OTA INSIGHT

Code Cleanup

A Data Scientist's Guide to

✨Sparkling Code✨

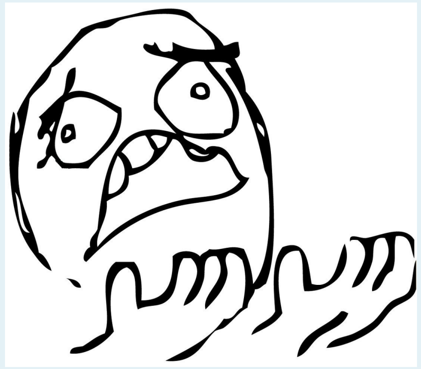
Corrie Bartelheimer
Senior Data Scientist

“Why should I care about clean code?”

– a young data scientist

“Why don’t they care about clean code?”

– an engineer working with data scientists



Lies we tell ourselves

- “I’m only gonna run this query once”
- “This won’t change”
- “No one's gonna need this again”
- “I don’t think we’ll run this analysis again”
- “We can just deploy the notebook to production”

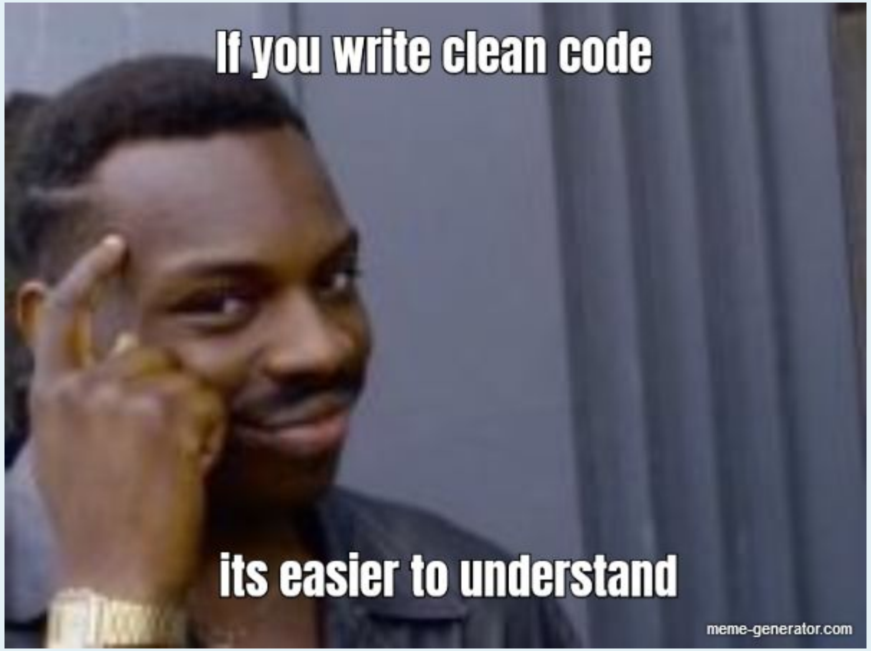
Reality

- “Can you check what happens if ….?”
- “Let me add just this one small thing”
- “Can you have a look at these old notebook from your former coworker?”
- “No!”

We read more code than we write!

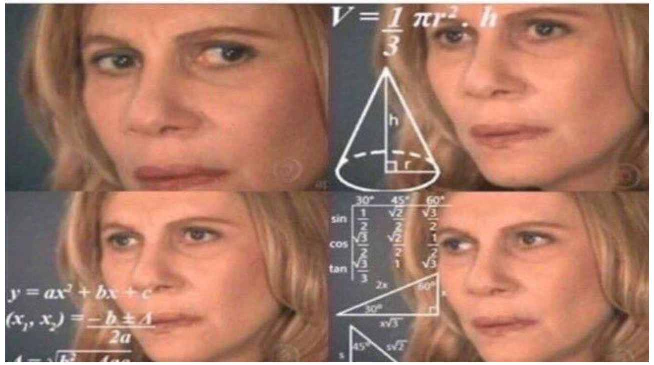
“Dirty” Code Slows Down:

