Python quality engineering
A tour of best practices

Paris Open Source Summit - 7 December 2017

Stéfane Fermigier
Founder & CEO, Abilian - Enterprise Social Software

abilian
CONNECTED, WE WORK!

Who am I ?

- Stefane Fermigier, Python developer since 1996
- Organizer of the ~~PyData Paris~~ PyParis conference (2015-now)
- Founder of Abilian SAS
  - Python shop, developing business application (collaboration, CRM, workflow...)
  - R&D activity (Wendelin -> Olapy)

abilian

0 - Intro

abilian
CONNECTED, WE WORK!

Our goal with this session

“Improve the quality of your development process - producing better software at a lower cost - by identifying and implementing proven practices and tools, when appropriate for our context”

abilian

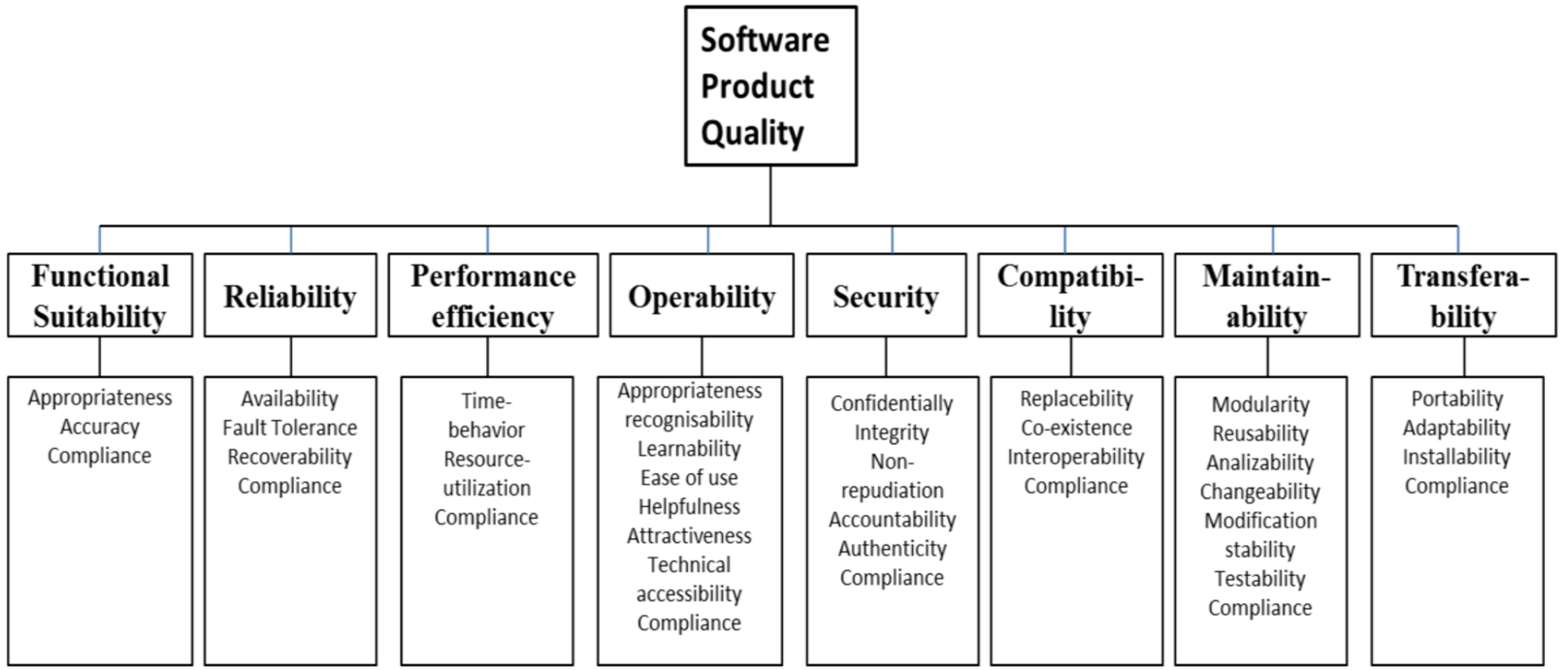
Some definitions

- Software quality ?
- Best practices ?

abilian

Software quality

- All the “-ities” that can be used to mesure software quality (ISO 25010):



Source: A Review of Software Quality Models for the Evaluation of Software Products - José P. Miguel, David Mauricio and Glen Rodríguez

abilian

Best practices?

