I will adopt Best Practices
I will adopt Best Practices
I will adopt Best Practices
I will adopt Best Practices
I will adopt Best Practices
I will adopt Best Practices
I will adopt Best Practices
I will adopt Best Practices
I will adopt Best Practices
I will adopt Best Practices
I will adopt Best Practices

Programming for Life Sciences

Introductory session

University of Basel, Fall 2025

Contact: Alex Kanitz (alexander.kanitz@unibas.ch) / Mihaela Zavolan (mihaela.zavolan@unibas.ch)

Why care about programming? (1/2)

- Like all other natural sciences, life sciences have become a ‘data science’
- ‘High-throughput’ experimental approaches may yield giga- or even terabytes of data for a single experiment!

➢ ***Figuring out what is in these data is impossible without computers!***

Why care about programming? (2/2)

- Without “general-purpose thinking machines”, we cannot make the computer understand a very high level description of the task to be accomplished and let it figure it out for itself
- In research we want to discover things that were not known before, and therefore were not considered in existing models or tools

➢ *We will need to refine old tools and workflows & develop new ones*

What can computers do for us?

- Fast calculations
- Visualizations
- Storage & organization (data, metadata, documentation)
- Automation

This is important for, e.g.:

- Data exploration
- Reproducible analyses
- Modeling
- Inference

What is this course about?

- ➢ Practice design & implementation of solutions to problems arising in the context of life science research

- ➢ We will learn about:
  - Building software tools with Python*
    *a widely used programming language, including in the life sciences
  - Stringing together multiple tools into a computational workflow*
    *sequences of processing steps applied to analyze data
  - Collaborative open source software (OSS) development
  - Best coding & documentation practices

- ➢ All this by working together on a ***'real-life' software project***!

What is this course ***not*** about?

- ➢ Basic introduction to computing/programming

  If you do not know
  - what a file is
  - what the command line is
  - basic notions of programming

  it will be too hard to get up to speed and follow the curriculum!

- ➢ Theory of probability, statistics, machine learning

  These are all large topics covered in specific courses.

Course requirements

- Basic knowledge of programming & computers
- Laptop
- Access to sciCORE infrastructure
- Linux/Unix or Mac OS strongly recommended

  Windows users please read article this to find out about ways how you can run Linux on your machine (we recommend WSL2 with Ubuntu 22.04)
- Python 3.8 or newer installed (recommended: 3.11)*

  *on WSL/VM, if on Windows
- Code editor installed (we recommend VS Code)

How is the course structured?

- A collaborative group project is introduced and you, along with 1-2-3 other students, will take responsibility for solving a part of it
- You will lead the discussion of new concepts in each session (45 min)* and you will apply them to your code iteratively (on site and at home)
  * so you will need to read the materials *ahead of time!*
- Discussion of problems in each session (45 min) and in Slack

Note that this is not an obligatory course, and so the assumption is that you are here to deepen your experience in programming by putting in the work!

How do I earn credit points?

- Active participation in sessions
- Work on one of the issues for the collaborative project and complete milestones

We understand that we cover a lot of material, and not everyone will be able to complete all milestones. This is fine as long as we see that you show your best effort.

Course schedule

DATE	SESSION TOPIC	HOMEWORK *
09/18	Course introduction / Q&A [†]	
09/25	Project introduction / task assignment	First iteration of code design
10/02	Python basics	First implementation of code
10/09	Algorithm design	Optimize design & code
10/16	**MILESTONE REVIEW: Task design**	**Resolve feedback from mentors**
10/23	Version control	Initialize repository & version control your code
10/30	Documentation	Document and annotate your code
11/06	Encapsulation & packaging	Encapsulate & package your code; add CLI
11/13	**MILESTONE REVIEW: Repo, executable & docs**	**Resolve feedback from mentors**
11/20	Linting, testing & continuous integration	Lint your code, write tests & set up CI
11/27	Dependency management & containerization	Add dependencies file and Dockerfile
12/04	**MILESTONE REVIEW: Final**	**Resolve feedback from mentors**
12/11	Workflows / Nextflow	Write workflow process for your tool
12/28	Course wrap up & demo	

** In addition to the listed homeworks, you are expected to read through the materials for the next session, if applicable; in each session, some of you are expected to guide the discussion of the materials*

[†] ***Virtual session** on Zoom: https://unibas.zoom.us/j/5260402868?pwd=M1V3ZWpqN1ZGcmx6OFBkRW1iR0VKZz09*

Please fill in info sheet!

Coding best practices

A short primer

So…

Writing reusable code...

