Practical Practicable Good Practices

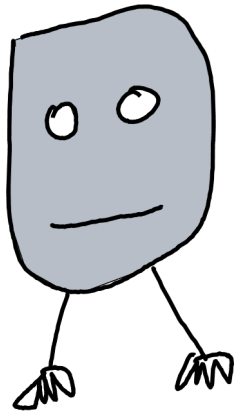
PyData Salamanca, 2021/03/18

Diego M. Rodríguez

twitter.com/diegoplan9

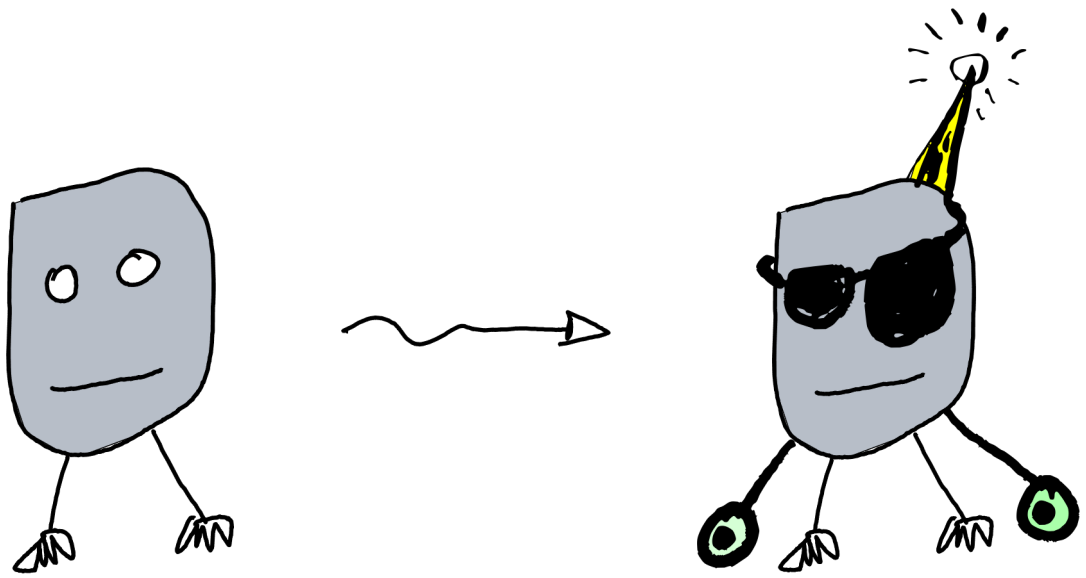
github.com/diego-plan9

Software is alive



