Clean and maintainable code

source: gettyimages.com

Goals of Programming and Priorities

The three laws of informatics

source: https://medium.com/@schemouil/rust-and-the-three-laws-of-informatics-4324062b322b

1. Programs must be correct.

2. Programs must be maintainable, except where it would conflict with the First Law.

3. Programs must be efficient (enough), except where it would conflict with the First or Second Law.

1. Correctness

A program produces correct results for **regular input**.

A program produces consistent results for **corner case input**.

A program detects **invalid input** and reports it.

A program **does not crash**.

2. Maintainability

Making changes is not a struggle.

It is easy to find and fix bugs.

Others can understand your program.

3. Efficiency

Not part of this presentation.

What is fast enough?

Don’t guess, measure first (use a **profiler**).

Optimization often needs good knowledge of programming language.

Optimization often needs good knowledge of algorithms and data structures.

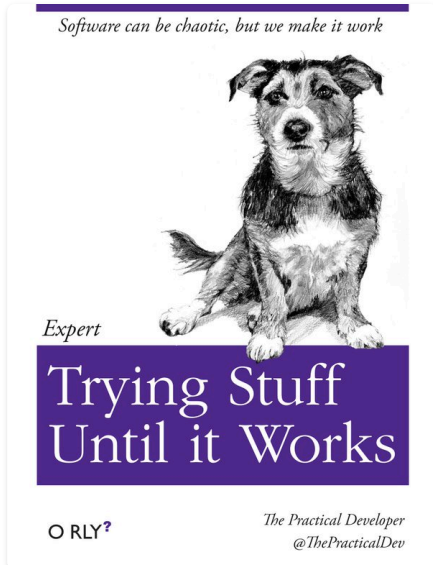
Maintainability:

1. General recommendations

General Recommendations

Understand what you do

- Read the documentation!
- Plan your program
- Don’t program by coincidence
- Understand bugs before you try to fix them
- Rethink what you do: are there alternative solutions?
- Be brave to trash your program and start from scratch



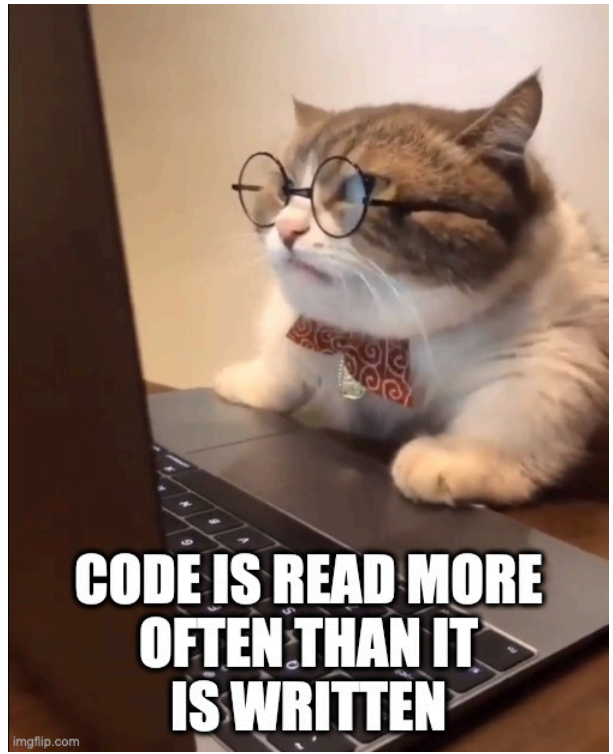
General recommendations, continued

Some principles

- `D.R.Y.`: Don’t repeat yourself (reduce code duplication).
  - Don’t copy paste, write functions instead!
  - Rule of thumb: duplicate code once, but not twice.
- `K.I.S.S.`: Keep it simple, stupid = avoid unnecessary complexity.
- Write **generic code**: split configuration and actual code
  - Simple case: set paths as variables at the beginning of a notebook
  - More complex: use config files and read them

Maintainability:

2. The power of readable code



source: https://i.imgflip.com/7sy18k.jpg

Readability: Reduce mental overhead

- Little scrolling needed.
- Keep related code together.
- Use empty lines to structure code.
- Avoid very long lines.
- Reduce nesting.
- Choose a code style and use it.

