eleven Hackathon

Python and GitHub best practices

To the attention of the Data Sciences
& Business Analytics master students
February 8th, 2021

CentraleSupélec ESSEC BUSINESS SCHOOL eleven strategy consultants London • Paris • Madrid

AGENDA

I  **Python best practices**

- Clean code
- Clean code with PEP8
- Documentation

II  Coding environment and folder template

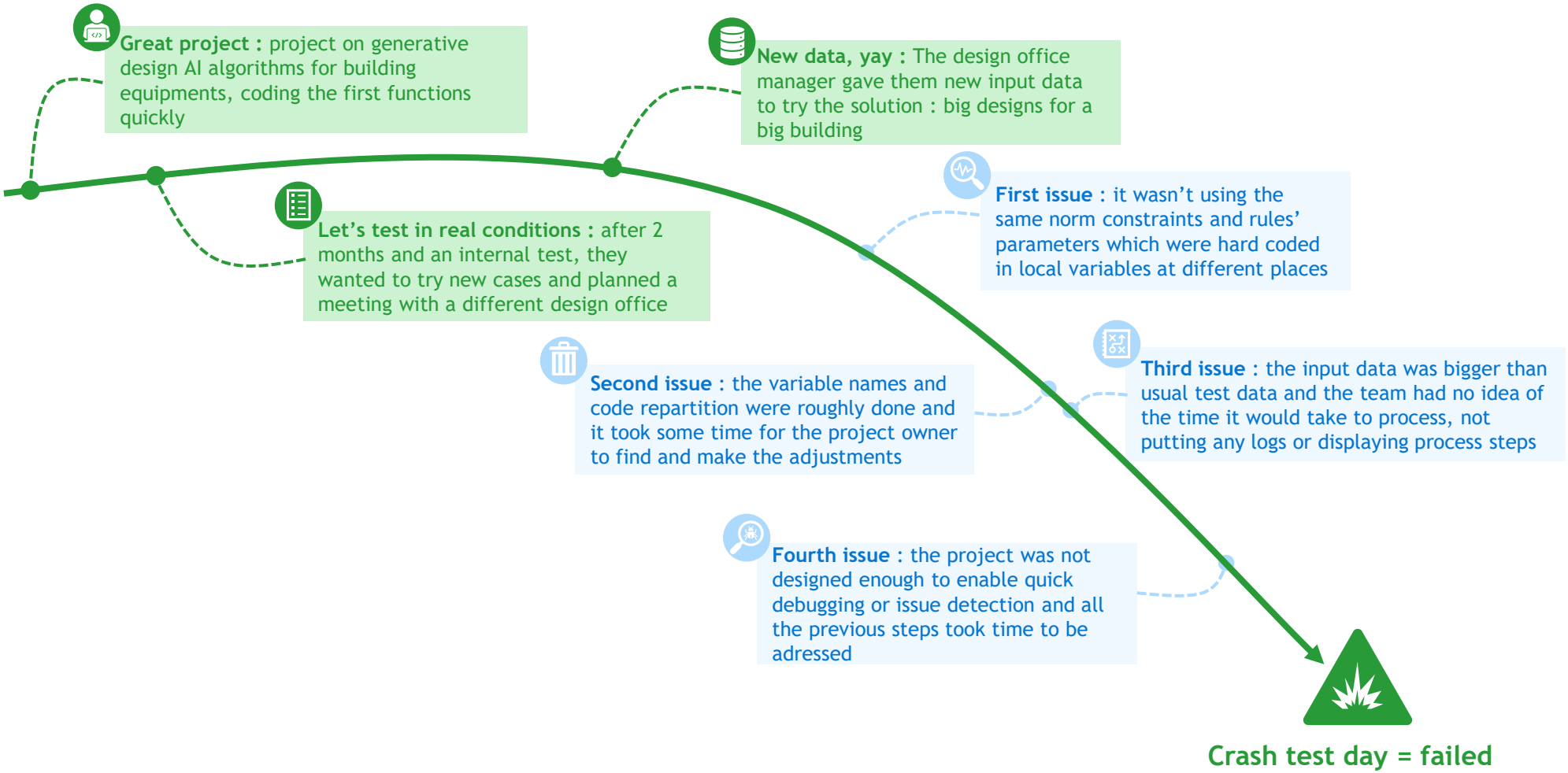
III  GitHub best practice

IV  Appendices

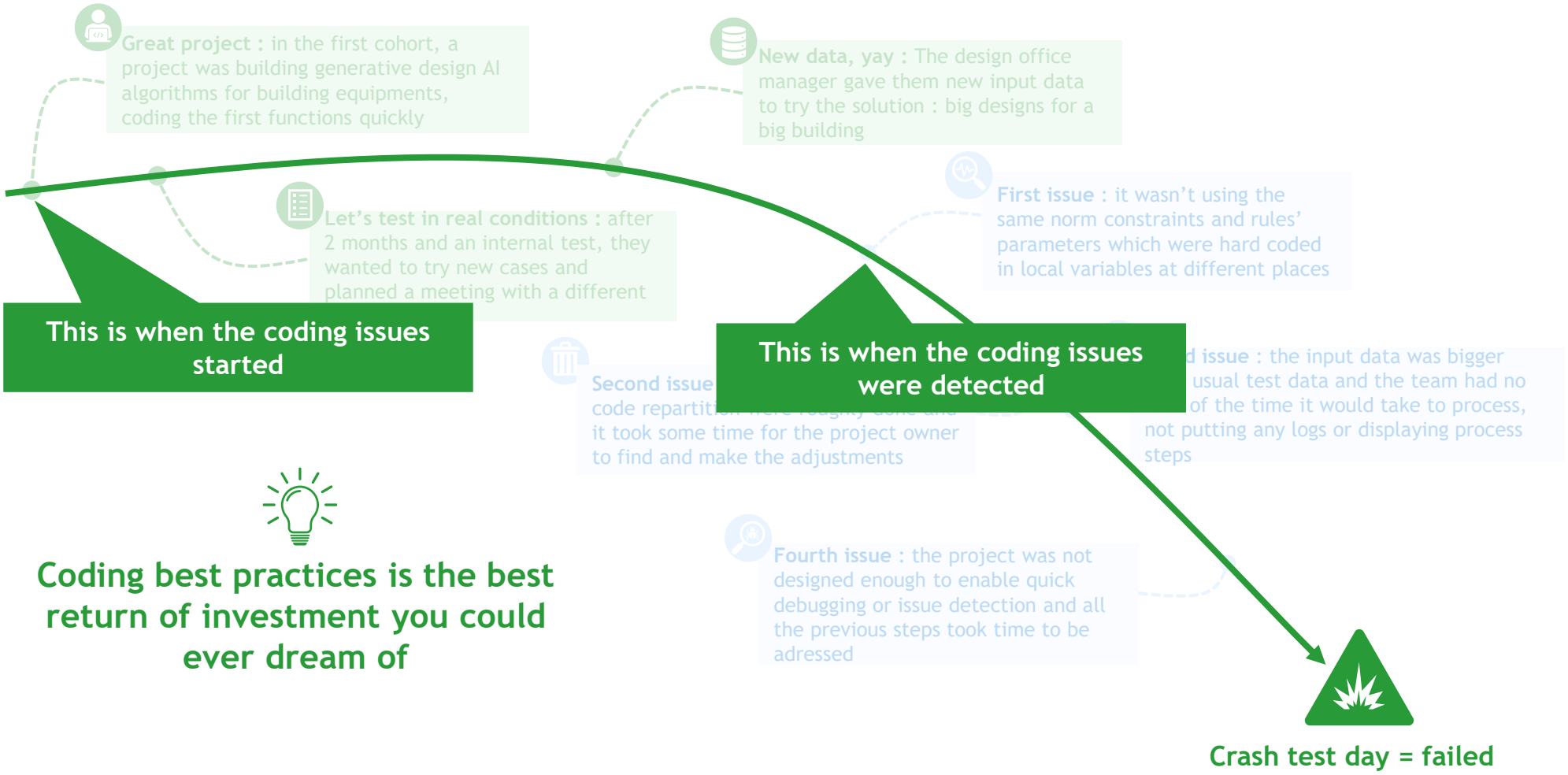
Why is it important to write clean code?

“Code is read much more often than it is written.”
Guido von Rossum, inventor of Python

Why is it important to write clean code?



Why is it important to write clean code?



