Best practices for scientific software development

Or, how to spend less time on making things to work and more on doing science

Stefano Alberto Russo - INAF / University of Trieste

Introduction

As with everything, even in software development there is a set of best practices which makes life easier at nearly no cost.

Interestingly enough, this set of common wisdom collected over the years of professional software development is often ignored in academia, causing to waste massive amounts of time which could be easily spared by just adopting the right approach and philosophy.

The seminar does not aim by any means to push scientist to become software engineers. Instead, it wants to provide a swiss army knife in a minimum effort - maximum yield logic to help them in everyday's life.

We will see tools and concepts as Git and version control, testing, debugging hints, managing dependencies, reproducibility and more, and how to use them effectively.

Why should you listen to me?

An hybrid profile:

- BSc in Computer Science
- MSc in Computational Physics

Started at CERN, as research fellow working on data analysis & Big Data

Then, 5 years in startups.

- Core team member of an IoT energy metering and analytics startup,
- Joined Entrepreneur First, Europe's best deep tech startup accelerator
- ..and launched my own one :)

Now back into research:

- INAF and UniTS, working on resource-intensive data analysis
- adjunct prof. of computer science at University of Trieste
- plus, experienced consultant for a number of private companies

The deal

1) I will use Python for the coding examples, but the concepts are 100% language-agnostic
2) Always interrupt if you have question, doubs, something not clear, curiosities. Let's try to keep it interactive
3) Over the talk, think about a concrete use case close to your work: we will discuss a few at the end.

Outline

- How to structure your code
  - Logic blocks and comments
  - Functions and scope
  - Objects and classes
  - Readability vs. performance
  - Sanity checks
- How to debug your code
  - Reproducibility
  - Dependencies
  - Naming variables
  - Logging
  - The Notebooks
- Testing
  - End-to-end testing
  - Unit testing
  - Continuous integration
- Version control and collaboration
  - Git, commits, tags
  - Versioning strategies
  - Branching and flows
  - Documentation
- Discussion

Outline

- How to structure your code
  - Logic blocks and comments
  - Functions and scope
  - Objects and classes
  - Readability vs. performance
  - Sanity checks
- How to debug your code
  - Reproducibility
  - Dependencies
  - Naming variables
  - Logging
  - The Notebooks
- Testing
  - End-to-end testing
  - Unit testing
  - Continuous integration
- Version control and collaboration
  - Git, commits, tags
  - Versioning strategies
  - Branching and flows
  - Documentation
- Discussion

How to structure your code

➜ *Logic blocks*

Logic blocks are the base unit of your piece of code, way before functions and classes

Use them to divide the code
in portions responsible
of a specific parts

```
#====================
# Base structures
#====================

class Point:
    ...

class DataPoint:
    ...

#====================
# Series structures
#====================

class PointSeries:
    ...

class DataPointSeries:
    ...
```

How to structure your code

➔ *Comments*

Exaggerate. Think about the future yourself reading your code in 5 years time.

```
def timezonize(timezone):
    """Convert a string representation of a timezone to its pytz object,
    or do nothing if the argument is already a pytz timezone."""

    # Checking if somthing is a valid pytz object is hard as it seems that they are spread around the pytz package.
    #
    # Option 1): Try to convert if string or unicode, otherwise try to instantiate a datetieme object decorated
    # with the timezone in order to check if it is a valid one.
    #
    # Option 2): Get all members of the pytz package and check for type, see
    # http://stackoverflow.com/questions/14570802/python-check-if-object-is-instance-of-any-class-from-a-certain-module
    #
    # Option 3) perform a hand-made test. We go for this one, tests would fail if something changes in this approach.

    if not 'pytz' in str(type(timezone)):
        timezone = pytz.timezone(timezone)

    return timezone
```

How to structure your code

➔ *Functions and scope*

Functions should be always self-consistent

- Do not access external variables
- Try to always return the result
- Process in-place only if you really have to

```
value = 5

def square():
    result = value*value
    return result
```

```
def square(value):
    result = value*value
    return result
```

How to structure your code

➔ *Functions and scope*

Functions should be always self-consistent

- Do not access external variables
- Try to always return the result
- Process in-place only if you really have to

```
result = None

def square(value, result):
    result = value*value
```

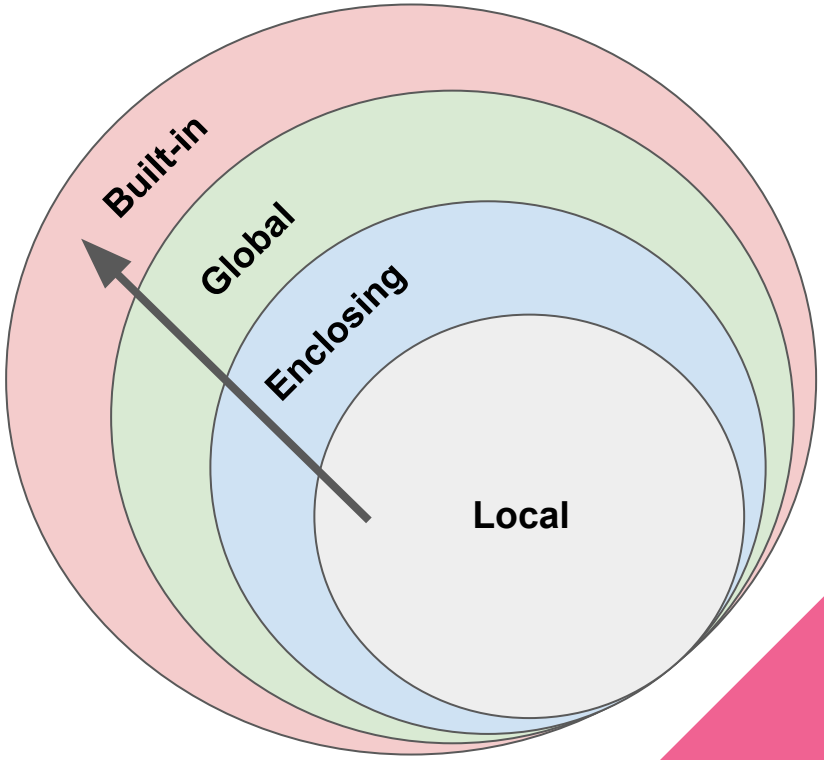
```
def square(value):
    result = value*value
    return result
```

How to structure your code

➔ *Functions and scope*

The LEGB rule works in nearly any programming language.