- Depend (a lot) on the context, e.g.:
  - (data) science vs. web / enterprise
  - small / medium / large teams
  - monolithic vs. microservices
  - single instance vs. single product vs. product line vs. multiple projects
  - single / mono repo vs. multiple repos
  - framework used (some are “opiniated”, some not)
  - individual opinions and team consensus
  - …

abilian

Trade-offs

- Build vs. reuse vs. buy
  - Standard library vs. 3rd-party
- Quick prototyping / MVP vs. long-term maintainability
  - Aka “Move fast vs. don’t break things”)
  - Relevant concept: “technical debt”
- Human aspects vs. technical aspects
- …

abilian

Best practices as patterns

- Apply only on certains contexts
- Several alternatives
- Imply some trade-offs

1 - “Dev”

abilian
CONNECTED, WE WORK!

Environments and dependencies management

Goals

- Ensure each developer in a project can work in an environment (Python runtime + dependencies) that:
  - is close enough to the production environment
  - has all the tools needed for development (including testing, quality control…)
  - doesn’t interfere with other environments on the developer’s machine
- Ensure that a closely similar environment is used for development, QA (Continuous Integration, preproduction), and production
- Easily manage deliberate dependencies upgrade

Virtualenv

- virtualenv is part of the standard library
- I can create and manage copies of a full Python installation + place to hold your specific dependencies
- A virtualenv typically needs to be “activated” by running
  “`. <path_to_my_env>/bin/activate`”
  - This can be done “automagically” when using some advanced shell configs, e.g Oh-my-zsh or others
  - virtualenv-wrapper and other tools provide additional magic to manage your virtualenvs

pip

- pip is now the standard way to install, list, upgrade, uninstall… dependencies, from a given repository, from PyPI (a central package repository), from your own package repository, etc.
- It used to build dependencies from sources, using lower-level existing tools, but now is able to:
  - Use a binary format, “wheels”, introduced in 2013 (PEP 427)
  - Cache builds locally

Specifying dependencies

- If you're using pip, dependencies can be specified in two ways:
  - In the `setup.py` file - they will automatically be fetched and installed when installing your package
  - In a `requirements.txt` file - you then have to install them explicitly using "`pip install -r requirements.txt`"
  - A third way (!) is to have `setup.py` parse `requirements.txt` at packaging time
- You usually need also to specify development dependencies (`dev-dependencies.txt` but there are other conventions), test dependencies, optional dependencies, etc.

Loose or strict dependencies?

- loose, and sparse, dependencies for libraries, e.g.

```
Flask>0.12<1.0
```

- pinned, and complete, dependencies for applications, e.g.

```
Flask==0.12.2
Jinja2==2.10
Werkzeug==0.12.2
```

abilian

pip-tools, preq

- These tools help you manage efficiently dependencies upgrades
- Keep a set a strict dependencies updated from a loose set
  - `requirements.in` -> `requirements.txt`
  - or `setup.cfg -> requirements.txt`
- Can show the graph of a project's dependencies (including unspecified transitive dependencies)

abilian

Pipenv

- Single tool to unify virtualenv, pip and pip-tool like upgrades
- May or may not become the standard way to to things in the future

See also

- **pyenv**: installer for many (319 at this time) Python versions and variants
- **tox**: test runner that allows you to run your tests against several different environments

Style guides and coding standards

Wikipedia definition

- “**Coding conventions are a set of guidelines** for a specific programming language [or team, or project] that recommend programming style, practices, and methods for each aspect of a program written in that language [or for this team / project].
- These conventions usually cover file organization, indentation, comments, declarations, statements, white space, naming conventions, programming practices, programming principles, programming rules of thumb, architectural best practices, etc.
- **These are guidelines for software structural quality.** Software programmers are highly recommended to follow these guidelines to help improve the readability of their source code and **make software maintenance easier**.”

