Technische Universität Braunschweig

Suresoft
SUSTAINABLE RESEARCH SOFTWARE

Clean Code and Refactoring

Sven Marcus, 09.11.23

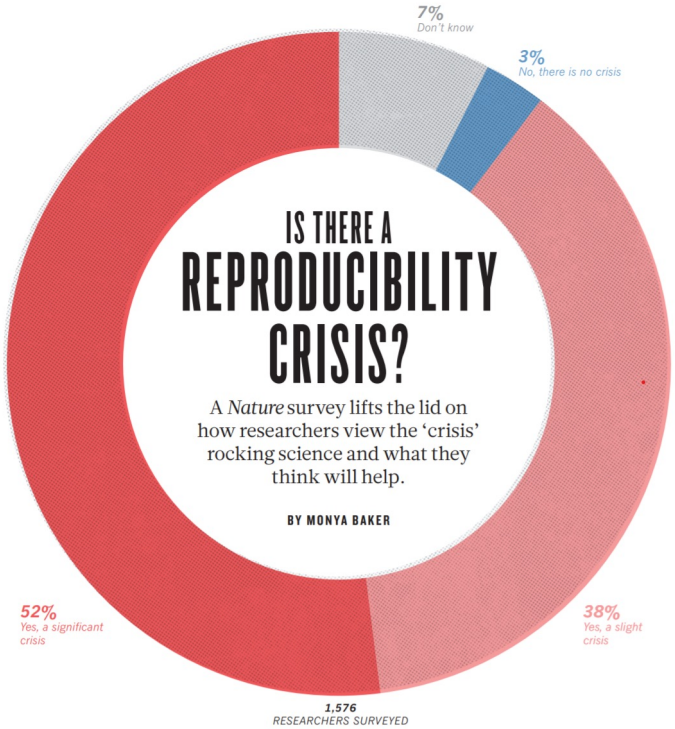
Content

- Motivation
- Clean code and Refactoring
- Python tips and tricks
- Let’s get to work: the Guilded Rose Kata

Motivation

Motivation

- 84% of scientists say that developing software is essential to their research. [1]
- Incorrect publications have led to a reproducibility and credibility crisis. [2, 3]



[1] Jo Hannay et al. "How Do Scientists Develop and Use Scientific Software?"
[2] Monya Baker. "1,500 scientists lift the lid on reproducibility".
[3] Zeeya Merali. "Computational science: Error, why scientific programming does not compute".

Motivation

Researchers

- often lack knowledge about principles and practices of the software engineering discipline [3, 4]
- don’t gain reputation for developing software
- are pressed to publish results as fast as possible [5]

[4] Lucas Joppa et al. “Troubling Trends in Scientific Software Use”.
[5] Mark De Rond and Alan N Miller. “Publish or perish: bane or boon of academic life?”

Consequences

Common problems of research software

- Low code quality
- Hard to understand
- Neither published nor documented

Broken Window Effect

[6] George L Kelling, James Q Wilson et al. “Broken windows”.

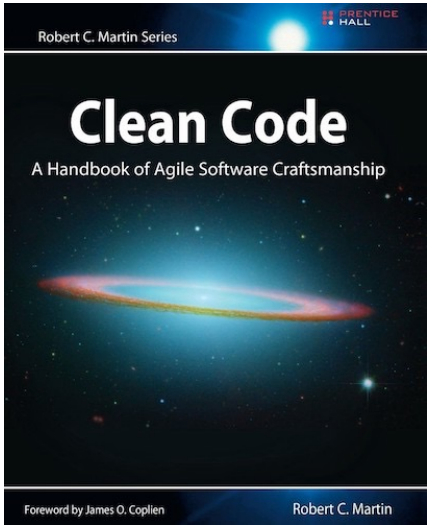
Motivation

“Indeed, the ratio of time spent reading versus writing is well over 10 to 1. We are constantly reading old code as part of the effort to write new code. ...[Therefore,] making it easy to read makes it easier to write.” – Robert C. Martin

Clean Code and Refactoring

What is clean code?

