Code Quality

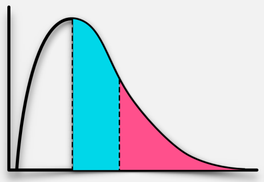
Because Therapy Isn’t Covered

Stefan Fröse, 24.06.2025

whoami

CTAO

Gamma-ray Astronomy (PhD@TUDO)



Statistics

Air Shower Physics (Taiwan)

Scientific Programming

Why?

```
def f(x,y=0):return[x[i]+y if x[i]>0else y-x[i]for i in range(len(x))]
```

```
def f(x,y=0):return[x[i]+y if x[i]>0else y-x[i]for i in range(len(x))]
```

This runs but do you trust it?

The Steps of Code Quality

1. Surface Quality ⟶ Readability

2. Semantic Quality ⟶ Understanding

3. Testability ⟶ Clarity

Static Code Checks

- Runs tools before runtime
- No code execution

Remember

Your code can be “clean” but still wrong

Surface Quality

Python's Style Guide

How to format your code

- Python Enhancement Proposals No. 8
- All about Readability
- Five important parts:
  - Code Lay-out
  - Whitespaces
  - Trailing Commas
  - Comments
  - Naming Conventions

Python's Style Guide

Example

```
def calc(x,y):return x+y+1
import os,sys
def DoSomething():print(  "Done" )
```

```
import os
import sys


def calc(x, y):
    return x + y + 1


def do_something():
    print("Done")
```

Tools

Overview

- pycodestyle -> strictly follows PEP8
- flake8 -> pyflakes + pycodestyle + plugins
- black -> opinionated formatter, ~PEP8 + strict rules (88 chars)
- isort -> sorts imports
- ruff -> all of the above + ultra fast

Tools

Overview

- pycodestyle -> strictly follows PEP8
- flake8 -> pyflakes + pycodestyle + plugins
- black -> opinionated formatter, ~PEP8 + strict rules (88 chars)
- isort -> sorts imports
- ruff -> all of the above + ultra fast

Recommended: Ruff

Ruff

Installation + Usage

- ‘pipx install ruff’ (pipx instead of pip!)
- Linter:
  - ‘ruff check’
  - Output:

```
test.py:1:1: F401 [*] `os` imported but unused
test.py:4:5: E302 expected 2 blank lines, found 1
```

- Formatter:
  - ‘ruff format’
  - Output: Linter hints already applied to files
  - Warning: no auto-sort of imports, do ‘ruff check --select I --fix’

Ruff

Configuration

- pyproject.toml (or ruff.toml) in your basedir
- Lots of options!
- Shamelessly stolen from Scientific-Python Cookie
- Be aware of what you use!
- Also check Ruff Docs!

```toml
[tool.ruff.lint]
extend-select = [
  "B",           # flake8-bugbear
  "I",           # isort
  "ARG",         # flake8-unused-arguments
  "C4",          # flake8-comprehensions
  "EM",          # flake8-errmsg
  "ICN",         # flake8-import-conventions
  "PGH",         # pygrep-hooks
  "PIE",         # flake8-pie
  "PL",          # pylint
  "PT",          # flake8-pytest-style
  "PTH",         # flake8-use-pathlib
  "RET",         # flake8-return
  "RUF",         # Ruff-specific
  "SIM",         # flake8-simplify
  "TID251",      # flake8-tidy-imports.banned-api
  "T20",         # flake8-print
  "UP",          # pyupgrade
  "YTT",         # flake8-2020
]
ignore = [
  "PLR",    # Design related pylint codes
  "RUF012", # Would require a lot of ClassVar's
]
```

Semantic Quality

Docstrings

- Why? -> Explains what your code does!
- How? -> Triple quotes (""" ... """)
- Complete sentence + explanation of parameters and returns
- Different styles available, choose one, be consistent

Numpy Style

```
def scale(data, factor):
    """
    Scale the input data by a constant factor.

    Parameters
    ----------
    data : np.ndarray
        Input array to scale.
    factor : float
        The scaling factor.

    Returns
    -------
    np.ndarray
        Scaled data array.
    """
```

Numpy Style

```
def scale(data, factor):
    """
    Scale the input data by a constant factor.

    Parameters
    ----------
    data : np.ndarray
        Input array to scale.
    factor : float
        The scaling factor.

    Returns
    -------
    np.ndarray
        Scaled data array.
    """

scale([1,2,3], 2) -> [1, 2, 3, 1, 2, 3]
```

→ Hints on types

Type Hints

Overview

- Python is dynamically typed
- Add types to Python to avoid unwanted behaviour
- These types are still not enforced at runtime!
  - But it will enhance the understanding of your code!

