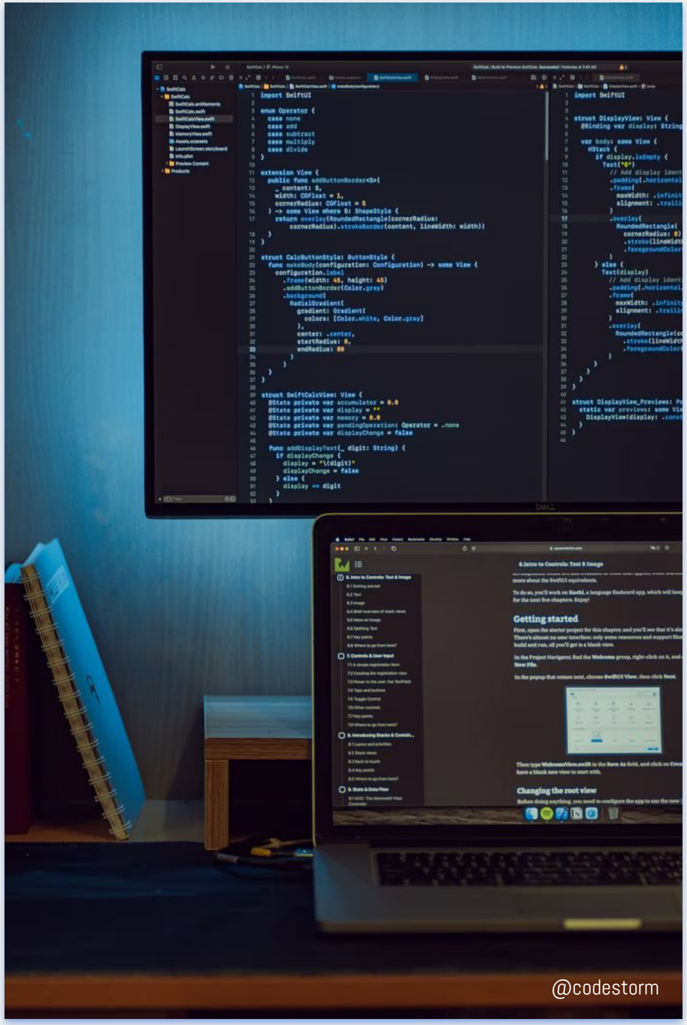


# Clean Code Principles for Data Science & Machine Learning

## Table of Contents

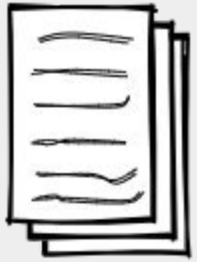
"Readability Counts" The zen of the Python

"The code is read much more often than it is written." Guido Van Rossum

## Coding Best Practices

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

Write clean and module code using informative naming conventions



Recfatoring and optimizing your code

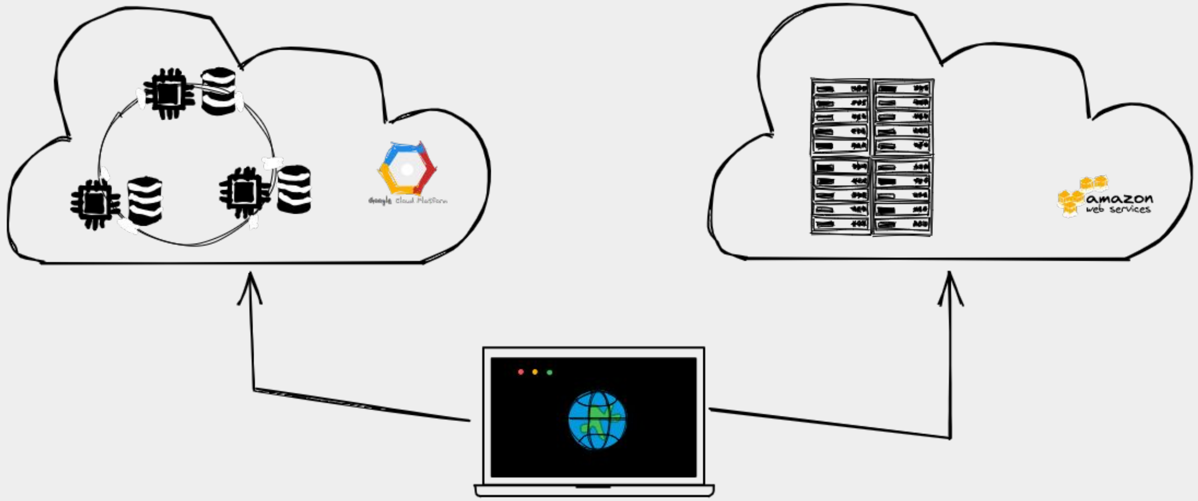
Ducument from in-line comments to docs string to project documentation