- Term coined by Robert C. Martin
- No strict rules, but a set of principles to make code easy to understand, extend and adapt



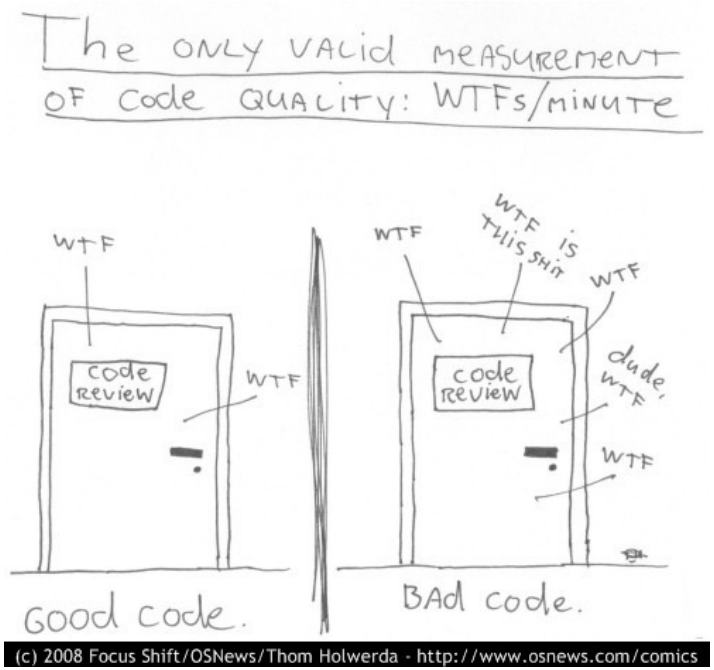
Clean Code

“Clean code always looks like it was written by someone who cares.”
– Michael Feathers

“Clean code reads like well-written prose.”
– Grady Booch

“Clean code can be read, and enhanced by a developer other than its original author.”
– Dave Thomas

Clean Code


The ONLY VALID MEASUREMENT OF CODE QUALITY: WTFs/MINUTE
WTF
code review
WTF
Good code.
WTF
WTF is THIS SHIT
WTF
code review
dude, WTF
WTF
BAd code.
(c) 2008 Focus Shift/OSNews/Thom Holwerda - http://www.osnews.com/comics

Refactoring

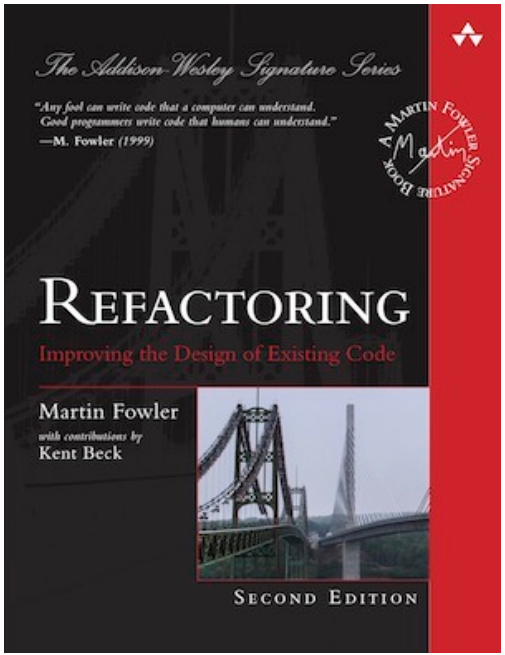
noun: *a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior*

verb: *to restructure software by applying a series of refactorings without changing its observable behavior.*

https://refactoring.com/

Refactoring

- Focus on improving the readability and structure of existing code
- Assigns clear names to the code changes
- Shares many ideas with Clean Code



Refactoring

“Any fool can write code that a computer can understand.
Good programmers write code that humans can understand.”

“The point of refactoring isn’t to show how sparkly a code base is - it is purely economic. We refactor because it makes us faster - faster to add features, faster to fix bugs.”

