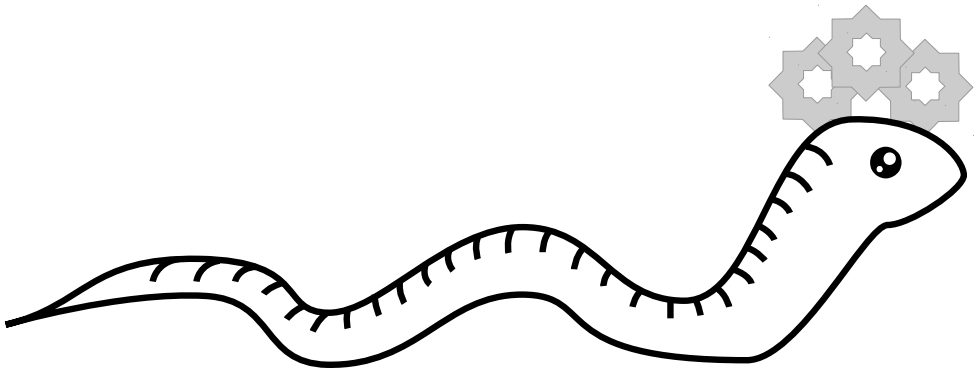
Python 3 for scientific computing

Lecture 9, 11.4.2018
Introduction to software engineering

Juha Jeronen
juha.jeronen@tut.fi



Spring 2018, TUT, Tampere
RAK-19006 Various Topics of Civil Engineering

TAMPERE
UNIVERSITY OF
TECHNOLOGY

Great virtues of a programmer?

- According to Larry Wall, the creator of the Perl programming language, the **three great virtues of a programmer** are:
  - **Laziness**: *The quality that makes you go to great effort to reduce overall energy expenditure. It makes you write labor-saving programs that other people will find useful, and document what you wrote so you don't have to answer so many questions about it.*
  - **Impatience**: *The anger you feel when the computer is being lazy. This makes you write programs that don't just react to your needs, but actually anticipate them. Or at least pretend to.*
  - **Hubris**: *Excessive pride, the sort of thing Zeus zaps you for. Also the quality that makes you write (and maintain) programs that other people won't want to say bad things about.*

DWIM [1] [2] [3] is a very Perl mindset, diametrically opposite to Python's *Simple is better than complex*; but impatience is a fine motivator for performance optimization, too.

SW Engineering is …

- *Tools and practices to develop correct and maintainable software quickly.*
  - Highly useful anywhere software is developed, including in research.
  - Even a bit of knowledge boosts programmer productivity significantly.

⚠ This lecture is only a quick run-through; large topic.

- **Practices**, such as:
  - **Comments** – code is for humans. Humans need comments.
  - **Tests** – *automated* testing. Catch *regressions*.
    - Dual role as usage examples, conveniently collected at one place.
  - **Assertions** – ensuring internal consistency; *design by contract.*
  - **Refactoring** – cleaning up by restructuring, without changing behavior.
  - **Automation** – *almost* always manual labor makes no sense.
  - Being sufficiently careful with **assumptions** (see e.g. Lehman, 2005).

- **Tools**, such as:
  - **Linters** – static analysis for potential problems and code style [1] [2]
  - **Debuggers** – interactive execution, examine program state
  - **Profilers** – dynamic analysis for finding the hot spots, reliably
  - **Version control**
    - Backtrack to any point in the project history
      - No need to keep zipped copies of the project
    - *Automatically* keep track of what exactly changed and when

Comments

- *Writing code is easy, reading code is hard*. –old programming adage (e.g. [1] [2])
- *When code and comments disagree, both are probably wrong.* –Norm Schryer

  - Reading someone else's code is especially hard.
    - But *also your past self is someone else*: six months from now, you'll have forgotten the mental model your code is built on.

  - In software development, the ratio of time spent reading vs. writing code is well over 10 : 1 [according to Robert C. Martin, in Clean Code].