- Takes more time to understand
- Harder to change
- Often hides bugs
- Makes reproducibility difficult



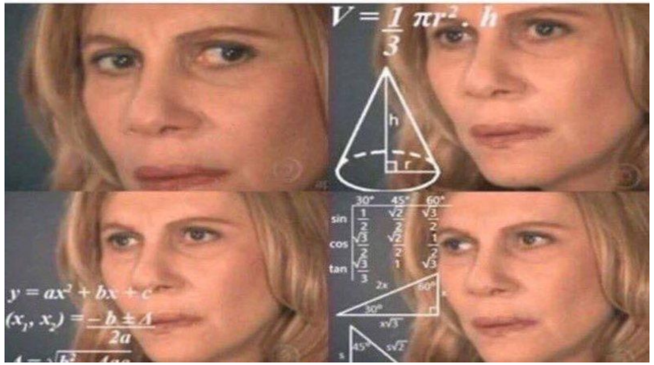
What makes code easy to read?

Meaningful Names



```python
def transform(data):
    for k, v in data.items():
        data[k] = round(v * 0.9, 2)
    return data
```

Meaningful Names



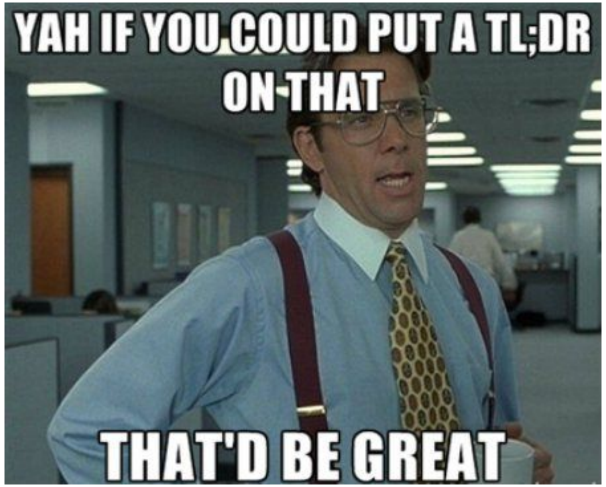
```python
def transform(data):
    for k, v in data.items():
        # calculate new price
        data[k] = round(v * 0.9, 2)
    return data
```

Meaningful Names

- Use descriptive names
  - Longer > shorter
  - Searchable
- Explain code through naming
- Avoid generic names like `transform, item, ..` etc

```python
def discount_prices(room_prices, discount=0.1):
    for room, price in room_prices.items():
        new_price = price * (1 - discount)
        room_prices[room] = round(new_price, 2)
    return room_prices
```

Use Short Functions!



```python
def compute_features(df_map, other_data):

    destination_cols = ["destination_id", "cdestination_name",
                        "latitude","longitude",]
    keep_cols = ["some_id","another_id","hotel_id"]

    # keep only unique hotels
    data = df_map[[*keep_cols, *destination_cols]].drop_duplicates(
        subset=["some_id", "destination_id"]
    )

    # mean of features
    mean_features = (
        df_map.groupby("some_id")[["stars","review_score"]].mean().reset_index()
    )

    # add prices
    df = (
        df_map[["some_id", "other_id"]]
        .dropna(subset=["some_id", "other_id"])
        .drop_duplicates()
    )
    room_price = ...

    min_price = (
        pd.merge(room_price, df, on="some_id", how="inner")
        .groupby(["other_id", "currency"])
        .agg(min_price=("price", np.min))
        .assign(min_price_usd=lambda x: x["min_price"].copy())
        .reset_index()
    )
    # align index
    data = pd.merge(hotel_df, min_price, on="some_id", how="left")

    # add room count
    rooms_count = (
        df_map.groupby("some_id")["other_id"].nunique().rename("rooms_count")
    )
    data = pd.merge(data, rooms_count.reset_index(), on="some_id", how="left")
```

Use Short Functions!