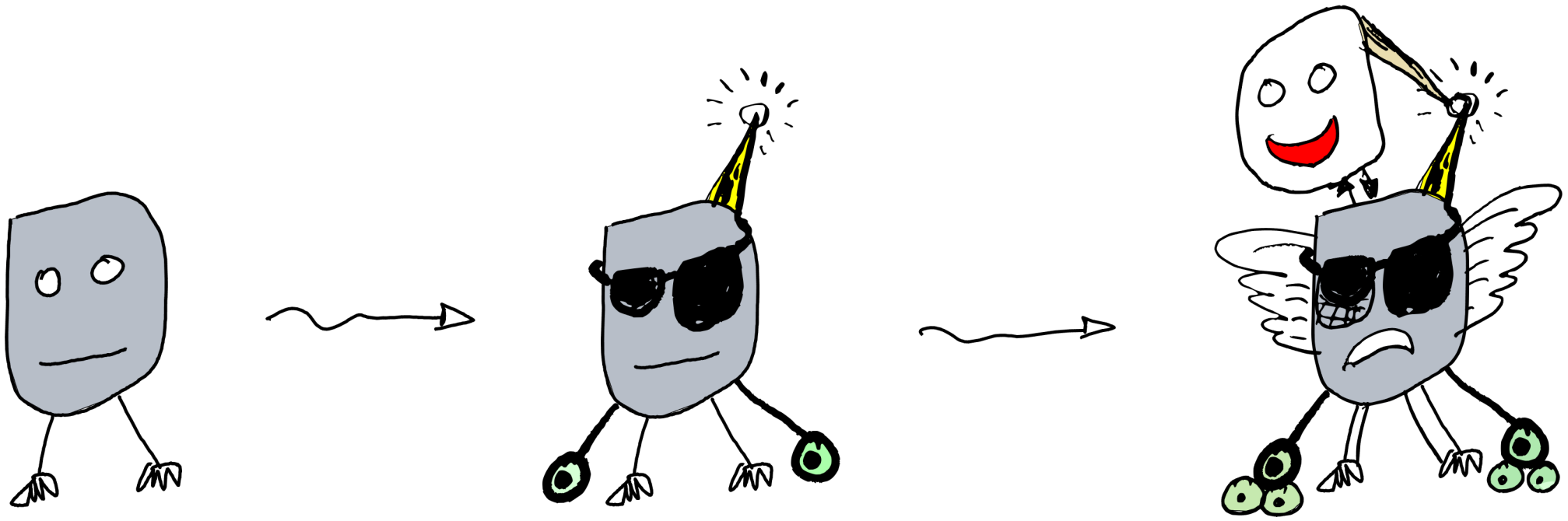
MVP1

Software is alive ... it evolves



MVP1 MVP2

Software is alive … it evolves … in unexpected ways



MVP1 MVP2 MVP3

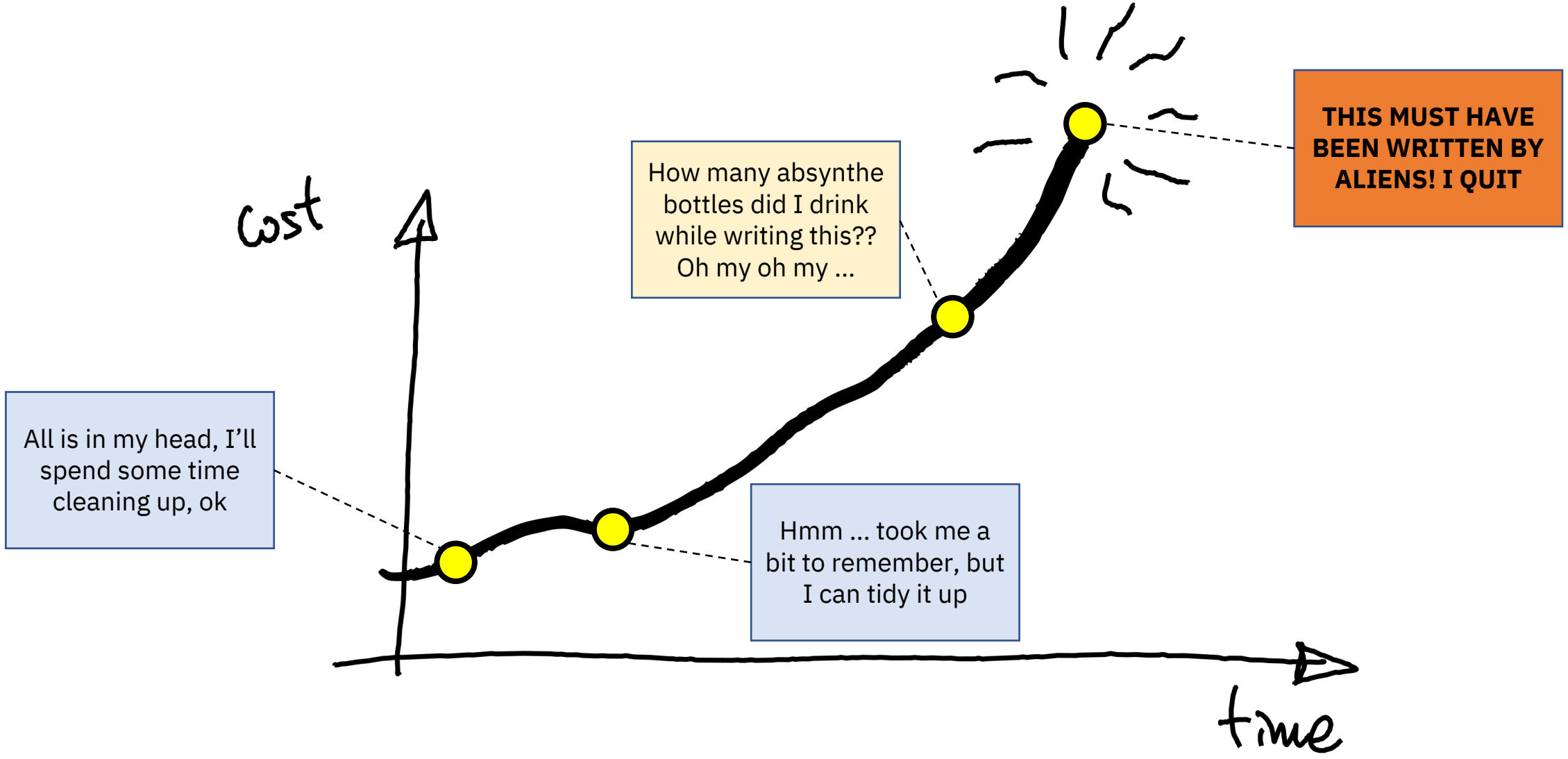
Write once, read many

Lines are written once, read **many** times

