

Clean code in Python

EuroPython

July 2016 - Bilbao, Spain


Mariano Anaya





/me

- Python developer
- Interests
 - Linux
 - Software development
 - Software Architecture / system design



 /rmariano

 @rmarianoa

 marianoanaya at gmail dot com

def

“You know you are working on clean code when each routine you read turns out to be pretty much what you expected.

You can call it *beautiful code* when the code also makes it look like the language was made for the problem.”

Ward Cunningham

In Python: magic methods → “Pythonic” code

Introduction / `__init__`

- What is “clean code”?
 - Does one thing well
 - Every $f(x)$ does what you'd expect
- Why is it important?
 - Code quality => Software quality
 - Readability
 - Agile development
 - Code: blueprint

What is not clean code

- Complex, obfuscated code
- Duplicated code
- Code that is not *intention revealing*

...Technical Debt

Meaning

```
def elapse(year):  
    days = 365  
    if year % 4 == 0 or (year % 100 == 0 and year % 400 == 0):  
        days += 1  
    for day in range(1, days + 1):  
        print("Day {} of {}".format(day, year))
```

Meaning and logic separation

```
def elapse(year):  
    days = 365  
    if year % 4 == 0 or (year % 100 == 0 and year % 400 == 0): ..... ?  
        days += 1  
    for day in range(1, days + 1):  
        print("Day {} of {}".format(day, year))
```

```
def elapse(year):  
    days = 365  
    if is_leap(year):  
        days += 1  
    ...
```

```
def is_leap(year):  
    ...
```

Duplicated code

- Often caused by the lack of meaningful abstractions
- Unclear patterns usually drive to code duplication

Problems:

- Hard to maintain, change, adapt
- Error prone

DRY principle

Don't **R**epeat **Y**ourself!

- Avoid code duplication at all cost
- Proposed solution: *decorators*

Duplicated code: decorators

General idea: take a function and modify it, returning a new one with the changed logic.

```
def decorator(original_function):  
    def inner(*args, **kwargs):  
        # modify original function, or add extra logic  
        return original_function(*args, **kwargs)  
    return inner
```

```
def update_db_indexes(cursor):
    commands = (
        """REINDEX DATABASE transactional""",
    )
    try:
        for command in commands:
            cursor.execute(command)
    except Exception as e:
        logger.exception("Error in update_db_indexes: %s", e)
        return -1
    else:
        logger.info("update_db_indexes run successfully")
        return 0
```

```
def move_data_archives(cursor):
    commands = (
        """INSERT INTO archive_orders SELECT * from orders
        WHERE order_date < '2016-01-01' """ ,
        """DELETE from orders WHERE order_date < '2016-01-01'
        """,)
    try:
        for command in commands:
            cursor.execute(command)
    except Exception as e:
        logger.exception("Error in move_data_archives: %s", e)
        return -1
    else:
        logger.info("move_data_archives run successfully")
        return 0
```

```
def db_status_handler(db_script_function):
    def inner(cursor):
        commands = db_script_function(cursor)
        function_name = db_script_function.__qualname__
        try:
            for command in commands:
                cursor.execute(command)
        except Exception as e:
            logger.exception("Error in %s: %s", function_name, e)
            return -1
        else:
            logger.info("%s run successfully", function_name)
            return 0
    return inner
```

```
@db_status_handler
```

```
def update_db_indexes(cursor):
```

```
    return (
```

```
        """REINDEX DATABASE transactional""",
```

```
    )
```

```
@db_status_handler
```

```
def move_data_archives(cursor):
```

```
    return (
```

```
        """INSERT INTO archive_orders SELECT * from orders  
WHERE order_date < '2016-01-01' """,
```

```
        """DELETE from orders WHERE order_date < '2016-01-01'
```

```
        """,
```

```
    ,
```

```
    )
```

Implementation details

- Abstract implementation details
- Separate them from business logic
- We could use:
 - Properties
 - Context managers
 - Magic methods

```
class PlayerStatus:
```

```
...
```

```
def accumulate_points(self, new_points):
```

```
    current_score = int(self.redis_connection.get(self.key) or 0)
```

```
    score = current_score + new_points
```

```
    self.redis_connection.set(self.key, score)
```

```
...
```

```
player_status = PlayerStatus()
```

```
player_status.accumulate_points(20)
```



```
class PlayerStatus:
```

```
...
```

```
def accumulate_points(self, new_points):
```

```
→ current_score = int(self.redis_connection.get(self.key) or 0)
```

```
    score = current_score + new_points ←
```

```
→ self.redis_connection.set(self.key, score)
```

```
...
```

```
-- implementation details
```

```
-- business logic
```