- Structure Long code into shorter functions

Also, **most comments are bad**

- Comments tend to go stale
- Instead of comments, explain the **what** in the function name
- Keep comments to explain the **why**

```
def compute_features(df_map, other_data):

    destination_cols = ["destination_id", "cdestination_name",
                        "latitude","longitude",]
    keep_cols = ["some_id","another_id","hotel_id"]

    # keep only unique hotels
    data = df_map[[*keep_cols, *destination_cols]].drop_duplicates(
        subset=["some_id", "destination_id"]
    )

    # mean of features
    mean_features = (
        df_map.groupby("some_id")[["stars","review_score"]].mean().reset_index()
```

```
def compute_features(df_map, other_data):

    data = get_unique_hotels(df_map)

    mean_features = get_mean_features(df_map)

    data = add_mean_price(data, mean_features)

    data = add_room_count(data, other_data)

    return data
```

```
    rooms_count = (
        df_map.groupby("some_id")["other_id"].nunique().rename("rooms_count")
    )
    data = pd.merge(data, rooms_count.reset_index(), on="some_id", how="left")
```

Avoid Mixing Abstraction Layers

Don’t put low-level operations on the same level as high-level functions

```
bad_ids = [hotel_ + str(id_) for id_ in [123, 42, 888]]

df = df[~df.hotel_name.isin(bad_ids)]

features = compute_features(df)
```

Avoid Mixing Abstraction Layers

- Group operations together that have the same level of abstraction
- Structure code into abstraction hierarchies by using functions
- Same for variable names

```
bad_ids = [hotel_ + str(id_) for id_ in [123, 42, 888]]

df = df[~df.hotel_name.isin(bad_ids)]

features = compute_features(df)
```

Better:

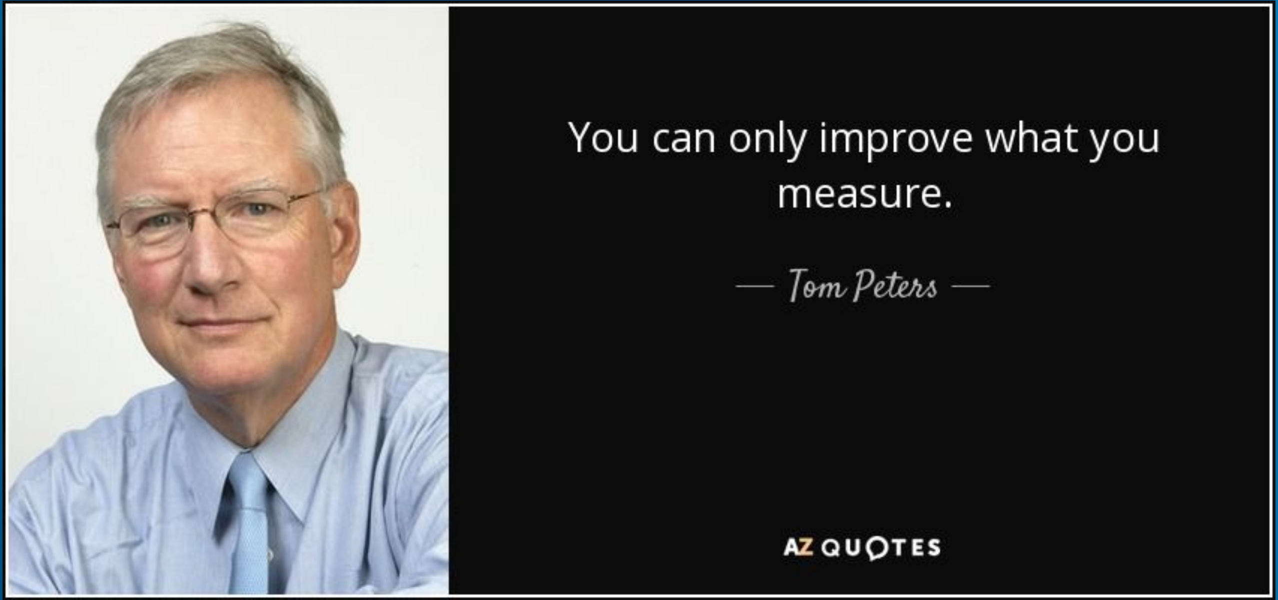
```
df_filtered = filter_out_bad_ids(df)

features = compute_features(df_filtered)
```

Ain’t got no time for this

Measure

Visualize

Prioritize



You can only improve what you measure.