... and usually not read by others: by **your future self**!

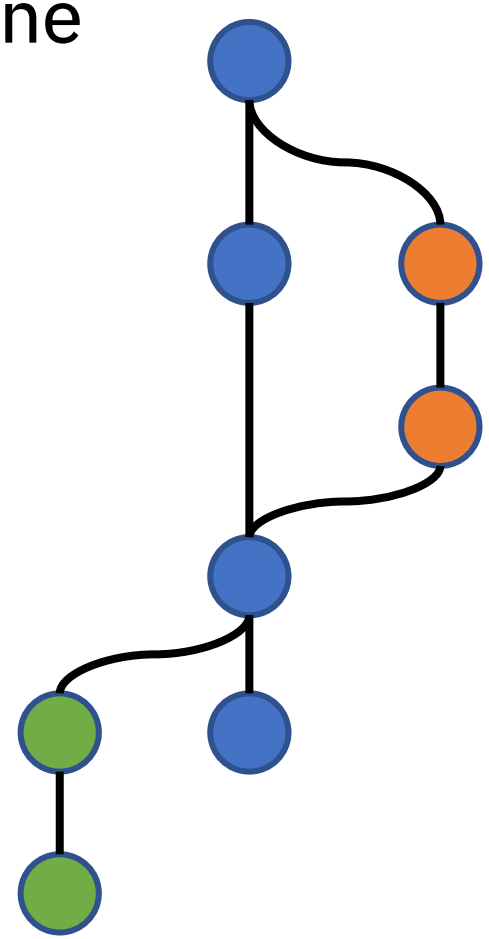
- Investing in good practices today saves on **time** tomorrow.
- Finding the right **balance** is key.
- Good practices help tame **complexity and evolution**.