Keep it in mind!



How to structure your code

➔ *Objects and classes*

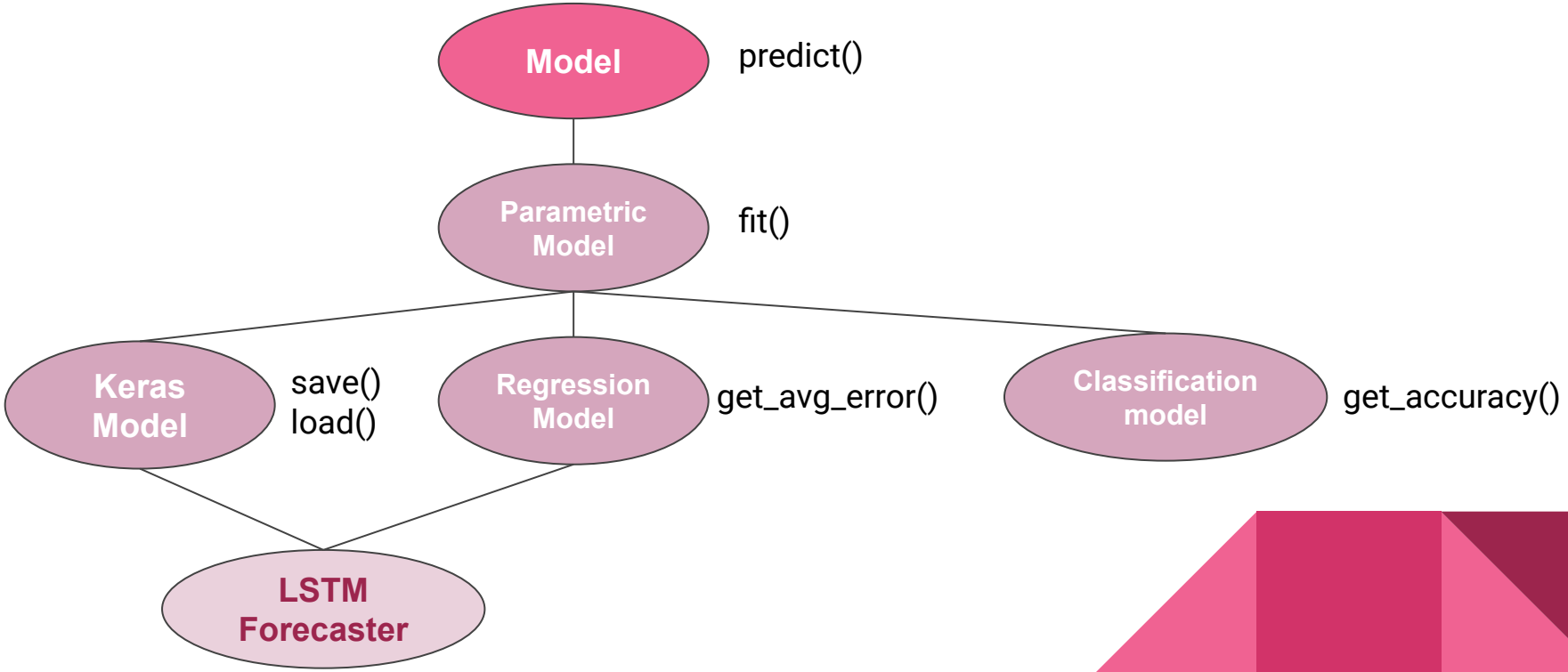
Modelling well the entities you have to deal with can help you a lot.

Ask yourself what you are modelling, and:

- use hierarchy and inheritance
- try not to change method interfaces
  - extend them if you really have to
- keep in mind double inheritance