The kind of access I'd like to have

```
player_status.accumulate_points(20)
```

```
player_status.points += 20
```

```
...
```

```
print(player_status.points)
```

```
player_status.points = 100
```

How to achieve it

```
class PlayerStatus:
```

```
    @property
```

```
    def points(self):
```

```
        return int(self.redis_connection.get(self.key) or 0)
```

```
    @points.setter
```

```
    def points(self, new_points):
```

```
        self.redis_connection.set(self.key, new_points)
```

@property

- Compute values for objects, based on other attributes
- Avoid writing methods like `get_*`(), `set_*`()
- Use Python's syntax instead

Looking for elements

```
class Stock:  
    def __init__(self, categories=None):  
        self.categories = categories or []  
        self._products_by_category = {}
```

```
def request_product_for_customer(customer, product, current_stock):
    product_available_in_stock = False
    for category in current_stock.categories:
        for prod in category.products:
            if prod.count > 0 and prod.id == product.id:
                product_available_in_stock = True
    if product_available_in_stock:
        requested_product = current_stock.request(product)
        customer.assign_product(requested_product)
    else:
        return "Product not available"
```

```
def request_product_for_customer(customer, product, current_stock):  
    product_available_in_stock = False  
    for category in current_stock.categories:  
        for prod in category.products:  
            if prod.count > 0 and prod.id == product.id:  
                product_available_in_stock = True  
    if product_available_in_stock:  
        requested_product = current_stock.request(product)  
        customer.assign_product(requested_product)  
    else:  
        return "Product not available"
```

Python was made for the problem

```
def request_product_for_customer(customer, product, current_stock):  
    if product in current_stock:  
        requested_product = current_stock.request(product)  
        customer.assign_product(requested_product)  
    else:  
        return "Product not available"
```


The magic method

```
product in current_stock
```

Translates into:

```
current_stock.__contains__(product)
```

Looking for elements

```
class Stock:
    ...
    def __contains__(self, product):
        self.products_by_category()
        available = self.categories.get(product.category)
        ...
```

Maintaining state

- Some functions might require certain pre-conditions to be met before running
- ... and we might also want to make sure to run other tasks upon completion.

Context Managers

```
class DBHandler:
    def __enter__(self):
        stop_database_service()
        return self

    def __exit__(self, *exc):
        start_database_service()
    ...
with DBHandler():
    run_offline_db_backup()
```

Context Managers

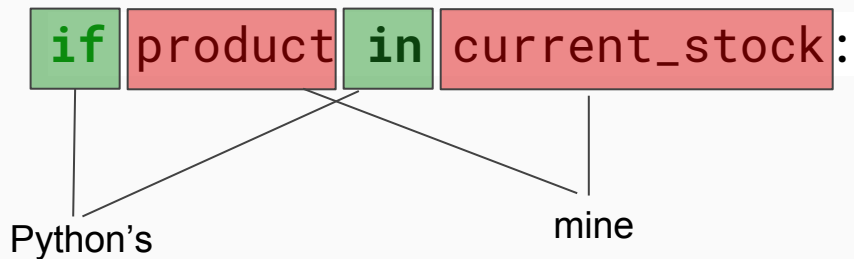
```
class db_status_handler(contextlib.ContextDecorator):  
    def __enter__(self):  
        stop_database_service()  
        return self
```

```
    def __exit__(self, *exc):  
        start_database_service()
```

```
@db_status_handler()  
def offline_db_backup():  
    ...
```

- Import contextlib
- Python 3.2+

Pythonic



A more Pythonic code, should blend with Python's words.

Summary

- Python's magic methods help us write more pythonic code.
 - As well as context managers do.
 - Use them to abstract the internal complexity and implementation details.
- Properties can enable better readability.
- Decorators can help to:
 - Avoid duplication
 - Separate logic

Achieving quality code

- PEP 8
 - Define coding guidelines for the project
 - Check automatically (as part of the CI)
- Docstrings (PEP 257)/ Function Annotations (PEP 3107)
- Unit tests
- Tools
 - **Pycodestyle**, Flake8, pylint, radon
 - coala

More info

- Python Enhancement Proposals: PEP 8, PEP 257, PEP 343
 - <https://www.python.org/dev/peps/>
- Clean Code, by Robert C. Martin
- Code Complete, by Steve McConnell
- Pycodestyle: <https://github.com/PyCQA/pycodestyle>
- PyCQA: <http://meta.pycqa.org/en/latest/>

Questions?

Thanks.