Best Practices

Scientific Programming with Python

Nicola Chiapolini

University of Zurich
Faculty of Science

23 June 2025

Based on talk by Valentin Haenel `https://github.com/esc/best-practices-talk`

Introduction

- We write code regularly
- We have not been formally trained

Best Practices

- evolved from experience
- increase productivity
- decrease stress
- still evolve with tools and languages

Development Methodologies

- e.g. Agile Programming or Test Driven Development
- lots of buzzwords
- still many helpful ideas

Outline

Outline

Coding Style

- readability counts (often more than brevity or speed)
- give things *intention revealing* names
  - For example: `numbers` instead of `n`
  - For example: `numbers` instead of `list_of_float_numbers`
  - See also: Ottinger's Rules for Naming

Example

```
def fun(n):
    """ no comment """
    r = 1
    for i in n:
        r *= i
    return r
```

Coding Style

- readability counts (often more than brevity or speed)
- give things *intention revealing* names
  - For example: `numbers` instead of `n`
  - For example: `numbers` instead of `list_of_float_numbers`
  - See also: Ottinger's Rules for Naming

Example

```
def my_product(numbers):
    """ Compute the product of a sequence of numbers. """
    total = 1
    for item in numbers:
        total *= item
    return total
```

Formatting Code

- use coding conventions, e.g: PEP-8
- conventions specify
  - layout
  - white-space
  - comments
  - naming
  - ...
- OR use a consistent style (especially when collaborating)

Formatting Code: Tools

Checker

- pylint (e.g. `pylint my_product.py`)
- pycodestyle (e.g. `pycodestyle my_product.py`)
- pydocstyle (e.g. `pydocstyle my_product.py`)
- flake8 (e.g. `flake8 my_product.py`)

Formatter

- autopep8 (e.g `autopep8 --in-place my_product.py`)
- yapf3 (e.g `yapf3 --in-place my_product.py`)
- black (e.g. `black my_product.py`)

Finding Bugs with Pylint

Example

```
def current_rotation(beta, iota, phi, sigma):
    """ calculate current rotation """
    return np.cos(beta)*np.cos(phi)*np.cos(sigma)
    +np.sin(beta)*np.cos(phi)*np.sin(sigma)*np.cos(iota)
    -np.sin(beta)*np.sin(phi)*np.sin(iota)
```

Finding Bugs with Pylint

Example

```
def current_rotation(beta, iota, phi, sigma):
    """ calculate current rotation """
    return np.cos(beta)*np.cos(phi)*np.cos(sigma)
    +np.sin(beta)*np.cos(phi)*np.sin(sigma)*np.cos(iota)
    -np.sin(beta)*np.sin(phi)*np.sin(iota)
```

```
[...]:7:4: W0101: Unreachable code (unreachable)
[...]:7:4: W0106: Expression "+np[...]" is assigned to nothing [...]
[...]:8:4: W0106: Expression "-np[...]" is assigned to nothing [...]
```

Documenting Code: Docstrings

Example

```
def my_product(numbers):
    """ Compute the product of a sequence of numbers. """
```