- ***Code is primarily meant for humans***: programs codify imperative knowledge.
  - *Programs must be written for people to read, and only incidentally for machines to execute*. –Abelson & Sussman, SICP, preface

- *Jeff Atwood: Code Tells You How, Comments Tell You Why*:
  - **The best comment is the one you can omit** – aim at clarity in the code itself.
  - When the code ***cannot*** *be made easier to understand*, add a comment.
  - The code already says *how*; **comment the *why***.
- Bill Sourour: Putting comments in code: the good, the bad, and the ugly.
- Consider who will be reading the code, and the kind of program you're writing.
  - **Explain what your target audience doesn't know**. E.g. in teaching examples, explanations of basic concepts and links to the language manual can be useful, but in production code, readers already know all that.
- Commenting in Python, incl. PEP8 guidelines; e.g. **keep comments up to date**.

The art of communication

- **No**!

```
# recursively sum f(k-1) and f(k-2)
def f(k):
    k = int(k)  # convert to integer
    if k < 0:
        raise ValueError('Error')
    if k < 2:
        return k
    else:
        return f(k-1) + f(k-2)
```

- The comment tries to explain how *f()* works, but does not mention *what it does* – i.e. ***why*** one would use this *f()*.
  - Also, the explanation is wrong; it is missing the error case and the termination condition. **The code already tells how, in detail. DRY!**
- The audience likely knows what int() does; useless comment adds noise.
- Useless error message.

- **Yes**:

```
# compute Fibonacci numbers
def fibo(k):
    k = int(k)
    if k < 0:
        raise ValueError('Expected k ≥ 0, got %d' % k)
    if k < 2:
        return k
    else:
        return fibo(k-1) + fibo(k-2)
```

- Descriptive function name.
  - Comment now better, but maybe just omit it to further reduce noise; the name already contains the same info.
  - Or convert to a docstring, to ease the minds of those users who are only 99% sure “fibo” is not a dog.
- Descriptive error message; easy to fix an erroneous call.

⚠ But don't actually use the above program; while *memoization* can improve it significantly, there's a much easier, very efficient approach with generators.

Tests

```
def inc(x):
    return x + 1

def test_answer():
    assert inc(41) == 42
```

The bare essentials of **unit testing** (using *pytest*).

- Testing ≈ *how to write more reliable programs*. Several levels, e.g.:
  - **Unit testing**: typically individual modules, and individual functions in them
  - **Integration testing**: interfaces between components; modules as a group
  - **System testing**: the complete, integrated system as a black box

- **Unit tests** are perhaps easiest to automate, while also very useful.

  - Often, *test()* or *main()* in the module itself, or *mylibrary.py* + *test_mylibrary.py*.

  - Particularly applicable in scientific computing:
    - Near-trivial control flow; can set up tests with decent coverage of code paths
    - Relatively few connections between parts; often, individually testable parts

  - When writing tests, consider also **edge** and **corner cases**, and **invalid inputs**.
  - When you fix a bug, add a test to alert if that particular issue ever comes back.
  - Functional programming helps testing by increasing modularity; objects with mutable state are harder to test, because state may affect behavior.

  - **Frameworks** exist; popular ones for Python: pytest, nose2. Also *unittest* (std lib).
  - Coverage.py measures which lines of code were or could have been executed.

- **Not a magic bullet**: *Testing will not catch every error in the program, because it cannot evaluate every execution path in any but the most trivial programs. This problem is a superset of the halting problem, which is undecidable*.–Wikipedia

Assertions

- ***Assertion**, n.*:
  - 1. *An expression, constructed by the programmer to always evaluate to truthy, on the condition that the code has no bugs*.
  - 2. *The act of testing an assertion (sense 1), and terminating the program if the test fails. A.k.a. an **assert**.*

- In Python, to make assertions, use the **assert** statement:

```
assert expr
assert expr, message    # variant with optional error message, to help debugging

assert x > 0, 'Expected positive x, but got {}'.format(x)
```

- If the test fails, Python raises AssertionError.

- **assert False** always fails. Useful for explicitly tagging a case as a *can't happen*.
  - Making a habit of this tends to catch some bugs early, and hence accelerate development.

- See also *help('assert')* . Note the quotes, since **assert** is a statement, not an object; *help(**assert**)* is a syntax error.

⚠ In many compiled languages (e.g. C, C++), *debug builds* have asserts enabled, whereas *optimized builds* (for production use) typically omit the code for them. Also Python; running with the -O command line argument omits asserts when compiling the bytecode, and disables asserts while running.

