RST Python Developer Guide Documentation

Release 0.0.1

Piotr Poteralski

Jan 31, 2018

Contents:

CHAPTER 1

Introduction

This is Employee Handbook for our current and future developers.

CHAPTER 2

Status

This is a work in progress. Actually, what you are reading now is a very early draft.

CHAPTER 3

About RST Software House

We are software house from Wroclaw we build online systems & we love to help startups. We create user-friendly web & mobile applications using best programming practices.

CHAPTER 4

tldr

- Be Agile.

  All good practices come from Agile manifesto. be familiar with this. Read more in Agile chapter.

- Write code for others (including "future you"). Check Code Style chapter.
- Practice Test-Driven Development (TDD).

  1. TDD (practiced correctly) helps come with better software design.
  2. It give us confidence that our product work properly, and that we can refactor them without fear of breaking things beyond repair.

  Read more in testing chapter.

- Create Ubiquitous Language for project.

  All project members should be familiar with business logic nomenclature.

- Practice "Documentation Driven Development". See documentation chapter in this document.

  Write documentation alongside the code. The simple act of trying to explain to others what we are trying to achieve with our software, and how, helps us clarify our thinking. If it's too hard to explain, then it's probably badly architected, designed or implemented.

- Practice Domain Driven Design (DDD).

  See the Architecture chapter in this document.

- Use on click test and deploy.

  We use docker for CI and deployment configuration. Read DevOps chapter for more information

CHAPTER 5

Table of Content

5.1 Introduction

One of the most important principle that should be understood and applied in our projects is Zen Of Python. Is collection of 20 software principles that influences the design of Python Programming Language. This document best reflects the philosophy of this language.

5.1.1 Zen of Python

- Beautiful is better than ugly.
- Explicit is better than implicit.
- Simple is better than complex.
- Complex is better than complicated.
- Flat is better than nested.
- Sparse is better than dense.
- Readability counts.
- Special cases aren't special enough to break the rules.
- Although practicality beats purity.
- Errors should never pass silently.
- Unless explicitly silenced.
- In the face of ambiguity, refuse the temptation to guess.
- There should be one—and preferably only one—obvious way to do it.
- Although that way may not be obvious at first unless you're Dutch.
- Now is better than never.

- Although never is often better than right now.[5]
- If the implementation is hard to explain, it's a bad idea.
- If the implementation is easy to explain, it may be a good idea.
- Namespaces are one honking great idea—let's do more of those!

5.2 Principles

In short, where this principles actually pay off:

- Readability—Having simple objects defined based on what they do make our life a lot easier coming back to the code we wrote months ago.
- Testability—Since the signatures of the objects are well-defined and very much contained, creating unit and integration tests is super straightforward and fast.
- Robustness—Simple objects allow you to focus on the specificities of each task individually and reduces the amount of input/output variables you need to consider at any given time. Thus making the whole process less error-prone.
- Onboarding—This approach has proven itself very helpful when handing down knowledge as the thought process is like a standard line protocol instead of a wibbly wobbly mix of instructions.
- Caching Layers—For scaling scenarios, you can cache objects using solutions such as Redis just by adding 2/3 lines of code to an object. As such you don't need interfere with the rest of the codebase.
- Reusability—Given all the examples we've seen, I think this speaks for itself.
- Less Issues—Considerably reduces cyclomatic complexity, hence, reducing the amount of defects

5.2.1 Single Responsibility Principle

Here is an example of violating this rule:

```
class CarWashService:
    def __init__(self, sms_sender):
        self.sms_sender = sms_sender

    def __call__(self, card_id, customer_id):
        car = Car.objects.get(id=card_id)
        customer = Customer.objects.get(customer_id)
        if car.wash_required:
            car.washed = True
            self.sms_sender.send(mobile_phone=customer.phone, text=f"Car %{car.plate}␣
↪whashed.")
```

After refactor:

```
class CarWashService:
    def __init__(self, repository, notifier):
        self.repository = repository
        self.notifier = notifier

    def __call__(self, car_id, customer_id):
        car = self.repository.get_car(car_id)
        customer = self.repository.get_customer(customer_id)
        if car.wash_required:
            car.washed = True
            self.notifier.wash_completed(customer.phone, car.plate)
```