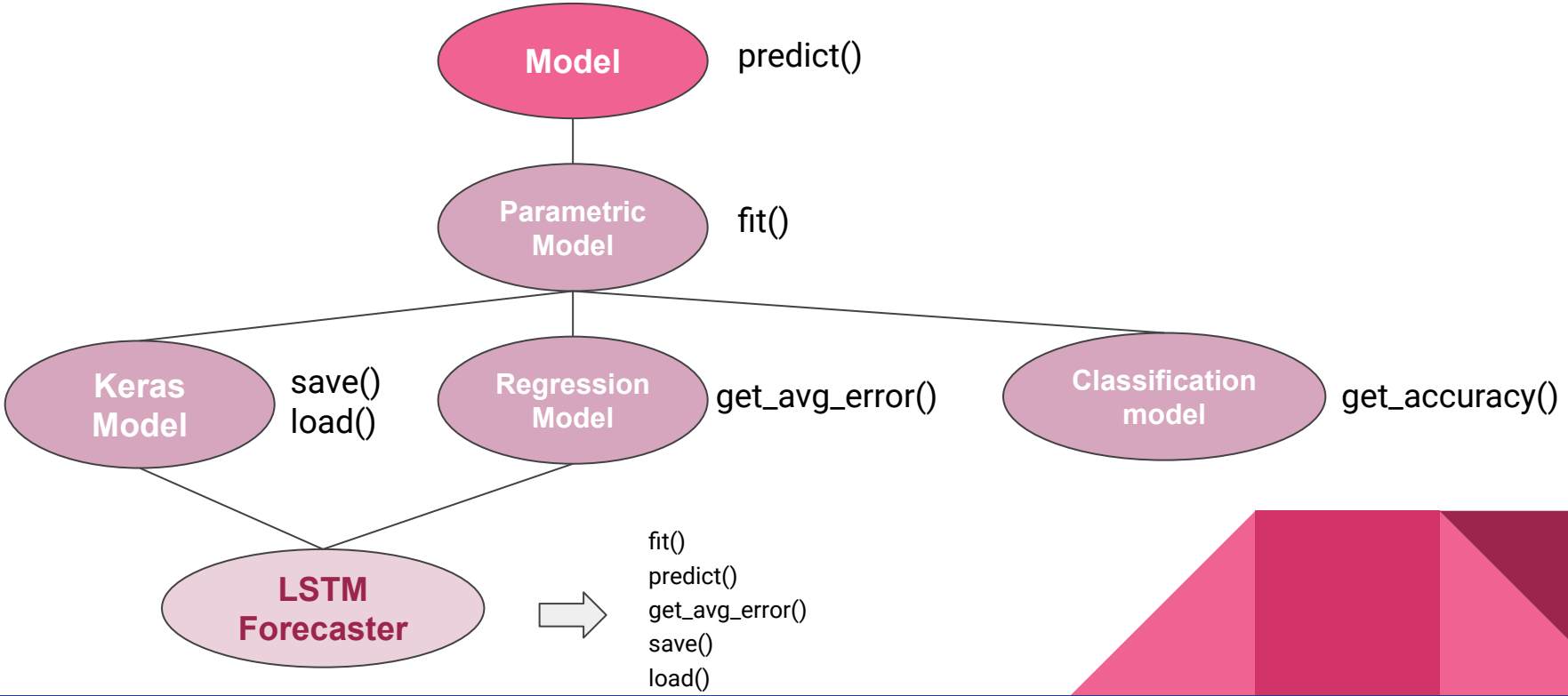
How to structure your code

➔ *Objects and classes*



How to structure your code

➔ *Objects and classes*



How to structure your code

➔ *Readability vs. performance*

Never code thinking about performance

Always try to be logically clear in what you write first.

- `compute_avg()` and `compute_var()`
  vs
- `compute_avg_and_var()`

- `compute_features()`, `train()`
  vs
- `compute_features_and_train(list)`

How to structure your code

➔ *Readability vs. performance*

Then, *profile* your code and see where you can improve it:

```
$ python3 -m cProfile -s tottime profiling.py

Done profiling
48567879 function calls (48557828 primitive calls) in 530.175 seconds

Ordered by: internal time

  ncalls  tottime  percall  cumtime  percall filename:lineno(function)
 4299877   87.457    0.000  150.125    0.000 datastructures.py:105(__getitem__)
10428202   50.614    0.000   50.614    0.000 {built-in method builtins.isinstance}
 2299931   30.683    0.000  134.900    0.000 datastructures.py:1494(__next__)
 2599935   26.472    0.000   39.964    0.000 __init__.py:1424(debug)
       1   22.502   22.502  479.374  479.374 transformations.py:250(process)
 4299877   21.061    0.000   21.061    0.000 {function Series.__getitem__ at 0x7fc9683ad160}
 3999910   20.135    0.000   20.135    0.000 datastructures.py:391(t)
   99997   17.895    0.000  131.834    0.001 operations.py:114(__call__)
  599992   16.965    0.000   33.056    0.000 time.py:125(dt_from_s)
   99997   14.618    0.000  307.165    0.003 transformations.py:21(_compute_new)
 2599937   13.492    0.000   13.492    0.000 __init__.py:1689(isEnabledFor)
```

How to structure your code

➔ *Sanity Checks*

Always ensure your working hypothesis are respected

- If you divide by n, check for n not equal to zero
- If you have to compute the derivative of an array, check it has at least two points
- If you have to read a file, check it exists first

In dynamic/duck typing context, enforce even the type checks*

- If you expect a number, ensure it is an integer or a floating point
- If you expect a name of a file, ensure it is of type string

Etc..

No global consensus on this

Outline

- How to structure your code
  - Logic blocks and comments
  - Functions and scope
  - Objects and classes
  - Readability vs. performance
  - Sanity checks
- How to debug your code
  - Reproducibility
  - Dependencies
  - Naming variables
  - Logging
  - The Notebooks
- Testing
  - End-to-end testing
  - Unit testing
  - Continuous integration
- Version control and collaboration
  - Git, commits, tags
  - Versioning strategies
  - Branching and flows
  - Documentation
- Discussion

How to debug your code

➔ *Reproducibility*

Ensure your code results does NOT depend on external factors

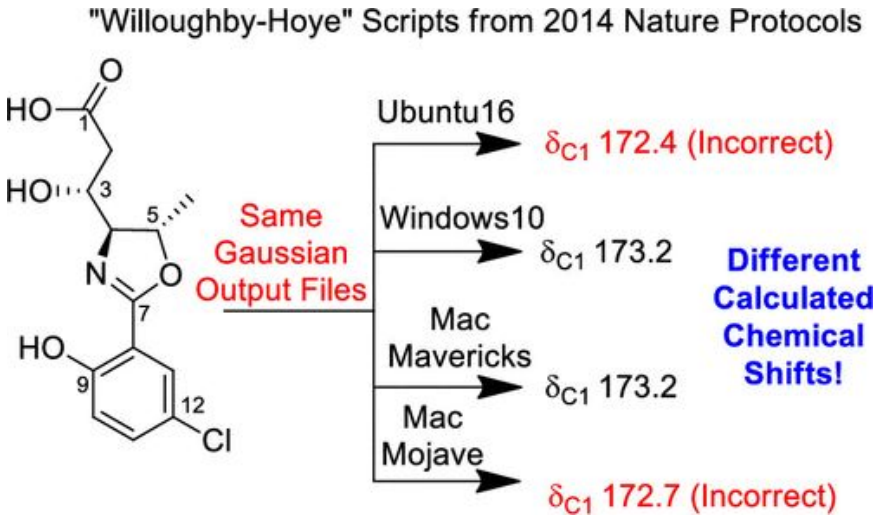
- Internet connectivity/resources
  - download them locally, always.
- Configuration files
  - read them at the beginning, and then use variables
- Moon phases
  - this means you are doing something wrong. Like treating a Python dictionary as an ordered data structure (spoiler alert: it isn't!)
- Threads (advanced stuff)
  - ensure you sync them. Signals. events, etc.