Coding best practices is the best return of investment you could ever dream of

AGENDA

I  **Python best practices**

- Clean code
- Clean code with PEP8
- Documentation

II  **Coding environment and folder template**

III  **GitHub best practice**

IV  **Appendices**

PEP8 is a set of conventions providing naming guidelines for Python

Type	Convention	Examples
Function	• Use lowercase words, separate words by underscores • Start with a verb when possible	• function, write_name, delete_date, …
Variable	• Use lowercase words • Separate words by underscores	• first_name, last_name, month, …
Class	• Camel case: start each word with a capital letter, no separation of words	• MyClass, Player, …
Method	• Use lowercase words • Separate words by underscores	• class_method, set_player, get_age, …
Constant	• Use uppercase words • Separate words by underscores	• CONSTANT, INITIAL_DATE, …
Module	• Use short lowercase words • Separate words by underscores	• module.py, feaure_engineering.py, …
Package	• Use short lowercase words • Do not separate words by underscores	• mypackage, …

Other Guidelines to name variables

- Use descriptive names to make it clear what the object represents and use pronounceable names
- Use English language
- Avoid starting a name with Python specifics words like list, from, dict, …
- Never use l, O, or I single letter names as these can be mistaken for 1 and 0
- And much more to be found here: Pep8

Illustration

✗

```
def transform_float(values, nb_deciles=10):
    sortedValues = sorted(values)
    aa = [sortedValues[int(k)] for k in np.linspace(0, len(values) - 1, nb_deciles, endpoint=False)]
    xx = np.digitize(values, aa)
    return (xx - xx.min()).astype('str')
```

✓

```
def get_percentile_values(values, number_of_quantiles=10):
    sorted_values = sorted(values)
    quantile_indices = [sorted_values[int(k)] for k in np.linspace(0, len(values) - 1,
                                                                   number_of_quantiles,
                                                                   endpoint=False)]
    quantiles = np.digitize(values, quantile_indices)
    return (quantiles - quantiles.min()).astype('str')
```

PEP8 also provides a set of rules for code layout

Type	Convention	Examples
Line length	• Limit all line to a maximum of 79 characters	`# This sentence is 79 characters long - you shouldn't write longer code lines.`

Indentation	• To keep lines under 79 characters, it can be necessary to break the codes with line continuation
	• Use 4 spaces per indentation level
	• Align the indented block either with the opening delimiter (1) or use a hanging incident (2) where every line is indented except the first one
	• Use '\' to separate an expression into two lines
		1
		`def function(arg_one, arg_two,`
		`arg_three, arg_four):`
		`return arg_one`
		
		2
		`def function(`
		`arg_one, arg_two,`
		`arg_three, arg_four):`
		`return arg_one`
		
		`var = function(`
		`arg_one, arg_two,`
		`arg_three, arg_four)`

Closing brace	• Line-up closing brace with the first non-whitespace character of the previous line or with the first character of the first line	`my_list = [`
		`1, 2, 3,`
		`4, 5, 6,`
		`]`
		or
		`my_list = [`
		`1, 2, 3,`
		`4, 5, 6,`
		`]`

White space	• Add a single whitespace before and after assignment operators (=, +=,…), comparisons (==, !=,…) and Booleans (and, or, …)
	• Use a whitespace after a ',' or ';'
	• Do not use a whitespace after '(' or '[' or '{'
		`x = 5`
		`y = 6`
		
		`print(x, y)`
		
		`def double(x):`
		`return x * 2`
		
		`list[3]`

For this hackathon, we recommend using a tool to check the code syntax and layout called Flake8

Quick introduction to linting and flake8

Lint

- Linting is the process of checking the code syntax and provide instructions on how to clean it
- A linter performs lint tasks on a given code or script
- An often used linter is Flake8

Flake8

- Linter which checks for pep8, pyflakes and circular complexity

Set up

- Flake only works with the python version used to install it
- e.g., python37 -m pip install flake8 will make flake 8 only work with python37