5.2.2 Open-Closed Principle

example:

```
class Rectangle(object):

    def __init__(self, width, height):
        self.width = width
        self.height = height
```

```
class AreaCalculator(object):

    def __init__(self, shapes):

        assert isinstance(shapes, list), "`shapes` should be of type `list`."
        self.shapes = shapes

    @property
    def total_area(self):
        total = 0
        for shape in self.shapes:
            total += shape.width * shape.height

        return total

def main():
    shapes = [Rectangle(2, 3), Rectangle(1, 6)]
    calculator = AreaCalculator(shapes)
    print(calculator.total_area)
```

after refactor You can see that it will be easy to extend the functionality:

```
from abc import ABCMeta, abstractproperty

class Shape(object):
```

```
    __metaclass__ = ABCMeta

    @abstractproperty
    def area(self):
        pass

class Rectangle(Shape):

    def __init__(self, width, height):
        self.width = width
        self.height = height

    @property
    def area(self):
        return self.width * self.height

class AreaCalculator(object):

    def __init__(self, shapes):
        self.shapes = shapes

    @property
    def total_area(self):
        total = 0
        for shape in self.shapes:
            total += shape.area
        return total

def main():
    shapes = [Rectangle(1, 6), Rectangle(2, 3)]
    calculator = AreaCalculator(shapes)

    print(calculator.total_area)
```

5.2.3 Liskov Substitution Principle

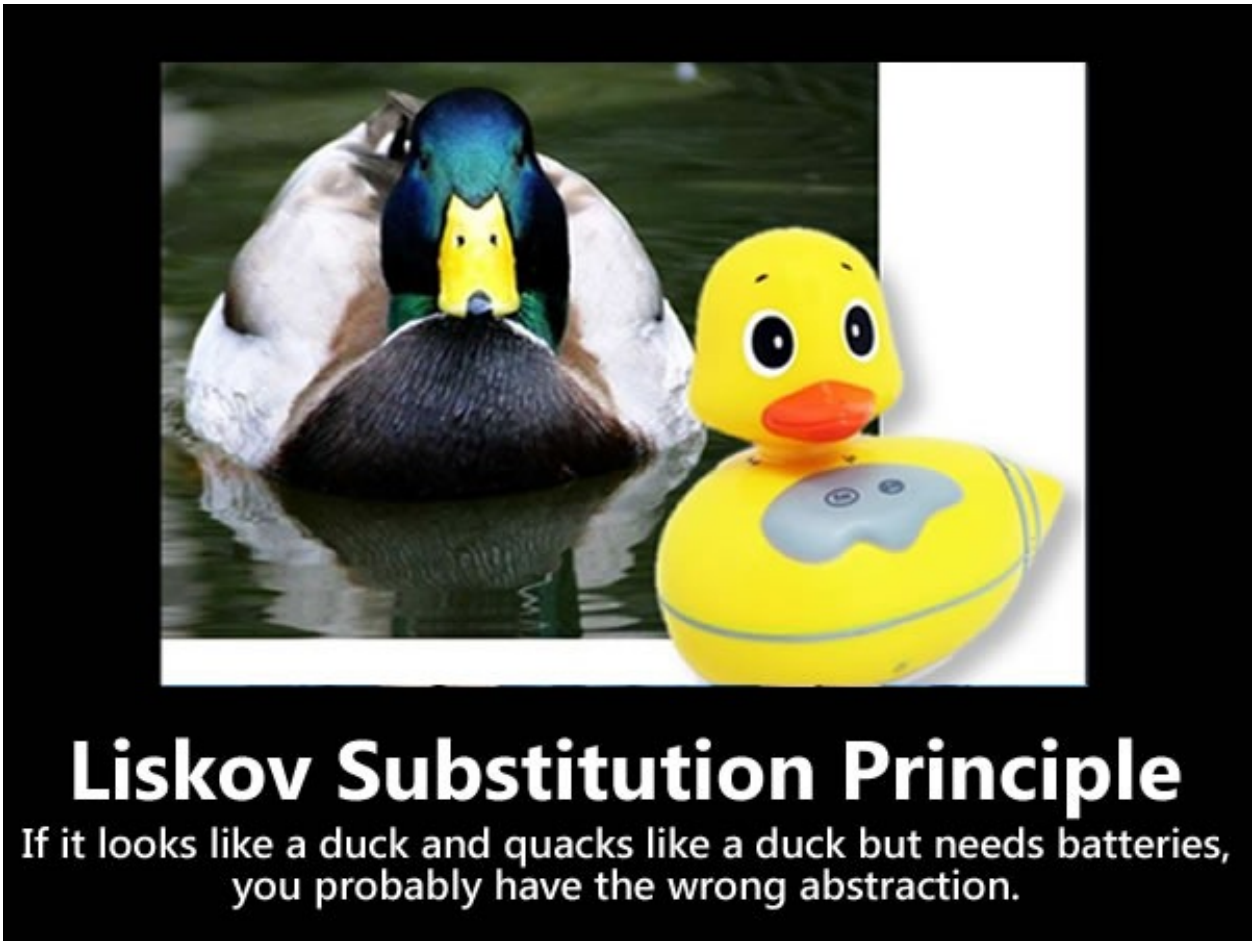
This is a scary term for a very simple concept. It's formally defined as "If S is a subtype of T, then objects of type T may be replaced with objects of type S (i.e., objects of type S may substitute objects of type T) without altering any of the desirable properties of that program (correctness, task performed, etc.)." That's an even scarier definition.