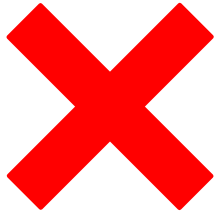
– Martin Fowler

Names

Use meaningful names

- Classes, variables, functions, modules, binaries, libraries…
- The name should describe the **purpose**

Names



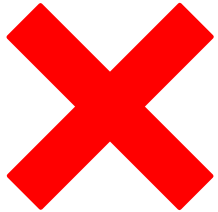
```
1  var = 7  # num days in week
```

Names

```
1  number_of_days_in_week = 7
```

Name of the Refactoring:
Rename Variable

Names



```
1  def copy_chars(a1, a2):
2      for i in range(0, len(a1)):
3          a2[i] = a1[i]
```

Names

Even if the function is simple,
it can be hard to understand what it does for a user

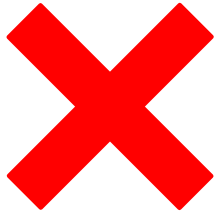
```
(a1: Any, a2: Any) → None
copy_chars()
```

Names

Make meaningful distinctions to make the code understandable

```
1  def copy_chars(source, destination):
2      for i in range(0, len(destination)):
3          destination[i] = source[i]
```

Names



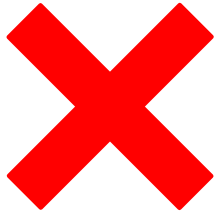
```
1  def genymdhms(): # generate date, year, month, day, minutes
```

Names

- Use names that are pronounceable and searchable
- Avoid encodings (unless they are commonly known)

```
1  def generate_timestamp():
```

Names



```
1  def do_the_next_thing():
```

Names

Use domain names

```
1  def apply_neumann_boundary_condition():
```

Name of the Refactoring:
Change function declaration, Rename function

Comments

„Don’t comment bad code - rewrite it.“

- Brian W. Kernighan and P.J. Plaugher

„The proper use of comments is to compensate for our failure to express ourself in code. Note that I used the word failure. I meant it. Comments are always failures."

- Robert C. Martin

Comments

If a comment must be used, it should describe the ***why*** and not the ***what*** or ***how***

Comments

```python
# check if i is a prime number
prime = True
for j in range(i):
    if i % j == 0:
        prime = False
        break
```

Comments

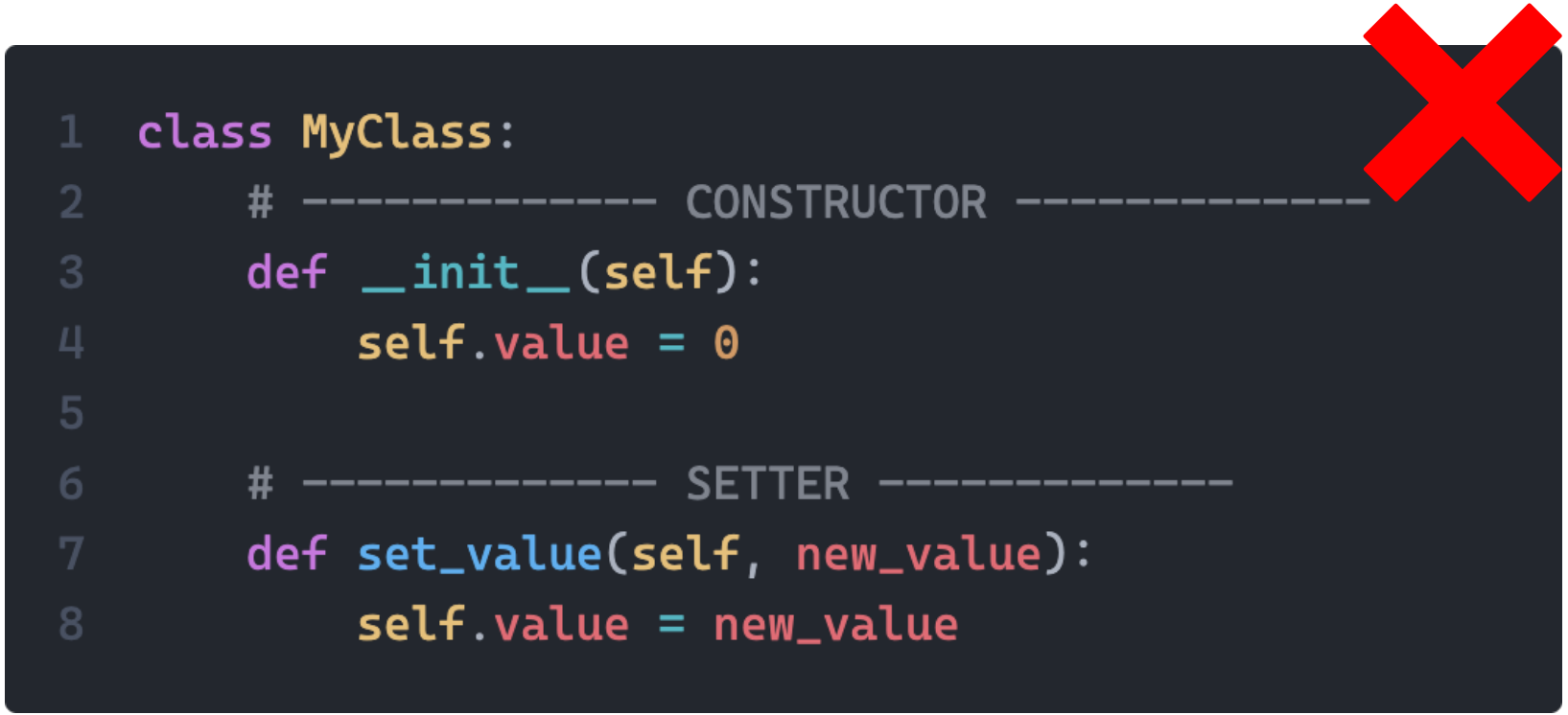
Express your intention in code instead of comments

```python
def is_prime(number: int) -> bool:
    for divisor in range(number):
        if number % divisor == 0:
            return False

    return True
```