- Tools:
  - mypy: ‘pip install mypy’ (good for CLI/CI)
  - pyright: `pip install pyright` (faster, VSCode integration)

Type Hints

Example

```
def calc(x, y):
    return x + y
```

```
def calc(x: int, y: int)-> int:
    return x + y
```

Run Static Type Checker (mypy test.py)

```
calc(“abc”, “def”)
```

```
test.py:7: error: Argument 1 to "calc" has incompatible type "str"; expected "int"  [arg-type]
test.py:7: error: Argument 2 to "calc" has incompatible type "str"; expected "int"  [arg-type]
```

Basic Hints

- Basic python types: int, float, str, bool
- Union: `int | float`
- Optional: `int | None`
- List: `list[int]`
- Sequence: `Sequence[int]` (list, tuple, …)
- Callable: `Callable[[int], int]` for methods
- Any: `Any` always passes -> The last resort

Any vs. object

- `Any` is a last resort, everything will pass this
- Why you should use `object` as default:
  - `Any` disables type checking for variable
  - catches invalid ops at type check time
  - Type narrowing! (next slide)

```
def with_object(x: object) -> None
    print(x)            # ✅ ok
    str(x)              # ✅ ok
    x + 1               # ❌ error
    x.upper()           # ❌ error
    x.nonexistent()     # ❌ error
```

Type Narrowing

- You function might accept Unions `int | str`
- Need to narrow the type for specific behaviour
- This allows you to keep track of all cases in an easy way
- Check type via `isinstance`

```
def format_id(id: int | str) -> str:
    return id.zfill(6)  # pads with zeros

format_id(42) # fails
format_id("42") # "000042"

def format_id(id: int | str) -> str:
    # narrow type
    if isinstance(id, int):
        id = str(id)
    return id.zfill(6)

format_id(42) # "000042"
format_id("42") # "000042"
```

Type Narrowing

- Other narrowing
  - These are called `TypeGuards`
- `issubclass` checks class
- `callable` checks method

```
def area(x: object) -> None:
    issubclass(x, MyClass):
        return ...
    ...

from collections.abc import Callable
def area(func: Callable[[int], float]) -> None:
    if callable(func):
        return ...
    ...
```

Type Narrowing

- Custom TypeGuards
- Self-defined function reveals no information about the type after narrowing
- TypeGuards are essentially a bool with extra info
- `TypeGuard[list[str]]`
- But TypeGuard does not narrow in `else`
  - For this case use `TypeIs`

```
def is_str_list(val: list[object]) -> bool:
  """Determines whether all objects in the list are strings"""
  return all(isinstance(x, str) for x in val)

def func1(val: list[object]) -> None:
    if is_str_list(val):
        reveal_type(val)  # Reveals list[object]
        print(" ".join(val)) # Error: incompatible type
```

```
from typing import TypeGuard
def is_str_list(val: list[object]) -> TypeGuard[list[str]]:
  """Determines whether all objects in the list are strings"""
  return all(isinstance(x, str) for x in val)
```

Advanced Types

- Lots of other cool features
- `TypeVar`
- `Literal`
- `TypedDict`
- `Annotated`
- `NewType`
- `Final`
- …

```
from typing import TypedDict

class Movie(TypedDict):
    name: str
    year: int

movie: Movie = {'name': 'Blade Runner',
                'year': 1982}
```

