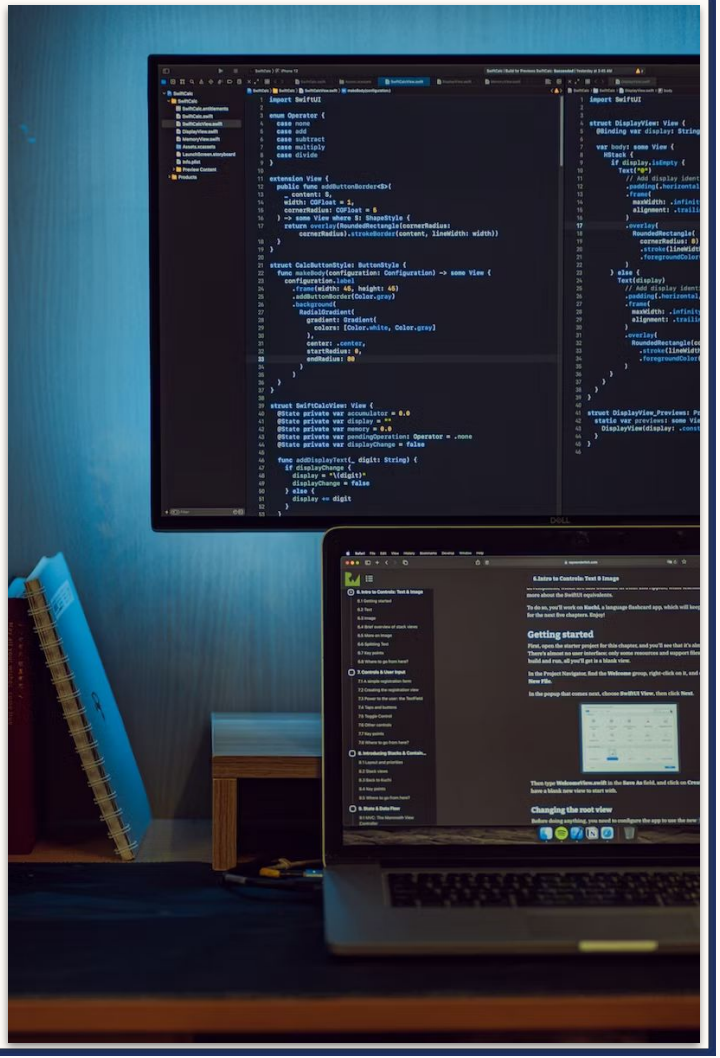
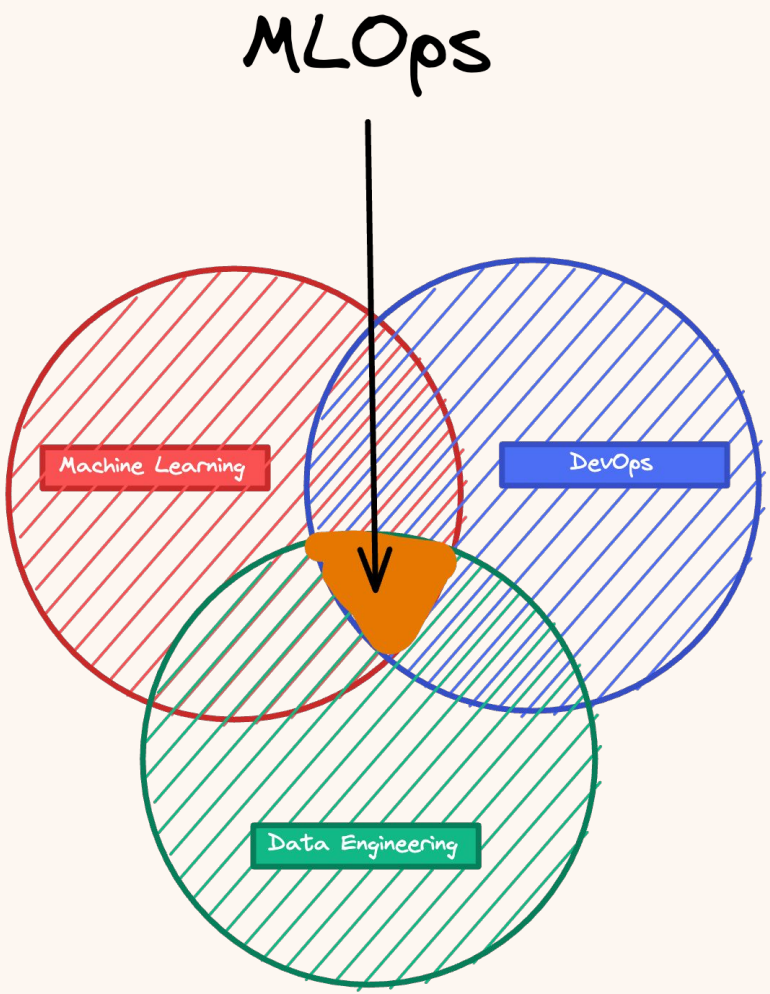