- Helps prevent bugs
- Writing better code
- Saves time in peer reviewer

- Example:

```
python3 -m flake8 path/to/scripts --statistics
```

Python version used to install flake8

Path to a folder containing many scripts or to a .py script

Get statistics on the detected errors

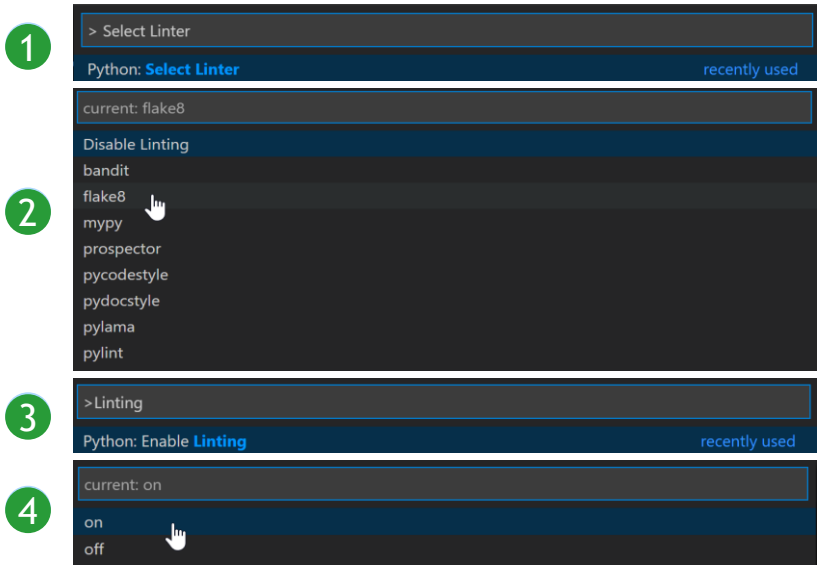
Output example

```
20    E122 continuation line missing indentation or outdented
1     E124 closing bracket does not match visual indentation
3     E125 continuation line with same indent as next logical line
```

Tutorial : Once installed, Flake8 can be directly used on VSC

Setup on VSC

- Make sure Flake8 is installed
- Open the command palette with CTRL + SHIFT + P
- Open the category “Python: Select Linter” and choose Flake8
- Enable linting



Illustration

```
34    for tweet in tweepy.Cursor(api.search,q="#airfrance",
```

twitter_scraping.py 2 of 69 problems

```
missing whitespace after ',' flake8(E231)
```

✖ 1 line is responsible for 10 linting problems !!

```
64    df = pd.read_csv('airfrance.csv', names = ['Date','Account','Text','Retweeted','Favorited','Locat
```

twitter_scraping.py 4 of 69 problems

```
unexpected spaces around keyword / parameter equals flake8(E251)
```

twitter_scraping.py 6 of 69 problems

```
missing whitespace after ',' flake8(E231)
```

twitter_scraping.py 10 of 69 problems

```
line too long (128 > 79 characters) flake8(E501)
```

✔ The code is now PEP8 compliant

```
df = pd.read_csv('airfrance.csv', names=['Date', 'Account', 'Text',
                                         'Retweeted', 'Favorited', 'Location',
                                         'Timezone', 'Coordinates'
                                         ]
                 )
```

Tutorial: A package called black can help you rewrite your code to pep8 format

Use black to rewrite your code

- On your terminal, run the following command to install the unittest package for Python

```
python -m pip install black
```

- You can then run the package on the file you want to rewrite in the command line

```
black code.py
```

- By default, it tolerates lines of 88 characters – which is accepted nowadays. You can change to respect PEP8 guidelines by adding the following option

```
black --line-length=79 code.py
```

Illustration

```
$ black --line-length=79 twitter_scraping.py
reformatted twitter_scraping.py
All done! ✨ 🍰 ✨
1 file reformatted.
```

```
173   # Create and generate a word cloud image:
174   wordcloud = WordCloud(max_font_size=40, background_color="white").generate(wc_airfrance)
twitter_scraping.py 13 of 40 problems
line too long (88 > 79 characters) flake8(E501)
```

```
# Create and generate a word cloud image:
wordcloud = WordCloud(max_font_size=40, background_color="white").generate(
    wc_airfrance
)
```

AGENDA

I Python best practices

- Clean code
- Clean code with PEP8
- Documentation

II Coding environment and folder template

III GitHub best practice

IV Appendices

Code documentation is essential to understand code you wrote in the past or to share your code with others

Code documentation

- Start a comment with "# "
- Explain WHY decisions are taken and not HOW
- Include information in the variable as much as possible
- DRY: Don't Repeat Yourself
- Use codetags (see PEP350) for tasks like TODO, TODOC, …

```
# A list of cities to keep
list = ['Paris', 'London']
```

```
cities_to_keep = ['Paris', 'London']
```

Potential risks and inefficiencies

- Over-commenting code
- Lack of comments
- Update the code but not the comments
- Misleading/controversy comment
- Obvious comments
- Comments just because you feel obliged
- Funny comments

- Clean code is a good substitute to many comments
- Comments should be mostly used to explain main ideas or specific details