Duck Typing

- “If it walks like a duck and it quacks like a duck, then it must be a duck”
- No inheritance need, just implement the methods that define a duck
- If you want to enforce users to implement API use `abc.abstractmethod`
- Duck Typing with `Protocol`

```python
from typing import Protocol


class HasArea(Protocol):
    def area(self) -> float: ...


class Circle:
    def __init__(self, r: float):
        self.r = r

    def area(self) -> float:
        return 3.14 * self.r * self.r


class Square:
    def __init__(self, a: float):
        self.a = a

    def area_of_a_square(self) -> float:
        return self.a * self.a


def print_area(shape: HasArea) -> None:
    print(f"Area = {shape.area()}")


print_area(Circle(2))  # OK
print_area(Square(3))  # fails
```

Testability

Testability

- Tomorrow lecture on **Testing**
  - You expect behaviour from a function or class
  - You run the function or class and check if it matches expectation
- Testability:
  - Write code that is easily testable
  - “Lower you expectations” = Functions should be simple

Pure Functions

- No side effects
- Same input → same output
- Don’t touch globals
- Don’t mutate input
  - Makes operations chainable
  - Numpy `a.T.mean().round(2)`

```
log: list[int] = []
def square(x: int) -> int:
    result = x * x
    log.append(result)
    return result

def square(x: int) -> int:
    return x * x

def double(values: list[int]) -> None:
    for i in range(len(values)):
        values[i] *= 2

def double(values: list[int]) -> list[int]:
    return [v * 2 for v in values]
```

Raise vs assert

- `Raise`
  - Use for input validation
  - Use if you want to **shout at the user**!
- `assert`
  - Internal consistency checks
  - Not for validating user input
    - Use if you want to **shout at the dev**!

```
if x <= 0:
    raise ValueError("x must be positive")

assert x > 0  # dev mistake if false
```

The Config Singleton

- Singleton is a class with only one instance
- Most common: Config
  - Often: load config once, reuse everywhere
- Brings pain later!
  - Becomes implicit global state
  - Hidden dependencies
  - Hard to override in tests

```
# Taken from MCEq/src/MCEq/config.py
# config.py acts a singleton since namespace

#: parameters for EarthGeometry
r_E = 6391.0e3  # Earth radius in m
h_obs = 0.0  # observation level in m
h_atm = 112.8e3  # top of the atmosphere in m

#: Default parameters for GeneralizedTarget
#: Total length of the target [m]
len_target = 1000.0
#: density of default material in g/cm^3
env_density = 0.001225
env_name = "air"
```

The Config Singleton

- Pydantic:
  - Data validation
  - Uses type hints!
- Create a config class based on `BaseModel`
- Similar to `dataclass` the `BaseModel` has fields
- Dump/Load configs

```python
from pydantic import BaseModel

class Config(BaseModel):
    name: str
    debug: bool = False
    threshold: float = 0.5

# create config
cfg = Config(name="MCEq")
```

Automation

The pre-commit Hook

- `pre-commit` = tool for automatically doing everything we spoke about
  - Run linter `ruff`
  - Run `mypy`
  - Run `codespell`: Checks typos
  - And much more
- `pip install pre-commit`

pre-commit

- Configure with `.pre-commit-config.yaml`
- Loads and install each tool
- `rev`: version tags on GitHub
- Run `pre-commit install`
- `pre-commit` runs automatically during every `git commit …`
- Available as GitHub Action -> Part of CI lecture

```yaml
repos:
 - repo: https://github.com/astral-sh/ruff-pre-commit
  # Ruff version.
  rev: v0.12.0
  hooks:
    # Run the linter.
    - id: ruff-check
      args: [ --fix ]
    # Run the formatter.
    - id: ruff-format

 - repo: https://github.com/pre-commit/mirrors-mypy
    rev: v1.16.1
    hooks:
      - id: mypy

 - repo: https://github.com/codespell-project/codespell
    rev: v2.4.1
    hooks:
      - id: codespell
```

Summary

Summary

1. Implement Surface Quality to make your code readable
2. Implement Semantic Quality to make your code understandable
3. Implement Testability to make your code verifiable

Resources

- PEP8
- Scientific Cookie
- pycodestyle
- flake8
- black
- isort
- ruff
- mypy
- pyright
- Google Style Guide
- PEP484-Type Hints
- Type Narrowing
- PEP544-Protocols
- pydantic
- codespell
- pre-commit