Code Style Example: Python's PEP 8 style guide

Most important points from `PEP 8`:

- Space after `,`, spaces around `+`, `*`, …
- Maximal line length: 80 characters.
- Indent by multiples of four spaces.
- `snake_case` for functions and variable names.
- `CamelCase` for class names.

Other style guides

- R: http://adv-r.had.co.nz/Style.html
- Java: https://google.github.io/styleguide/javaguide.html, https://www.oracle.com/technetwork/java/codeconventions-150003.pdf
- Matlab: https://www.mathworks.com/matlabcentral/fileexchange/2529-matlab-programming-style-guidelines

Code style: tools

Tools to reformat your code:

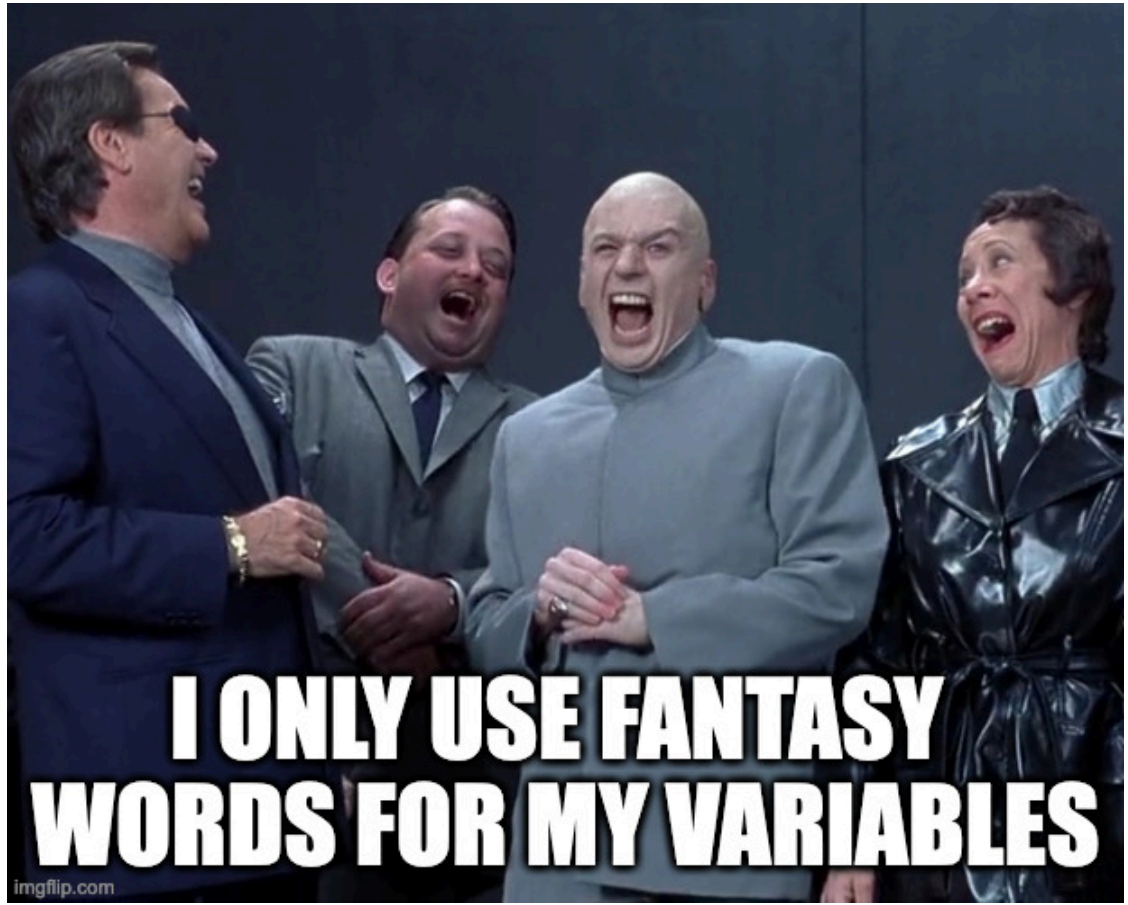
- Python: `ruff`. *PyCharm* has builtin code formatter, Visual Studio Code can use `ruff`.
- R: `formatR`, `styler`. *RStudio* has builtin code formatter.
- Java: `google-java-format`, *Eclipse*, *IntelliJ*.
- Matlab: https://github.com/davidvarga/MBeautifier