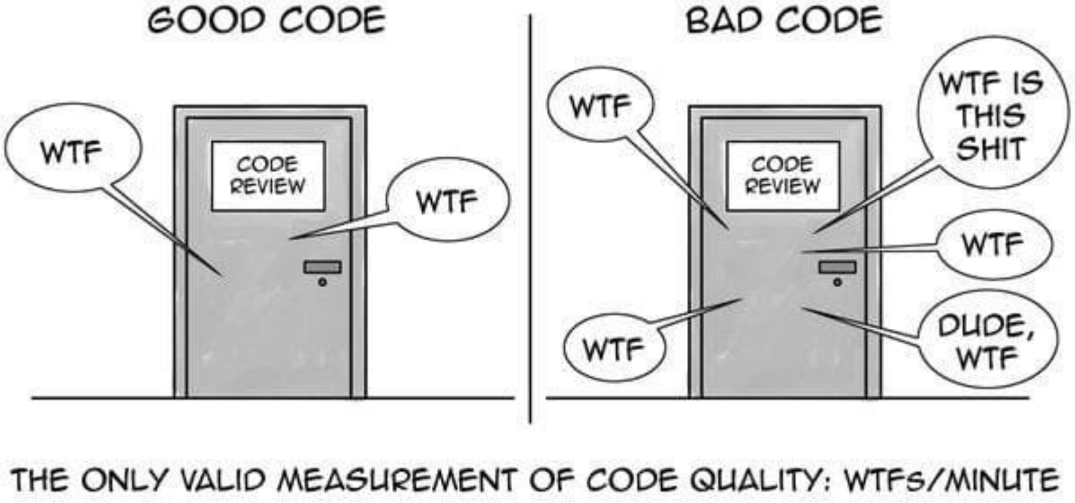
- Improves your **coding skills**
- Promotes **reproducibility**
- Benefits the **community**
- Gains you **recognition**
- **Saves time** (and money) *eventually*
- May attract **external contributors**
- Imparts **meaning** to your work

And remember:

My code will be published eventually…

…so I may as well do it right!

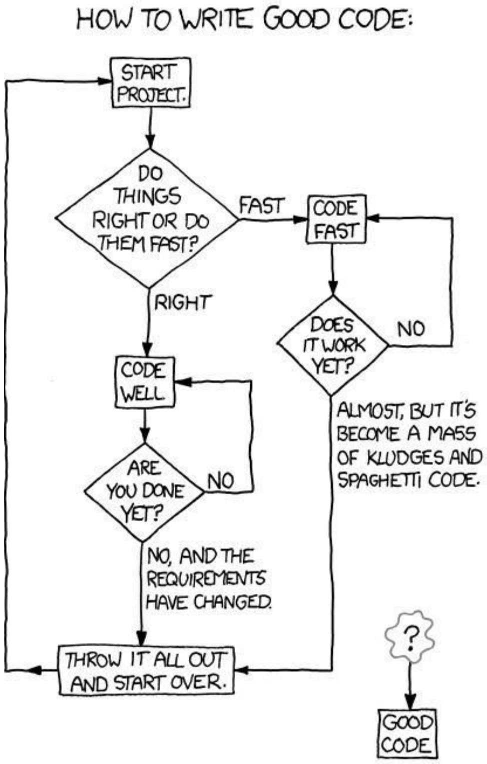
What is good code?



Good code is...

- **Correct**
  - Solves the problem at hand robustly (for reasonable range of inputs)
  - Gives *correct* results for regular input
  - Gives *consistent* results for irregular input
  - Validates inputs and returns helpful error messages
- **Maintainable**
  - Easy to read / understand (no unnecessary complexities)
  - Well documented / annotated
  - Easy to extend
- **Efficient**
  - “Fast enough“
  - Quantifiable and consistent performance
  - Time, memory & storage requirements scale well across reasonable range of inputs

So how to write good code?


HOW TO WRITE GOOD CODE:
START PROJECT.
DO THINGS RIGHT OR DO THEM FAST?
FAST
CODE FAST
DOES IT WORK YET?
NO
ALMOST, BUT IT'S BECOME A MASS OF KLUDGES AND SPAGHETTI CODE.
RIGHT
CODE WELL
ARE YOU DONE YET?
NO
NO, AND THE REQUIREMENTS HAVE CHANGED.
THROW IT ALL OUT AND START OVER.
?
GOOD CODE

Solution: Software engineering

> *“The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.”*
>
> IEEE Std 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology

Okay, but *HOW?*

Some software *design patterns*

- KISS (*“Keep It Simple, Stupid!“*)
  - Simplicity should be your key design goal
  - Antipattern: *spaghetti code*
- DRY (*“Don’t Repeat Yourself!“*)
  - Abstract solutions to maximize maintainability & reusability
  - Antipatterns: *reinvent the wheel*, *copy-and-paste programming*
- YAGNI (*“You Ain’t Gonna Need It!“*)
  - Focus on your current use case
  - Antipattern: *overengineering* (also: *feature creep*)

- More principles: https://en.wikipedia.org/wiki/List_of_software_development_philosophies
- More antipatterns: https://en.wikipedia.org/wiki/Category:Anti-patterns
- Related book: Thomas D & Hunt A, The Pragmatic Programmer, 1999, Addison Wesley

How about some concrete advice?