PEP8

- Python has one since 2001

Python »» Python Developer's Guide »» PEP Index »» PEP 8 -- Style Guide for Python Code

PEP 8 -- Style Guide for Python Code

PEP:	8
Title:	Style Guide for Python Code
Author:	Guido van Rossum <guido at python.org>, Barry Warsaw <barry at python.org>, Nick Coghlan <ncoghlan at gmail.com>
Status:	Active
Type:	Process
Created:	05-Jul-2001
Post-History:	05-Jul-2001, 01-Aug-2013

abilian

Enforcing a coding standard using formatting tools

- Go: gofmt
- JS: prettier
- Python:
  - **autopep8**: will enforce some of the PEP8 stylistic rules for you
  - **yapf**: similar algorithm as gofmt, works quite well except when it doesn't
  - **isort**: will sort your import according to your taste

Static analysis

Problem / goals

- Use automated tools to pinpoint quality issues in a given code base:
  - Adherence to a coding standard
  - Statically detect errors thats would / could happen at runtime (e.g. type errors)
  - Measure key metrics and check that they are within acceptable bounds
- We’re using an IDE or IDE-like text editor, and want to help the IDE help us be more efficient

Command-line tools

- flake8
- pylint
- mypy
- mccabe

Pylint and Flake8

- Both are able to statically check for:
  - Violation of parts of a given coding standards (including parts of PEP8)
  - Possible runtime errors
- Both are customisable and extensible through plugins
  - You can write your own to address specific issues in your own coding standards

Flake8 and Pylint: comparison

- Flake8 is faster than Pylint
- Both (specially Pylint) can give you a lot of false positives (or report on issues you don’t care about)
- Pylint has a cool “`pylint --py3k`” mode which will warn you (with a few false positives :( ) about things that won’t work in Python 3 in your legacy code base

Type annotations

- Introduced by PEP 484 and PEP 526
  - Dedicated syntax only available in Python 3
  - Workaround (using comments) for legacy code bases
- Introduce optional type annotation (aka “gradual typing”) that can be checked by a command-line tool (mypy) or an IDE (PyCharm)
- Still a WIP (important changes expected in Python 3.7)

Measurements

Application Architecture Standards
- Multilayer design compliance (UI vs App Domain vs Infrastructure/Data)
- Data access performance
- Coupling Ratios
- Component (or pattern) reuse ratios

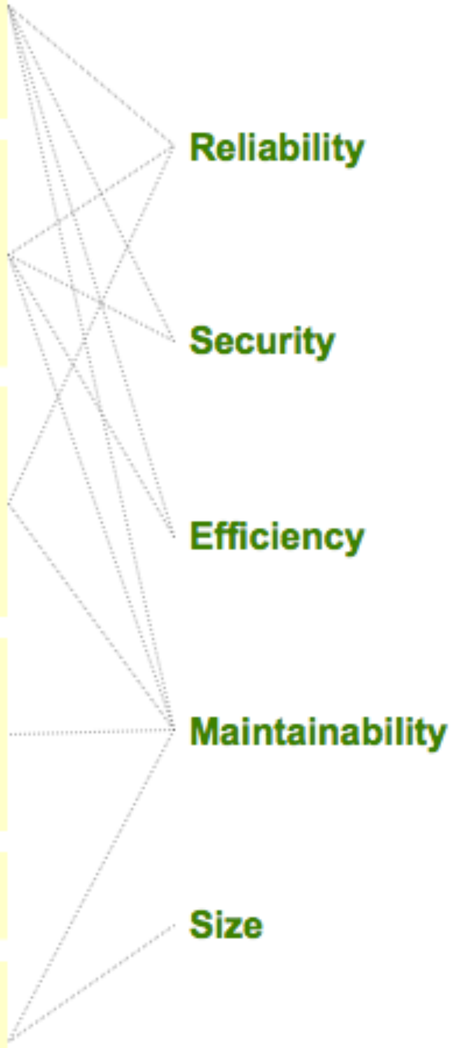
Coding Practices
- Error/exception handling (all layers UI/Logic/data)
- If applicable - compliance with OO and structured programming practices
- Secure controls (access to system functions, access controls to programs)

Complexity
- Transaction
- Algorithms
- Programming practices (eg use of polymorphism, dynamic instantation)
- Dirty programming (dead code, empty code…)

Documentation
- Code readability and structuredness
- Architecture -, program, - and code-level documentation ratios
- Source code file organization

Portability: Hardware, OS and Software component and DB dependency levels

Technical and Functional Volumes
- # LOC per technology, # of artifacts, files
- Function points - Adherence to specifications (IFPUG, Cosmic references..)



abilian

Complexity metrics

- The McCabe cyclomatic complexity index, and similar metrics (Halstead)
- High cyclomatic complexity of a given function / method / class / module is often correlated to:
  - Poor understandability
  - Poor maintainability
  - Number of bugs
  - Poor test coverage
- Pylint and Flake8 + mccabe are able to report excessive (you get to set the threshold) complexity, according to this, and other, metrics

Online metrics assessment tools

- Sonar
- CodeClimate
- Codacy

abilian

Testing
CT / CI

abilian
CONNECTED, WE WORK!