The best explanation for this is if you have a parent class and a child class, then the base class and child class can be used interchangeably without getting incorrect results. This might still be confusing, so let's take a look at the classic Square-Rectangle example. Mathematically, a square is a rectangle, but if you model it using the "is-a" relationship via inheritance, you quickly get into trouble.

Example:

```
class Rectange:

    def __init__(self):
        self.width = 0
        self.height = 0

    def set_width(self, width):
        self.width = width

    def set_height(self, height):
```

```
        self.height = height

    def set_color(self, color):
        self.color = color

    def render(self):
        # ...

class Squere(Rectange):

    def set_width(self, width):
        self.width = width
        self.height = width

    def set_height(self, height):
        self.height = height
        self.width = height
```



After refactor:

```
class Shape:
    def set_color(self, color):
        self.color = color

    def render(self):
```

```
    # ...

class Rectange(Shape):
    def __init__(self, width, heigh):
        self.width = width
        self.height = heigh

    def get_area(self):
        return self.width * self.height

class Square(Shape):
    def __init__(self, length):
        self.length

    def get_area(self):
        return self.length * self.length
```

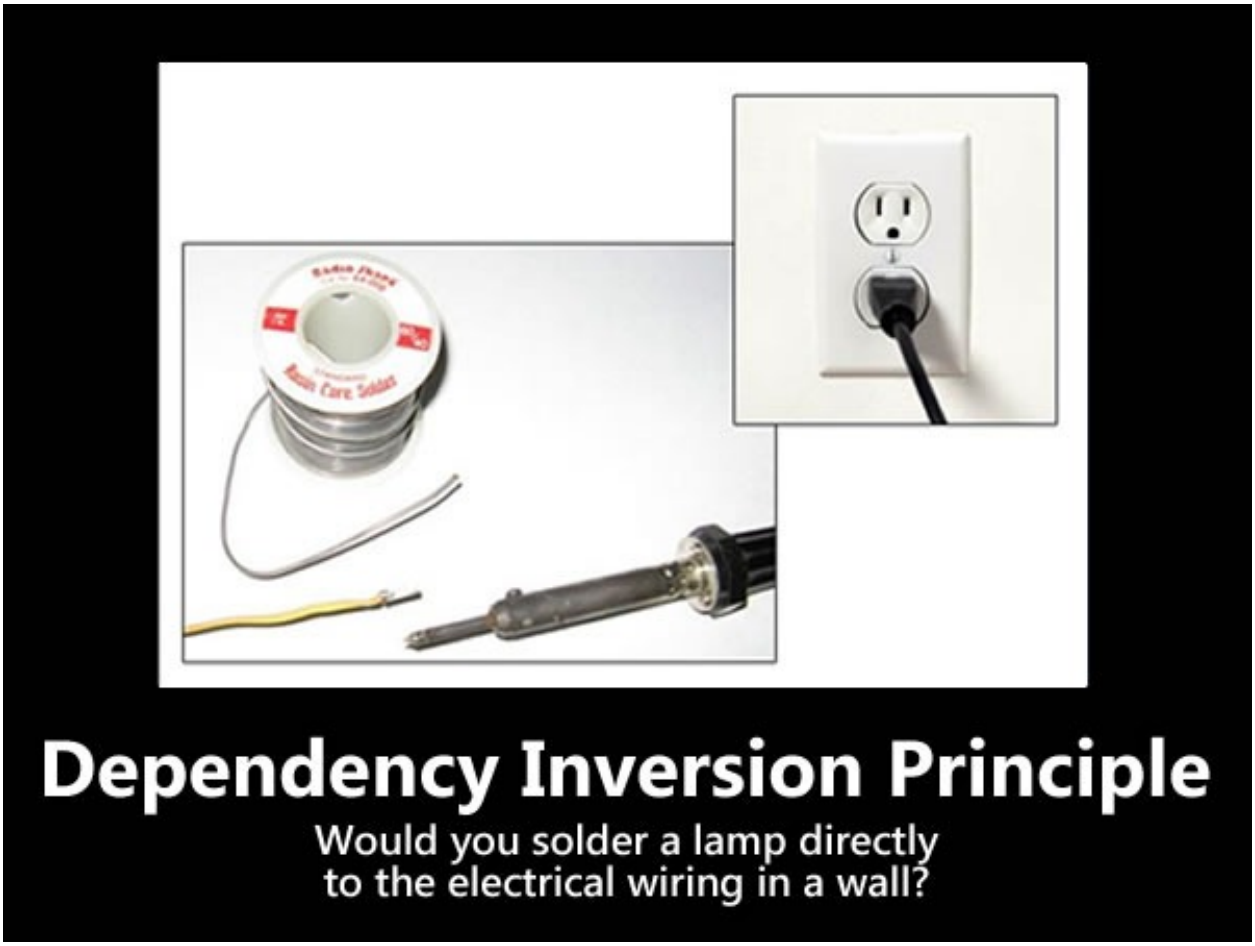
5.2.4 Interface Segregation Principle

5.2.5 Dependency Inversion Principle

`Depend of abstractions. Do not depend upon concretion.`

Example with Global State Problem, Implicit Dependency Problem and Concrete API:

```
class CarWashService:
    def __init__(self, repository):
        self.repository = repository

    def __call__(self, car_id, customer_ids):
        car_wash_job = CarWashJob(car_id, customer_id)
        self.repository.put(car_wash_job)
        SMSNotifier.send_sms(car_wash_job)
```



After refactor:

```
class CarWashService:
    def __init__(self, notifier, repository):
        self.repository = repository
        self.notifier = notifier

    def __call__(self, car_id, customer_id):
        car_wash_job = CarWashJob(car_id, customer_id)