It’s all about trade-offs: an investment



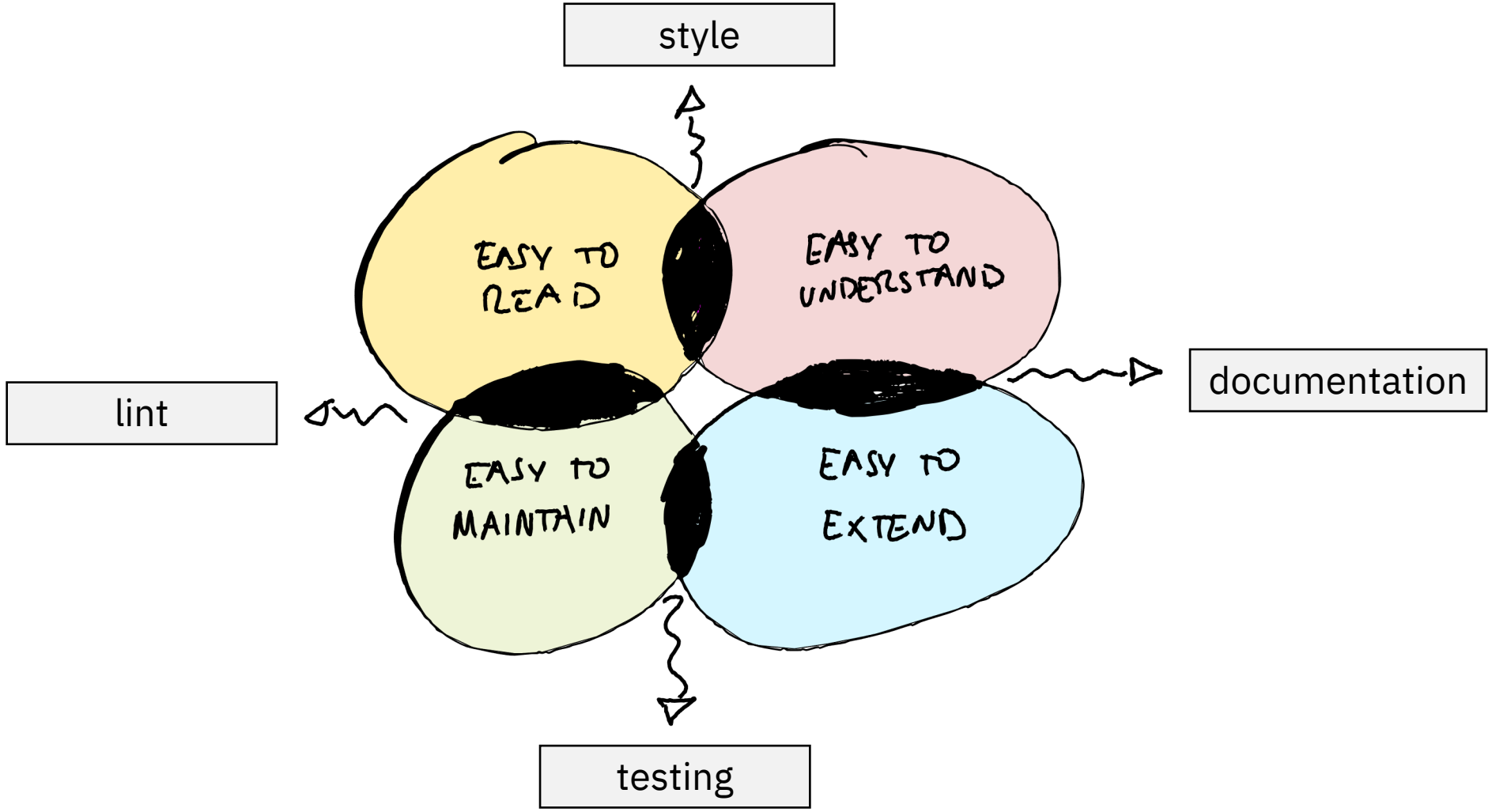
Tool #0: Version control

- A tool for **conquering time** – your personal time machine
- Keep an (annotated, granular) **history** of your project
- Complex, but worth it:
  - **commits**: annotated small changesets
  - **diff**: compare two points in time
  - **branches**: isolate different features
  - **forks**: easier coordination (+backup)



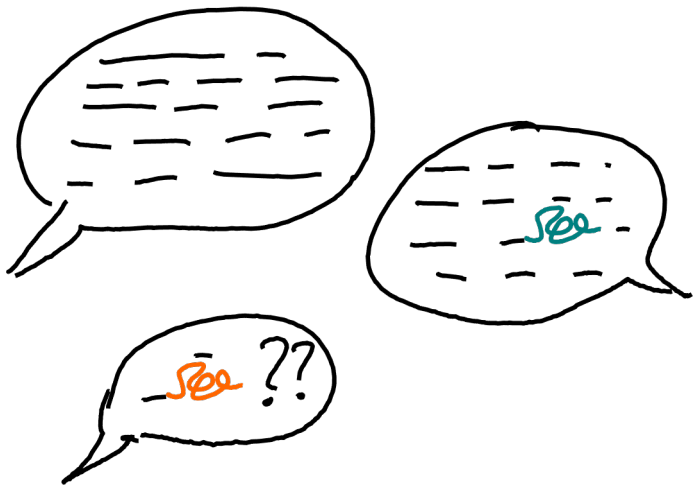
- Excellent tool and services support
  - CLI, IDEs, graphical clients; GitHub, GitLab, BitBucket