— Tom Peters —

AZ QUOTES

Or someone else…

Measuring Code Complexity

Function Length

- Checks length of a function
- Checks number of input and output parameter

```
def some_long_function(
    first_parameter: int,
    second_parameter: int,
    third_parameter: int,
):
    first_parameter = (
        first_parameter +
        second_parameter +
        third_parameter
    )
    first_parameter = (
        first_parameter -
        second_parameter +
        third_parameter
    )
    first_parameter = (
        first_parameter +
        second_parameter -
        third_parameter
    )
    first_parameter = (
        first_parameter
        second_parameter
        third_parameter
    )

    return first_parameter
```

Cognitive Complexity

- Increments for breaks in the flow
  - loops & conditionals
  - catch, switch statements
  - breaks, continue

- Increments for nested structures

```python
def f(a, b):
    if a:
        for i in range(b):
            if b:
                return 1
```

Abstraction Layer

- Only allow generic variable names in small functions
    - items, var, variables, result, ...
    - The more generic, the smaller the function

```
def foo(variables)
    items = []
    for var in variables:
      items += [var]
    return items
```

Expression Complexity

- Measures complexity of expressions
- Rule of Thumb: if expression goes over multiple lines, consider splitting it

```
if (df.sold_out.any() and
    df[~(df.is_new_hotel & ~df.is_covid)
        | df.price_value.isna()].any()):
        do_something(df)
```

All of these available as flake8 plugin:

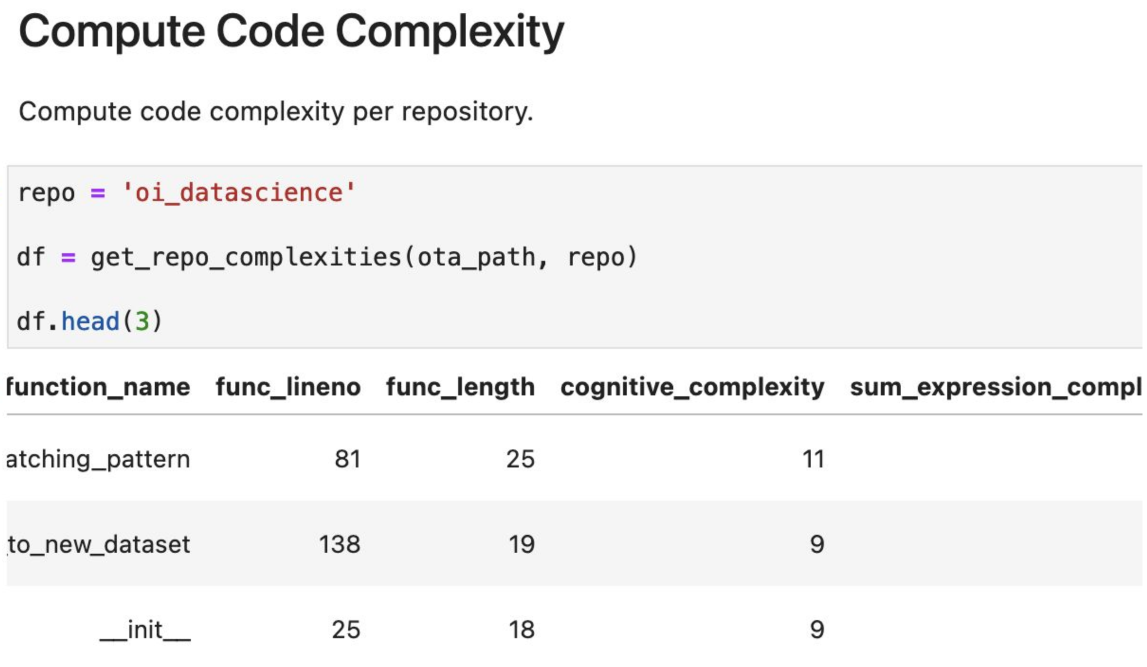
- flake8-cognitive-complexity
- flake8-adjustable-complexity
- flake8-expression-complexity
- flake8-functions

flake8

Now on to collect some data...