Assertions

- **In contrast to unit tests**, assertions are:
  - Placed anywhere in the code; not centralized into a separate tester.
    - Typically used to check internal consistency, i.e. **detect situations that are logically impossible if the code is bug-free**, instead of checking computed results for correctness.
    - Asserts are sometimes also used as a poor man's unit testing framework, which terminates the whole test run at the first failed test.
      - Proper frameworks run all the tests, and then report which ones succeeded and which failed. But that needs a library.
    - Some test frameworks allow using the **assert** syntax to specify unit tests (e.g. *pytest*).
- **In contrast to error handling** [Wikipedia]:
  - Most error conditions are *logically possible*, although some are extremely unlikely to get triggered in practice.

Assertions

- **Why** use assertions:
  - To **make internal assumptions explicit**, especially where functions interact.
    - Jim Shore: Fail Fast (IEEE Software, September/October 2004, 21–25)
  - To **fail fast**, so that the computation terminates immediately upon encountering an internal inconsistency, instead of continuing with a possibly corrupt state.
    - This is important to **make bugs easier to find**. Without fail-fast, often the context in which the error arose will have been long lost, when the error finally manifests itself – gets detected by error handlers, or (hopefully) noticed by the user as bogus output.
- **Where not to use** assertions:
  - Unit testing… unless:
    - Too lazy to install a framework – never mind the tool, *having* tests is better than not.
    - The chosen framework hooks into **assert** as syntax to specify unit tests.
  - Error handling – **raise** an exception instead.
    - In the real world, many errors can be handled without terminating the whole program.
      - In scientific codes, perhaps better to keep the program simple; no need for a proper error handler, since the user will anyway stop the program, fix the hard-coded parameters, and re-run it. (Good for both documentation and reproducibility.)
      - But crashing by raising a well-chosen exception type (e.g. TypeError, ValueError, RuntimeError) may express the intent of the code more clearly than crashing with an AssertionError (which makes the situation look like there is a bug).

Contracts

- Assertions are also useful for ***design by contract*** (a.k.a. *contract programming*).
  - Introduction (in the original context of Eiffel); contracts in Racket.

- A function is a *contract*, which comes with:

  - ***Preconditions***: what the function **expects** from its caller, before it can run.
    - E.g. an argument *k* must be a positive integer.

  - ***Postconditions***: what the function **guarantees** to its caller, after it exits.
    - E.g. the return value of *cos(x)* is a real number between -1 and +1.

  - ***Invariants***: what the function **maintains**: properties that are assumed on entry, and guaranteed on exit.
    - See Class invariant (programming); similar concept: Loop invariant.

- Contracts help catch some bugs early, speeding up development.
- Advanced programming technique, not common (so far) in numerical codes.

- For Python:
  - PyContracts provides a nice set of features with a nice interface. (Try this first?)
  - A rudimentary DIY recipe from the Python Decorator Library.
  - No direct language support; suggested in PEP 316 in 2003, which was deferred.

Refactoring

- ***Refactoring**, n.*:
  - *The act of cleaning up code by restructuring it (i.e. changing its* factoring*), without altering its behavior.*

- **Why**? Refactoring:
  - Reduces *technical debt*, e.g. by:
    - Eliminating bad design decisions (which are easier to see in hindsight).
    - Applying some logical organization to an "organically grown" program.
    - Generalizing the design to enable the program to support new requirements.
  - Helps keep the codebase in a maintainable state, by improving readability.

- Many IDEs support some automatic low-level refactoring operations.
  - However, Spyder doesn't (as of v3.2.4)!
  - See rope for a refactoring tool for Python.

- Examples, for the general idea:
  - Renaming attributes or function parameters to give them clearer names.
  - Extracting some common operation into a function or a method.
  - In OOP, moving a method into a superclass or into a subclass.
  - Splitting a class or function into several to improve separation of concerns.
    - I.e. split a tangled mess into modular components. Must be done manually.
  - Introducing an abstract interface, and then changing the existing (possibly only) implementation of a feature to work through that interface, to enable adding new implementations of that feature (e.g. different output file formats).