Comments

```python
class Container:

    def __init__(self, logger) -> None:
        # The logger associated with this container
        self.logger = logger
```

Comments



```python
class MyClass:
    # ------------- CONSTRUCTOR -------------
    def __init__(self):
        self.value = 0

    # ------------- SETTER -------------
    def set_value(self, new_value):
        self.value = new_value
```

Comments

Don’t require the use of doc comments in private functions if they don’t add value

Use doc comments in public APIs

```python
def _is_sad(maybe_sad: str) -> bool:
    """Checks if string is sad

    Args:
        maybe_sad(str): a string

    Returns:
        bool
    """
    return ":(" in maybe_sad or "T_T" in maybe_sad


def make_happy(sad_string: str) -> str:
    """Makes a sad string happy

    Args:
        sad_string(str): A string with sad smileys

    Returns:
        str
    """
    if not _is_sad(sad_string):
        return sad_string

    return sad_string.replace(":(", ":)").replace("T_T", "^_^")
```

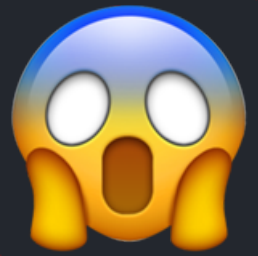
Comments

```python
def load_config(parser: ConfigParser, config_path: str) → list[str]:
    """
    Loads configuration from a config file.
    Falls back to the defaults file on fail.
    """
    config = parser.read(config_path)
    if not config:
        config = parser.read(".defaults")

    return config
```

Comments

```python
def load_config(parser: ConfigParser, config_path: str) → list[str]:
    """
    Loads configuration from a config file.
    Falls back to the defaults file on fail.
    """
    config = parser.read(config_path)
    if not config:
        config = parser.read(".defaults")

    return config
```

LIES!!!



Good Comments

```
1  # We're doing this, because of ...
2
3  # This code could lead to a dead lock
4  # which I wasn't able to prevent so far
5
6  # TODO: pass additional parameter to implement XYZ model
7
8  # Cutting out spaces here is important, because ...
```

Explanation of intent

Warning

Amplification

Functions

```python
a = [1, 2, 3, 4]
with open("log_a.txt", "w") as f:
    for num in a:
        f.write(str(num) + "\n")

b = [1, 2, 3, 4]
with open("log_b.txt", "w") as f:
    for num in b:
        f.write(str(num) + "\n")
```

Functions

DRY – DON'T REPEAT YOURSELF!

```python
def write_list_to_file(a_list: list[int], filename: str) -> None:
    with open(filename, "w") as f:
        for num in a_list:
            f.write(str(num) + "\n")

write_list_to_file(a, "log_a.txt")
write_list_to_file(b, "log_b.txt")
```

Name of the Refactoring: Extract Function

Functions

DRY – DON’T REPEAT YOURSELF!