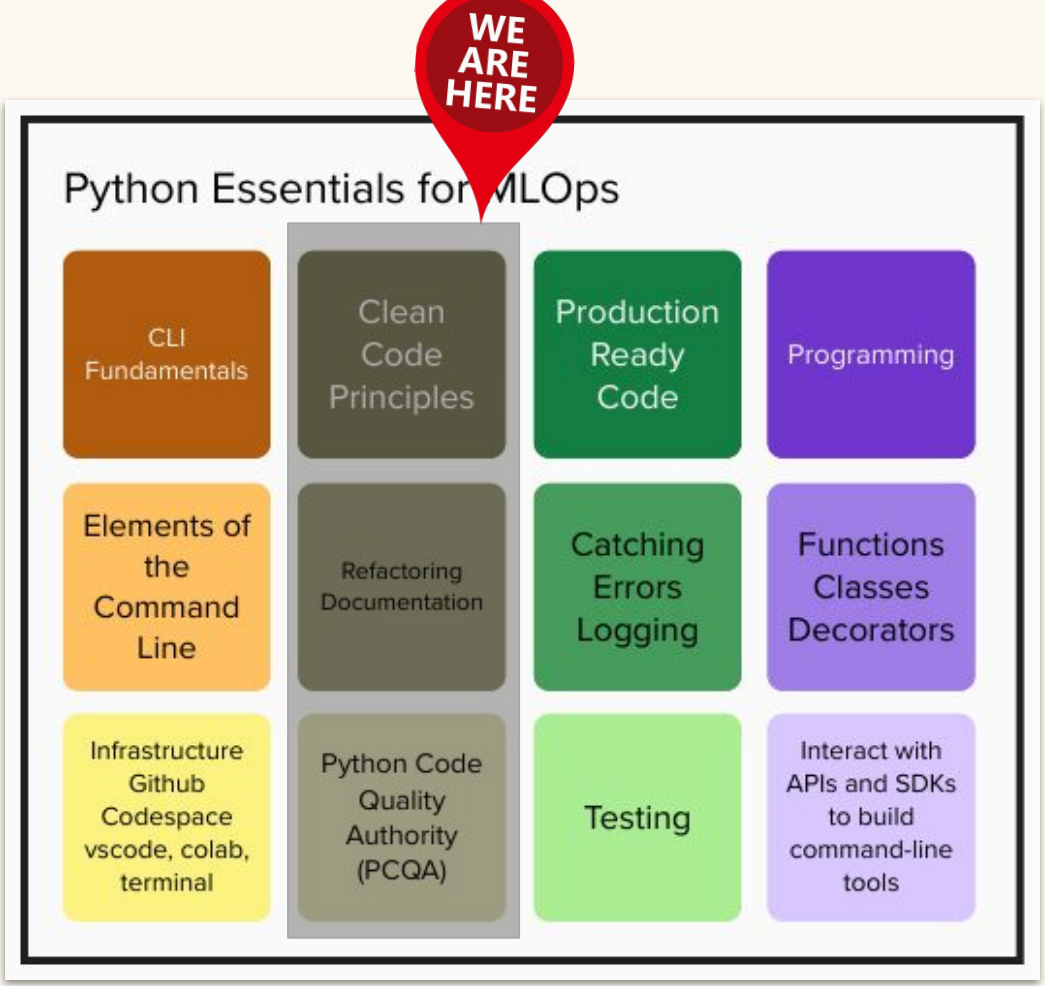
Clean Code Principles