Automation

- A mindset somewhat unique to programmers: *if it can be automated, it should*.
  - Many tasks repeat; save time in the long run, by running a script instead of manually inputting many commands each time.
  - Document a particular task, to eliminate the need to remember the details.
  - Make tasks reliably reproducible, also much later.
  - Make a solution easy and fast to apply to new instances of the problem class.
- We have already seen some automation:
  - Unit tests – a form of automated testing.
  - Build scripts – to compile, e.g. setup.py for Cython projects, or Makefiles.
- *What else can be automated?*
  - Code generation – don't want to write Fortran? Write a Python program that does that for you! (Useful module in SymPy. Real-world example.)
  - Static code analysis – make a program to generate static call graphs; get up-to-date graphics of your function dependencies at the push of a button.
  - Discretization of weak forms, in the context of solving PDE problems.
- Limitations:
  - If you need to generalize the problem first, solving that may take much longer.
  - Whether automation pays off depends on how often the task repeats and how long one instance of it takes.

Assumptions

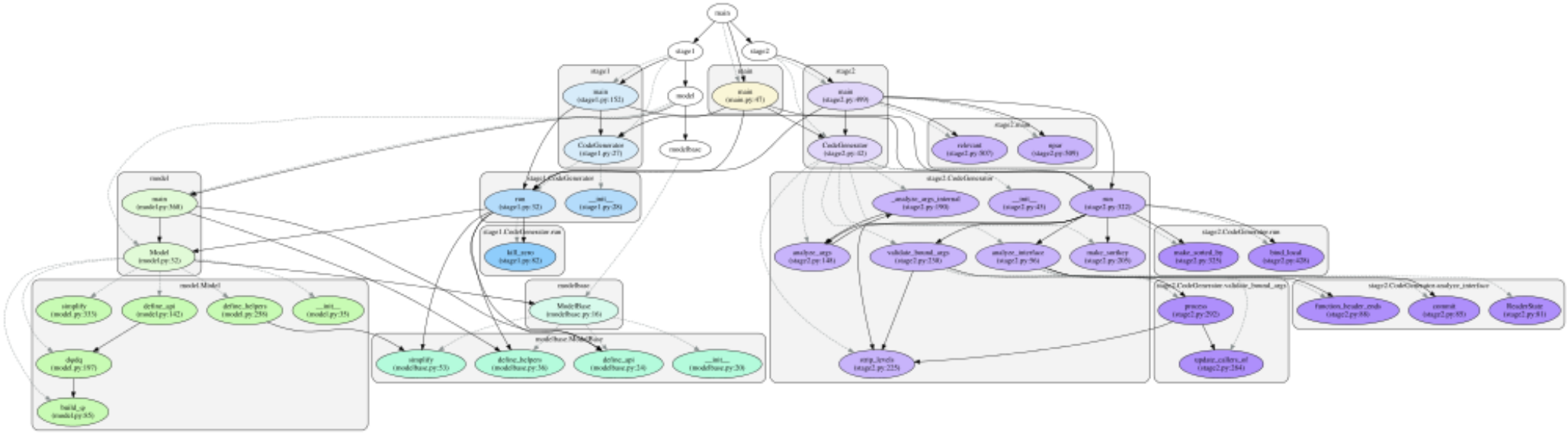
- Making assumptions in SW development often introduces *cross-cutting concerns*:
  - The effects an assumption has on the code rarely appear neatly in one place in the codebase, but rather span across it; undoing an assumption later is almost impossible.
    - Hence, be careful not to “code yourself into a corner”, to avoid the need to later scrap the whole project and start over; re-use is a major source of programmer productivity.
    - But it's a game of balance: generality brings with it extra work and extra complexity.
  - Examples:
    - Y2K, where any code handling dates needed changes.
    - Using any particular library directly to perform a task for which multiple libraries exist; first consider whether there may arise a need to switch to a different library later.
      - If so, add your own layer of abstraction, and put the actual calls inside there; much easier to switch implementation, or provide several.
      - This is how multi-platform threading libraries and GUI toolkits work; in each OS, the same things may have a completely different API.
    - (In some languages) Choosing a datatype to represent numbers. Different bit widths for ints (e.g. 32, 64), different float precisions, real vs. complex. If there is any chance that the type may need to be switched later, typedef it (Cython: **ctypedef**), don't use directly.
      - (This is the raison d'être for the silly-sounding feature of renaming an existing type.)
    - Element type in finite element codes. Linear vs. other; triangle vs. quad; scalar vs. vector (e.g. Raviart–Thomas or Nédélec). Assume, or make general – i.e. pay which price?