Functions

Duplication is dangerous, because

- it increases the chance of making mistakes.
- when requirements change, all duplicated code sections have to be adjusted. It becomes easy to forget one of the implementations.

“Duplication may be the root of all evil in software.”
– Robert C. Martin

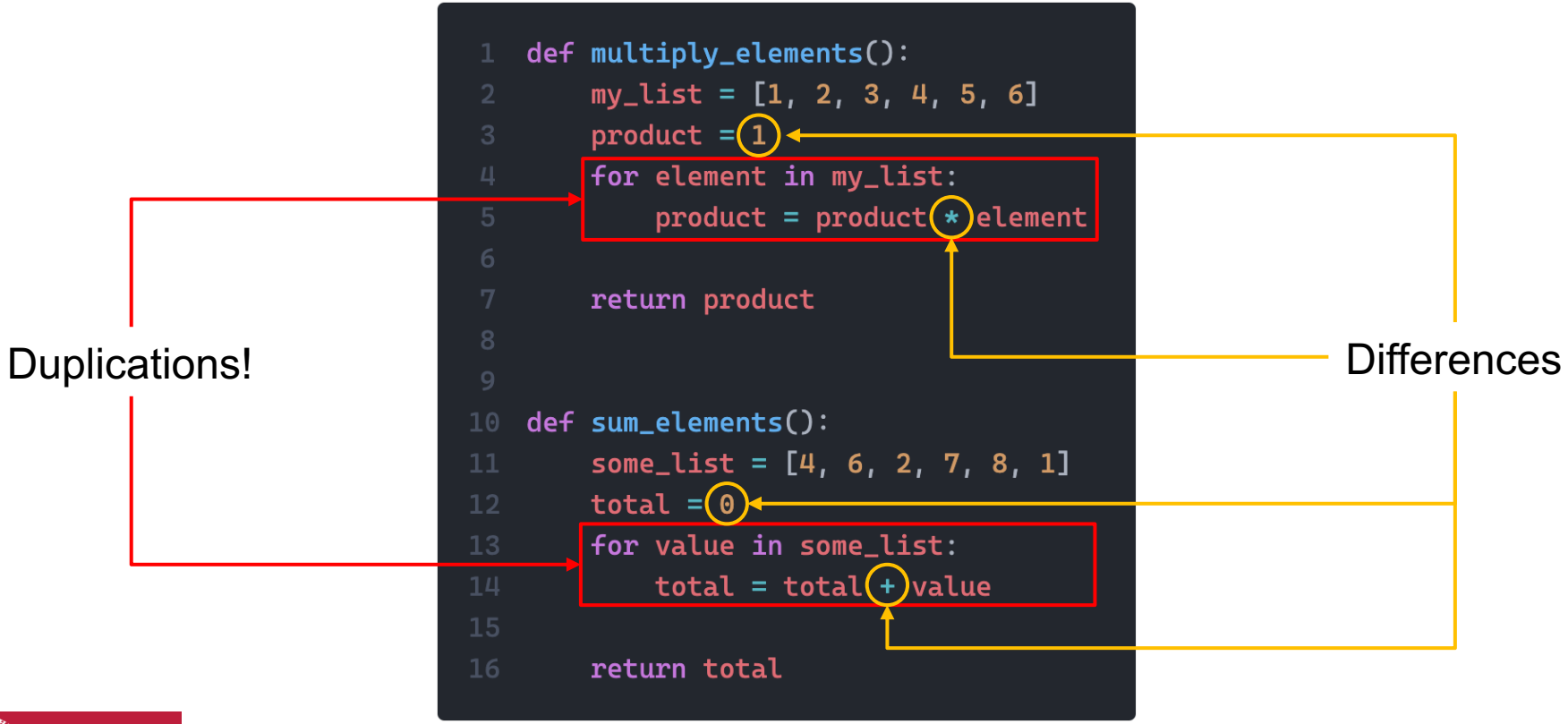
Functions

Challenge:
Spot the duplication!