Code Styles

Don’t discuss details, do your work!

Maintainability:

3. The power of good names



source: https://i.imgflip.com/7sy514.jpg

Use names to reveal intention

Is this program correct?

```
def compute(a, b):
    return a + b
```

With appropriate names:

```
def area_rectangle(height, width):
    return height + width
```

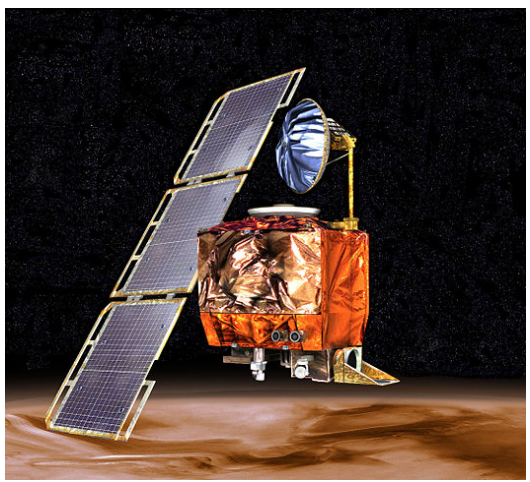
Good names reduce comments

```
time_tolerance = 60  # in seconds
```

```
time_tolerance_in_seconds = 60
```

https://en.wikipedia.org/wiki/Mars_Climate_Orbiter#/Cause_of_failure

($327.6 million lost)



Good names reduce comments

```
mem_used = mem_used / 1073741824  # bytes to gb
```

Introduce a constant for such "magic numbers":

```
BYTES_PER_GB = 1073741824
...
mem_used = mem_used / BYTES_PER_GB
```

- Use capital letters for constants.
- Constants also reduce *undetected* typing errors.

Names should not encode types

```
names_list = read_names_as_list()

for name in names_list:
    ...
```

```
names = read_names()

for name in names:
    ...
```

If types change you will have to change your code, else your code will lie!

Explanatory variables to express intent

```
if taste_score > 30:
    print("I like this")
```

```
is_sweet = (taste_score > 30)

if is_sweet:
    print("I like this")
```

Explanatory variables to reduce duplication

```
if width * height > 30:
    print("area", width * height, "is too large.")
```

```
area = width * height

if area > 30:
    print("area", area, "is too large.")
```

Now imagine more complicated expressions!

If good names are not enough: write comments!

- Don’t comment the obvious.
- Only comment the unexpected.
- If you use a solution from the internet: add a comment with the link.
- Check if your comment is understandable for others.

