Good practices on Python

(or coding can be fun and less difficult)

About me

Claudia Ramírez
PhD | Data Scientist

- Physics degree
  - Code design
  - Fortran
  - PA on Fortran for physics
- PhD on Computational Chemistry
  - Computational Simulations x2
  - High Performance Computing x2
  - BASH
  - Git
  - Developer in a project with +100 contributors and +8.000 cites
- Data Scientist
  - SQL
  - PHP, HTML
  - D3
  - Python (Pandas, Numpy, Matplotlib, Scipy, TensorFlow, Pytorch, Seaborn, Streamlit, Pyplot,...)
  - Statistical Analysis
  - Forecasting
  - Stochastic Matching
  - Reinforcement Learning
  - Generative modeling
  - Natural Language Processing
  - Teacher at the university for "Introduction to Python"
- Pos doc

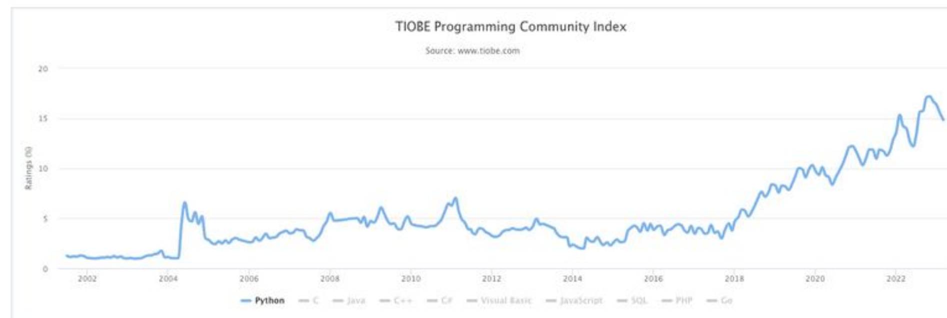
datacamp WE'RE HIRING

CONTENTS

12 Top Data Science Programming Languages in 2023

- Python
- R
- SQL
- Java
- Julia
- Scala
- #C/C++
- JavaScript
- Swift
- Go
- MATLAB

Ranked first in several programming languages popularity indices, including the **TIOBE** Index and the **PYPL** Index, the popularity of Python has boomed in recent years and it remains the most popular programming language. Python is an open-source, general-purpose programming language with broad applicability not only in the data science industry, but also in other domains, like web development and video game development.



Source: **TIOBE Index**

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

How to start coding something?

Think the steps

Use libraries

Code

```
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
```

```
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
```

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Think the steps

Use your brain

Organize your ideas: think on computer steps, sizes, dimensions, inputs/outputs and everything you may need. **Don’t start coding without knowing what to code.**

1. Initialize variables
    a. Read input* variables
    b. Define random variables
2. Run main code* (usually using a for or while)
    a. Do some calculation
    b. Update variables (and maybe store them)
    c. Calculate metric(s)
3. Store results*
4. Analyze results* (tables, plots, etc)
5. Make README (you will thank yourself)

*create folders!

Something general that could help (or not)

Years coding	Average Lines Per Day
1-5	100
5-10	80
10-15	60
15-20	40
20+	20

According to medium

Use libraries

Save time

Every basic (and even not that basic) thing is already coded in an optimal way.
Use their knowledge, please.

1.

Ask to your favorite search engine

Be as detailed as possible, you want to find the closest solution to you problem but on “mainstream”

2.

Read the documentation

(and then some others)