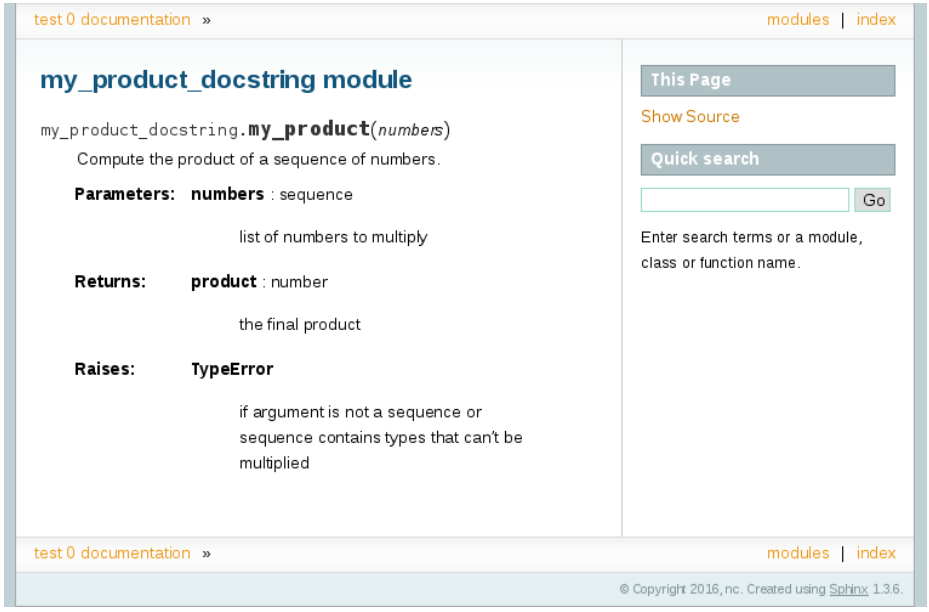
- at least a single line
- also for yourself
- is on-line help too
- Document arguments and return objects, including types
- For complex algorithms, document every line, and include equations in docstring
- Use docstring conventions: PEP257 and/or numpy

Example Docstring

```
def my_product(numbers):
    """ Compute the product of a sequence of numbers.

    Parameters
    ----------
    numbers : sequence
        list of numbers to multiply

    Returns
    -------
    product : number
        the final product

    Raises
    ------
    TypeError
        if argument is not a sequence or sequence contains
        types that can't be multiplied
    """
```

Documenting Your Project

- tools generate website from docstrings
  - pydoc
  - sphinx
  - Overview List
- when project gets bigger
  - how-to
  - FAQ
  - quick-start



Outline

`import`

- Don't use the *star import*: `from module import *`
  - not obvious what you need
  - modules may overwrite each other
  - Where does this function come from?
  - will import *everything* in a module
  - . . . unless you have a very good reason: e.g. `pylab`, interactive
- Put all imports at the beginning of the file. . .
- . . . unless you have a very good reason

Example

```
import my_product as mp
mp.my_product([1,2,3])
```

```
from my_product import my_product
my_product([1,2,3])
```

`import`: Pitfalls

Python evaluates the imported code at import time.

```
""" Bad Things happen here. """

def append_one(list_=[]):
    """ Do not use mutable default values """
    list_.append(1)
    return list_

def default_arg(bad=1 / 0):
    """ Do not trigger exceptions in keyword-arguments """
    return bad

def constants():
    """ This can not be imported in Python < 3.6 """
    return 9999999 ** 9999999
```

Exceptions

- use `try`, `except` and `raise`
- often better then `if` (e.g. `IndexError`)

Example

```
try:
    my_product(1, 2, 3)
except TypeError as e:
    raise TypeError("'my_product' expects a sequence") from e
```

- don't use *special* return values:
  `1`, `0`, `False`, `None`
- Fail early, fail often
- use built-in Exceptions

assert

- Not intended for checks needed in production.

Example

```
def withdraw(balance,amount):
    assert balance > amount, "Balance too small"
    return balance - amount
print(withdraw(50, 100))
```

running: `python3 withdraw.py`

```
Traceback (most recent call last):
    [...]
    assert balance > amount, "Balance too small"
           ^^^^^^^^^^^^^^^^
AssertionError: Balance too small
```

assert

- Not intended for checks needed in production.

Example

```
def withdraw(balance,amount):
    assert balance > amount, "Balance too small"
    return balance - amount
print(withdraw(50, 100))
```

running: `python3 -O withdraw.py`

assert

- Not intended for checks needed in production.