- Design pattern/philosophies: https://en.wikipedia.org/wiki/List_of_software_development_philosophies
- Antipatterns: https://en.wikipedia.org/wiki/Category:Anti-patterns
- Related book: Thomas D & Hunt A, The Pragmatic Programmer, 1999, Addison Wesley

Let’s focus on some core aspects

➢ Iterative development

➢ Readability

➢ Modularity / Reusability

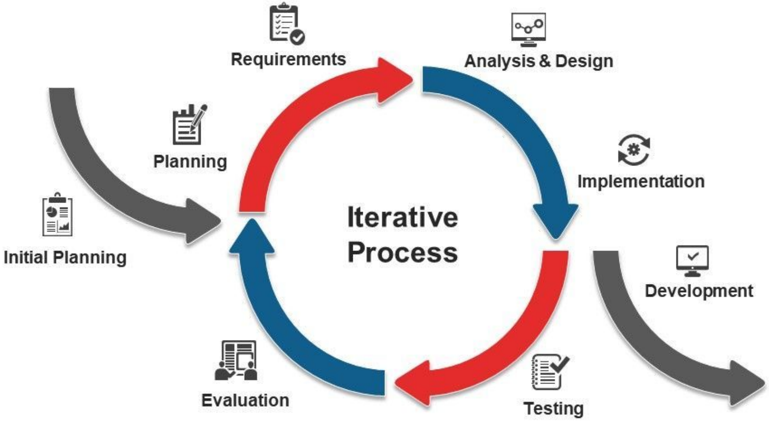
➢ Open source

More details in later sessions!

Iterative development

- Use version control system
- Get to MVP (minimum viable product) fast
- Improve incrementally, one feature/fix at a time

➢ Agile software development



Related design patterns/antipatterns:

- KISS (“Keep It Simple, Stupid!”)
- YAGNI (“You Ain’t Gonna Need It!”)
- RERO (“Release Early, Release Often”)

- Spaghetti code
- Feature creep

Modularity / Reusability

- Avoid code duplication
- Consider parametrizing/abstracting your code
- Write small units of code that do *one thing* (and do it well)
- Write unit tests

Related design patterns and antipatterns:

- DRY ("Don't Repeat Yourself!")
- Rule of three

- Copy-and-paste programming
- God object

Readability

- Use consistent coding style
  - For Python, follow at least PEP 8, but consider a stricter guide, e.g., Google's Python style guide)
- Use expressive names for variables, functions, classes etc.
- Where applicable, use constants or enumerations
- Comment code that is not obvious

Open source: 4OSS recommendations

- Develop publicly accessible open source code **from day one**
- Make software **easy to discover** by providing software metadata via a popular community registry
- Adopt a **license** and comply with the licence of third-party dependencies
- Have a clear and **transparent** contribution, governance and communication processes

- Details: https://softdev4research.github.io/recommendations/
- Paper: https://doi.org/10.12688/f1000research.11407.1

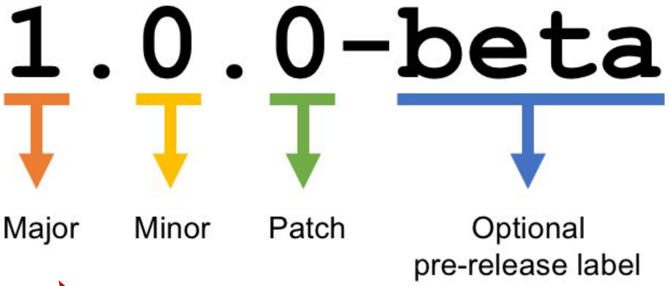
Open source: Free software licenses

Most commonly used (~80% combined in 2018 survey):

- MIT
  - Permissive license, essentially: *"Do whatever you want with it, just don't sue me!"*
- Apache 2.0
  - Permissive license, but with explicit granting of patent license and protection against patent treachery; changes must be documented
    ⇒ better for substantial software
- GPL (different variants)
  - Compatible with above but *copyleft* (viral!) license: modified code needs to be released open source and *under same license*!

- Details and more licenses (GitHub): https://choosealicense.com/
- Details and more licenses (FSF): https://www.gnu.org/licenses/license-list.en.html

Open source: Semantic versioning



1.0.0-beta

Major

Minor

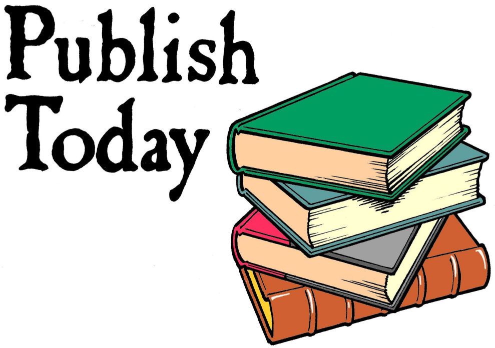
Patch

Optional
pre-release label

Breaking
changes!

- Details and specification: https://semver.org/

Open source: Publishing



- Code repository (GitHub, GitLab, BitBucket)
- Snapshots with DOI: Zenodo
- Tools
    - Package index (e.g., PyPI for Python)
    - Bioconda & Biocontainers
- Workflows
    - Dedicated registries (e.g., Workflow Hub, Dockstore)

Summary

Take-home messages

- *Design* your code
- Use appropriate tooling
- Follow good practices
- *Always* version control your code
- Publish your code (consider Open Source)
- Involve the community