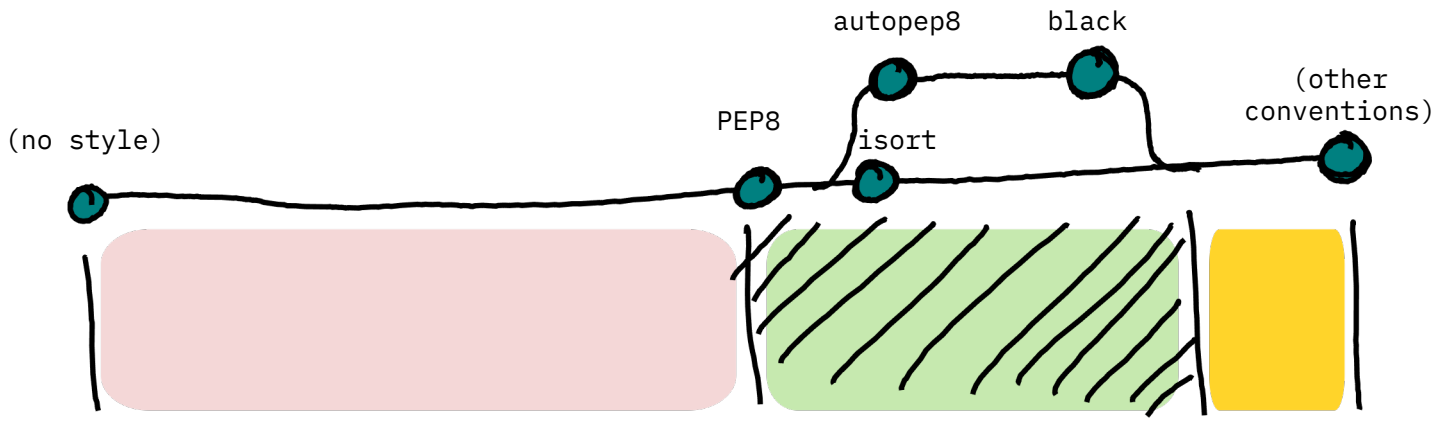
Venn-ish diagram of desires



Tool #1: Style

- Allows everyone to speak a “**common language**”
- Reduces distractions, keeps the **focus** on the important bits
- Sets **conventions** at different levels
- Good support in IDEs and tooling (automatable)
- *Python*: `PEP8`, `pycodestyle`, `black`, `isort`





Tool #1: Style (example)

```python
def Calculate(A, B= {}, print = True):
    if A == None:
        if print:
            print('error: A is not valid')
        return


    elif A != None:
        if print:
            print('calculating ...', \
                  "Using ", A)
    C = {}
    C['orig'] = A
    #C['comp'] = A*2??????
    C['comp'] = A *3.21868
    return C
```

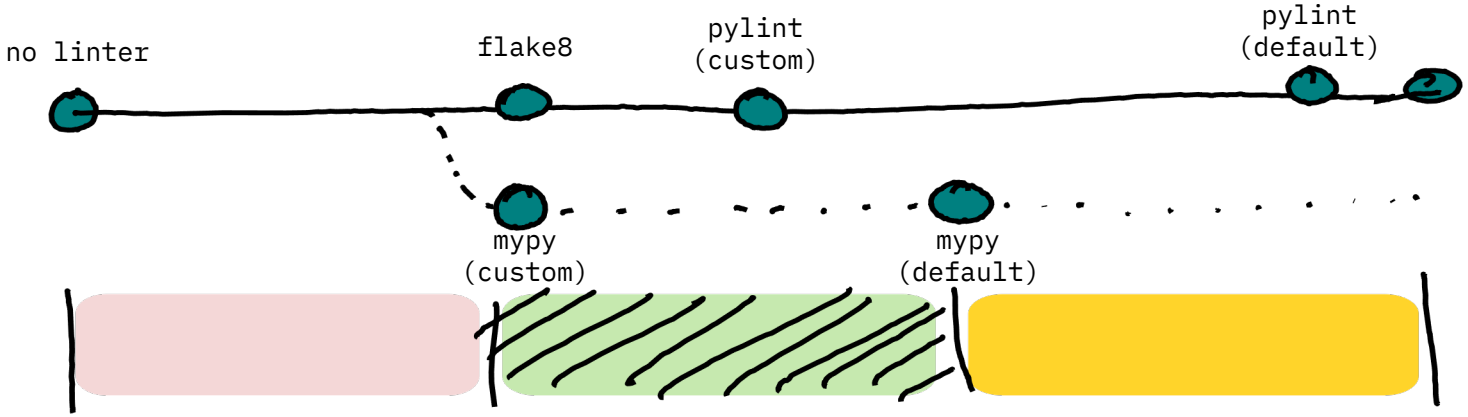
```python
def Calculate(A, B={}, print=True):
    if A is None:
        if print:
            print("error: A is not valid")
        return

    elif A is not None:
        if print:
            print("calculating ...", "Using ", A)
    C = {}
    C["orig"] = A
    # C['comp'] = A*2??????
    C["comp"] = A * 3.21868
    return C
```

```
$ black practices2.py
$ pycodestyle practices2.py
```

Tool #2: Lint

... a static code analysis tool used to flag programming errors, bugs, stylistic errors, and suspicious constructs.