How to debug your code

➔ *Reproducibility*

Characterization of Leptazolines A–D, Polar Oxazolines from the Cyanobacterium Leptolyngbya sp., Reveals a Glitch with the "Willoughby–Hoye" Scripts for Calculating NMR Chemical Shifts

Jayanti et al - 2019



How to debug your code

➔ *Reproducibility*

If using randomness:

- fix the seed

If evaluating a stochastic process:

- sample it

  → *fix a set of seeds*

Until your last, glorious, nobel-winning run: make your code deterministic.

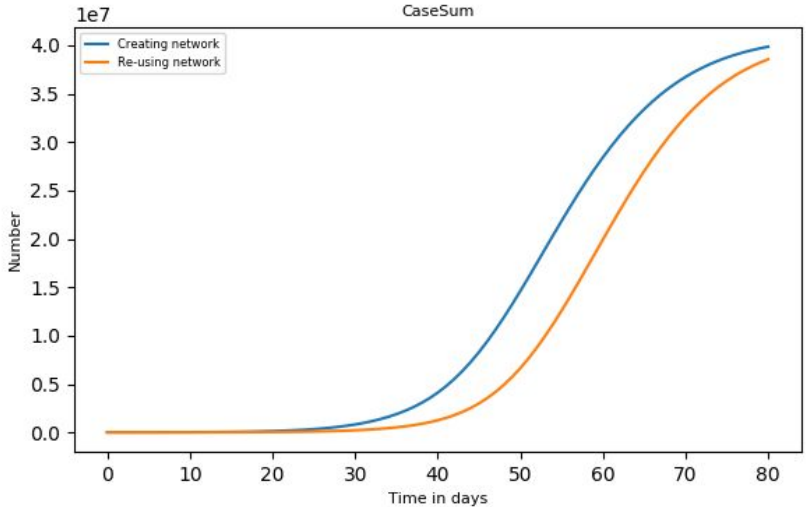
How to debug your code

➔ *Reproducibility*

The Ferguson model case:

"We are aware of some small non-determinisms when using multiple threads to set up the network of people and places. This has historically been considered acceptable because of the general stochastic nature of the model."

A team member



https://github.com/mrc-ide/covid-sim/issues/116

How to debug your code

➜ *Dependencies*

Make sure to know what dependencies your code relies upon and FIX them.

Do not just do a:

```
pip install keras numpy tensorflow scikit-learn
```

Instead:

```
pip install Keras==2.7.0 numpy==1.21.4 scikit-learn==1.0.1
```

You then *might* use different dependencies in some contexts,
but ensure you have a set of versions known as working.

How to debug your code

➔ *naming*

Variables

- `t` vs `temperature`
- `a0` vs `first_element_first_row`

Functions

- `compute()` vs `compute_monthly_averages()`

Classes

- `Model` vs `KerasLSTMOptimizedModel`

p.s. in Python, variables and class instances always lowercase_with_underscores, classes CamelCase.

How to debug your code

➔ *logging*

Logging must be *verbose* and give as much context as possible:

- `Error`
  vs
- `Cannot convert element #32 of the list of type "str" and value "ciao"`

Logging levels (if using the logging modules)

- `CRITICAL` is for errors which crash the entire program
- `ERROR` is for errors you can deal with
- `WARNING` is for particular conditions to be notified
- `INFO` is for giving informations about the execution
- `DEBUG` is for debug messages

How to debug your code

➔ *the Notebooks*

The Jupyter/R Notebooks are a big source of issues.

1) They allow for unordered execution
2) They have tons and tons of hidden state that's easy to screw up and difficult to reason about
3) For beginners, with dozens of cells and more complex code, this is utterly confusing