For Data Science & Machine Learning

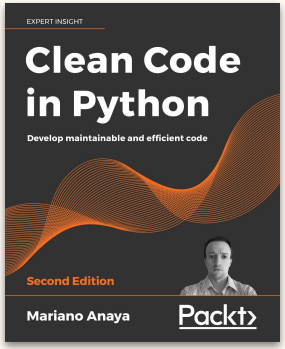
DCA0305
ivanovitch.silva@ufrn.br

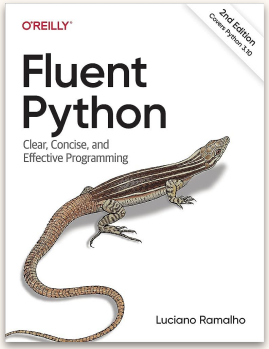



MLOps
Machine Learning
DevOps
Data Engineering


WE ARE HERE
Python Essentials for MLOps
CLI Fundamentals
Clean Code Principles
Production Ready Code
Programming
Elements of the Command Line
Refactoring Documentation
Catching Errors Logging
Functions Classes Decorators
Infrastructure Github Codespace vscode, colab, terminal
Python Code Quality Authority (PCQA)
Testing
Interact with APIs and SDKs to build command-line tools

Agenda





1. Coding best practices
2. Writing clean and modular code
3. Refactoring
4. Efficient code and documentation
5. Following PEP8

"Readability Counts" The zen of the Python
"The code is read much more often than it is written." Guido Van Rossum

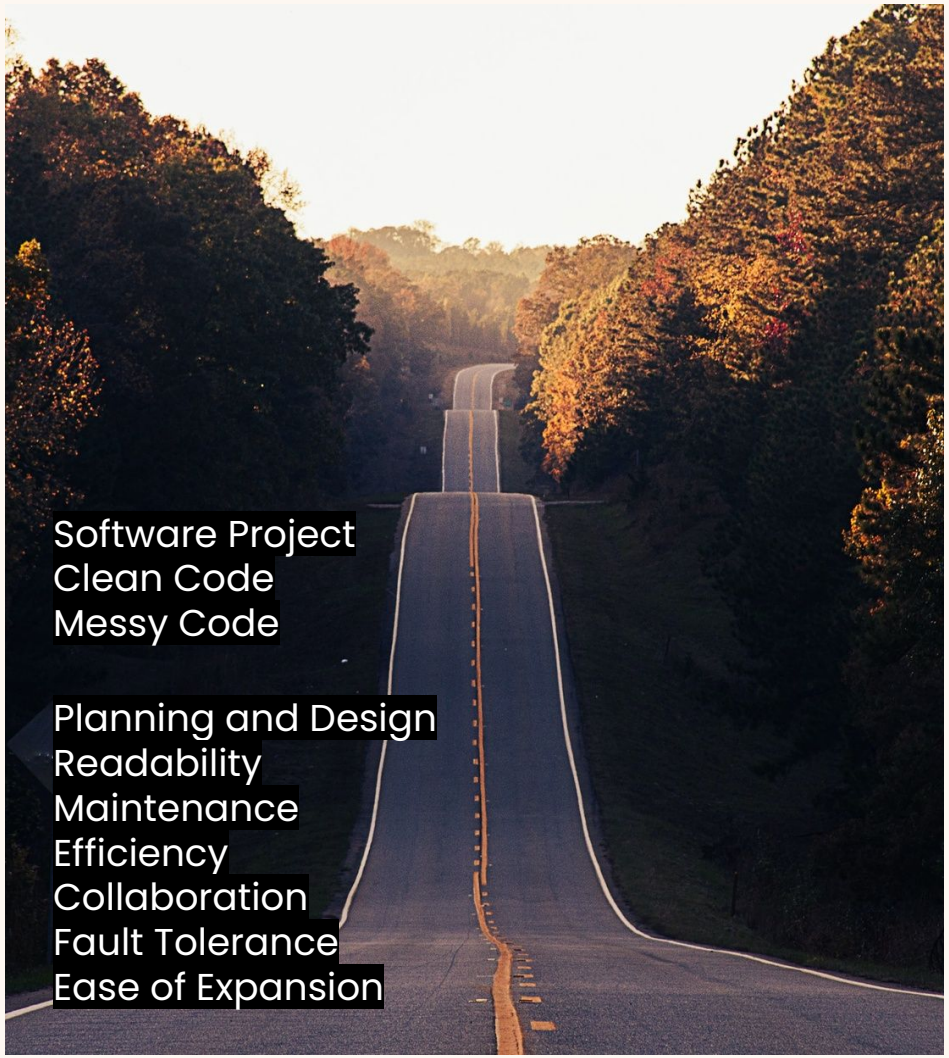
This is not about clean code (...)