Compute Complexity Heuristics

- Used the flake8 implementation to compute complexity heuristics
- Depending on needs, decide how often to run
- Jupyter notebook might be good enough 😜
- Store in your favorite database



Compute Code Complexity

Compute code complexity per repository.

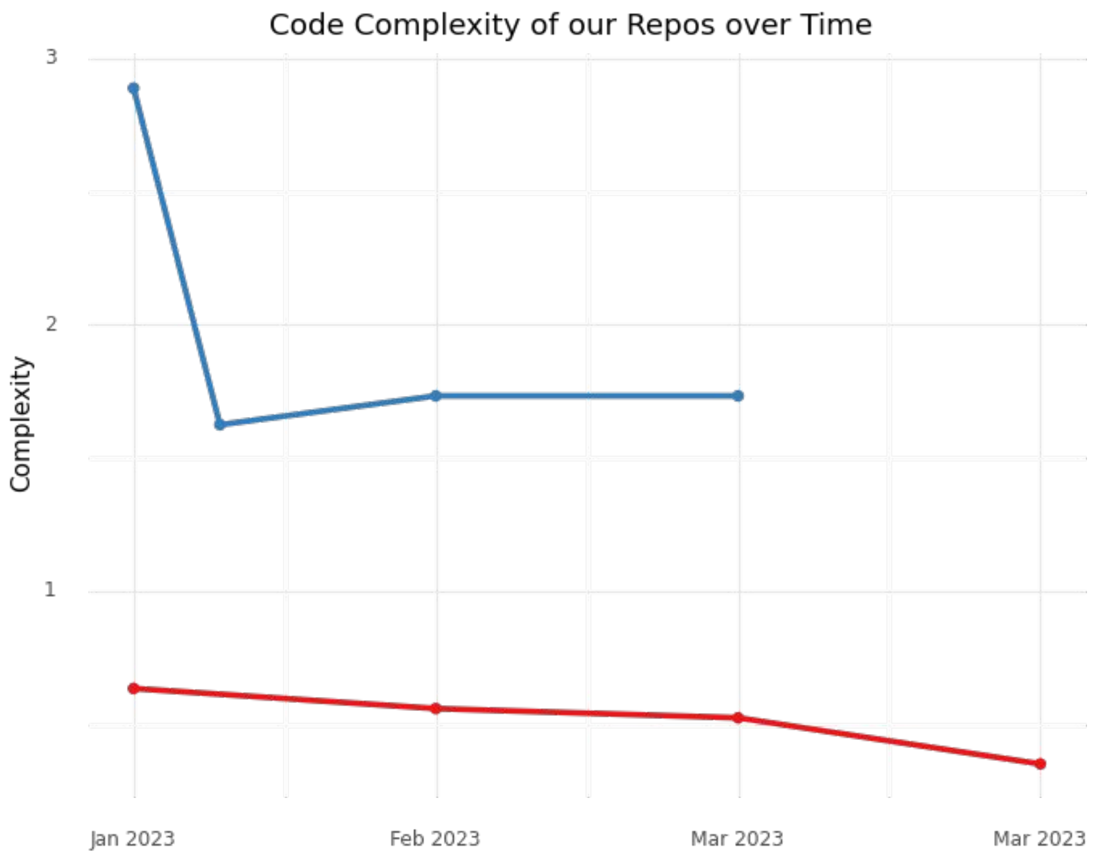
```
repo = 'oi_datascience'

df = get_repo_complexities(ota_path, repo)

df.head(3)
```

function_name	func_lineno	func_length	cognitive_complexity	sum_expression_compl
atching_pattern	81	25	11	
to_new_dataset	138	19	9	
__init__	25	18	9	