Example

```
def withdraw(balance,amount):
    assert balance > amount, "Balance too small"
    return balance - amount
print(withdraw(50, 100))
```

running: `python3 -O withdraw.py`

```
-50
```

`mypy`: Type Hints and Static Checking

- ensures variables and functions are used correctly

Example

```
def factorial(n):
    res = 1
    for i in range(1,n+1):
        res *= i
    return res

def concat(str1, str2):
    return str1+str2


n = 4.
res = factorial(n)
concat(f"{n}! = ", res)
```

`mypy`: Execute with Python

```
python3 factorial.py
```

```
Traceback (most recent call last):
  File ".../factorial_nohint.py", line 26, in <module>
    res = factorial(n)
          ^^^^^^^^^^^^
  File ".../factorial_nohint.py", line 3, in factorial
    for i in range(1,n+1):
             ^^^^^^^^^^^^
TypeError: 'float' object cannot be interpreted as an integer
```

- `n` must be integer
- once fixed, we get the next error: `str2` must be string

`mypy`: Adding Type Hints

- can add type info to variables and function definitions

Example

```
def factorial(n: int) -> int:
    res = 1
    for i in range(1,n+1):
        res *= i
    return res

def concat(str1: str, str2: str) -> str:
    return str1+str2


n = 4.
res = factorial(n)
concat(f"{n}! = ", res)
```

`mypy`: Check with `mypy`

```
python3 -m mypy factorial.py
```

```
factorial.py:15: error: Argument 1 to "factorial" has incompatible
    type "float"; expected "int"  [arg-type]
factorial.py:16: error: Argument 2 to "concat" has incompatible type
    "int"; expected "str"  [arg-type]
Found 2 errors in 1 file (checked 1 source file)
```

- spots both problems
- must be run separately (ideally automatically)

`mypy`: Summary

Cons

- additional work for programmer
- defining correct types can be tricky
  (type hints cheat sheet)
- code gets more verbose and maybe confusing

Pros

- prevent a lot of possible bugs
- valuable even if not all code covered

Outline

Keep it Simple (Stupid) – KIS(S) Principle

Keep it Simple

> Debugging is twice as hard as writing the code in the first place.
> Therefore, if you write the code as cleverly as possible,
> you are, by definition, not smart enough to debug it.
>
> – Brian W. Kernighan

Don’t Repeat Yourself (DRY)

- No copy & paste!
- Not just lines code, but knowledge of all sorts
- Do not express the same piece of knowledge in two places. . .
- . . . or you will have to update it everywhere
- It is not a question of *if* this may fail, but *when*

Don't Repeat Yourself (DRY): Types

Example

- Copy-and-paste a snippet, instead of refactoring it into a function
- Repeated implementation of utility methods
  - because you don't remember
  - because you don't know the libraries

    ```
    numpy.prod([1,2,3])
    ```

  - because developers don't talk to each other
- Version number in source code, website, readme, package filename

- If you detect duplication: refactor!

Outline

Refactoring

- re-organise your code without changing its functionality
- rethink earlier design decisions
- break large code blocks apart
- rename and restructure code
- will improve the readability and modularity
- will usually reduce the lines of code

Common Refactoring Operations

- Rename class/method/module/package/function
- Move class/method/module/package/function
- Encapsulate code in method/function
- Change method/function signature
- Organise imports (remove unused and sort)
- Always refactor one step at a time, and ensure code still works
  - version control
  - unit tests

Refactoring Example

```
def product_minus_sum(numbers):
    """ Subtract sum of numbers from product of numbers. """
    total = 0
    for item in numbers:
        total += item
    total2 = 1
    for item in numbers:
        total2 *= item
    return total - total2
```

- split into functions
- use libraries/built-ins
- fix bug

Refactoring Example

```python
from my_math import my_product, my_sum

def product_minus_sum(numbers):
    """ Subtract sum of numbers from product of numbers. """
    sum_value = my_sum(numbers)
    product_value = my_product(numbers)
    return sum_value - product_value
```

- split into functions
- use libraries/built-ins
- fix bug

Refactoring Example

```python
from numpy import prod, sum

def product_minus_sum(numbers):
    """ Subtract sum of numbers from product of numbers. """
    sum_value = sum(numbers)
    product_value = prod(numbers)
    return sum_value - product_value
```

- split into functions
- use libraries/built-ins
- fix bug

Refactoring Example

```python
from numpy import prod, sum

def product_minus_sum(numbers):
    """ Subtract sum of numbers from product of numbers. """
    sum_value = sum(numbers)
    product_value = prod(numbers)
    return product_value - sum_value
```

- split into functions
- use libraries/built-ins
- fix bug

Outline

What is a Development Methodology?