Add a five-second delay.

E.g., **Thread.Sleep(5000)**;

Behind every common action in your software, after 3 months, when your users are going crazy, remove the delays and tell them that you've worked tirelessly to improve performance, and suddenly everyone will love you.

Is clean code only about formatting and structuring the code?

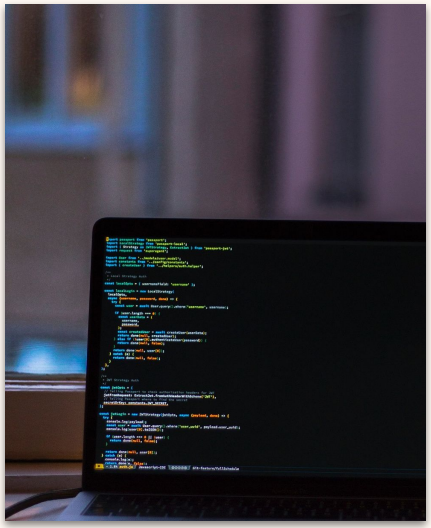

Software Project
Clean Code
Messy Code
Planning and Design
Readability
Maintenance
Efficiency
Collaboration
Fault Tolerance
Ease of Expansion

Some Exceptions

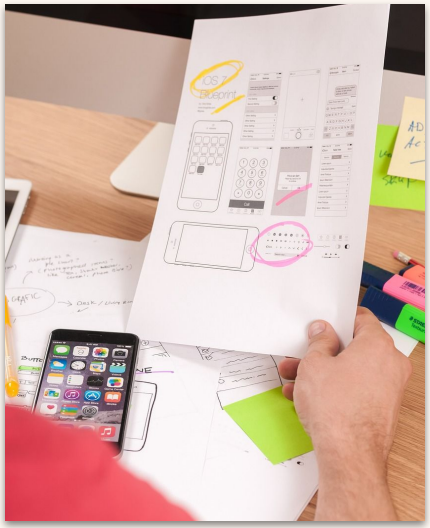
a) Hackathons, b) *If you're writing a simple script for a one-off task, c) Code Competitions, d) When developing a proof of concept, e) When developing a prototype, f) When you're working with a legacy project that will be deprecated, and it's only in maintenance mode for a fixed, short-lived period of time*

Coding Best Practices

Makes you a better engineer and preparing you for writing production-level code



Writing clean and module code using informative naming conventions



Refactoring and optimizing your code

Transform in-line comments into docstrings and subsequently into comprehensive project documentation