jupyter ❤️

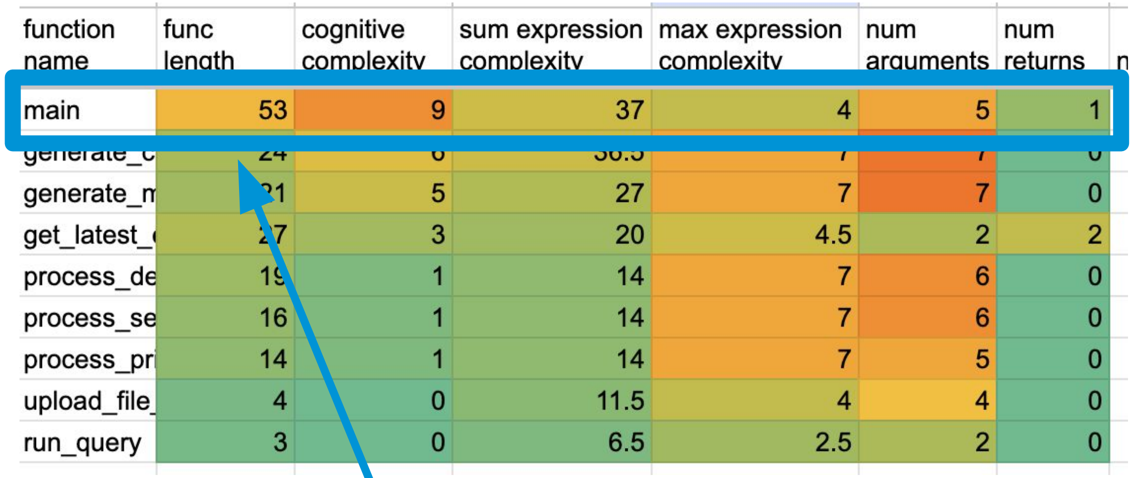
Visualize & Prioritize

- How are your repositories developing over time?



Visualize & Prioritize

- How are your repositories developing over time?
- Which files/functions should we tackle first?
- Set aside some time to focus on improving your code base.
- As little as 2hrs a months every month can have a big impact

