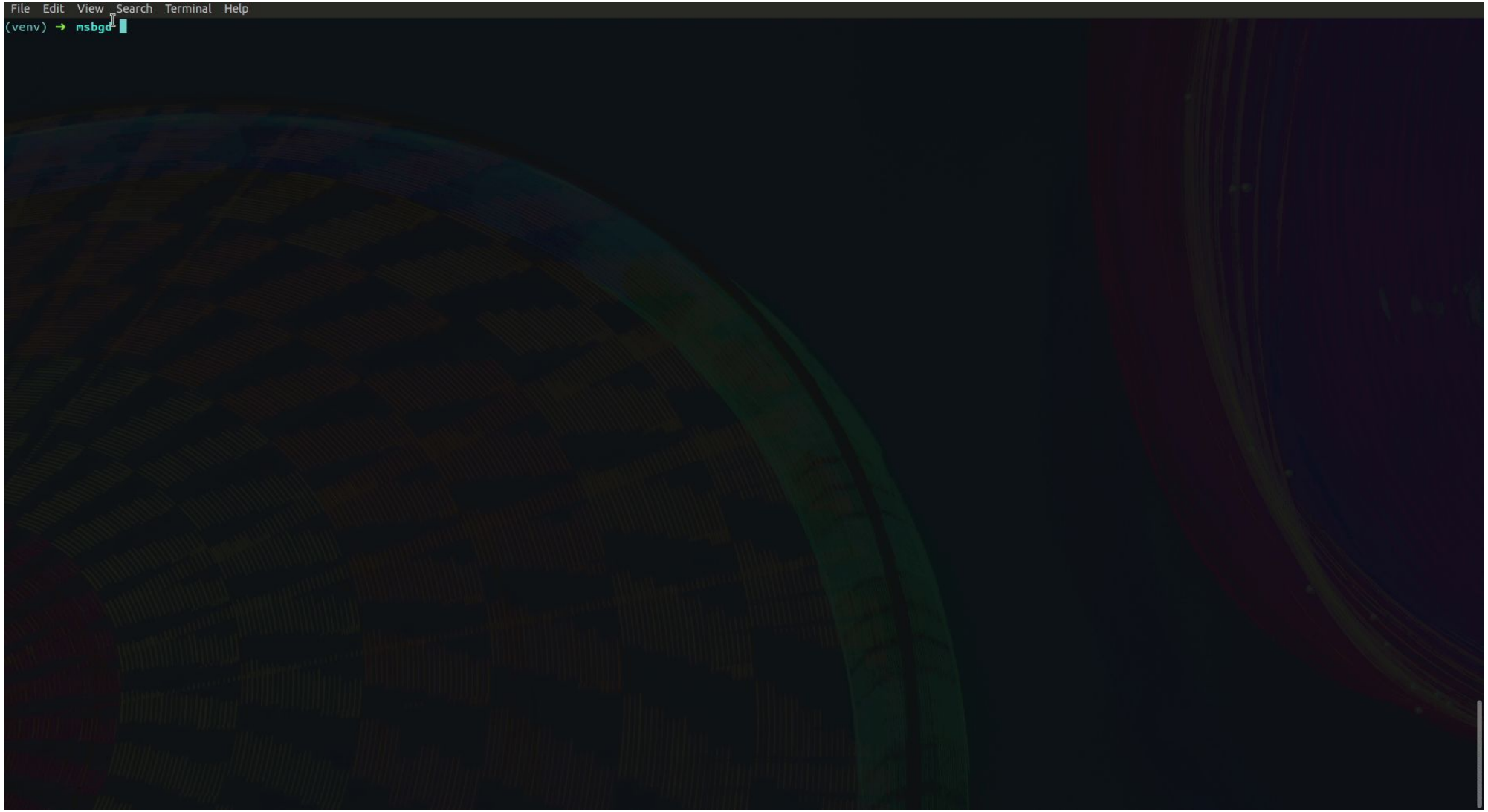
- Virtualenv: what for?
  - make clean install of your project dependencies
  - make sure your projects has (only) the necessary dependencies

- Steps to follow for any new project:
  - virtualenv -p *mypath/to/python3.7 my_venv_name*
  - add a file requirements.txt at the root of your project, containing all **versioned** dependencies.
  - activate the venv and install the dependencies:
    - $: source *my_venv_name*/bin/activate
    - $: pip install -r requirements.txt
  - install ipython or any other useful development stuff INSIDE your virtual env
  - whenever your project requires a new dependency, just add it to the requirements.txt file and re-run the pip install command
  - ipython inside a venv:
    - Run the command: *alias ipy="python -c 'import IPython; IPython.terminal.ipapp.launch_new_instance()'"*

- This way, you have:
  - A clean development environment
  - Reproducible setup creation
  - A shareable development environment