Automate putting our best practices to use autopep8 and pylint

Contents

- Introduction
- A Foolish Consistency is the Hobgoblin of Little Minds
- Code Lay-out
  - Indentation
  - Tabs or Spaces?
  - Maximum Line Length
  - Should a Line Break Before or After a Binary Operator?
  - Blank Lines
  - Source File Encoding
  - Imports
  - Module Level Dunder Names
- String Quotes
- Whitespace in Expressions and Statements
  - Pet Peeves
  - Other Recommendations
- When to Use Trailing Commas
- Comments
  - Block Comments
  - Inline Comments
  - Documentation Strings
- Naming Conventions
  - Overriding Principle
  - Descriptive: Naming Styles
  - Prescriptive: Naming Conventions
    - Names to Avoid
    - ASCII Compatibility
    - Package and Module Names
    - Class Names

PEP 8 – Style Guide for Python Code

Author: Guido van Rossum <guido at python.org>, Barry Warsaw <barry at python.org>, Nick Coghlan <ncoghlan at gmail.com>

Status: Active
Type: Process
Created: 05-Jul-2001
Post-History: 05-Jul-2001, 01-Aug-2013

▶ **Table of Contents**

Introduction

This document gives coding conventions for the Python code comprising the standard library in the main Python distribution. Please see the companion informational PEP describing style guidelines for the C code in the C implementation of Python.

This document and PEP 257 (Docstring Conventions) were adapted from Guido's original Python Style Guide essay, with some additions from Barry's style guide [2].

This style guide evolves over time as additional conventions are identified and past conventions are rendered obsolete by changes in the language itself.

Many projects have their own coding style guidelines. In the event of any conflicts, such project-specific guides take precedence for that project.

https://peps.python.org/pep-0008

Clean and Modular Codes

Using meaningful names

Type	Naming Convention	Examples
Function	Use a lowercase word or words. Separate words by underscores to improve readability.	`function`, `my_function`
Variable	Use a lowercase single letter, word, or words. Separate words with underscores to improve readability.	`x`, `var`, `my_variable`
Class	Start each word with a capital letter. Do not separate words with underscores. This style is called camel case or pascal case.	`Model`, `MyClass`
Method	Use a lowercase word or words. Separate words with underscores to improve readability.	`class_method`, `method`
Constant	Use an uppercase single letter, word, or words. Separate words with underscores to improve readability.	`CONSTANT`, `MY_CONSTANT`, `MY_LONG_CONSTANT`
Module	Use a short, lowercase word or words. Separate words with underscores to improve readability.	`module.py`, `my_module.py`
Package	Use a short, lowercase word or words. Do not separate words with underscores.	`package`, `mypackage`

Clean and Modular Codes

Using meaningful names

Avoid abbreviations and single letters
Maintain consistency while ensuring clear distic.

```
# wrong
# student test scores
s = [88, 92, 70, 93, 85]

# print mean of test scores
print(sum(s)/len(s))

# curve scores with square root method and store in new list
s1 = [x ** 0.5 * 10 for x in s]

# print mean of curved test scores
print(sum(s1)/len(s1))
```

```
# correct
import math
import numpy as np

# student test scores
test_scores = [88, 92, 70, 93, 85]

# print mean of test scores
print(np.mean(test_scores))

# curve scores with square root method and store in new list
curved_test_scores = [math.sqrt(score) * 10 for score in test_scores]

# print mean of curved test scores
print(np.mean(curved_test_scores))
```

Clean and Modular Codes

Using meaningful names

Be descriptive and indicate type

```
# wrong

ages = [47, 12, 28, 52, 35]
for i, age in enumerate(ages):
    if age < 18:
        minor = True
        ages[i] = "minor"
# some other code
```