Functions

```python
def multiply_elements():
    my_list = [1, 2, 3, 4, 5, 6]
    product = 1
    for element in my_list:
        product = product * element

    return product


def sum_elements():
    some_list = [4, 6, 2, 7, 8, 1]
    total = 0
    for value in some_list:
        total = total + value

    return total
```

Functions


Duplications!
Differences
1 def multiply_elements():
2 my_list = [1, 2, 3, 4, 5, 6]
3 product = 1
4 for element in my_list:
5 product = product * element
6
7 return product
8
9
10 def sum_elements():
11 some_list = [4, 6, 2, 7, 8, 1]
12 total = 0
13 for value in some_list:
14 total = total + value
15
16 return total

Functions

Extract the duplications and parameterize the differences!

Functions

```python
BinaryOperator = Callable[[Number, Number], Number]


def accumulate(
    iterable: Iterable[Number],
    operator: BinaryOperator,
    initial: Number,
) -> Number:
    for element in iterable:
        initial = operator(initial, element)

    return initial
```

Functions

```python
def multiply(a, b):
    return a * b


def add(a, b):
    return a + b

```

```python
def multiply_elements():
    my_list = [1, 2, 3, 4, 5, 6]
    return accumulate(my_list, multiply, 1)


def sum_elements():
    some_list = [4, 6, 2, 7, 8, 1]
    return accumulate(some_list, add, 0)
```

Functions

```python
@dataclass
class TemperatureData:
    datetime: datetime
    value_in_kelvin: float

    @staticmethod
    def from_csv(csv: str) -> "TemperatureData":
        dt, kelvin = csv.split(",")
        return TemperatureData(datetime.fromisoformat(dt), float(kelvin))


def write_celsius_values_in_timeframe_to_log(start: datetime, end: datetime) -> None:
    data = []
    for line in (DIR / "values.csv").read_text().splitlines():
        data.append(TemperatureData.from_csv(line))

    celsius_values: list[float] = []
    for d in data:
        if not math.isnan(d.value_in_kelvin):
            if start < d.datetime < end:
                celsius_values.append(d.value_in_kelvin - 273.15)

    (DIR / "logfile.txt").write_text("\n".join(map(str, celsius_values)))
```

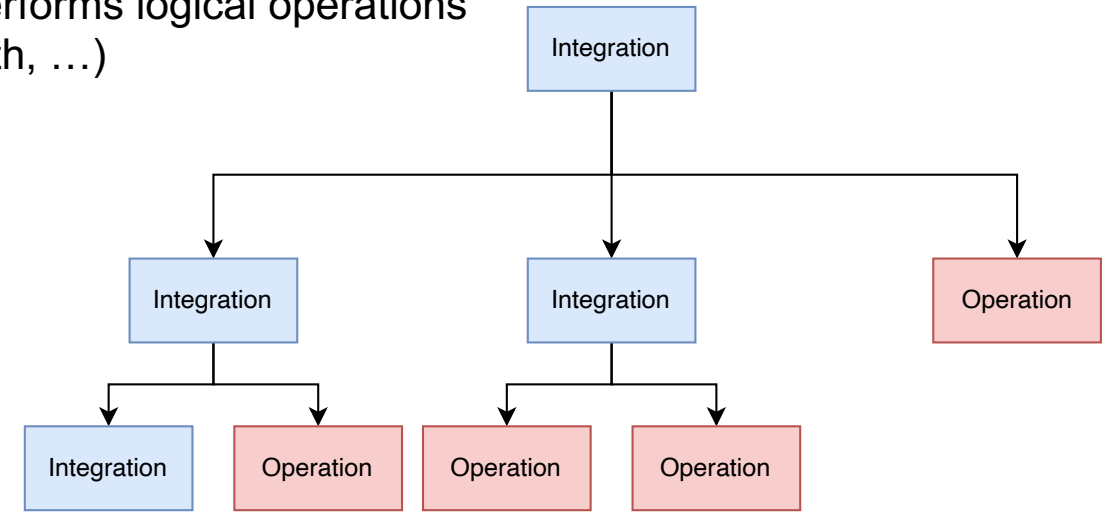
Functions

Functions should only have one task!