Functions and classes must be well-documented using docstring

Three main conventions to document classes and functions

Google Docstrings

```
"""Gets and prints the spreadsheet's header columns

Args:
    file_loc (str): The file location of the spreadsheet
    print_cols (bool): A flag used to print the columns to the console
        (default is False)

Returns:
    list: a list of strings representing the header columns
"""
```

Numpy/SciPy Docstrings

```
"""Gets and prints the spreadsheet's header columns

Parameters
----------
file_loc : str
    The file location of the spreadsheet
print_cols : bool, optional
    A flag used to print the columns to the console (default is False)

Returns
-------
list
    a list of strings representing the header columns
"""
```

reStructured Text

```
"""Gets and prints the spreadsheet's header columns

:param file_loc: The file location of the spreadsheet
:type file_loc: str
:param print_cols: A flag used to print the columns to the console
    (default is False)
:type print_cols: bool
:returns: a list of strings representing the header columns
:rtype: list
"""
```

Recommended docstring

Type hinting

- You can specify the type of the inputs and the outputs of functions on Python, which will raise and error if the function is run with wrong types inputs

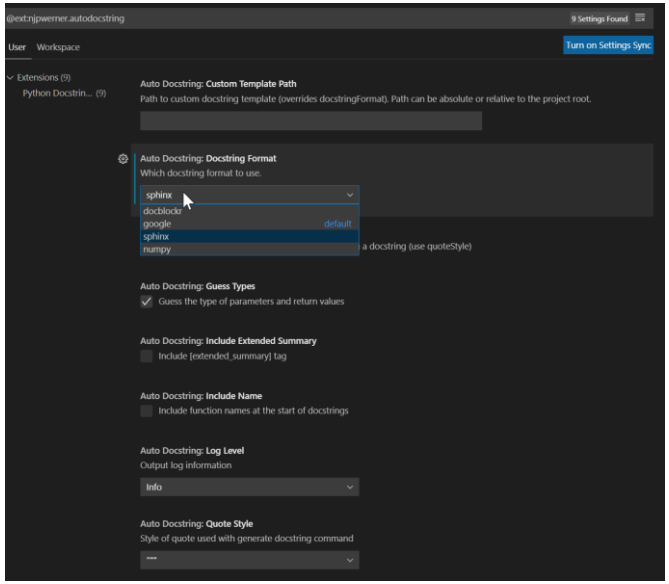
```
def compute_bmi(weight: int, size: int)
-> float:
```

```
239    compute_bmi(3, 'okay')
⊗ twitter_scraping.py 17 of 39 problems
Change this argument; Function "compute_bmi" expects a different type
```

Tutorial: On VSC there is an automatic docstring generation extension providing the different templates mentioned before

Setup on VSC

- On VSC extensions platform, look for "Python Docstring Generator" by Niels Werner and install it
- In the extension settings you can choose the docstring format by default



Illustration

The documentation is automatically generated with type """ + ENTER

```python
def compute_bmi(size: int, weight: int) -> float:
    """[summary]

    Parameters
    ----------
    size : int
        [description]
    weight : int
        [description]

    Returns
    -------
    float
        [description]
    """
    bmi = weight / (size**2)

    return bmi
```

AGENDA

I  **Python best practices**

- Clean code
- Clean code with PEP8
- Documentation

II  **Coding environment and folder template**

III  **GitHub best practice**

IV  **Appendices**

When working on a project, it is necessary to create an environment with all the necessary dependencies to run the project

Use venv to create your virtual environment

- On your terminal, run the following command to install the create a virtual environment at the root of your project

```
python3 -m venv my_env_name
```

- You can specify the exact version of Python you woud like to use with the following command line

```
py -3.7 -m venv my_env_name
```

- You can then activate your environment and deactivate with the following commands

```
• source my_env_name/Scripts/activate
• deactivate
```

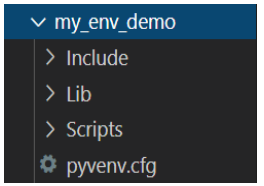
Illustration

- Creation and activation of a virtual environment

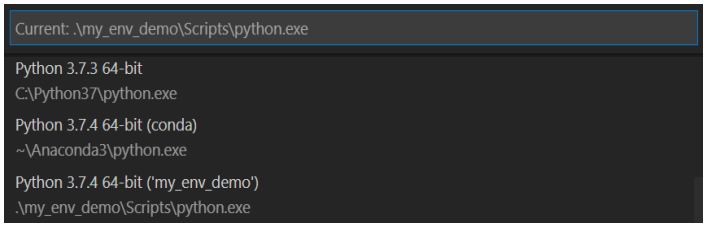
```
$ py -3.7 -m venv my_env_demo
```

```
$ source my_env_demo/Scripts/activate
(my_env_demo)
```