Goals

- Ensure (up to a point) correctness of your code
- Help developers come with better (more decoupled) design (if you believe the TDD credo)
- Make it a lot safer so refactor and more generally update your code (including your dependencies)

Automated testing

- Don’t say “unit tests” even if the standard module is called “unittest”
- Personal opinion: use Pytest instead
  - unittest, inspired by JUnit, leads to complex inheritance hierarchies
- Distinguish between
  - Unit test: isolated, fast (< 1s)
  - Integration / end-to-end test

Two neat pytest tricks

- “`pytest --ff`”: when a test failed in the previous run, will rerun it first, shortening the fix -> test cycle
- Use the pytest-randomly plugin (unit tests aren’t supposed in a specific order)

abilian

Other cool tools

- **Coverage.py**: you should aim for 100% code coverage with your tests - and fail your test if you don’t. Some excellent projects do it, some don’t. Still, that’s both a valuable and achievable goal for many kinds of applications.
- **Hypothesis**: property based testing library, which helps create tests “which are simpler to write and more powerful when run, finding edge cases in your code you wouldn’t have thought to look for.”
- **Cosmic Ray**: mutation testing (WIP).

Continuous testing / integration

- tox when doing it locally
- Several available server tools
  - Jenkins, Buildbot… (self hosted)
  - Travis, Circle… (SaaS)
- IMHO it’s best if you’re able to leverage tox in your server config, which is not always fully compatible with the way the SaaS vendor wants things to work

Documentation

abilian
CONNECTED, WE WORK!

Goal

- Good documentation serves several goals:
  - Marketing material for your code
  - Quickly onboarding users and contributors
  - Reference for everyone (e.g. API)

README files

- A `README.md` or `README.rst` is usually first the piece of documentation your prospects and users will discover, specially on places such as PyPI or GitHub / Bitbucket / GitLab
  - You must pay a lot of attention to make it clear for the target audience, and always up to date
  - Luckily, there is a way to reuse it in the “official” doc when you are using Sphinx (next slide)
- Additional important text documents at the root of your project could include: `CHANGES.rst`, `AUTHORS.rst`, `CONTRIBUTING.rst`, `COC.rst`, etc.

Sphinx

- Sphinx is pretty much the standard for Python (and non-Python!) projects nowadays
- Cleverly mixes standalone text (usually from a `docs` subdirectory) with docstrings
  - Main benefit: gives you API documentation “for free”
- Uses the RestructuredText (ReST) syntax and can produce multiple output formats (HTML, PDF, ePub…)
- Still needs some discipline to get right (including tools such as pydocstyle)
- Can easily be published on https://your-project.readthedocs.org/

abilian

Packaging

abilian
CONNECTED, WE WORK!

Topics

- Versionning
  - Scheme (semantic vs. date-based)
  - Tools (bumpversion, setuptools_scm...)
- Making packages
  - Setuptools
  - Wheels (for platform-dependent packages)
  - Pyroma

abilian

Pyroma

- Simple tool to check your package information (setup.py, setup.cfg) for best practices

```
➜  ~ pyroma -p flask
------------------------------
Checking flask
Found Flask version 0.12.2
Looking for documentation
Downloading Flask-0.12.2.tar.gz to verify distribution
Found Flask
------------------------------
Your package does not have keywords data.
You might want to host your documentation on readthedocs.org.
You should have three or more owners of the project on PyPI.
------------------------------
Final rating: 8/10
Philadelphia
------------------------------
```

abilian

Automation, configuration & “project management”

abilian
CONNECTED, WE WORK!

