# The art of developing scientific software

Inga Ulusoy, Scientific Software Center, Institute for Scientific Computing, Heidelberg University

March 2021

# Unit 2: Collaborative software development

- The programming paradigm: Planning a piece of software
- Technical debt and clean coding
- Linting and performance of code
- Documentation using sphinx (doxygen)
- More on git: “Clean” repositories

A python module will be developed.

# The programming paradigm

The programming paradigm is an approach of solving a programming problem.

declarative ⟺ imperative

Programming through defining goals but not on how to achieve these.
Focus on *what to do*.

- Logic programming paradigm
- Functional programming paradigm (Java script)
- Database processing approach

Programming through step-by-step instructions (explicit status updates of the program).
Focus on *how to do something.*

- *Procedural programming paradigm*
- *Object-oriented programming paradigm*

**Different languages support different paradigms.**
Python: Imperative, Procedural, Object-oriented, Functional

# The programming paradigm

- Procedural: Statements are structured into procedures (subroutines/`functions`) with main program calling the procedures.
  Allows reuse of procedures through modules/libraries
  Multitude of modules can lead to overhead, duplication and difficulty finding correct calls
- Object-oriented: Objects contain both data and methods (`classes`).
  Data hiding (security), code reusability, inheritance
  Programming becomes quite complex, harder to implement logic
- Functional: Statements are formulated as evaluations of mathematical functions using `lambda` calculus.
  Simple to understand and debug
  Low performance of code, hard to implement

# Planning a piece of software

- Define output
- Define required input and data flow
- Think about the programming style you want to follow
- Define the processing from input to output (the logic) of the program

Separate overhead from logic
(input and output handling)

Bottom-up:
1. Define specifics
2. Design the logic

Top-down:
1. Write logic in pseudo code
2. Work out specifics

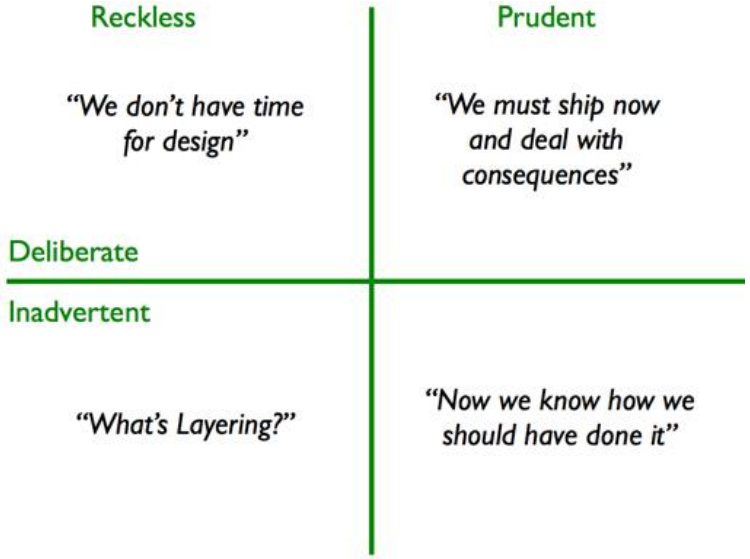
*Bottom-up approach better at detecting dependencies that influence top level*

# Planning a piece of software

- Take a piece of paper and draft a program based on the jupyter notebook that you wrote. Consider: programming paradigm (style), top-down or bottom-up? Do this without your ***team***.
- In the design, consider that your ***team*** will also have contributions to the program.

# Technical debt

- The result of prioritizing speed over clean code.