        self.repository.put(car_wash_job)
        self.notifier.job_completed(car_wash_job)
```

5.3 Style

5.3.1 Little Zen for Code Style

Barry Warsaw, one of the core Python developers, once said that it frustrated him that "The Zen of Python" ([PEP 20][pep20]) is used as a style guide for Python code, since it was originally written as a poem about Python's **internal** design. That is, the design of the language and language implementation itself. One can acknowledge that, but a few of the lines from PEP 20 serve as pretty good guidelines for idiomatic Python code, so we'll just go with it.

Beautiful is better than ugly

This one is subjective, but what it usually amounts to is this: will the person who inherits this code from you be impressed or disappointed? What if that person is you, three years later?

Explicit is better than implicit

Sometimes in the name of refactoring out repetition in our code, we also get a little bit abstract with it. It should be possible to translate the code into plain English and basically understand what's going on. There shouldn't be an excessive amount of "magic".

Flat is better than nested

This one is really easy to understand. The best functions have no nesting, neither by loops nor *if* statements. Second best is one level of nesting. Two or more levels of nesting, and you should probably start refactoring to smaller functions.

Also, don't be afraid to refactor a nested if statement into a multi-part boolean conditional.

Readability counts

Don't be afraid to add line-comments with #. Don't go overboard on these or over-document, but a little explanation, line-by-line, often helps a whole lot. Don't be afraid to pick a slightly longer name because it's more descriptive. No one wins any points for shortening "*response*" to "*rsp*". Use doctest-style examples to illustrate edge cases in docstrings. Keep it simple!

Errors should never pass silently

The biggest offender here is the bare *except: pass* clause. Never use these. Suppressing **all** exceptions is simply dangerous. Scope your exception handling to single lines of code, and always scope your *except* handler to a specific type. Also, get comfortable with the *logging* module and *log.exception(...)*.