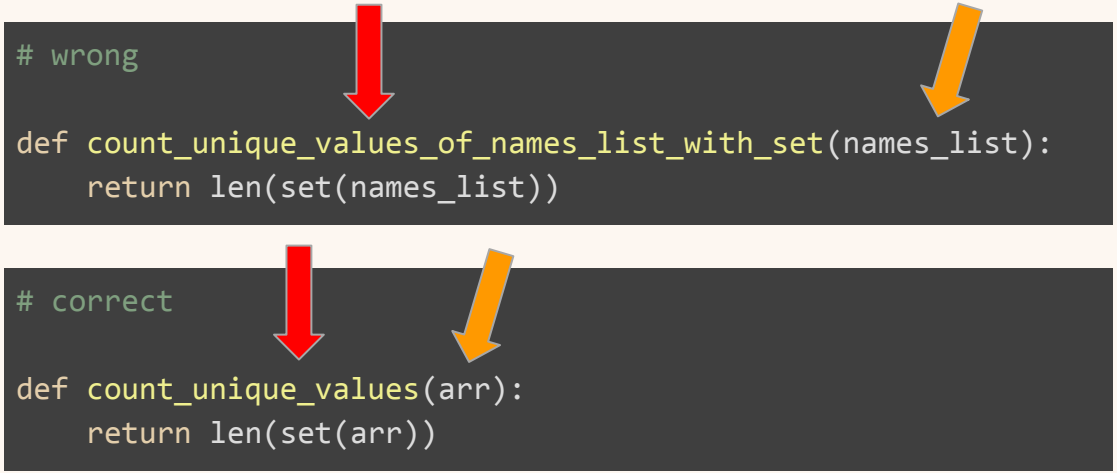
```
# correct

age_list = [47, 12, 28, 52, 35]
for i, age in enumerate(age_list):
    if age < 18:
        is_minor = True
        age_list[i] = "minor"
# some other code
```

Clean and Modular Codes

Using meaningful names

Long names aren't the same as descriptive names



Clean and Modular Codes

Writing Modular Codes

DRY (Don't repeat yourself)
Abstract out logic to improve readability
Function do one thing

```
# wrong

s = [88, 92, 79, 93, 85]
print(sum(s)/len(s))

s1 = []
for x in s:
    s1.append(x+5)

print(sum(s1)/len(s1))

s2 = []
for x in s:
    s2.append(x+10)

print(sum(s2)/len(s2))

s3 = []
for x in s:
    s3.append(x ** 0.5 * 10)

print(sum(s3)/len(s3))
```

Clean and Modular Codes

Writing Modular Codes

DRY (Don't repeat yourself)
Abstract out logic to improve readability
Function do one thing

```
# correct

import math
import numpy as np

def flat_curve(arr, n):
    return [i + n for i in arr]

def square_root_curve(arr):
    return [math.sqrt(i) * 10 for i in arr]

test_scores = s = [88, 92, 79, 93, 85]
curved_5 = flat_curve(test_scores, 5)
curved_10 = flat_curve(test_scores, 10)
curved_sqrt = square_root_curve(test_scores)

for score_list in test_scores, curved_5, curved_10, curved_sqrt:
    print(np.mean(score_list))
```

Writing_Clean_Code.ipynb

Acronyms to live by

- *DRY/OAOO*

  Don't repeat yourself (DRY) and **Once and Only Once** (OAOO). *You should avoid duplication at all costs.*

- YAGNI

  You Ain't Gonna Need It *(YAGNI). You might want to keep in mind very often when writing a solution if you do not want to over-engineer it.*