- A folder is automatically created at the root of the project



- The environment created can be found on VSC



When working on a project, it is necessary to create an environment with all the necessary dependencies to run the project

Package management with venv

- Once your virtual environment is activated, you can install the necessary packages using pip with the following command

```
pip install my_package
```

- At any moment you can check the list of packages used in the project with the following commands

```
pip freeze
pip list
```

- You can then activate your environment and deactivate with the following commands

```
pip freeze > requirements.txt
```

Illustration

- Installation of a new package in your venv

```
$ pip install numpy
Collecting numpy
  Using cached https://files.pythonhosted.org/packages/
Installing collected packages: numpy
Successfully installed numpy-1.19.4
```

- Creation of a requirements.txt file with pip freeze

```
$ pip freeze > requirements.txt
```

requirements.txt

```
1  numpy==1.19.4
2
```

- requirements.txt is updated everytime you use pip freeze

```
$ pip install ipykernel
$ pip freeze > requirements.txt
```

requirements.txt

```
1  backcall==0.2.0
2  colorama==0.4.4
3  decorator==4.4.2
4  ipykernel==5.3.4
5  ipython==7.19.0
6  ipython-genutils==0.2.0
7  jedi==0.17.2
8  jupyter-client==6.1.7
9  jupyter-core==4.6.3
10 numpy==1.19.4
11 parso==0.7.1
```

When you start a collaborative project, some files should be present at the root of the project

README

- Describe your project, how to install the dependencies and how your folders are organised

Requirements.txt

- Place at the root of the repository
- Specifies the dependencies required to contribute to the project
- Avoid version conflict

```
# This file is auto-generated from environment.yml, do not modify.
# See that file for comments about the need/usage of each dependency.

numpy>=1.16.5
python-dateutil>=2.7.3
pytz
asv
cython>=0.29.21
black==20.8b1
cpplint
flake8
flake8-comprehensions>=3.1.0
isort>=5.2.1
```

It is possible to specify the minimum version of the package or the exact version

.gitignore

- You might want to keep some files locally – for instance private credentials – and not push them to the remote branches. You can do so by adding elements in the .gitignore
- Some well-made templates are available online (gitignore.io)

```
.#*
*\#*\#
[#]*#
*~
*$
*.bak
*flymake*
*.iml
*.kdev4
*.log
*.swp
*.pdb
.project
.pydevproject
.settings
.idea
.vagrant
.noseids
.ipynb_checkpoints
```

AGENDA

I  **Python best practices**

- Clean code
- Clean code with PEP8
- Documentation

II  **Coding environment and folder template**

III  **GitHub best practice**

IV  **Appendices**

Several guidelines have to be respected to work efficiently on GitHub

Code version control system

- The project's code is in a GitHub repository shared by all team members
- New features are continuously added