NOT LIKE THIS

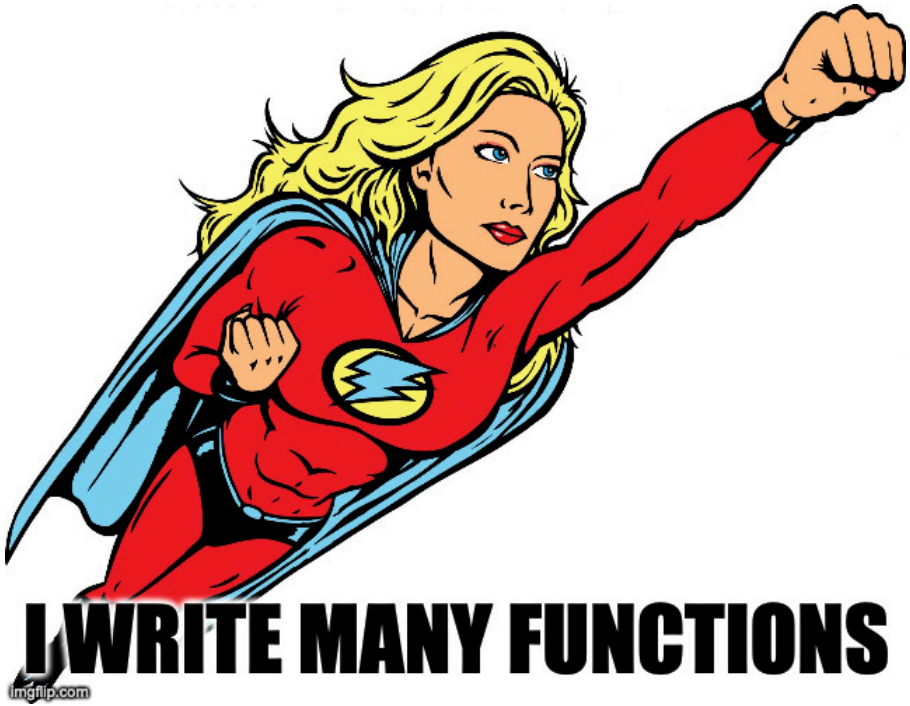
```
# open file:
fh = open("sequece.txt")

# read all lines:
lines = fh.readlines()

# compute number of sequences:
number_of_sequences = len(lines)
```

Maintainability:

4. The power of writing functions



source: https://i.imgflip.com/7sy0qk.jpg

Write many functions

- D.R.Y: Functions avoid code duplication
- Readability: Functions can reduce nesting level
- Good function names reduce comments
- Well written functions simplify code structure
- Well written functions support re-usability
- Well written functions support testability

A function should only do "one thing"!

```
def ask_and_check_if_prime():
    number = int(input("give me a number: "))
    divisor = 2
    while divisor * divisor <= number:
        if number % divisor == 0:
            print(number, "is not prime")
            return
        divisor += 1
    print(number, "is prime")
```

A function should only do "one thing"! Continued

Instead:

```
def is_prime(number):
    divisor = 2
    while divisor * divisor <= number:
        if number % divisor == 0:
            return False
        divisor += 1
    return True

def ask_and_check_if_prime():
    number = int(input("give me a number: "))
    if is_prime(number):
        print(number, "is prime")
    else:
        print(number, "is not prime")
```

Naming functions

Loose recommendations:

- Function names should be verbs if the function changes sth
- Nouns if they’re used to return a certain value.

```
def upload_data(data):
    ...

def full_address(person_id):
    ...
```

If you feel the need to use `and` in the name it is most likely doing multiple things and should be split!

Separate pure and non-pure functions

A **pure function** always produces the same result for the same input(s).

Examples:

- `math.sin`
- `is_prime` from the previous examples.

A **non-pure function** usually has "side effects", e.g. by interacting with the environment.

Examples:

- reading or writing a file
- querying a Web API

Separate pure and non-pure functions

Why?

- pure functions are predictable and easier to reason about
- non-pure functions can be unpredictable and thus harder to get to work in all situations.
- pure functions are easier to test (more about this later).