Automation

- Lots of development tasks should be automated: setup, build, test, package, release, deploy, clean, format, update…
- Instead of having to document / remember all these tasks by hand, I prefer to put them in a `Makefile`
  - Bonus: shell autocompletion
- Other similar build tools may be used, depending on your taste

Configuration

- A lot of the tools we've surveyed so far depend on configuration files present in your project repository
  - `setup.py`
  - `setup.cfg`
  - `MANIFEST.in`
  - `tox.ini / .travis.yml / circle.yml`
  - `pylint.ini`
  - …
- These configuration need to be provided with sensible defaults (depending on your engineering policy), and upgraded when this policy changes

Cookiecutter

- Cookiecutter allows you to bootstrap a new project repository using from a project template and your answers to a of questions (“What’s the project name”, etc.)
- Think “scaffolding” but for a whole project
- You can (and should) make your own templates according to your own policy
- Note that this only addresses project creation and not policy updates

abilian

Medikit

- Manage (and update) and your projects assets.
- Combinable features (pytest, django, nodejs, webpack, sphinx+ custom).
- Manage all your projects the same way:
  - One interface for the release engineers (Projectfile).
  - One interface for the software engineers (Makefile).

abilian

Quality is a team sport!

Code reviews

- GitHub & Gitlab provide integrated code review tools
- Gerrit (cf. “Gerrit is Awesome” by Mike Bayer)
- We should also talk about branching models, but let’s consider this an advanced (and controversial!) topic

Relevant communities

- Work groups under the PSF:
  - **PyPA**: the Python Packaging Authority (https://www.pypa.io/)
  - **PyCQA**: the Python Code Quality Authority (http://meta.pycqa.org/)
- Others
  - **Write The Docs**: a series of conferences and local meetups focused on all things related to software documentation (http://www.writethedocs.org/)

2 - “Ops”

(With a focus on server / web apps)

Deployment

abilian
CONNECTED, WE WORK!

Options

- Old-school bare-metal / VPS
- PaaS, Serverless, containers (kubernetes…)

abilian

PaaS & Containers

- Your PaaS software / provider or your container architecture has probably decided for you
  - a specific workflow
  - specifically, which tools to use (git, a specific CLI tool, etc.)
  - a set of principles / best practices (ex: “12 factor app manifesto”)

Your own servers

- Web deploying apps, one should probably distinguish provisioning from app deployment
- Provisioning can be done using dedicated tools (Ansible, Salt, BundleWrap, Chef, Puppet, Fabric+Fabtools…)
- For application deployment, I’m personally using Fabric+Fabtools, but there are some issues (is the software still maintained?) and there are several options

Possible deployment strategies

- Build Debian packages on your CD platform or a dedicated server, push them to a private repo, install on the servers using apt
  - Details -> cf. Nylas blog post, Hynek Schlawack blog post, Parcel...
- Build Python wheels on your CD platform or a dedicated server (same env as production!), push them to a private PyPI server, etc.
- Tag your git repository, push it to your private git repo, then pull the specific tag on the servers and run the install or update procedure ("`pip install .`")
  - Details -> "git-based fabric deploys are awesome" blog post
- In all cases, Fabric can be used to orchestrate the actual deployments on all your servers (if you have several)

Advanced topics

- Blue / green deployment
- Staged deployment
- Immutable deployments
- Data migration

abilian

Monitoring

abilian
CONNECTED, WE WORK!

Tools

- Error reporting: Sentry
- Log collection
- Metrics collection: Prometheus, Statsd, Graphite, Influxdb, OpenTSDB, Gnocchi...
- Application Performance Monitoring (APM): Newrelic, Librato, OpenTracing...
- Security monitoring and protection: Sqreen

3 - Levelling-up

abilian
CONNECTED, WE WORK!

Further reading

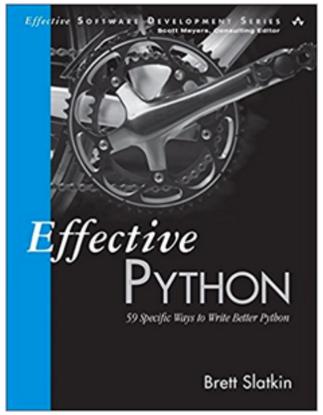
- Tarek Ziadé, “Expert Python Programming (2nd ed)” - Packt
- Julien Danjou “The Hacker's Guide to Python” - self-published.
- Frederic Lepied, “Quality Python Development” - self-published

abilian

Good advanced-level Python Books

- Luciano Ramalho, “Fluent Python” - O’Reilly

- Brett Slatkin, “Effective Python” - Wiley



abilian

4 - Conclusion

abilian
CONNECTED, WE WORK!

Conclusion

- It’s not because you’re using Python you shouldn’t be serious about software engineering

abilian

More info

- Slides will appear soon on https://speakerdeck.com/sfermigier/
- Repo for this talk: https://github.com/abilian/talks
- Contact: sf@abilian.com

abilian