If the implementation is hard to explain, it's a bad idea

This is a general software engineering principle – but applies very well to Python code. Most Python functions and objects can have an easy-to-explain implementation. If it's hard to explain, it's probably a bad idea. Usually you can make a hard-to-explain function easier-to-explain via "divide and conquer" – split it into several functions.

Testing is one honking great idea

OK, we took liberty on this one – in "The Zen of Python", it's actually "namespaces" that's the honking great idea.

But seriously: beautiful code without tests is simply worse than even the ugliest tested code. At least the ugly code can be refactored to be beautiful, but the beautiful code can't be refactored to be verifiably correct, at least not without writing the tests! So, write tests! Please!

5.3.2 Strict Rules

- Each project must have a flake8 linter.
- Each project must follow PEP8 (with 99 chars limit).
- Docstrings must follow PEP257.
- Do not use wildcard imports.
- Don't use single letter variable names, unless within a list comprehension.
- Use Python idioms.
- Avoid redundant labeling.
- Prefer reverse notation.
- Sort and divide import statements.

5.3.3 Usefull tools

- https://bitbucket.org/pytest-dev/pytest-pep8
- https://github.com/fschulze/pytest-flakes
- https://github.com/cbrueffer/pep8-git-hook

5.4 Code Review

5.4.1 Before you create PR

make sure:

- acceptance criteria from task are met
- you wrote automated tests and they pass
- remove unnecessary debugging code
- you assigned at least 2 persons as reviewers to your PR
- code meets guidelines

If you are a reviewer:

- Do code reviews as soon as possible. Of course your tasks have higher priority, but put yourself in another person's place - you want to create a PR and get reviewed.
- Check code according to the following guides.
- Click "Approve" when you accept changes.

5.4.2 Basics

- Identify places which were over-engineered.
- Merge is done by the reviewer who approves the PR as the last one, except for a situation when deployment of the code in PR requires some additional configuration in target environment (setting env variables, migrations etc).

- Identify ways to simplify the code while still solving the problem.
- If discussions turn too philosophical or academic, move the discussion face to face or hipchat. The outcome of this discussion should be placed in the comment.
- Offer alternative implementations, but assume the author already considered them.
- **What do you think about changing it one-line if-statement?**
  - In our guidelines we us ‘ instead of “, is there any particular reason you used “ here?
  - Try to understand the author’s perspective.
- Don’t be sarcastic
- You can mark you comment as optional using [opt]

5.4.3 When your code is reviewed:

- Don’t take it personally. It’s a review of the code, not you.
- Try to understand the reviewer’s perspective
- You should respond to almost every comment - this means that every comment from a reviewer should result in a response or a code change.
- Can merge if you have at least two accepts and all comments and discussions are exhausted.

5.4.4 When you review someone’s code

- Remember that you can ask for clarification.
- Avoid using terms that could be seen as referring to personal traits. [dumb, stupid]. Assume everyone is intelligent and well-meaning.
- Be explicit. Remember people don’t always understand your intentions.

5.5 Design Architecture

5.5.1 DDD Introduction

Coupling

Coupling is the degree of interdependence between software modules. All project should be loosely coupled. To do this you follow SOLID principles and useone of above clean architecture examples.

Tight Coupling symptoms

- fat controllers
- fat models
- hard to test
- hard to explain
- hard to read

- spaghetti code

Domain Driven Design Architecture List

- Clean Architecture
- Haxagonal Architecture (Ports and Adapters)
- Onion Architecture
- Screaming Architecture

Layered Structure

In DDD approach project should be layered to at least 3 parts.

- Application (glue code)
- Domain (Business Logic)
- Framework (Framework related Interfaces, all can be used as Open Source)

Characteristics

- Architectures should tell about the system.
- Not about the frameworks.
- New programmers should be able to learn all the use cases of the system.
- And still not know how the system is delivered.
- They may come to you and say: "We see some things that look sorta like models, but where are the views and controllers", and you should say: "Oh, those are details that needn't concern you at the moment, we'll show them to you later."