Automate putting our best practices to use autopep8 and pylint

## Clean and Modular Code

Writing code in a way that is clean and modular.

- Production Code
- Clean Code
- Modular code
- Module



Production code just means code that is run on production servers. Ideally, this code should meet a number of criteria to ensure reliability and efficiency before become public.

## Clean and Modular Code

Writing code in a way that is clean and modular.

- Production Code
- Clean Code
- Modular code
- Module & Functions

Clean: readable, simple and conside

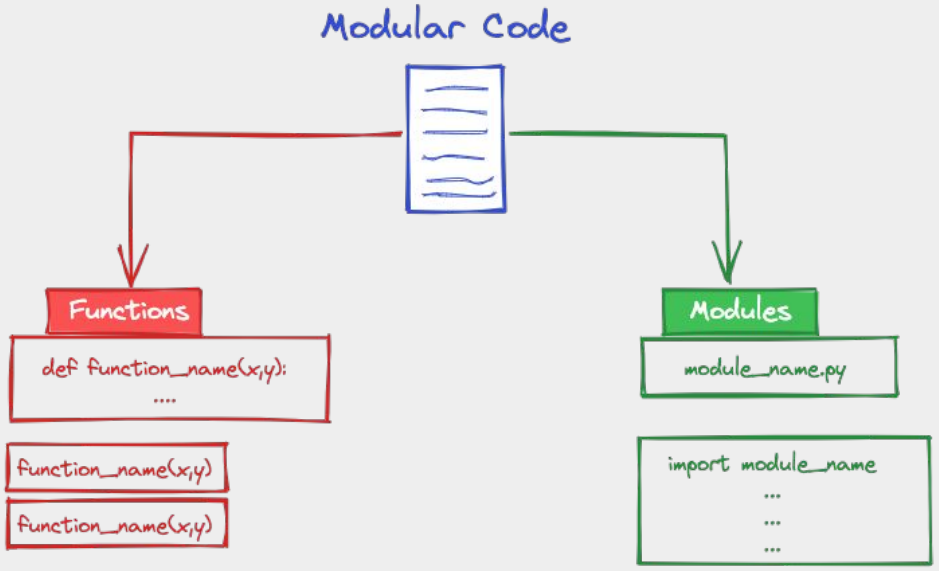
"One could observe that your shirt has been sullied due to the orange color on syour shirt that apperars to be similar to the color of a certain kind of juice."

"It looks like you spilled orange juice on your shirt"

# Clean and Modular Code

Writing code in a way that is clean and modular.