Functions

The Integration-Operation-Segregation-Principle (IOSP)

- Functions should perform **only integrations** or **only operations**
- An integration is a function that calls other integration or operation functions
- An operation is a function that performs logical operations (e.g. if statements, for loops, math, …)



Functions

```python
def read_temperature_data(path: Path) -> list[TemperatureData]:
    return [TemperatureData.from_csv(line) for line in path.read_text().splitlines()]

def is_valid_value(d: TemperatureData) -> bool:
    return not math.isnan(d.value_in_kelvin)

def is_in_timeframe(d: TemperatureData, start: datetime, end: datetime) -> bool:
    return start < d.datetime < end

def celsius(d: TemperatureData) -> float:
    return d.value_in_kelvin - 273.15

def celsius_values_in_timeframe(
    data: list[TemperatureData], start: datetime, end: datetime
) -> list[float]:
    return [
        celsius(d) for d in data if is_valid_value(d) and is_in_timeframe(d, start, end)
    ]

def write_results(result_file: Path, values: list[float]) -> None:
    result_file.write_text("\n".join(map(str, values)))

if __name__ == "__main__":
    data = read_temperature_data(INPUT_FILE)
    celsius_values = celsius_values_in_timeframe(data, START_DATE, END_DATE)
    write_results(RESULT_FILE, celsius_values)
```

Functions

Function Arguments

- Attempt to use no more than 3 arguments
- Use custom data structures to combine arguments
- Many arguments are an indicator that a function does too much
- Avoid boolean flags as they don’t express intent

Functions

```
1  def throw_trajectory(v0, h, a, t, g=10.0):
2      x = v0 * math.cos(a) * t
3      return x, -0.5 * g / (((v0 * math.cos(a)) ** 2)) * x**2 + math.tan(a) * x + h
```

Functions

```python
def throw_trajectory(initial_speed, initial_height, throwing_angle, time, gravity=10.0):
    x = horizontal_position(initial_speed, throwing_angle, time)
    y = vertical_position(x, initial_speed, initial_height, throwing_angle, gravity)
    return (x, y)

def horizontal_position(initial_speed, throwing_angle, time):
    return initial_speed * math.cos(throwing_angle) * time

def vertical_position(
    horizontal_position, initial_speed, initial_height, throwing_angle, gravity
):
    a, b = coefficients_of_throw_function(initial_speed, throwing_angle, gravity)
    return quadratic_function(a, b, initial_height, horizontal_position)

def quadratic_function(a, b, c, x):
    return a * x ** 2 + b * x + c

def coefficients_of_throw_function(initial_speed, throwing_angle, gravity):
    a = -0.5 * gravity / (((initial_speed * math.cos(throwing_angle)) ** 2))
    b = math.tan(throwing_angle)
    return a, b
```

Functions

- Steps are clearer now, follows IOSP
- Still a lot of parameters that make the code hard to read

Functions

- Grouped related parameters together into custom data class
- Tradeoff:
  easier to read,
  but a lot more code!

```python
@dataclass
class Throw:
    speed: float
    height: float
    angle: float

def throw_trajectory(throw, time, gravity=10.0):
    x = horizontal_position(throw, time)
    y = vertical_position(throw, gravity, x)
    return (x, y)

def horizontal_position(throw, time):
    return throw.speed * math.cos(throw.angle) * time

def vertical_position(throw, gravity, horizontal_position):
    a, b = coefficients_of_throw_function(throw, gravity)
    return quadratic_function(a, b, throw.height, horizontal_position)

def quadratic_function(a, b, c, x):
    return a * x ** 2 + b * x + c

def coefficients_of_throw_function(throw, gravity):
    a = -0.5 * gravity / (((throw.speed * math.cos(throw.angle)) ** 2))
    b = math.tan(throw.angle)
    return a, b
```

Names of the Refactorings:
Extract Function,
Replace Parameter with Query

Functions

```python
def print_shape_area(shape):
    if isinstance(shape, Square):
        print(f"The area is {shape.side_length ** 2}")
    elif isinstance(shape, Circle):
        print(f"The area is {shape.radius ** 2 * math.pi}")
    elif isinstance(shape, Rectangle):
        print(f"The area is {shape.width * shape.height}")
```

Functions

```python
class Shape(Protocol):
    def area(self) -> float:
        pass

@dataclass
class Square:
    side_length: float

    def area(self) -> float:
        return self.side_length ** 2


def print_shape_area(shape: Shape):
    print(f"The area is {shape.area()}")
```

Name of the Refactoring:
Replace Conditional with Polymorphism

Formatting

- Code is read much more often than written
- Well formatted code helps us to focus on the essentials