5.5.2 DDD Recipes

Common Project Structure

Framework layer

- utils.py
- commons.py
- generics.py
- base.py
- extensions.py
- adapters.py

Domain Layer

- services.py
- schemas.py
- models.py
- factories.py
- repositories.py
- exceptions.py

Application Layer

- views.py
- routes.py
- exceptions.py
- app.py
- wsgi.py
- tasks.py
- permissions.py
- authentications.py

Flask/SQLAlchemy Project Structure

TODO

Django Project Structure

TODO

5.6 Open Source

5.6.1 Our open source projects

We are the main developers of the following open source projects:

- SQLXerion (not packaged yet)
- Flask-AdminLogin (not packaged yet)
- Flask-FileAlchemy (not packaged yet)
- Flask-ImageAlchemy
- Flask-PyFCM
- Flask-Airbrake

5.6.2 Guides

- Project should have good coverage of test
- Project should have configured CI on travis or similiar
- All practices from this documents should be implemented

5.7 Process [WIP]

5.7.1 Idea

Agile Manifesto

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

http://agilemanifesto.org/

5.7.2 Prioritizing and scheduling tasks

Scrum & Kanban

TODO.

Bug tracking

IEEE Standard Classification for Software Anomalies:

- **Blocking**: Testing is inhibited or suspended pending correction or identification of suitable workaround.
- **Critical**: Essential operations are unavoidably disrupted, safety is jeopardized, and security is compromised.
- **Major**: Essential operations are affected but can proceed.
- **Minor**: Nonessential operations are disrupted.
- **Inconsequential**: No significant impact on operations.

Tools

TODO For open source project we should use github's issue page For private project we use Jirra

5.7.3 Versionning and releasing

(Semantic) Versionning

See:

- <http://semver.org/>
- <https://www.python.org/dev/peps/pep-0440/>

Releasing

TODO

5.8 Python Libraries

Favorite libraries:

- flask - Flask is a microframework for Python based on Werkzeug, Jinja 2 and good intentions.
- aiohttp - Asynchronous HTTP Client/Server for Python and asyncio
- apistar - A smart Web API framework, designed for Python 3.
- sqlalchemy - The Python SQL Toolkit and Object Relational Mapper
- marshmallow - is an ORM/ODM/framework-agnostic library for converting complex datatypes, such as objects, to and from native Python datatypes.
- injector - Python dependency injection framework, inspired by Guice.

5.9 Resources

The best sources of knowledge.

5.9.1 Books

- Domain-Driven Design: Tackling Complexity in the Heart of Software
- Clean Code: A Handbook of Agile Software Craftsmanship
- Extreme Programming Explained: Embrace Change
- Refactoring to Patterns
- The Clean Coder: A Code of Conduct for Professional Programmers
- Code Complete (Developer Best Practices

5.9.2 Blogs

- http://blog.cleancoder.com/ by Robert C. Martin (Uncle Bob)
- http://lucumr.pocoo.org/ by Armin Ronacher (Flask)
- https://emacsway.github.io by Ivan Zakrevskyi (XP and DDD expert)
- https://www.joelonsoftware.com by Joel Spolsky (Stack Overflow CEO)
- https://rhettinger.wordpress.com by Raymond Hettinger (Python core developer)

Articles - http://www.ballofcode.com/python/domain-driven-design/2013/12/22/exploring-domains-with-python - https://github.com/anthony-tresontani/methodic-python/blob/master/DomainDrivenDesign.rst - https://stevewedig.com/2014/07/31/value-objects-in-java-and-python/ - https://blog.fedecarg.com/2009/03/11/domain-driven-design-and-mvc-architectures/

5.9.3 stackoverflow questions

5.9.4 YouTube Videos

- Clean application architecture https://www.youtube.com/watch?v=NXzPRVLEmUE
- Deliver domain driven designs dynamically https://www.youtube.com/watch?v=p7PHOFRtI04
- The Clean Architecture in Python https://www.youtube.com/watch?v=DJtef410XaM
- Building highly decoupled systems in Python https://www.youtube.com/watch?v=3MEsh44XZDo
- Clean Architecture in Python (web) apps https://www.youtube.com/watch?v=4X1hNuW7WGo
- Beyond PEP 8 – Best practices for beautiful intelligible code https://www.youtube.com/watch?v=wf-BqAjZb8M

5.9.5 YouTube Channels

- https://www.youtube.com/user/pyconpl
- https://www.youtube.com/user/PyWaw

5.9.6 Podcasts

- https://talkpython.fm/ - A podcast on Python and related technologies

5.9.7 Sample Projects

- https://github.com/jordifierro/abidria-api
- https://github.com/MichaelDiBernardo/ddd-flask-example
- https://github.com/basco-johnkevin/ddd-python-django

5.10 Documentation

5.10.1 Glossary

The domain and the technical experts should create document of all the terminology that your team uses in code and issue tracker. after reading this, there should be no doubt what is happening in the code or what should be done in the task.