Look at the examples, run the notebooks: **see how easy it is to customize**

Find someone to explain it

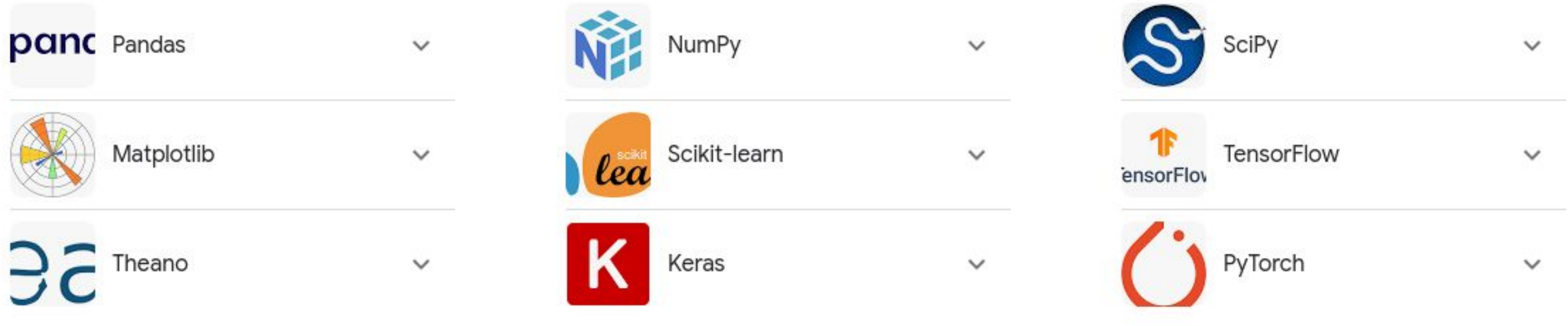
On YouTube:

- DataEngineerOne
- QuantumBlack
- ...find your flavor

To read:

- Towards Data Science
- Medium
- ..

Common libraries



https://www.linkedin.com/pulse/top-10-python-libraries-data-science-2023-akshay-gangshettiwar

https://www.datacamp.com/blog/top-python-libraries-for-data-science

https://www.dataquest.io/blog/15-python-libraries-for-data-science/

https://www.simplilearn.com/top-python-libraries-for-data-science-article

3.

Take your time

This may seem as a “waste of time”, but it’s not: **you will earn time on the long run**

Code

Where the fun begins

Here are some tips that could help you make a more “pythonic” code (aka: a better code)

Use a venv

Avoid problems

Separate your project dependencies with your system dependencies. Allow reproducibility in time.

Use structure

Avoid headaches

Separate your I/O files and your pipelines into folders, name your files in a useful way.

For large projects I recommend **kedro**.

If it's a small project, it may not be worth it

(or will it be?)

project
data
input
output
notebooks
src
prep
main
post
README.md

git

Make functions

Save energy

Put as many steps as you can into functions with descriptive names

1.

Add readability

Everyone who will deal with your code will thank you **(even your self after holidays)**

Initial code

38 BAD lines

```
def gp sampler(
        data: pd.DataFrame , db: pd.DataFrame ,
        features: List[ str], targets: List[ str], features_nn: List[ str], targets_prediction:
List[str],
        nn_calculator: NearestNeighbors , max_it: int, max_trials: int, n_jobs: int = 1,
):
    doi = features + targets + [ 'Tg_Tmax', 'quality', 'nn_distance' ]
    res = pd.DataFrame( columns=doi)
    res[data.columns] = data
    x train = data[features].values
    y train = data[targets].values
    yield {f'[0-{res.shape[ 0]}]': res}

    model = FOM( features=features, targets=targets)
    for n in range(max it):
        print(f'Sampling number  {n}')
        model.fit( x train=x train, y target=y train)
        new_x, fx = model.optimize( max_trials=max_trials, n_jobs=n_jobs)

        distance, index = nn calculator.kneighbors( X=new_x, n_neighbors=1)
        df = db.loc[index[ 0]]
        new y = df[targets].values
        new extra vars = df[[ 'Tg Tmax', 'quality']].values
        x_nn = df[features].values

        np.append(x train , new_x, axis=0), np.append(y_train , new_y, axis=0)
        idx = len(res.index)
        res.loc[idx , features] = new x[ 0]
        res.loc[idx , features nn] = x nn[ 0]
        res.loc[idx , targets] = new y[ 0]
        res.loc[idx , targets prediction] = fx
        res.loc[idx , ['Tg Tmax', 'quality']] = new_extra_vars[ 0]
        res.loc[idx , 'nn_distance' ] = distance[ 0]

        if (res.shape[ 0] % 100) == 0:
            idx = res.shape[ 0]
            ini idx = (idx -  100)
            yield {f'[{ini_idx}-{idx}]': res[ini_idx: idx]}
```