- KIS

  Keep It Simple *(KIS) relates very much to the previous point.*

- *EAFP/LBYL*

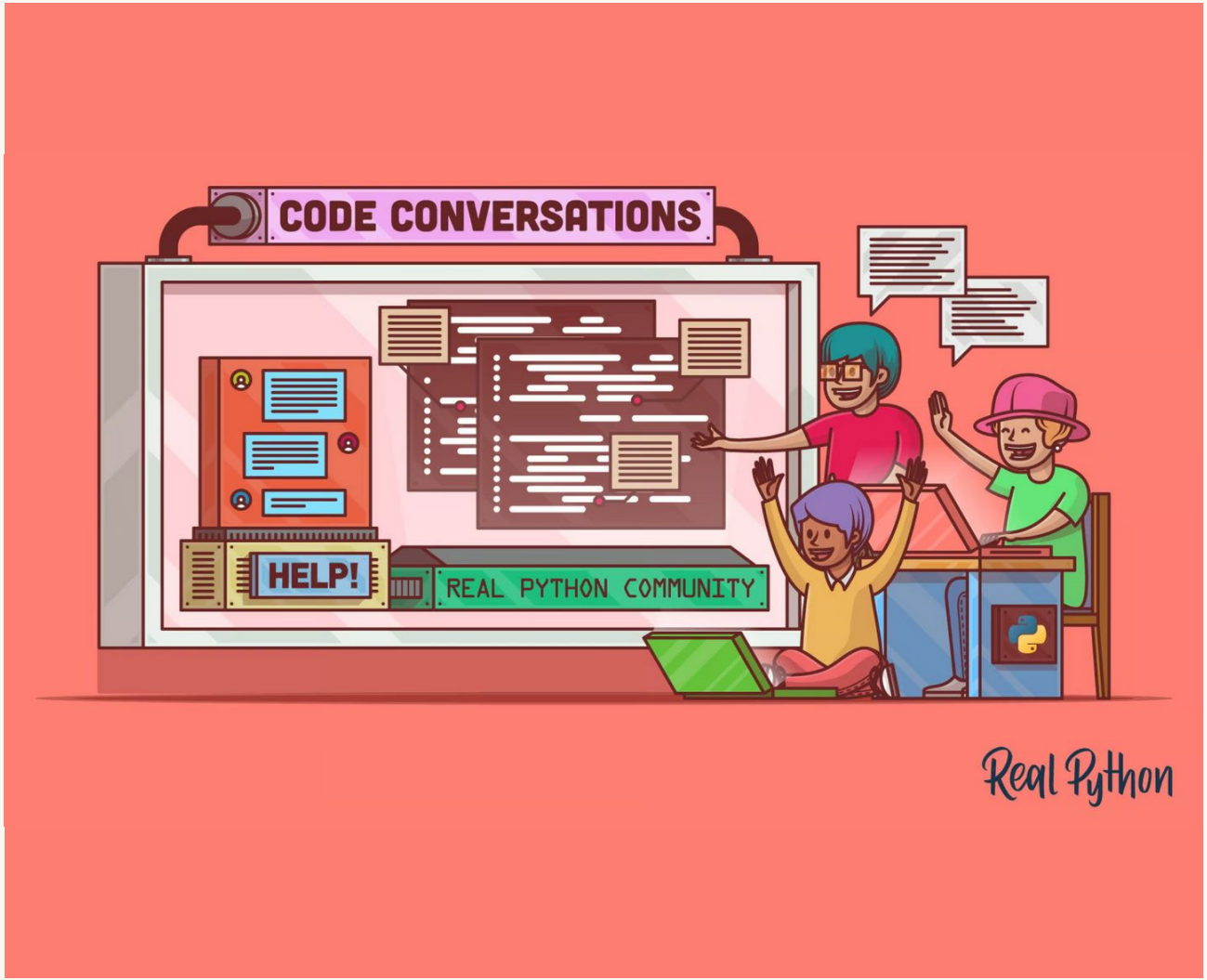
  Easier to Ask Forgiveness than Permission *(EAFP) and* ***Look Before You Leap*** *(LBYL).*

Refactoring Code

It could be difficult to know exactly what functions would best modularize

Restructuring code to improve internal structure without changing external functionality

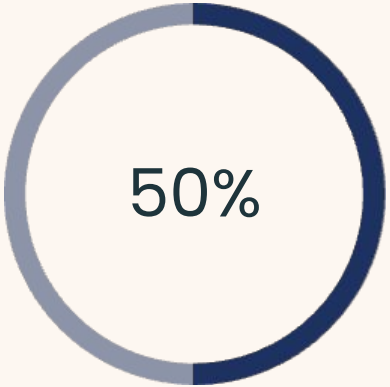
refactoring_code.ipynb



Efficient Code

Reducing runtime vs space in memory

Execute faster



Take up less space in memory/storage

Knowing **how to write code that runs efficiently** is another essential skill in **software development**. The project on which you're working determines which of these is more important to optimize for your company or product. When you're performing lots of different transformations on large amounts of data, this can make orders of magnitudes of difference in performance.

jupyter optimizing_code_common_books_example.ipynb