Consists of:

- process used for development
- tools to support this process

Help answer questions like:

- How far ahead should I plan?
- What should I prioritise?
- When do I write tests and documentation?

Right methodology depends on scenario.

What is a Development Methodology?

Consists of:

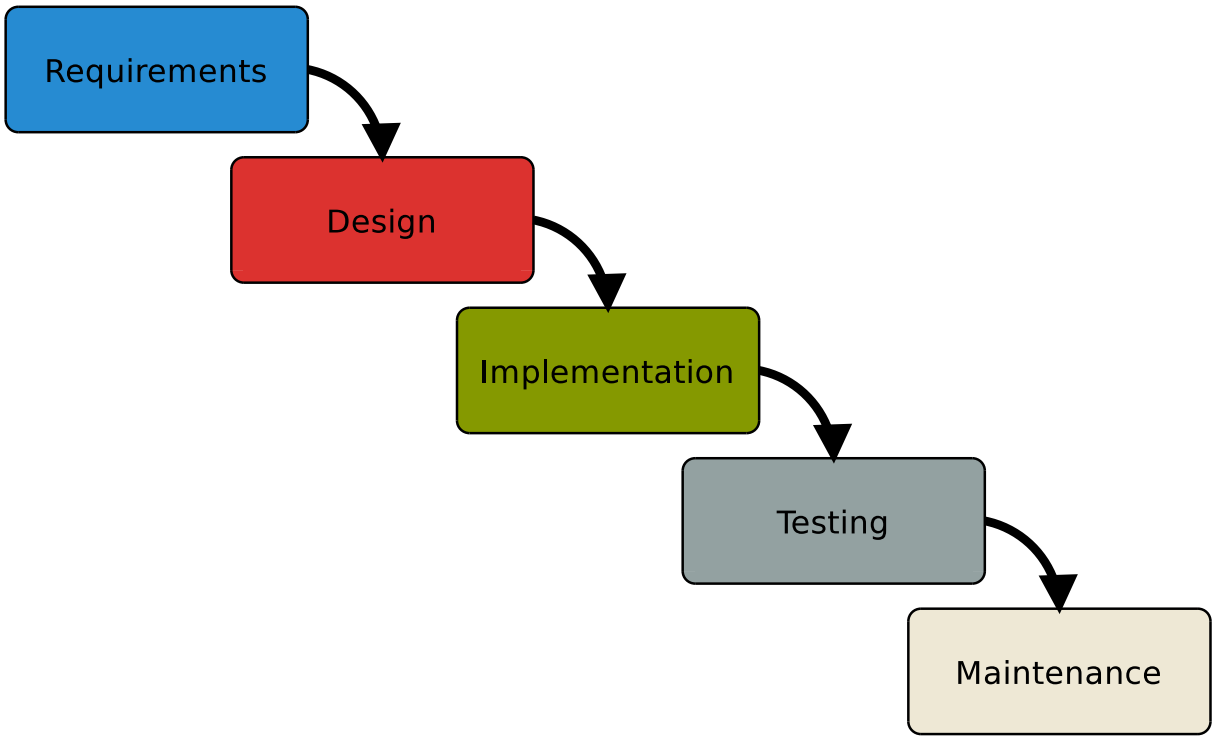
- process used for development
- tools to support this process

Help answer questions like:

- How far ahead should I plan?
- What should I prioritise?
- When do I write tests and documentation?

Right methodology depends on scenario.