Formatting

```python
def important_function():
    hello = ("hello" +
        "world"
    )

def other_function(num):
    return [
    1, 2,
    3,

        num]
```

Formatting

```python
def important_function():
    hello = "hello" + "world"


def other_function(num):
    return [1, 2, 3, num]
```

Formatting

```python
def very_important_function(
    too: int,
    many: float,
    args: str,
    to: str,
    keep: tuple[int, int],
    track: dict[str, str],
    of: str,
):
    pass
```

Formatting

Look out for

- vertical size
- horizontal size
- consistent spacing

Python tips and tricks

Type hinting

Python supports **type hints** since version 3.5

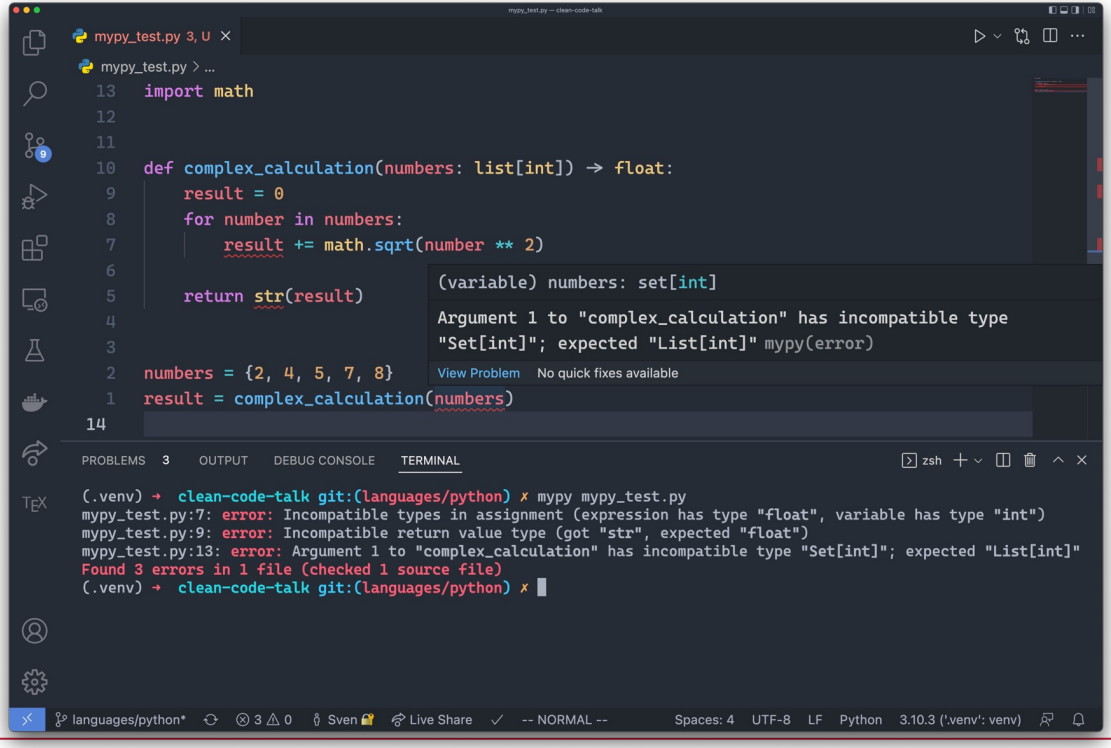
- Type hints communicate which types work with your functions
- Readers and users of your code won't have to guess which types are expected
- Use at least in interfaces intended to be used by others
- Recommendation: use type hints everywhere, they will help you understand your older code as well!

Type hinting

```python
BinaryOperator = Callable[[Number, Number], Number]


def accumulate(
    iterable: Iterable[Number],
    operator: BinaryOperator,
    initial: Number,
) -> Number:
    for element in iterable:
        initial = operator(initial, element)

    return initial
```

Type hinting

Use *mypy* to check correct type usage across your code base



Formatting

Use a PEP8 conformant formatter like *autopep8* or *black*

Recommendation: *black*

Linting

Linters come with many code style rules for naming, formatting and more

Recommendation: *ruff*

Let’s get to work!

The Gilded Rose Kata

Gilded Rose