Linters, static analyzers

- ***Linting***: automatic static analysis of code quality, style, etc.
  - Find common problems (e.g. uninitialized variables, assigned to but unused variables) already before actually running your program.
  - Find whether your code conforms to a particular code style such as PEP8, and what to change if not.
- **For Python**:
  - pyflakes – a lightweight linter.
    - Integrated in Spyder; the engine behind the exclamation marks in the margin.
  - pylint – a much more thorough but slower linter. Includes code style analysis.
    - Also integrated in Spyder. Menu: *Source* ▷ *Run static code analysis* (or F8).
    - Requires a configuration file (*pylintrc*) to be useful. Here is one; place a copy into your project folder.
    - For more information (including how to generate a new base configuration and what customizations were made in the above), see the lecture notes, p. 102.
  - Pyan3 – static *call graphs*; see which of your functions/classes use which others.
  - Vulture – find *unreachable code*.

Linters, static analyzers

- Example of a **static call graph** as produced by Pyan3:



- Boxes represent namespaces; white ovals represent modules; a solid black arrow means "*uses*"; a dashed gray arrow means "*defines*". Hue denotes file, lightness namespace nesting depth.
- This reveals the "wiring" of the program in a visual manner:
  - A modular program appears as a set of units connected in a somewhat organized way.
  - A tangled mess of arrows means the program could use some refactoring.

- The analysis is only approximate, but still rather useful.

- Pyan3 analyzes the code and generates a GraphViz graph description file.
- GraphViz [1] [2] renders the picture (e.g. PNG or SVG), using various layout algorithms.

- This is useful when reading someone else's code (to see which parts to read first), or to quickly generate an overview of your code for your coworkers.

Debuggers

- ***Debugger***: a tool for interactive execution, to help find bugs more easily.
  - Eliminates the need to pepper the code with print calls to see what is going on.
  - Nowadays almost all languages have at least one; although GPU, heterogeneous or distributed programming environments might not (CUDA, MPI, …).
  - Familiar to many MATLAB users.

- **Typical features**:
  - **Pause** at a given line in the source code (via a breakpoint), either always or conditionally.
  - **Step**: run the current line and pause again.
  - **Step into** the next function call in the current line.
  - **Step return**: run until the next **return** and pause just before returning.
  - **Continue** until the next breakpoint, or until the program exits.
  - Examine the program state – whenever paused, look at values of any variables in scope.

- **For Python**:
  - Spyder has a graphical debugger for Python.
  - For debugging in the terminal – or programmatically – see *pdb* (standard library). Intro.
  - Cython has only a terminal debugger, cygdb, which extends GDB (the GNU Debugger).

- Requires the program being analyzed to be instrumented for debugging.
  - In some languages (e.g. C, C++), requires a *debug build*, which contains *debug symbols*.
    - The *setup.py* for Cython projects given in lecture 8 has a debug setting.
  - In pure Python, no need to do anything special.

- Because the debugger must trace the program execution, the program will run **much more slowly** in the debugger than when run normally.

Spyder debugger

- **Set or clear a breakpoint** (whenever paused, or not debugging):
  - Double-click in the left marginal (with the line numbers), on the desired line; or
  - Press F12 when the text cursor is on the desired line.

  - To set or edit a *conditional* breakpoint, Shift+F12 with the text cursor on the line.
  - See also the *Debug* menu.

  - ⚠ To be able to have a breakpoint, *the line must contain executable code*.
    If it doesn't, it may appear Spyder didn't hear you.

  - ⚠ Setting a breakpoint at the **def** line of a function pauses when Python **defines the function** – probably not what you want. Instead, set it on the first line *inside the function body* to pause when the function is entered.

- **Start debugging** (once you have some breakpoints):
  - *Debug* ⊳ *Debug* (Ctrl+F5) to **load** the program into the debugger
  - *Debug* ⊳ *Continue* (Ctrl+F12) to **run** it
    - The program runs until it hits a breakpoint, or exits.