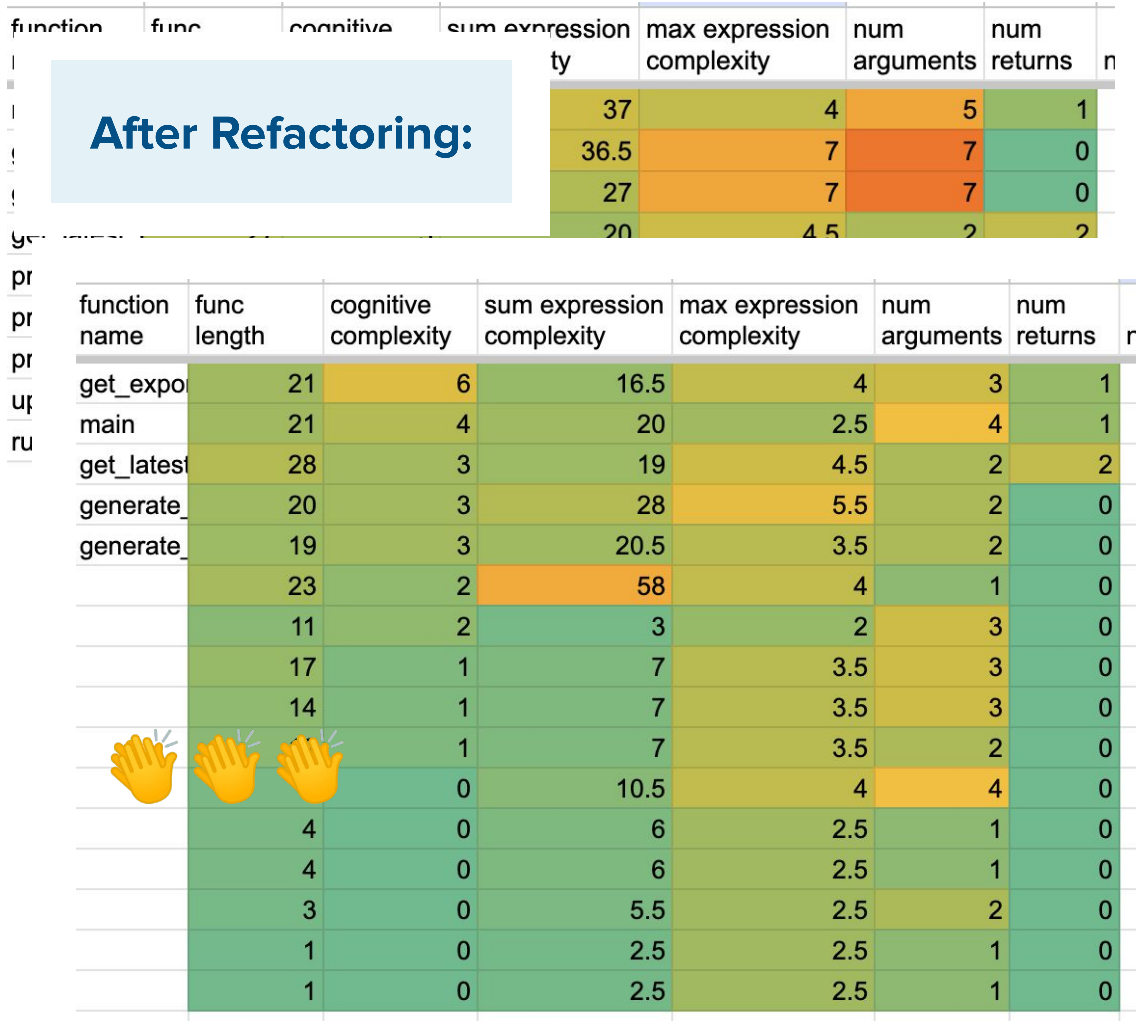


function name	func length	cognitive complexity	sum expression complexity	max expression complexity	num arguments	num returns
main	53	9	37	4	5	1
generate_c	24	6	30.5	7	7	0
generate_m	21	5	27	7	7	0
get_latest_	27	3	20	4.5	2	2
process_de	19	1	14	7	6	0
process_se	16	1	14	7	6	0
process_pri	14	1	14	7	5	0
upload_file_	4	0	11.5	4	4	0
run_query	3	0	6.5	2.5	2	0

Good candidate to refactor

Visualize & Prioritize

- How are your repositories developing over time?
- Which files/functions should we tackle first?
- Set aside some time to focus on improving your code base.
- As little as 2hrs a months every month can have a big impact



function	func	cognitive	sum expression ty	max expression complexity	num arguments	num returns
			37	4	5	1
			36.5	7	7	0
			27	7	7	0
ge			20	4.5	2	2
pr						
pr						
pr						
up						
ru						

After Refactoring:

function name	func length	cognitive complexity	sum expression complexity	max expression complexity	num arguments	num returns
get_expo	21	6	16.5	4	3	1
main	21	4	20	2.5	4	1
get_latest	28	3	19	4.5	2	2
generate_	20	3	28	5.5	2	0
generate_	19	3	20.5	3.5	2	0
	23	2	58	4	1	0
	11	2	3	2	3	0
	17	1	7	3.5	3	0
	14	1	7	3.5	3	0
	[illegible]	1	7	3.5	2	0
	[illegible]	0	10.5	4	4	0
	4	0	6	2.5	1	0
	4	0	6	2.5	1	0
	3	0	5.5	2.5	2	0
	1	0	2.5	2.5	1	0
	1	0	2.5	2.5	1	0

In the end, it’s about culture

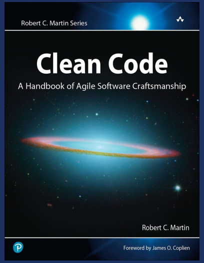
How to foster a clean code culture in your team:

- Regularly sync & discuss which code needs improvement
- Fixed time per month to work on code quality
- Use pair programming to ease burden & additional knowledge sharing
- Learn about best practices through e.g. reading groups

OTA INSIGHT

Thank you for your time!

Resources


Robert C. Martin Series
Clean Code
A Handbook of Agile Software Craftsmanship
Robert C. Martin
Foreword by James O. Coplien

- Clean Code [pdf, videos] by Uncle Bob
  - All in Java but still worth a read
  - Recommended Chapters: 1-5 and 17. Chp 6-10 go deeper into data structures, classes etc. 14-16 are Java very specific and less interesting for python developers
- Notebook to compute complexity