Final code

64 BETTER lines
(including functions)

```
def gp sampler (
        data: pd.DataFrame , db: pd.DataFrame ,
        features: List[ str], targets: List[ str], features_nn: List[ str], targets_prediction:
List[str],
        nn_calculator: NearestNeighbors , max_it: int, max_trials: int, n_jobs: int = 1,
):
    res, x train, y train = initialize variables(data , features, targets)
    yield results(res , initialize=True)

    model = FOM( features=features , targets=targets)
    for n in range(max it):
        print(f'Sampling number  {n}')
        model.fit( x train=x train, y target=y train)
        new x , fx = model.optimize( max trials=max trials , n jobs=n_jobs)
        new y , new extra vars , x nn, nn dist = get values from db(
            x=new_x, db=db, knn=nn_calculator , features=features, targets=targets
        )
        x train , y train = extend_train_set(new_x , new_y, x_train, y_train)
        store values(
            features , features nn , fx, new_extra_vars , new_x, new_y, nn_dist, res, targets,
            targets_prediction , x_nn
        )
        if (res.shape[0] % 100) == 0:
            yield results(res)
```

Don’t scroll

If you have to scroll to read a complete function,
you need to add more functions

2.

Recycle

(also when coding)

If you have code that is used in another place **(even once)**

Initial code

26 BAD lines

```
data = {
    'lhs': {
        'name': 'LHS', 'color': '#E7B10A',
        'inliers': lhs[lhs.quality =='ignition'],
        'outliers': lhs[lhs.quality !='ignition'],
        'df': lhs
    },
    'gp_ori': {
        'name': 'GP_original', 'color': '#898121',
        'inliers': gp_ori[(gp_ori.quality =='ignition') | (gp_ori.quality =='interest')],
        'outliers': gp_ori[(gp_ori.quality !='ignition') & (gp_ori.quality !='interest')],
        'df': gp_ori
    },
    'gp_new': {
        'name': 'GP_fixed', 'color': '#00425A',
        'inliers': gp_treated[(gp_treated.quality =='ignition') | (gp_treated.quality =='interest')],
        'outliers': gp_treated[(gp_treated.quality !='ignition') & (gp_treated.quality !='interest')],
        'df': gp_treated
    },
    'gp_nn': {
        'name': 'GP_nn', 'color': '#8B1874',
        'inliers': new_treated[(new_treated.quality =='ignition') | (new_treated.quality =='interest')],
        'outliers': new_treated[(new_treated.quality !='ignition') & (new_treated.quality !='interest')],
        'df': new_treated
    }
}
```

Final code

13 BETTER lines

```
[...]
        data = {
            'lhs': create_dict(df=lhs, name='LHS', color='#E7B10A'),
            'gp_ori': create_dict(df=gp_ori, name='GP_original', color='#898121'),
            'gp_new': create_dict(df=gp_treated, name='GP_fixed', color='#00425A'),
            'gp_nn': create_dict(df=new_treated, name='GP_nn', color='#8B1874')
        }
[...]

def create_dict(df: pd.DataFrame, name: str, color: str) -> Dict:
    return {
            'name': name, 'color': color,
            'inliers': df[(df.quality == 'ignition') | (df.quality == 'interest')],
            'outliers': df[(df.quality != 'ignition') & (df.quality != 'interest')],
            'df': df
        }
```

Don’t copy-paste

When you copy-paste code and latter need to modify it, you will have to do so in all the places (and you will have to remember all those places). Instead **use functions**

Create good variable names

Save lives

Nobody charges you for writing more letters, and the limitation in characters doesn’t exist in python. There is no excuse: **Use descriptive variable names.**

Naming matters

Snake case

this_is_snake_case

Use it for variables and functions

Capwords

ThisIsCapWords

Use it for classes

Uppercase

UPPERCASE or also

THIS_IS_UPPER_CASE

Use it for constants

Kebab case

this-is-kebab-case

Use it on git*

You can know more things if you read: https://peps.python.org/pep-0008/

Readability counts

- x, y, i, j, *_clean_* are not good variable name
  (unless you are programming something extremely flexible that you could use for many different problems, but probably is not the case)
- train, test, res, idx, n_row, n_col, batch_size are ok

  (this are names that could be "broad" but their are still clear)

Tips

Make your work easier

There are things that make coding easier, use those things.

Integrated Development Environment

PyCharm is **awesome**

Documenting, reducing overhead and exploiting your IDE, all in one hint

Tool that suggest you coded functions (if you have a clear name), and it's free for students!

BREAK

Let's code!

Have some fun

On a terminal:
> cd <working-dir>
> source seminar-venv/bin/activate
> jupyter lab

Thanks!

Any questions?

You can find me at
Room A45
clramirez@laas.fr