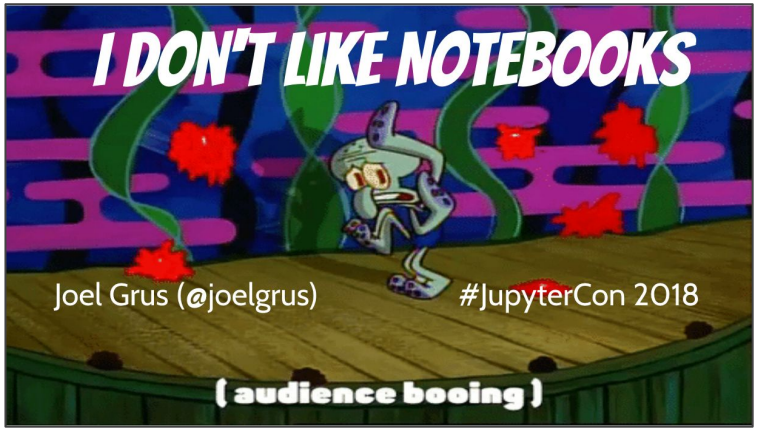
2) and 3) are by Joel Grus

How to debug your code

➔ *the Notebooks*

Joel Grus - I don't Like Notebooks @ PyCon2018

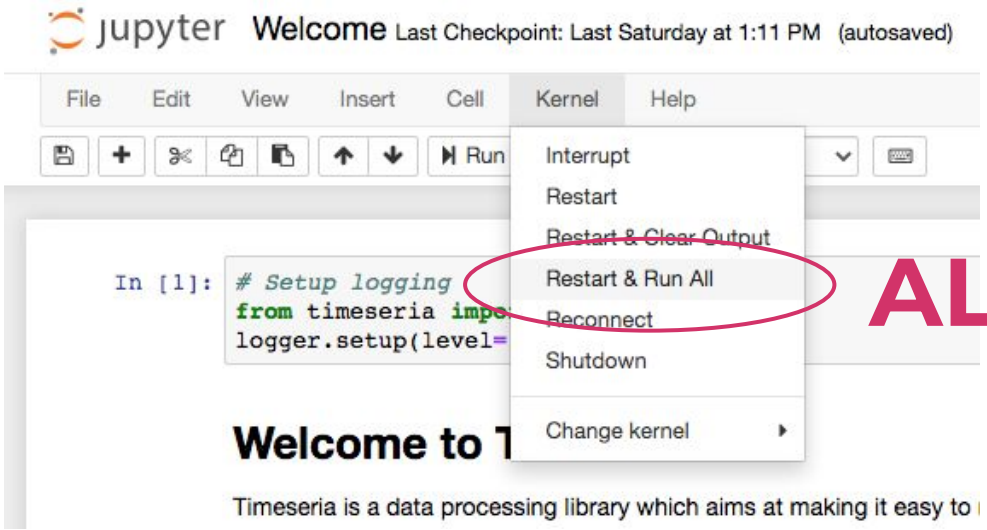
- https://docs.google.com/presentation/d/1n2RlMdmv1p25Xy5thJUhkKGvjtV-dkAIsUXP-AL4ffI/edit#slide=id.g38857eff70_0_4
- Will teach you everything you need to know about what can go wrong in Notebooks
- Plus, presentation has Smurfs



How to debug your code

➔ *the Notebooks*

The best thing we can do for now:



ALWAYS

Outline

- How to structure your code
  - Logic blocks and comments
  - Functions and scope
  - Objects and classes
  - Readability vs. performance
  - Sanity checks
- How to debug your code
  - Reproducibility
  - Dependencies
  - Naming variables
  - Logging
  - The Notebooks
- Testing
  - End-to-end testing
  - Unit testing
  - Continuous integration
- Version control and collaboration
  - Git, commits, tags
  - Versioning strategies
  - Branching and flows
  - Documentation
- Discussion

Testing

➔ *what is testing?*

- How would you test a pen?

Testing

➔ *what is testing?*

- How would you test a pen?
- How would you test the sum function?

Testing

➔ *what is testing?*

- How would you test a pen?
- How would you test the sum function?
- How would you test a forecasting model?

Testing

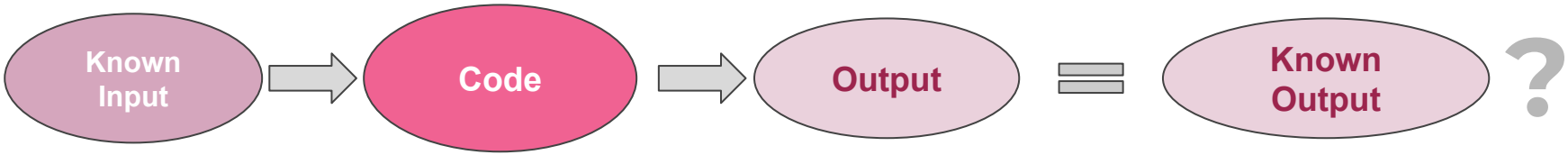
➔ *what is testing?*

- How would you test a pen?
- How would you test the sum function?
- How would you test a forecasting model?
- How would you test a database connection module?

Testing

➔ *End to end testing*

- Test the behaviour of a big entity (i.e. your entire code) from one end to the other.



- If the test fails, it does not tell you where you screwed up.

  → But it is like having a seat belt fasten, and comes nearly for free.

Testing

➔ *End to end testing*

Always create at least one end-to-end testing.

It can be a shell script, a Jupyter Notebook, whatever: we are not picky.

But do it. Like, tomorrow.

Testing

➔ *End to end testing*

Always create at least one end-to-end testing.

It can be a shell script, a Jupyter Notebook, whatever: we are not picky.

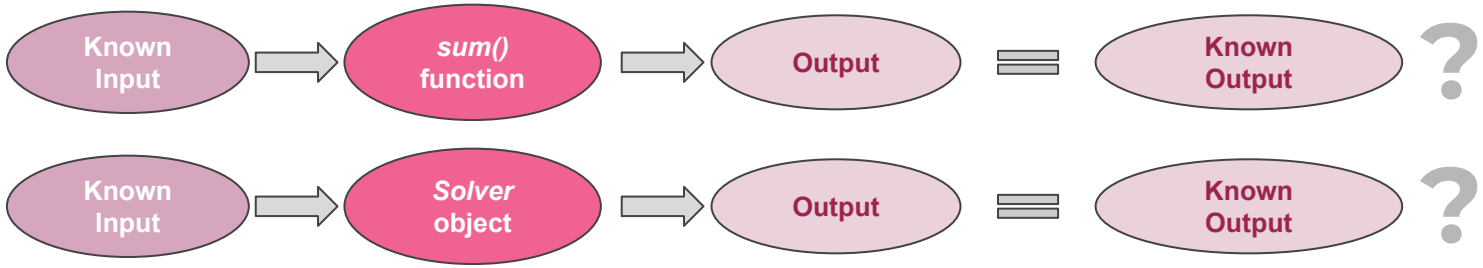
But do it. Like, tomorrow.