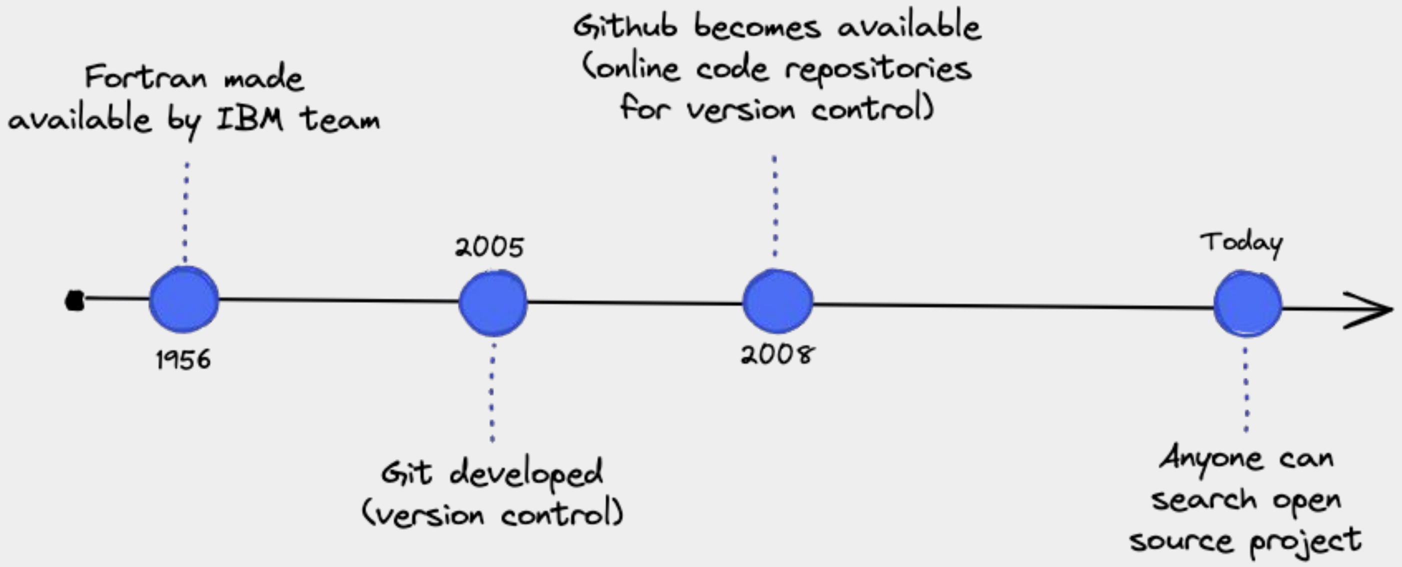
- Production Code
- Clean Code
- Modular code
- Module & Functions



Making your code modular makes it easier:
a) reuse, b) write less code,
c) read and d) collaborate on code.

# Clean and Modular Code

## History of Clean Code


Fortran made available by IBM team
1956
2005
Git developed (version control)
Github becomes available (online code repositories for version control)
2008
Today
Anyone can search open source project

# Clean and Modular Code

## Writing Clean Code: use meaningful names vs use whitespace properly

```python
# bady
# 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))
```

```python
# better
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 Code

## Writing Clean Code: use meaningful names vs use whitespace properly

```python
# bad

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

```python
# better

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 Code

### Writing Clean Code: use meaningful names vs use whitespace properly

```python
# bad

def count_unique_values_of_names_list_with_set(names_list):
   return len(set(names_list))
```

```python
# better

def count_unique_values(arr):
   return len(set(arr))
```

# Clean and Modular Code

## Writing Modular Code

- DRY (Don't Repeat Yourself)
- Abstract out logic to improve readability
- Minimize the number of entities (functions, classes, modules, etc.)
- Functions should do one thing
- Arbitrary variable names can be more effective in certain functions
- Try to use fewer than three arguments per function

# Clean and Modular Code

## Writing Modular Code

```python
# bady

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))
```

```python
# little better

import math
import numpy as np

test_scores = [88, 92, 79, 93, 85]
print(np.mean(test_scores))

curved_5 = [score + 5 for score in test_scores]
print(np.mean(curved_5))

curved_10 = [score + 10 for score in test_scores]
print(np.mean(curved_10))