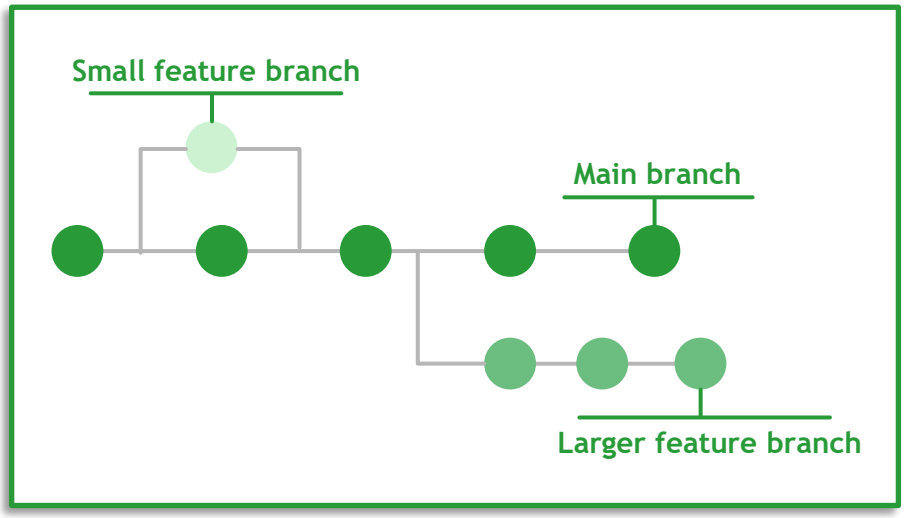


Illustration of a GitHub project evolution

GitHub best practices

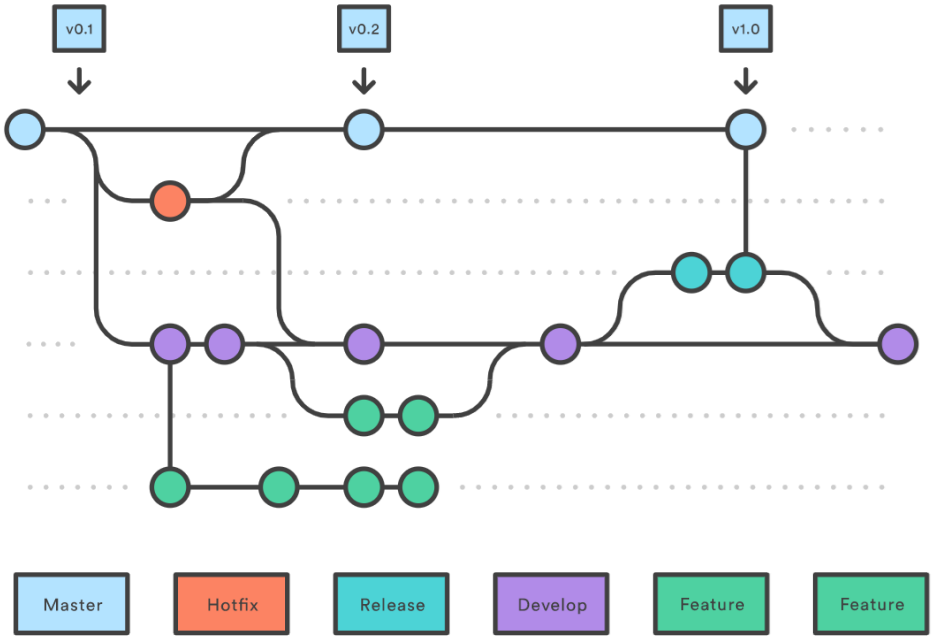
- Use a clear GitHub workflow (details in next slide)
- Restrict access to main branch (it can not be modified without the approval of at least one person)
- Have a .gitignore to avoid pushing files that should remain local (credentials, large datasets,…)
- Make sure your branch is up to date with remote branches to avoid conflicts when merging
- Commit regularly and with a clear commit message, usually starting with a verb.
- Commit when you finish a sub-task and/or at least once a day
- Push your commits at least once a day
- Delete feature branches after their merge into develop

How to work collaboratively with GitHub: GitFlow workflow

Branches type

Main	• Official release history • Usually protected branch • Commits are usually tagged with a number
Develop	• Integration branch for features • It will contain the complete history of the project
Feature	• Each new feature has its own branch • Features branch are based on develop • Branch name should be of the format feature/my_feature_name • When a feature is complete, it gets merged back into the develop branch and the feature branch is deleted
Release	• Prepare a clean release while other teams continue to develop new features for the next release
Hotfix	• Quickly patch production release • Base on the master branch

Illustration



AGENDA

I Python best practices

- Clean code
- Clean code with PEP8
- Documentation

II Coding environment and folder template

III GitHub best practice

IV Appendices

References

- PEP8: https://realpython.com/python-pep8/
- PEP257 for docstring: https://www.python.org/dev/peps/pep-0257/
- Python Code Quality: Tools & Best Practices : https://realpython.com/python-code-quality/
- Visual studio automatic docstring generation: https://marketplace.visualstudio.com/items?itemName=njpwerner.autodocstring
- Configuration autodocstring: https://stackoverflow.com/questions/51716465/python-visual-studio-code-autodocstring-configuration
- Sphinx tutorial: https://www.youtube.com/watch?v=b4iFyrLQQh4
- Different Git workflows: https://www.atlassian.com/git/tutorials/comparing-workflows
- README Cheatsheet: https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet
- .gitignore generator: https://www.toptal.com/developers/gitignore