- **Stop debugging** (terminate program):
  - *Debug* ⊳ *Stop* (Ctrl+Shift+F12)

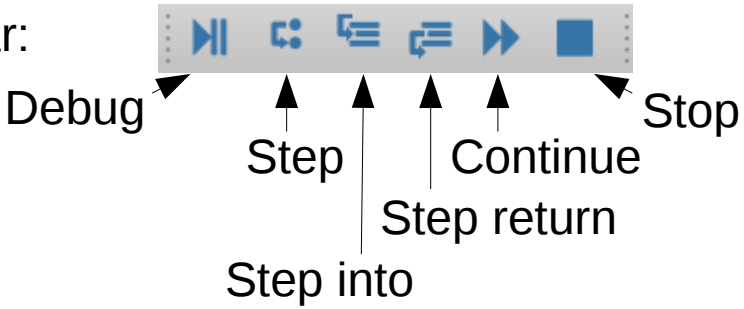
- There is no separate “restart” command. **Stop, then start again**.

Spyder debugger

- **Stepping**, once paused:
  - *Debug* ⊳ *Step* (Ctrl+F10)
  - *Debug* ⊳ *Step Into* (Ctrl+F11)
  - *Debug* ⊳ *Step Return* (Ctrl+Shift+F11)

- **Resuming**:
  - *Debug* ⊳ *Continue* (Ctrl+F12)

- Can also use the blue icons in the toolbar:



- **Examining** program state: *Variable Explorer*, at the right edge of the window.

  - Supports standard Python containers (e.g. list, set, dict) and NumPy arrays.

  - If a name is missing, the datatype of the value bound to it is likely not supported.
    - May be able to print something about it in the *ipdb* pane: ***p expr***
      (where *expr* is any Python expression). See also ***help***.
    - Unfortunately, tab completion in the Spyder / ipdb combo is currently broken.

Profilers

- ***Profiler***: a *dynamic analysis* tool to measure things such as:
  - Time spent in various functions or specific lines of code (*performance profiling*)
  - Memory usage (*memory use profiling*). (How to profile memory usage in Python.)

- Performance profiling is the most common.
- Familiar to many MATLAB users.

- **Why** profile:
  - *Pareto principle*: very roughly, a program often spends 80% of its run time in 20% of its code (hot spots). Performance profilers **help identify that 20%**.
  - Measuring (one way or another) is the **only** way to reliably locate the hot spots.

- **Types of profilers**:
  - *Tracing* – hooks into the program like a debugger, but for gathering statistics.
    - *cProfile* (function level), line_profiler (individual source line level), memory_profiler (memory usage)
    - Due to tracing, the program will run **much more slowly** than normally.
  - *Statistical* – samples the call stack at regular intervals.
    - pyinstrument (function level)

- Spyder has cProfile by default (*Run* ▷ *Profile*, F10); plugins are available to add line_profiler (*Run* ▷ *Profile line by line*) and memory_profiler (*Run* ▷ *Profile memory line by line*).
  - For line_profiler and memory_profiler, the functions to be profiled **must be tagged with the *@profile* decorator**. This magically springs into existence when the program is being run under one of these profilers – when running normally, the name does not exist.
  - Hence, use cProfile first (no configuration needed) to get a rough idea; then drill down to the details with line_profiler.

Version control

- **Perhaps the single most important tool to make a developer's life easier.**

- *Records the full project history.*
  - Nowadays also locally on your computer, not only on a remote server.
  - Allows temporarily backtracking to any point in the history
    - …and optionally branching out from it.

- *Looks just like a regular folder,* ***with only the up-to-date files visible***.
  - No need stash important versions manually, or hunt for the most recent one.

- *Automatically* keeps track of *what exactly changed and when*.
  - Allows also a short description to be logged with each complete set of changes.
    - This is very useful for searching later – like comments, but at a higher level.
- Allows tracking down exactly which set of changes introduced a particular bug.
- Allows *tagging* important versions for easily finding them (much) later.

- Allows comparing (*diff*) any two versions
  - …and creating a change-set (*patch*) out of those differences.
    - This can be used for backporting features and bugfixes.
  - Basically the only requirement for the *diff* features is that the files must be plain text. Binaries can be stored, but there is no comparison or patching support.