curved_sqrt = [math.sqrt(score) * 10 for score in test_scores]
print(np.mean(curved_sqrt))
```

## Clean and Modular Code

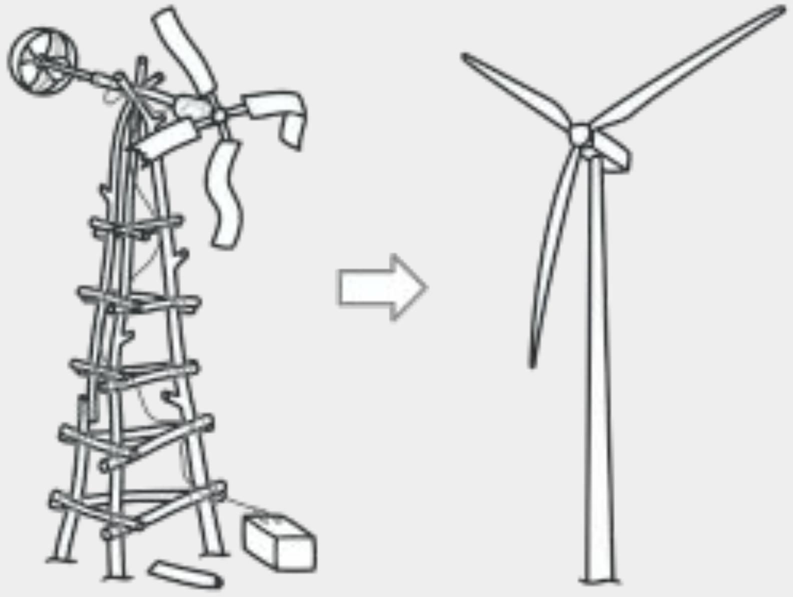
### Writing Modular Code