A function should have few arguments

```
def my_algorithm(input_data, n_iter, epsilon, tau, variant):
    ...
```

We use a dictionary instead (or e.g. a named list in `R`):

```
def my_algorithm(input_data, configuration):

    n_iter = configuration["n_iter"]
    epsilon = configuration["epsilon"]
    tau = configuration["tau"]
    variant = configuration["variant"]
    ...
```

A function should have few arguments

```
distance = distance_3d(1, 2, 3, 3, 2, 1)
```

Which coordinate of the two points is which argument?

```
def distance_3d(x0, y0, z0, x1, y1, z1):
    dx = x1 - x0
    dy = y1 - y0
    dz = z1 - z0
    return (dx ** 2 + dy ** 2 + dz ** 2) ** .5

distance = distance_3d(1, 2, 3, 3, 2, 1)
```

A function should have few arguments

```
from dataclasses import dataclass

@dataclass
class Point3D:
    x: float
    y: float
    z: float

def distance_3d(p1, p2):
    dx = p2.x - p1.x
    dy = p2.y - p1.y
    dz = p2.z - p1.z
    return (dx ** 2 + dy ** 2 + dz ** 2) ** .5

distance = distance_3d(Point3D(1, 2, 3), Point3D(3, 2, 1))
```

A function should have few arguments

Solutions:

- R: named vectors and lists
- Python: dictionaries, data classes, named tuples
- Generally: implement classes

The code of a function should operate on the same level of abstraction

Avoid long functions with sections! Instead:

```
def workflow():
    configuration = read_configuration()
    data = read_data(configuration)
    results = process(data, configuration)
    write_result(results, configuration)
```

```
def read_data(configuration):
    input_path = configuration['input_file']
    if input_path.endswith(".csv"):
        return read_csv(input_path)
    elif input_path.endswith(".xlsx"):
        return read_xlsx(input_path)
    else:
        raise NotImplementedError(f"file {input_path} not supported")
```

The code of a function should operate on the same level of abstraction

- Clear expression of underlying approach.
- Easier navigation within code (dive deep where you want to).

Return early

```
def is_prime(number):
    if number >= 1:
        ...
        ...
    else:
        return False
```

Better:

```
def is_prime(number):
    if number < 1:
        return False
    ...
    ...
```

- reduces indentation

Use break / continue early

```
for value in values:
    if value > 0:
        # a long code section follows
        ...
        ...
```

Better:

```
for value in values:
    if value <= 0:
        continue
    # a long code section follows
    ...
    ...
```

Maintainability:

5. Be Consistent

Be consistent

- Consistent naming (e.g. either `index` or `idx`, not both)
- Consistent physical units
- Consistent style
- Consistent error handling (`return None` vs `raise ...`)

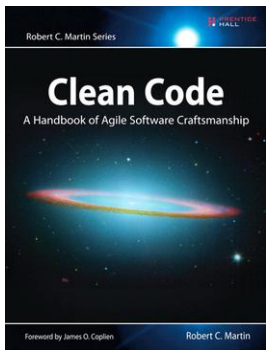
Be consistent

NOT LIKE THIS:

```
def lookup_pubchem(keyword):
    ...
    if not_found:
        return None
    ...

def lookup_kegg(keyword):
    ...
    if not_found:
        raise ValueError("not found")
    ...
```

Robert C Martin: Clean Code



source: https://www.oreilly.com

Take home messages:

- Understand what you do!

- Write with style!

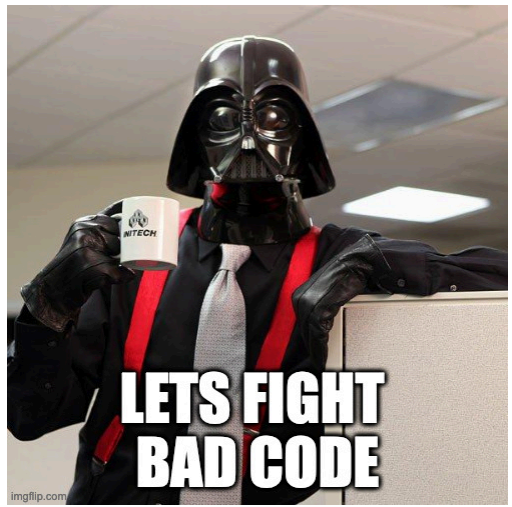
- Think about good names

- Write many good functions

- Be consistent

- Exercise and automate best practices

Questions?



source: https://i.imgflip.com/7sy4ha.jpg

Thanks for your attention!