> Ciao Ste. Volevo dirti che grazie a come mi hai istruita sulla costruzione dei programmini (modularità, testing eccetera) sono sopravvissuta indenne all'aggiornamento di sistema delle workstation SISSA.
>
> Mi è bastato cambiare due righe e reinstallare una cosa e runnare il test per vedere se funzionava tutto
>
> 🎉🎉🎉🎉🎉🎉
>
> ***My Sister***

Testing

➔ *Unit testing*

- Test the behaviour of all the single entities (unit) of a bigger one (i.e. your code)



●●●

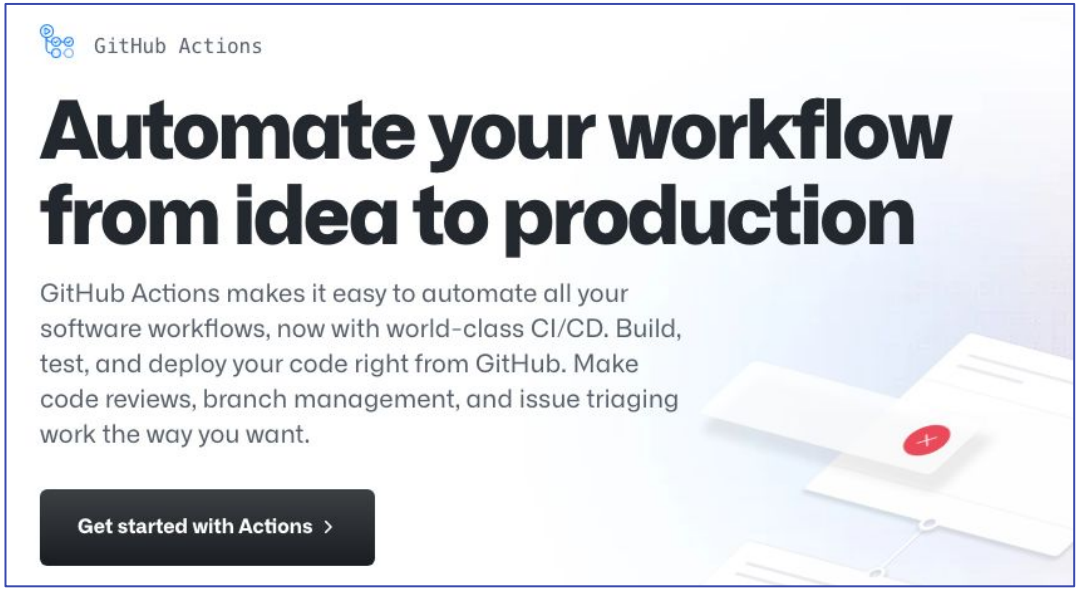
- If a test fails, it will tell you exactly where you screwed up.

  → But unit-testing requires more effort and a testing suite.

Testing

➔ *Continuous integration*

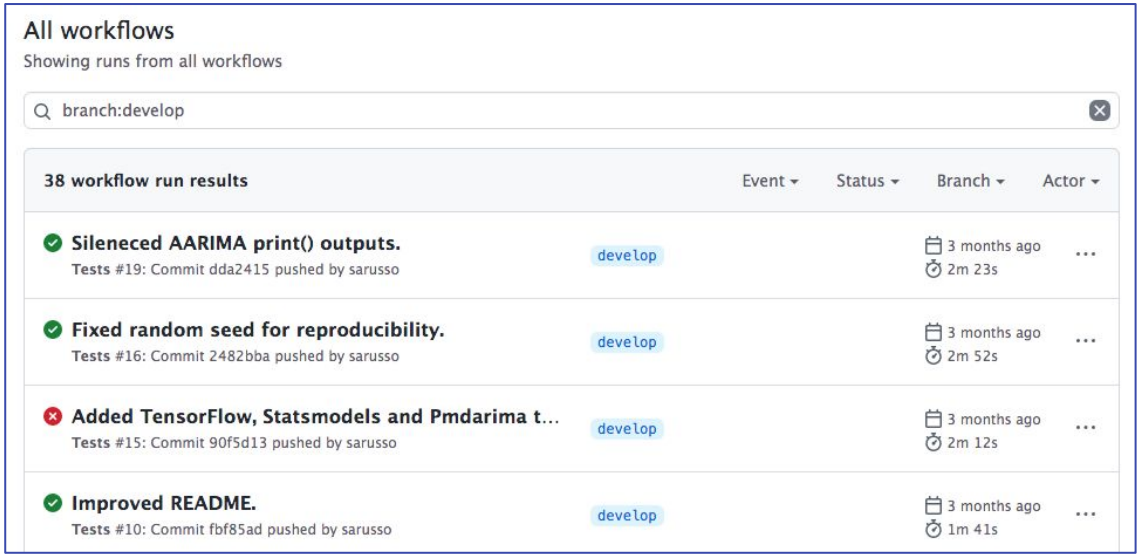
Continuous integration allows to automatically perform one or more tests every time a codebase changes. It stays for to “continuously integrating” changes (relying on testing).



Testing

➔ *Continuous integration*

Continuous integration allows to automatically perform one or more tests every time a codebase changes. It stays for to “continuously integrating” changes (relying on testing).



Outline

- How to structure your code
  - Logic blocks and comments
  - Functions and scope
  - Objects and classes
  - Readability vs. performance
  - Sanity checks
- How to debug your code
  - Reproducibility
  - Dependencies
  - Naming variables
  - Logging
  - The Notebooks
- Testing
  - End-to-end testing
  - Unit testing
  - Continuous integration
- Version control and collaboration
  - Git, commits, tags
  - Versioning strategies
  - Branching and flows
  - Documentation
- Discussion

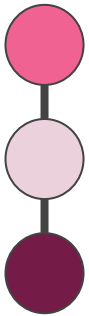
Version control and collaboration

➔ *Git, commits and tags*

Git is the versioning system de-facto standard for modern development.

Invented by Linus Torvalds for developing the Linux Kernel, after he got enough with versioning systems not working properly

Saving files in Git means to “commit” them, and to generate *hashes*



Codebase status @ hash d4ee02a

Codebase status @ hash 84c909f

Codebase status @ hash e5fe764

Version control and collaboration

➜ *Git, commits and tags*

Git is fully distributed, there is no centralized authority: no one assign version numbers. This is why it works so well.

The “origin”, which is usually GitHub, is just a copy of your Git repository. Do not confuse Git (a technology) with GitHub (a platform)

Git is fully deterministic: you cannot rewrite the history for a given hash!