ARMIS

# The Python development setup: virtualenv



# The Python development setup: Docker

- Docker: what for?
  - abstracts away the OS-specific things
  - your application runs in a **container** having all (and only) the necessary dependencies to make your app work
  - This container can communicate with the “outside world” (ex: the host) by exposing ports
  - Example of container contents for a Python app:
    - The base OS: Ubuntu? Debian? Alpine? ⇒ depends on your needs. The simpler, the better.
  - Once the docker container / image is built, your application is ready to be deployed almost anywhere.
  - Just think a built container

- Docker is composed of:
  - The  docker daemon / server: will be the one building your image and executing it.
  - The docker command line (CLI): will be used to send instructions to the docker daemon

- DockerHub
  - DockerHub is to docker what GitHub is to git
  - You can export your docker container definition to DockerHub, and use it in other projects.

# The Python development setup: Docker

- Docker (super) QuickStart:
  - Container definition lies in the **Dockerfile**
  - The Dockerfile contains all the necessary Docker commands to build your image:
    - The base image on which to build from
    - Your code
    - The command to run to execute your code
  - Once the Dockerfile is written, you **build** your container with the **docker build command**
  - It produces an **image**, that you can **run** with the **docker run command**

# The Python development setup: Docker

- Things to keep in mind when using Docker:
  - Docker builds by **layers**, and identifies which layers **need to be re-built** when source has changed
    - ⇒ ordering of the steps matters a lot!
  - Docker CLI sends a build context to the docker daemon when building container
    - Use dockerignore file to avoid adding big files / directories to the build context
  - Use Docker volumes when developing (to be seen during TP1)
  - Demo:
    - git clone this repo: https://github.com/ngallot/docker-python-helloworld
    - follow the steps and run your first dockerized python code!

# The Python development setup: Docker-compose

- On top of Docker, to build multi-services apps: Docker-compose
  - Extremely convenient when developing, to recreate real environment
    - Example: multiple services talking to each other
    - Database access without connecting to the production instance
  - Just 1 configuration file: the docker-compose.yaml
    - Describes the list of services
    - Describes volumes, base images, environment variables, build arguments, etc...
    - Handles networking between services: create custom endpoints
  - Command line to start / stop / build all services
  - Demo to be seen during TP1

# EXPOSING PYTHON CODE

# Exposing Python code

- Making python packages:
  - everything lies in the setup.py file: at the root of your package
  - sub-modules: must contain an __init__.py (even empty).
  - Example: a minimal Python package (GitHub repo here)
    - Create a virtual env
    - Install ipython
    - pip install git+https://github.com/ngallot/python-package-helloworld.git

```
Python 3.7.3 (default, Jul 19 2019, 22:23:21)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.9.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: from pyhw.core.models import Hello

In [2]: hello_msbgd = Hello(hello_who='msbgd 2020')

In [3]: print(hello_msbgd)
Hello msbgd 2020
```

ARMIS

# Exposing Python code: REST apis

- REST(ful) apis concepts:
  - Representational State Transfer. Roy Fielding's thesis in 2000: *Architectural Styles and the Design of Network-based Software Architectures*.
  - Basically:
    - code hosted on a remote infrastructure.
    - based on HTTP protocole
    - This code exposes **methods** to process incoming **http requests**.
    - After processing: it sends a **response** back to the client.
  - Multiple response formats:
    - XML
    - CSV
    - **JSON:** we will focus on this one

# Exposing Python code: REST apis

- The JSON format:
  - JavaScript Object Notation
  - Format that describes classes, with typed fields
  - Limited types: only basic types are supported
  - almost like a dictionary in Python
  - Standard way to exchange data between services over the network

# Exposing Python code: REST apis

- The REST operations:
  - **GET**: retrieves a resource. Parameters can be passed to the endpoint:
    - Directly in the url
    - Through parameters (dictionary in Python)
  - **POST**: creates a new resource (ex. a new entry in database). Will take a json as input
  - **PATCH** / **PUT**: modifies a resource. Takes a json as input.
  - **DELETE**: will delete a resource. Will take the same kind of parameters as GET.

# Exposing Python code: REST apis

- REST apis in Python?
  - Flask
  - Django
  - Pyramid
  - ...
  - **FastApi**:
    - One of (if not the) fastest Python framework
    - Supports async requests
    - Extremely easy and fast to develop REST apis
    - Typed requests, based on the pydantic library.
    - Handles requests validation automatically (through usage of pydantic). Custom validation if needed.
    - Automatically raises nice errors with detailed messages.
    - Included Swagger documentation, inferred from your code and usage of type hints.

# TP 1: LET'S BUILD A REST API

# TP1: Chuck Norris Facts API

Let’s build a REST API in Python!

→ git for git@github.com:ngallot/chuck-norris-facts-api.git

ARMIS


ARMIS
DÉVELOPPER LES VENTES EN MAGASIN