```python
# bady

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))
```

```python
# better

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 = [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))
```

# Refactoring Code

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



Restructuring code to improve internal structure without changing external functionality

Exercise 01!!!

## Efficient Code

### Reducing runtime vs space in memory

Knowing how to write code that runs efficiently is another **essential skill in software development**. Optimizing code to be more efficient can mean making it:

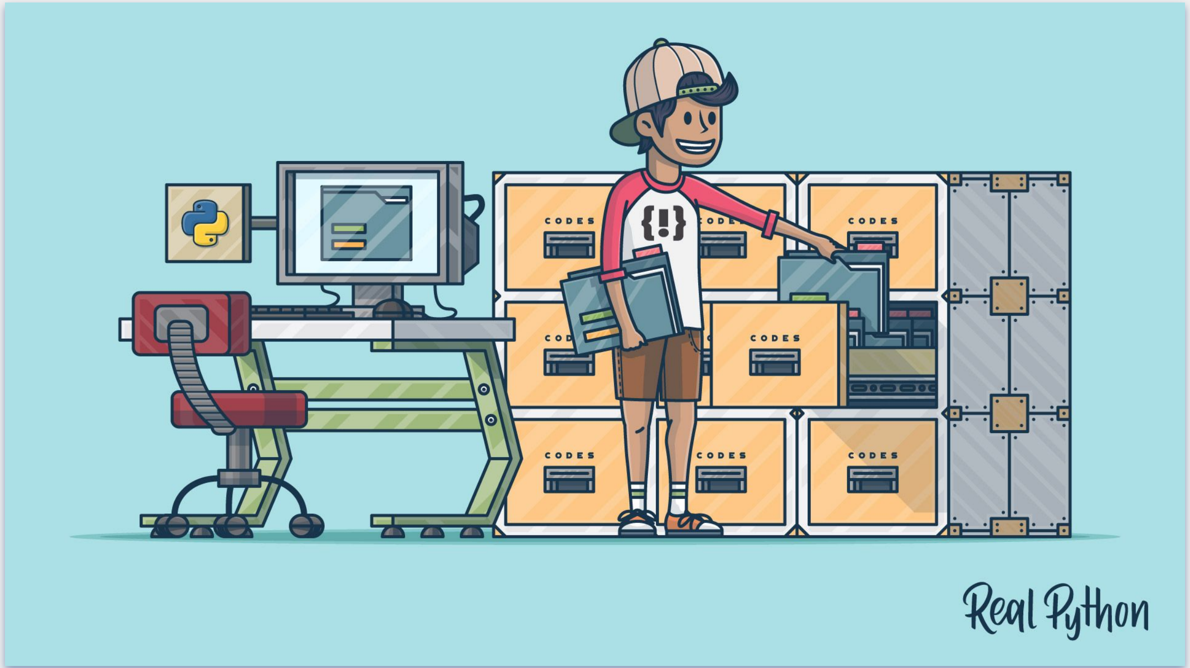
- Execute faster
- Take up less space in memory/storage

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.

Exercises 02 and 03 !!!

## Documentation

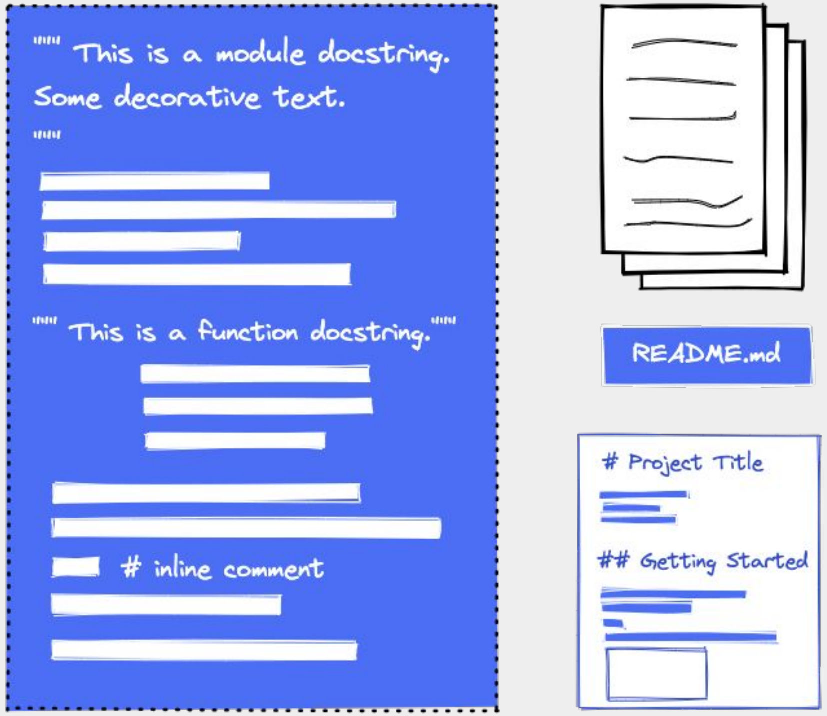
Clarify complex parts, navigate easily, describe use and purpose of components



Types of Documentation

- Line level
- Function or module level
- Project level

## Documentation



Comments are valuable for explaining where code cannot. For example, the history behind why a certain method was implemented a specific way. Sometimes an unconventional or seemingly arbitrary approach may be applied because of some obscure external variable causing side effects. These things are difficult to explain with code.

## Documentation

Docstring Conventions https://www.python.org/dev/peps/pep-0257/

```python
def population_density(population, land_area):
    """Calculate the population density of an area."""
    return population / land_area
```

```python
def population_density(population, land_area):
    """Calculate the population density of an area.

    Args:
    population: int. The population of the area
    land_area: int or float. This function is unit-agnostic, if you pass in values
in terms of square km or square miles the function will return a density in those
units.

    Returns:
    population_density: population/land_area. The population density of a
    particular area.
    """
    return population / land_area
```

# Auto Python Enhancement Proposal (PEP-8)

Key points that you can use to make your code more organized and readable

| | |
|---|---|
| https://github.com/PyCQA/pycodestyle | Working on notebook cells, indicate changes |
| https://github.com/PyCQA/pylint | Working on command-line with python script, show a score |
| https://pypi.org/project/autopep8/ | Working on command-line with python script, automatize the style |
| https://github.com/nbQA-dev/nbQA | Working on notebook and script |

## Lesson Conclusion

In this lesson, you learned 5 key factors to coding best practices:

1. Writing clean and modular code
2. Refactoring code
3. Optimizing code to be more efficient
4. Writing documentation
5. Following PEP8