- more **intelligent** than style checker
- **discover** potential bugs and issues
- doubles up as an "**assistant**" providing pro-tips
- *Python*: `pylint`, `flake8`, (`mypy`)



Tool #2: Lint++

- Gateway to software engineering **techniques and patterns**:
  - architecture
    - focus on defining the **interfaces**
    - **modularize** and split
    - vet the **dependencies**
  - simplicity and clarity
    - keep it **simple**
    - **generalize**, but not too soon
    - invest time in **naming**
  - idiomatic code
    - make use of **type hints**
    - explore standard library and de-facto standards

Tool #2: Lint (example)

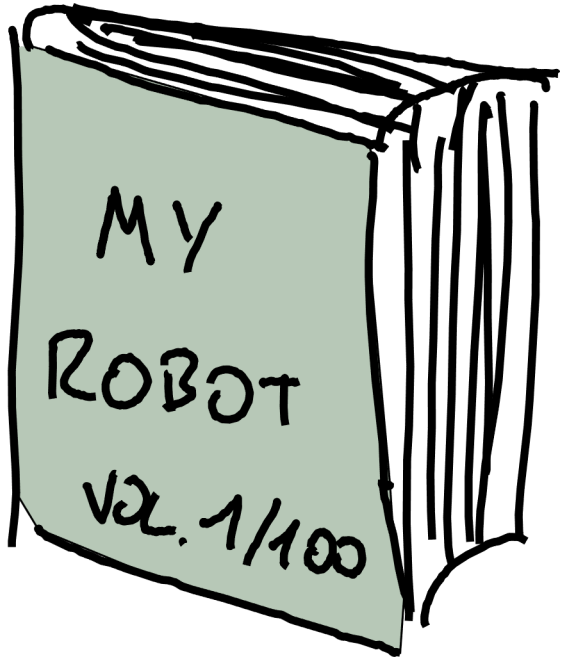
```python
def Calculate(A, B={}, print=True):
    if A is None:
        if print:
            print("A is not valid")
        return

    elif A is not None:
        if print:
            print("calculating ...", "Using ", A)
    C = {}
    C["orig"] = A
    # C['comp'] = A*2??????
    C["comp"] = A * 3.21868
    return C
```

```python
def calculate(distance, print_output=True):
    if distance is None:
        raise Exception('distance is not valid')

    if print_output:
        print("calculating ...", "Using ", distance)

    return {
        "orig": distance,
        "comp": distance * 3.21868
    }
```

```
$ pylint practices2.py
************* Module practices2
practices2.py:1:23: W0622: Redefining built-in 'print' (redefined-builtin)
practices2.py:1:0: W0102: Dangerous default value {} as argument (dangerous-default-value)
practices2.py:1:0: C0103: Argument name "A" doesn't conform to snake_case naming style (invalid-name)
practices2.py:1:0: C0116: Missing function or method docstring (missing-function-docstring)
practices2.py:2:4: R1705: Unnecessary "elif" after "return" (no-else-return)
practices2.py:1:0: R1710: Either all return statements in a function should return an expression, or none should. (inconsistent-return-statements)
practices2.py:1:17: W0613: Unused argument 'B' (unused-argument)
...
```

Tool #3: Documentation

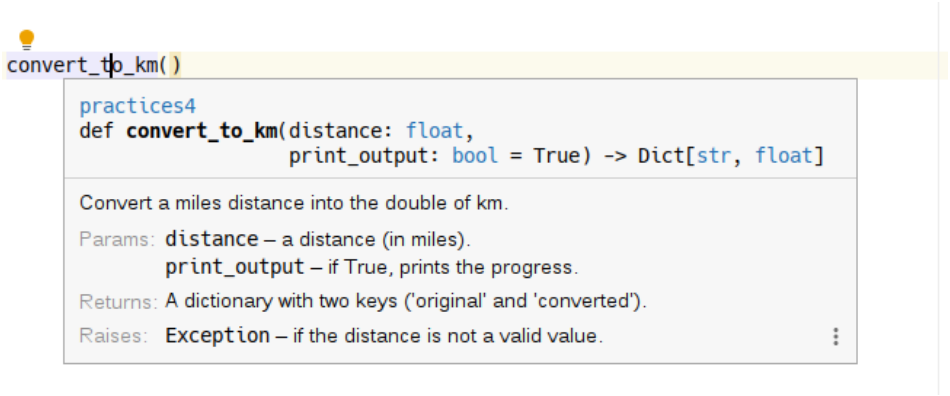
- Convey the "**why**" instead of the "how"
- Notes for **others and your future self**
- Avoids the need of being a **mind-reader**
- Several levels of documentation:
  - inline comments
  - docstrings
  - technical docs
- Integration with IDEs and tools for free