5.10.2 Readme

A README file at the root directory should give general information to both users and maintainers of a project. It should be raw text or written in some very easy to read markup, such as reStructuredText or Markdown. It should contain a few lines explaining the purpose of the project or library (without assuming the user knows anything about the project), the URL of the main source for the software. This file should also have information how to configure and run projects for multiple environments (local, prod, testing). This file is the main entry point for readers of the code.

example file:

```
# Project Title
One Paragraph of project description goes here

## Getting Started
These instructions will get you a copy of the project up and running on your local␣
↪machine for development and testing purposes. See deployment for notes on how to␣
↪deploy the project on a live system.

### Prerequisites
What things you need to install the software and how to install them
```
Give examples
```

### Installing
A step by step series of examples that tell you have to get a development env running
```
Give the example
```

## Running the tests
Explain how to run the automated tests for this system

## Deployment
Add additional notes about how to deploy this on a live system
```

5.10.3 API specification

All backend project should have documented endpoints in apiary.apib file. With this blueprint file you can already get a mock, documentation and test for your API. This file describe File should be always and up-to-date. Apiary ensures

example file:

```
FORMAT: 1A

# The Simplest API
This is one of the simplest APIs written in the **API Blueprint**.

### List All Questions [GET]
+ Response 200 (application/json)

    + Attributes (array[Question])

## Data Structures

### Question
+ question: Favourite programming language? (required)
```

```
+ published_at: `2014-11-11T08:40:51.620Z` (required)
+ url: /questions/1 (required)
+ choices (array[Choice], required)

### Choice
+ choice: Javascript (required)
+ url: /questions/1/choices/1 (required)
+ votes: 2048 (number, required)
```

5.10.4 Docstring

All domain classes should have docstring for class definition. We should create them also for complex classes and functions that will be used in other places.

```
class CarWashService:
    """
    This class describe how to wash a car.
    Service performs advanced and complex actions to wash the customer's car.
    Require repository that could be injected using Dependency Injection.
    After all, calls the function responsible for notifications.
    """
    def __init__(self, repository, notifier):
        self.repository = repository
        self.notifier = notifier

    def __call__(self, car_id, customer_id):
        car = self.repository.get_car(car_id)
        customer = self.repository.get_customer(customer_id)
        if car.wash_required:
            car.washed = True
            car.washed_at = utcnow()
            self.notifier.wash_completed(customer.phone, car.plate)
        return car
```

5.10.5 Type Hint

If there is such a possibility, we should use it wherever possible. This will allow showing explicitly what we expect and what will be returned.

```
class CarWashService:
    """
    This class describe how to wash a car.
    Service performs advanced and complex actions to wash the customer's car.
    Require repository that could be injected using Dependency Injection.
    """
    def __init__(self, repository: MongoRepository, notifier: SMSNotifier) -> None:
        self.repository = repository
        self.notifier = notifier

    def __call__(self, car_id: int, customer_id: int) -> Car:
        """
        :param car_id:              Unique Identifier of a Car
        :param customer_id:         Unique Identigier of a Customer
        :return:
```

```
        """
        car = self.repository.get_car(car_id)
        customer = self.repository.get_customer(customer_id)
        if car.wash_required:
            car.washed = True
            car.washed_at = utcnow()
            self.notifier.wash_completed(customer.phone, car.plate)
        return car
```

5.11 Refactoring [WIP]

5.11.1 The Boy Scout Rule

5.11.2 Refactoring Patterns

5.11.3 PyCharm tips and tricks

5.12 Modeling [WIP]

TODO

5.13 Testing [WIP]

5.13.1 Unit tests

5.13.2 Integration tests

5.13.3 Mocking

5.13.4 Patching

5.13.5 TDD

5.13.6 pytest

5.13.7 utils

5.14 DevOps

5.14.1 Continous Integration

5.14.2 Infrastructure

5.14.3 Tools

5.14.4 Scaling