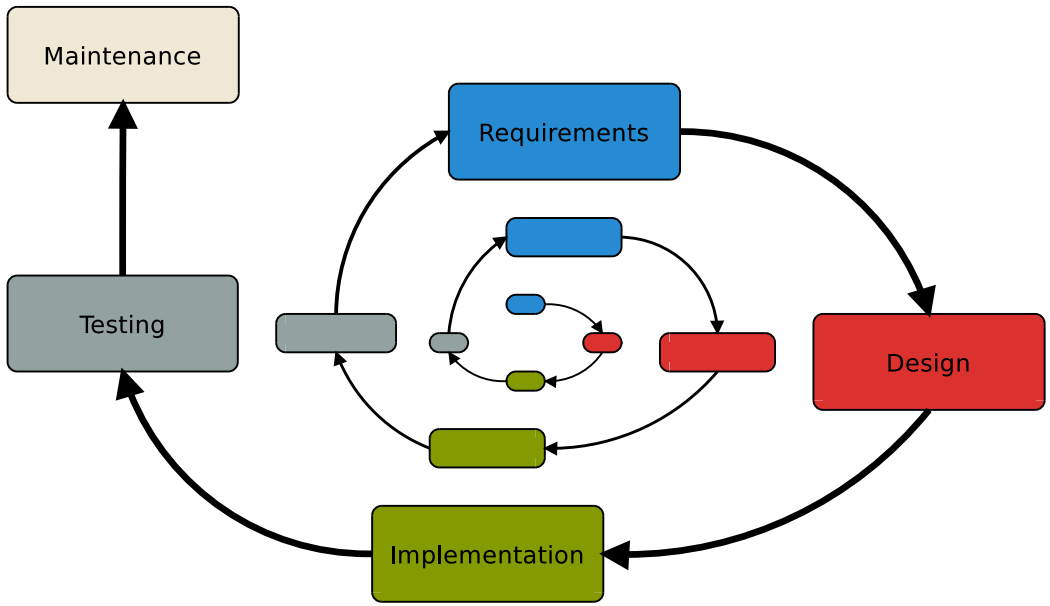
The Waterfall Model, Royce 1970



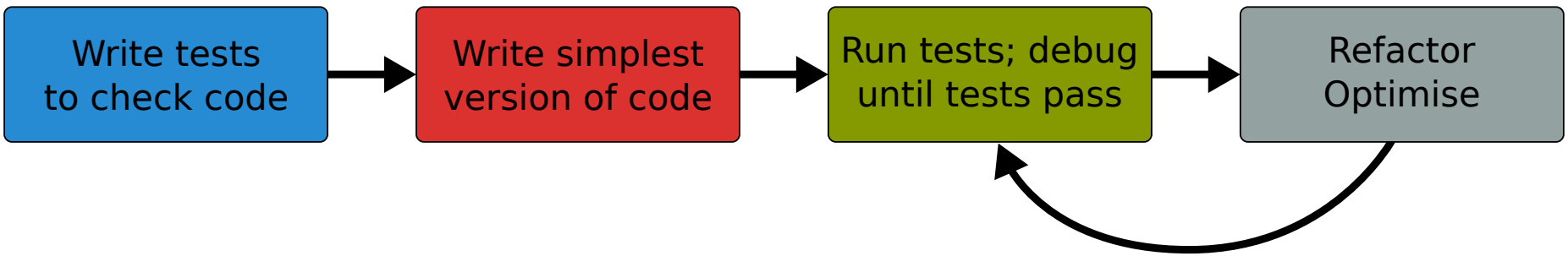
- sequential
- from manufacturing and construction

Agile Methods (late 90's)

- minimal planning, small development iterations
- frequent input from environment
- very adaptive, since nothing is set in stone



Test Driven Development (TDD)



- Define unit tests first!
- Develop one unit at a time!
- more tomorrow

An Almost Unrelated Note: Using VirtualEnv

The Problem

- different tools need different versions of a module
- your Linux distribution does not include a module

The Solution: virtualenv

- initialise folder `school_venv` to store modules of this project

```
python -m venv --system-site-packages ~/school_venv
```

- update the search-paths to include folders in `~/venv`

```
. venv/bin/activate
```

- run your code or install libraries with pip
- undo changes to search-paths

```
deactivate
```