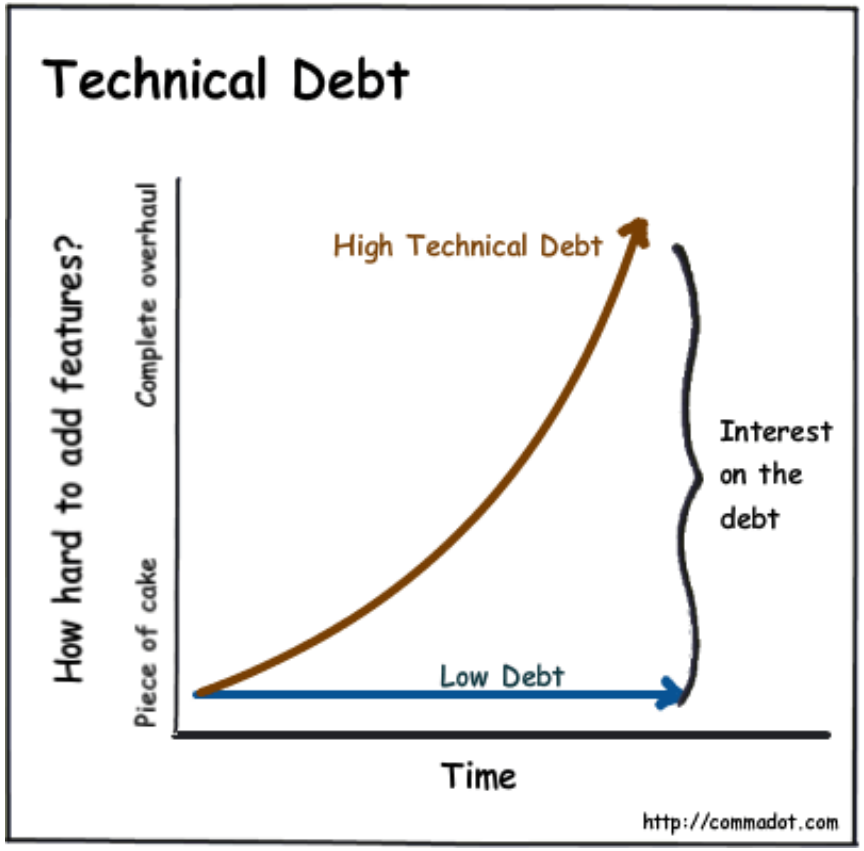



https://martinfowler.com/bliki/TechnicalDebtQuadrant.html

- Refactoring: The process of restructuring existing code and thereby removing technical debt.

# Technical debt hinders development

- A high technical debt makes it much more cumbersome to implement new features.




*You will save time in the long run by keeping your code clean.*

https://commadot.com/the-ux-of-technical-debt/

# Clean code: Coding best practices

### write comments and documentation

**Advice:** actively use comments in the documentation through tools like i.e. `sphinx` or `doxygen` – this way your comments will not become outdated!

### write efficient but readable code

Making use of functions and classes (methods) can be very efficient, but sometimes hard to read.

### proper styling

The indentation and styling should be consistent throughout the project. Avoid deep nesting and too long lines.

### proper naming conventions

The names of variables and procedures should be meaningful. *(ideally in English)*

### no imaginary future cases

Don't write code for imaginary future scenarios. Only write what you need.

### accuracy before speed

Make your piece of code work first, then improve efficiency.

### no hardcoding

Those should be constants and declared in the overhead.

### reuse functionality

Implement features in a way that enables reuse of functionality in different contexts. Do not repeat yourself.

### develop iteratively

Give breaks, don't write all code in one go. Iterate over (pieces of) the code several times.

### use helper methods

A method should only do what it is supposed to. Anything else should be contained in other functions/methods.

### use existing libraries

You will not be able to code more efficient than the pros.

### refactor code

Whenever you have time, refactor parts of your code.