- *Python*: `sphinx`, `apidoc`, `doxygen`

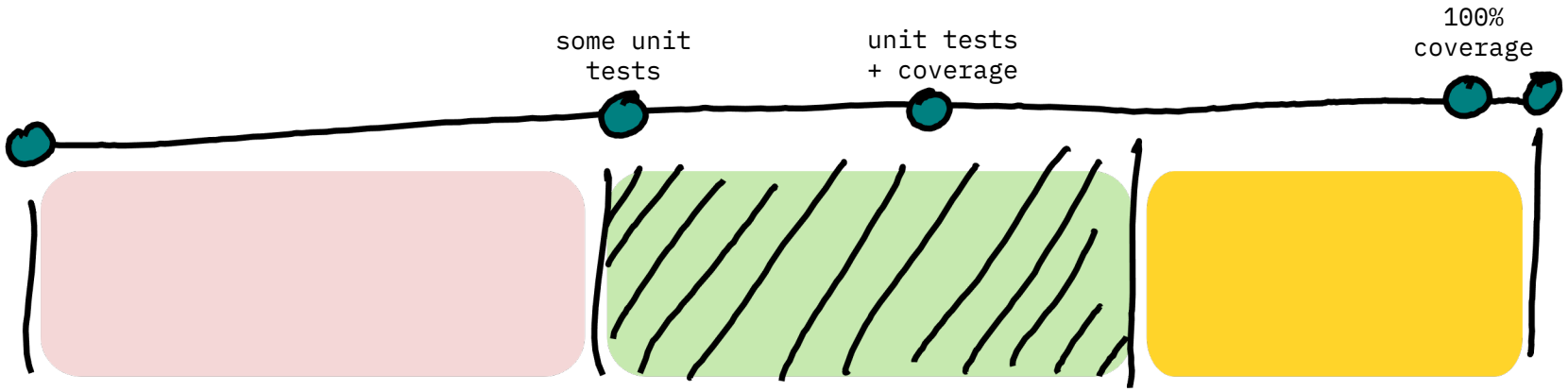
Tool #3: Documentation (example)

```python
def convert_to_double_km(distance: float, print_output: bool=True) -> Dict[str, float]:
    """Convert a miles distance into the double of km.

    Args:
        distance: a distance (in miles).
        print_output: if True, prints the progress.

    Returns:
        A dictionary with two keys ('original' and 'converted').

    Raises:
        Exception: if the distance is not a valid value.
    """
    if distance is None:
        raise Exception('distance is not valid')

    if print_output:
        print("calculating ...", "Using ", distance)

    return {
        "original": distance,
        # The constant 2*1.60934 is used as the robot is magic
        # and covers twice the distance if specified in km.
        "converted": distance * 3.21868
    }
```



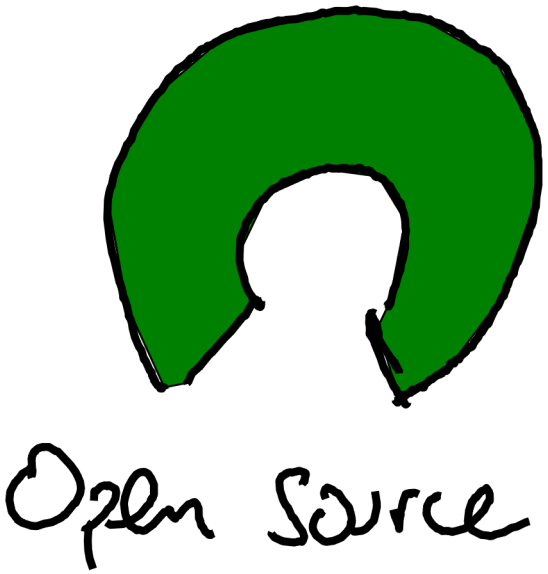
Tool #4: Testing

- Ensure that new changes **don't break** existing features
- Can also double-up as **documentation**
- Allow **repeatability** and checking different conditions
- Provide unvaluable **Peace Of Mind**
- (Whole topic by itself!)
- Python: `unittest`, `pytest`, `nose`, `coverage`