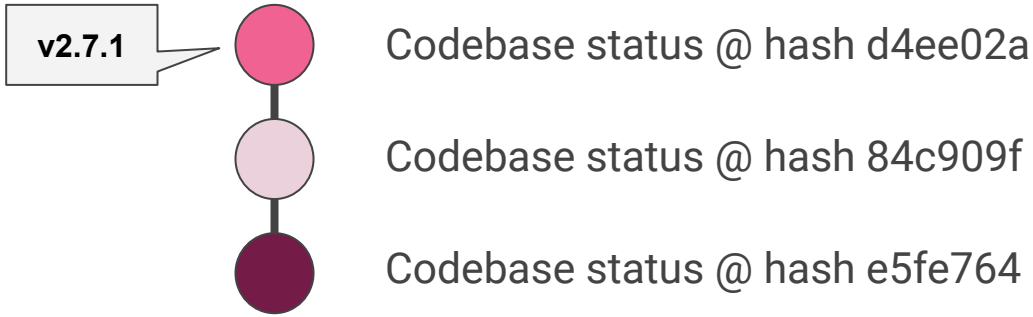
And if you do it, you change all the hashes: no one can cheat, not even a bug.

Version control and collaboration

➔ *Git, commits and tags*

Tags are instead “labels”, and can be changed.

→ beware of this, as people might act “unwisely”

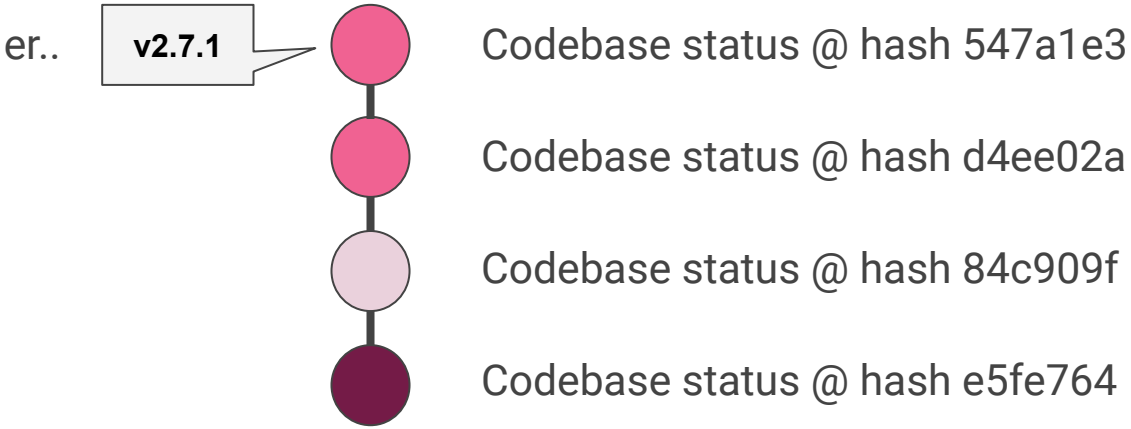


Version control and collaboration

➔ *Git, commits and tags*

Tags are instead “labels”, and can be changed.

→ beware of this, as people might act “unwisely”



Version control and collaboration

➜ *Git, commits and tags*

Refer to hashes if you can, not on tags

..and definitely not on a default or a specific a branch, like “master” → they can change!

```
$ git clone https://github.com/user/repo
$ git clone https://github.com/user/repo && git checkout master
$ git clone https://github.com/user/repo && git checkout v2.7.1
$ git clone https://github.com/user/repo && git checkout d4ee02a
```

Terrible

Bad

Better

Best

Version control and collaboration

➔ *Versioning strategies*

- Sequential numbers
  → revision 19721
- Dates
  → Ubuntu 20.04
- Semantic versioning
  → v2.7.1
- Hashes
  → 0f68b421

Version control and collaboration

➔ *Versioning strategies*

- Sequential numbers
  → revision 19721
- Dates
  → Ubuntu 20.04
- Semantic versioning
  → v2.7.1
- Hashes
  → 0f68b421

Given a version number MAJOR.MINOR.PATCH, increment the:

MAJOR version when you make incompatible API changes,

MINOR version when you add functionality in a backwards compatible manner, and

PATCH version when you make backwards compatible bug fixes.

https://semver.org/

p.s. v1.0.0 is as soon as someone starts using your software!

Version control and collaboration

➔ *Branching and flows*

“A branching strategy refers to the strategy a software development team employs when writing, merging, and shipping code in the context of a version control system like Git.”

https://launchdarkly.com/blog/git-branching-strategies-vs-trunk-based-development

→ The *Gitflow* branching strategy is a good compromise between complexity and effectiveness.

Version control and collaboration

➔ *Branching and flows*

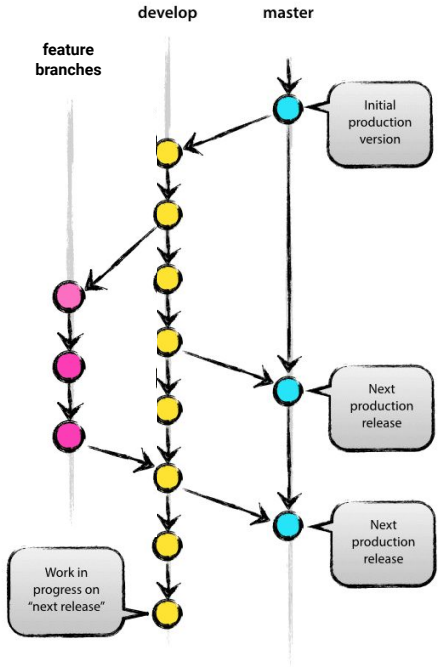
Gitflow foresees three classes of branches:

- The *main* or *master* branch, were releases happen
- The *develop* branch, where features are tested
- And n *feature* branches, one for each feature