- **To try it quickly, to see if you find it useful**: lecture notes, pp. 80–88 has a crash course on the basics of the popular *git* DVCS, with examples.

Version control

- Keep your VCS clean:
  - **Yes**:
    - Source code (.py, .pyx, .pxd)
    - Build scripts and similar (*setup.py*, Makefile, …)
    - README, LICENSE, AUTHORS, CHANGELOG
    - Documentation (.pdf, .txt, .md, .rst, .tex, .bib, .lyx, …, .docx?)
    - Assets needed by your program, e.g. images (although they are binary files)
    - *.gitignore* (lecture notes, p. 88)
  - **No**!
    - Object files (.obj, .o)
    - Compiled binaries (.exe, [bare name], .dll, .so) and bytecode (.pyc)
    - *Generated* source code (e.g. .c files compiled by Cython from .pyx sources)
    - Temporary files
    - Editor backup files (*.*~, *.bak)
    - Simulation output data (*or at least create a separate repository to save those!*)

- Make a *.gitignore* to **automatically** ignore the files that do not belong.

- Keep your commits clean:
  - Each commit should be related to one “thing” only, e.g. some feature or bugfix.
    - Even a simple whitespace cleanup should go into a separate commit!
  - Include a **descriptive** commit message; one line is enough.
  - This makes it much easier – or possible at all! – to extract exactly the desired change-sets later, if needed.

Version control

- **Not a replacement for backups**!
  - The data is still just in one place – unless backed up, or uploaded to a server.
    - A remote “origin” repository can be useful.
  - Deleting the folder still destroys the whole project history (from that computer).
  - Rewriting the history (not recommended!) can still destroy everything.

- GitHub ≈ *social medium of source code version control*
  - As of 2018, popular for open-source projects; people will look here first.
  - Online collaboration: anyone can fork any project, work on their own copy, and then propose a pull request, asking for the changes to be included in the original.
  - Workflow: use *git* on your own computer; the “origin” repository lives on GitHub.
    - Some things can be managed from GitHub's web interface.
  - To try GitHub, start at Hello World. Also, lecture notes, p. 77.
  - BitBucket is the other popular online DVCS collaboration site.

- **Material** on *git*:
  - Sam Livingston–Gray: Think Like (a) Git
  - Ryan Tomayko: The Thing about Git [is never having to say, “you should have”]
    - A good explanation of *git add --patch*, or how to keep your commits clean even if you have worked on several things at once.
  - What is the difference between Mercurial and Git?
    - (Mercurial is the other popular open-source DVCS.)
  - Scott Chacon, Ben Straub (2014): Pro Git, 2nd ed. (book, available online)
    - For a quick taste, see esp. 3.2 Basic branching and merging

Disclaimer: IANAL – this is not legal advice.

Licensing, in 3 minutes

- By default, *copyright law forbids anyone from distributing anyone else's code*.
  - This would make open-source projects impossible, but **licensing** provides a solution.

- Hence, if you intend to publish your code, it is **critically important to include a license**, so that others can modify the code and publish their version legally.
  - License does not mean “all rights reserved”; open-source licenses are specifically designed to *provide additional rights* on top of what copyright allows.
  - See Open Source Licenses; BSD and GPL are perhaps the most popular ones.
    - BSD is a permissive free software license; the GPL is a copyleft license.
  - For works other than software, Creative Commons have licenses in a similar spirit.

- The copyright holder may license (and re-license, later or simultaneously) the work under any terms they wish – open-source, commercially, or both.
  - In your own (free-time) projects, this is you.
  - ⚠ In your work projects, **this is typically your employer** (many work contracts require a blanket transfer of copyright for anything written on the job).
    - Hence, must first check with them if publishing the code is ok at all, and which licenses they consider acceptable.

- In open-source projects, **choose carefully**: in the case of several contributors (without transfer of copyright to a central entity governing the project), **written permission** must be obtained *from every contributor* to change the license later.

  - If some contributors cannot be reached, or if even one says no, **then the license cannot be changed**, except by rewriting the affected contributions from scratch – to be safe, without looking at the original.

Meta

Next time

- Starting the final topic of the course: Functional programming.
- See you next week!