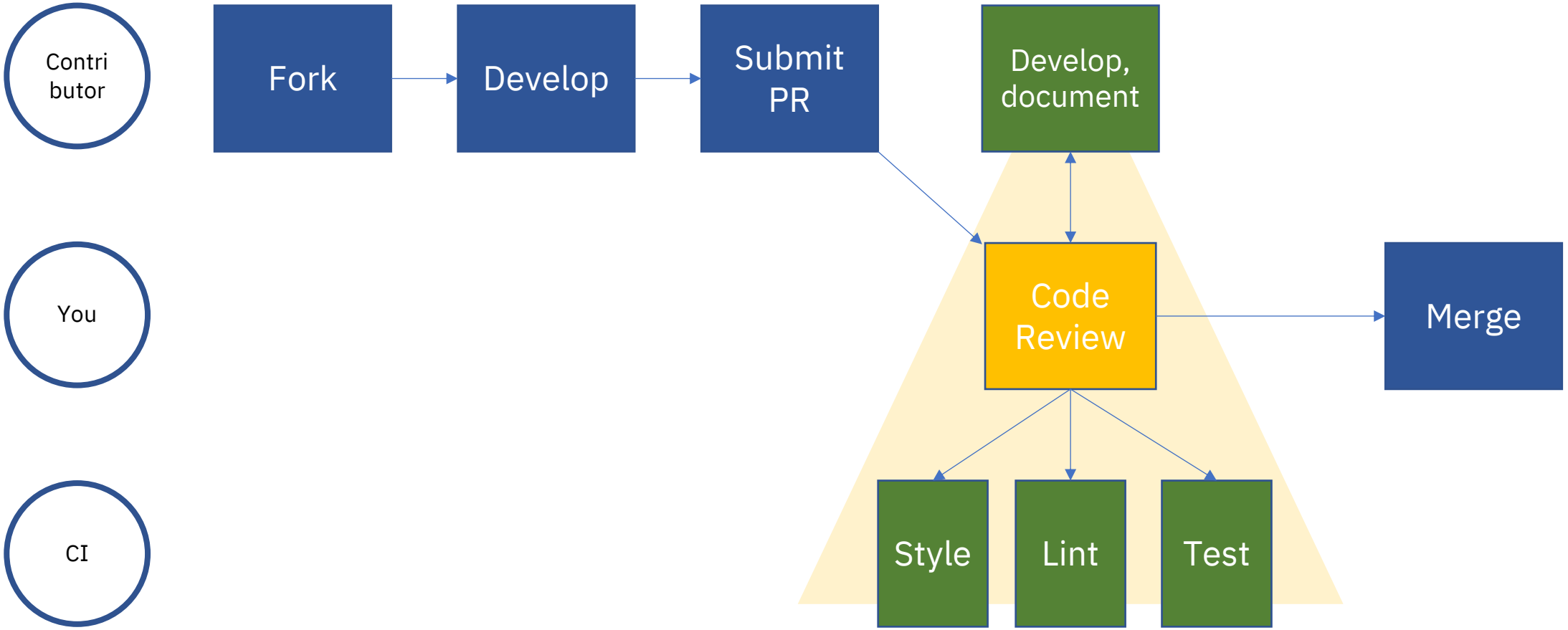


Open source & good practices symbiosis

- **Open sourcing** your software will force you to apply good practices
  - More contributions ➔ more diversity ➔ more complexity
  - More users ➔ more reports ➔ more ideas
  - (Not so different in corporate environment
    - Contributors ≈ team mates; Users ≈ management and customers)
- Good practices as tools for facilitating:
  - cooperation
  - communication
- Hand-in-hand with “meta” good practices:
  - Code reviews
  - Continuous integration & automation
  - Support and feedback



Open source: GitHub example model



Recap

- All software **evolves** through **time**
- Invest in **good practices**: pays off
- **Toolbelt**:
  - Version control
  - Style
  - Linter
  - Documentation
  - Testing
- For both open source / large projects …
- … and personal ones!