It allows to collaborate in small-mid teams without much merge conflicts

it works great even if working in solo mode

https://nvie.com/posts/a-successful-git-branching-model/

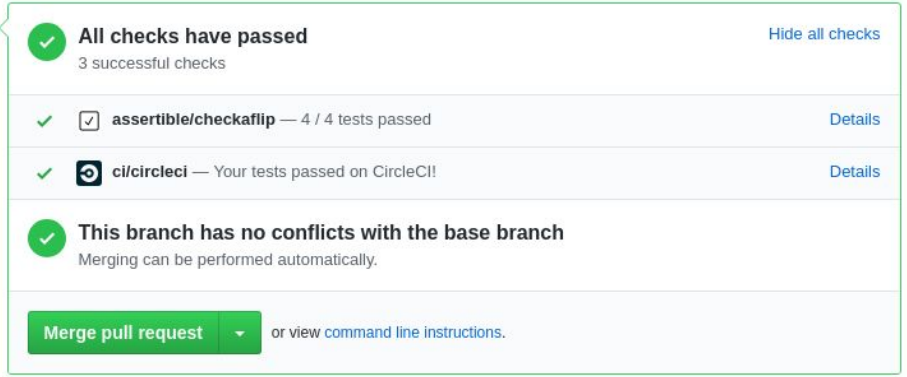


Version control and collaboration

➜ *Branching and flows*

Pull requests adds a more powerful mode to handle flows:

- a pull request asks the maintainers to “pull” a feature
- pull requests can be accepted and merged or rejected
- if continuous integration is set up, they get automatically tested



Version control and collaboration

➜ *Documentation*

A README!

Then,

Docstrings are your best friend:

If you write then as you write your code, you get the documentation for free with tools as Sphinx

Services as readthedocs can grab your commits from GitHub and automatically generate the documentation for you

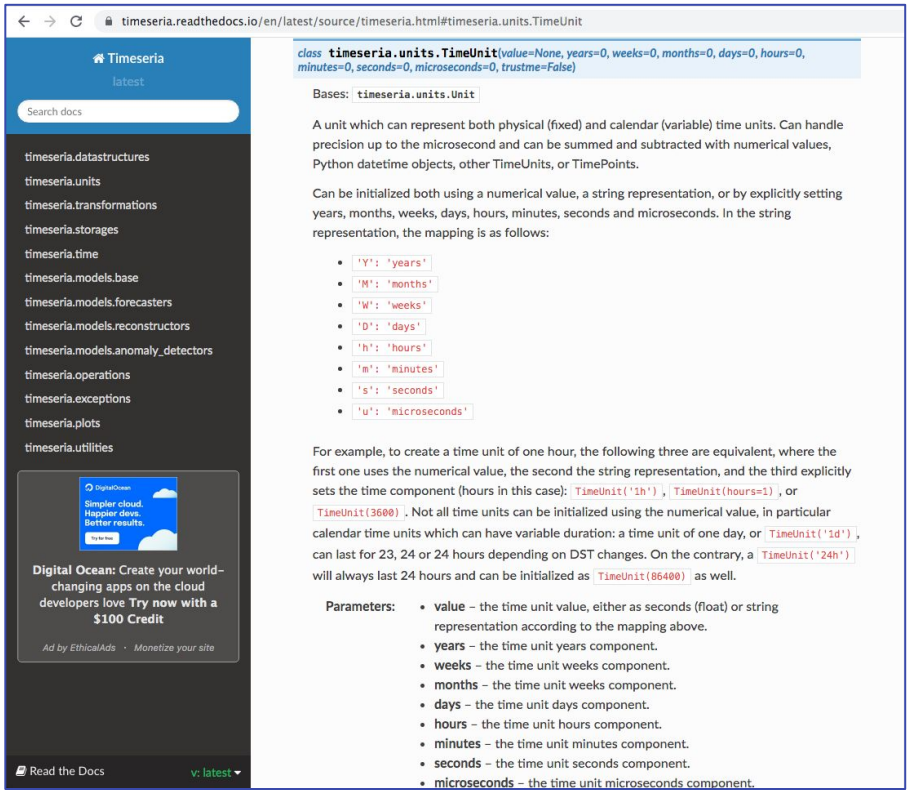
Version control and collaboration

➜ *Documentation*

```
class TimeUnit(Unit):
    """A unit which can represent both physical (fixed) and calendar (variable) time units.
    Can handle precision up to the microsecond and can be summed and subtracted with numerical
    values, Python datetime objects, other TimeUnits, or TimePoints.

    Can be initialized both using a numerical value, a string representation, or by explicitly setting
    years, months, weeks, days, hours, minutes, seconds and microseconds. In the string representation,
    the mapping is as follows:

        * ``'Y': 'years'``
        * ``'M': 'months'``
        * ``'W': 'weeks'``
        * ``'D': 'days'``
        * ``'h': 'hours'``
        * ``'m': 'minutes'``
        * ``'s': 'seconds'``
        * ``'u': 'microseconds'``

    For example, to create a time unit of one hour, the following three are equivalent, where the
    first one uses the numerical value, the second the string representation, and the third explicitly
    sets the time component (hours in this case): ``TimeUnit('1h')``, ``TimeUnit(hours=1)``, or ``TimeUnit(3600)``.
    Not all time units can be initialized using the numerical value, in particular calendar time units which can
    have variable duration: a time unit of one day, or ``TimeUnit('1d')``, can last for 23, 24 or 24 hours depending
    on DST changes. On the contrary, a ``TimeUnit('24h')`` will always last 24 hours and can be initialized as
    ``TimeUnit(86400)`` as well.

    Args:
        value: the time unit value, either as seconds (float) or string representation according to the mapping above.
        years: the time unit years component.
        weeks: the time unit weeks component.
        months: the time unit weeks component.
        days: the time unit days component.
        hours: the time unit hours component.
        minutes: the time unit minutes component.
        seconds: the time unit seconds component.
        microseconds: the time unit microseconds component.
        trustme: a boolean switch to skip checks.
    """
```

Version control and collaboration

➜ *Documentation*



Thanks!

➔ *Questions?*

stefano.russo@gmail.com

https://sarusso.github.io

https://twitter.com/_sarusso