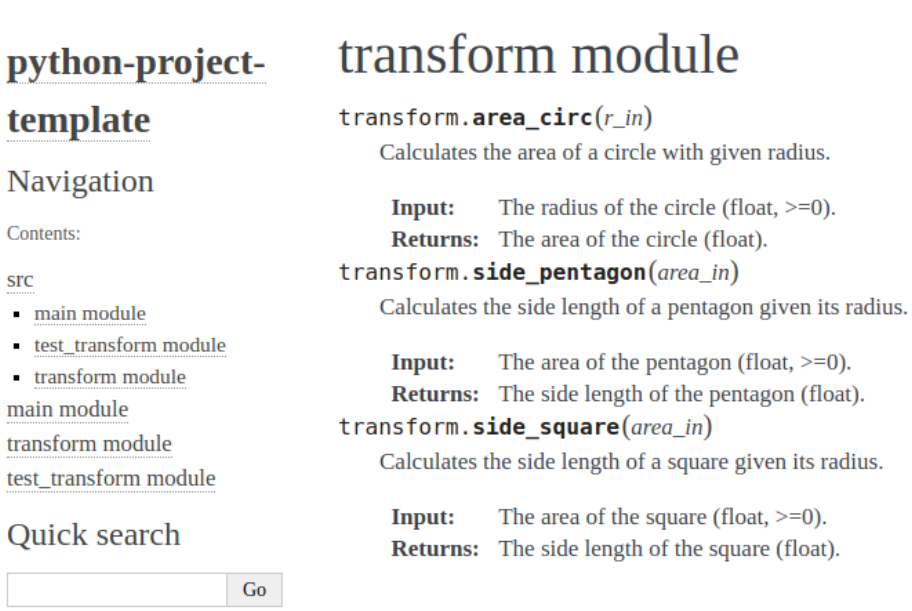
# Documentation using `sphinx`

https://www.sphinx-doc.org/en/master/
https://pythonhosted.org/an_example_pypi_project/sphinx.html

```
def area_circ(r_in):
    """Calculates the area of a circle with given radius.

    :Input: The radius of the circle (float, >=0).
    :Returns: The area of the circle (float)."""
```

use documentation string in function head



**python-project-template**

Navigation

Contents:

src
- main module
- test_transform module
- transform module

main module
transform module
test_transform module

Quick search

Go

transform module

`transform.`**`area_circ`**(*r_in*)

Calculates the area of a circle with given radius.

**Input:** The radius of the circle (float, >=0).
**Returns:** The area of the circle (float).

`transform.`**`side_pentagon`**(*area_in*)

Calculates the side length of a pentagon given its radius.

**Input:** The area of the pentagon (float, >=0).
**Returns:** The side length of the pentagon (float).

`transform.`**`side_square`**(*area_in*)

Calculates the side length of a square given its radius.

**Input:** The area of the square (float, >=0).
**Returns:** The side length of the square (float).

# Linting

https://pypi.org/project/flake8/
https://simpleisbetterthancomplex.com/packages/2016/08/05/flake8.html

A Linter automatically highlights styling and syntax errors, as well as suspicious constructs in your code.

Most IDEs run linters in the background as you work on your code, and directly highlight problems. Resolve all issues before pushing to your repository.

Additionally, you can check your code in the terminal by running the linter manually (ie. `flake8`). You will also want to run the linter in your `github actions` (we will get to this in unit 4).

# Performance of code

https://docs.python.org/3/library/timeit.html
https://docs.python.org/3/library/profile.html

You can check performance by using `timeit` or `cProfile`.

`timeit` will measure the execution time of specified bits of code.

`cProfile` enables deterministic profiling of your code, including how many times functions were called and how long these took to execute, but will add some overhead to the overall run time of your program.

# Installing additional packages

- Type `pip install –U sphinx` or `conda install sphinx`
- `pip install recommonmark` to use mark-up syntax
- `pip install flake8`
- `pip install pytest`

# Live lesson

- In the beginning of the live lesson, you will discuss with your ***team*** how to implement the jupyter notebook as a python module. Discuss the strategy that you came up with and your reasoning.
- Identify overlap in the logic/specifics and discuss the implementation considering reusability and clean code.
- Agree on some coding standards: Naming of constants/variables/functions/classes, code styling etc.
- Adhere to the ***coding best practices***!

# Live lesson

- Start your implementation as discussed with your ***team*** and make use of tools that help you adhere to ***coding best practices***.

`sphinx`

`flake8`

`timeit/cProfile`

Your IDE

# Documentation using `sphinx`

- Navigate into your `doc` directory
- Type `sphinx-quickstart`
- Answer "y" - "your project name" - "Author names" - "release version" - I would choose 1.0 as that would be the first official release version; "project language"
- Open `conf.py` and uncomment `import os, import sys, sys.path.insert(0, os.path.abspath('.'))`
- Put the correct path – ie. `sys.path.insert(0, os.path.abspath('../src/'))`
- Add `extensions = ['sphinx.ext.autodoc']`
- For a selection of themes, visit https://www.sphinx-doc.org/en/master/usage/theming.html
- Type `make html`
- Open the index.html file in your build/html directory – it should open in your browser and display the initial documentation page
- Use autodoc to generate the modules.rst file: `sphinx-apidoc -o source/ ../src`
- Type `make html`
- Again check index.html – it should have added your source code docstrings in modules