Informal introduction to Python 3 for scientific computing

Juha Jeronen

October 7, 2017

Department of
Mathematical Information Technology
University of Jyväskylä

Laboratory of
Electrical Energy Engineering
Tampere University of Technology

juha.jeronen@{jyu,tut}.fi

LECTURE NOTES
Version 1.0.0

Contents

Foreword

These lecture notes are aimed mostly at students and researchers interested in learning Python, studying or working in scientific computing, and already having experienced MATLAB and/or Fortran. We assume no previous background in computer science or software engineering, but basic or intermediate programming skill is useful.

This does *not* aim to be a comprehensive general guide to Python for general-purpose programming. For that, see the book by Mark Lutz*: Learning Python, 5th ed., O'Reilly, 2013*. For newcomers to programming in general, there is also Zed A. Shaw: *Learn Python 3 the Hard Way, Addison–Wesley, 2017*.

Instead, these notes bring together — in an extremely compressed form — various aspects of Python as seen through the lens of scientific computing. Unfortunately, it is impossible to cover everything at a hands-on level, because that would expand this material into a proper book, which I don't have the time to write right now.

Hence, links to existing online material are provided, and the reader is expected to follow them where appropriate. All the links are clickable. Although the text does show the raw URLs so that no information is lost when printing, it may be more convenient to read this material digitally (and while connected to the internet) than to print it out.

Still, this material is rather long. Wherever this does not stem simply from my own failure to summarize adequately, it is because programming in Python, even when restricted to the use case of scientific computing, is a large topic. The task of explaining Python briefly is akin to that of explaining large-deformation elasticity in five minutes — up to, and including, tensor calculus in general coordinate systems.

Some aspects not traditionally emphasized in a scientific computing context are explored, including the very basics of software engineering. From my somewhat unusual vantage point, which is a superposition state, $1/\sqrt{3}$ each, of information technology, pure and applied mathematics, and physics and engineering sciences, I feel that IT and CS have a lot to offer to scientists who program, and especially to those who either apply or study numerics.

Of all things, why software engineering? In an online essay on Python, physicist Steve Byrnes[1] makes the point that scientists often write code, but many scientists do not know how to write *good, correct software quickly*. But this is a solved problem! Software engineering is, precisely, a set of tools and practices to do exactly that. It is not only for software professionals, and contrary to popular belief, it's not scary.

Of course, an awareness of the very basics is more than enough to get started in a scientific setting, and that is the extent to which we will touch upon this topic in the final advanced section of these notes. In fact, with MATLAB, you may have already used some software engineering tools: a debugger, a performance profiler, and a (realtime) static code analyzer that alerts you to problems such as assigned to but unused variables.

In my own research work, I have used MATLAB in 2008–2012, and Python since 2012, at which point I felt it provided all the tools that my work needed, and that the benefits outweighed the cost of switching. The practical motivating factor behind writing these notes is that there has now been interest toward Python for scientific computing, at both University of Jyväskylä and at Tampere University of Technology.

Why teach Python right now? In my personal opinion: Python's capabilities have for the most part equalled, and in some respects, surpassed MATLAB; Python 3 can be considered a stable platform to build science on; and Python's popularity shows not the faintest sign of waning — having observed, for the last 15 years, Python take the programming world by storm in slow motion. I'm by no means an early adopter of new programming languages!

Although there are those who swear by the names of even more advanced languages such as Haskell and Scheme, Python has the advantage that it has entered the mainstream, and comes with ample software libraries and community support. Someone, somewhere on the internet always knows how to do the particular technical thing you want — and likely, also how to obtain optimal performance while at it.

Also, Python is a huge step up in a scientist's productivity when compared to C, C++ and Fortran. Simultaneously, Python is a proper general-purpose programming language, in contrast to a domain-specific collection of tools like MATLAB. *Python is a general high-level language done right (at least to a large extent)*.

This material contains a large number of links to the internet. If any links have gone dead, or if you find mistakes, please do tell me. Or even, feel free to send a PR on GitHub.

Oh, and, welcome to the revolution.

–JJ 2017-10-07

[1] Appendix 3 in http://sjbyrnes.com/python/

License

Homepage

https://github.com/Technologicat/python-3-scicomp-intro

(Please excuse the temporary lack of proper branding.)

Changelog

This project adheres to **semantic versioning**:

http://semver.org/

...as far as it makes sense for a set of lecture notes to adhere to a standard of *software* versioning. Thus, we interpret the rules as follows:

The API is the table of contents (TOC). If the TOC changes in any way that would invalidate existing external references to this text by section number, the major version will change (e.g. 1.0.0 to 2.0.0). Note that page numbers, on the other hand, are not considered stable, and should not be used to refer to parts of this text.

Backwards-compatible changes in functionality are any changes in content that do not break the API. This includes content changes in existing sections. This also implies that new sections or subsections may still be added, as long as any existing references by section number remain valid. Changes of this kind bump the minor version (e.g. 1.0.0 to 1.1.0).

Backwards-compatible bugfixes only correct mistakes, without breaking the API or introducing changes in content. Changes of this kind bump the patch version (e.g. 1.0.0 to 1.0.1).

Version history below.

- v1.0.0 [2017-10-07]: initial release

0 Motivation

What is Python? Let's hear it from the horse's mouth:

```
jje@arcturus:~$ man python

PYTHON(1)                  General Commands Manual                 PYTHON(1)
        [...]
        DESCRIPTION          Python is an interpreted, interactive, object-oriented
                             programming language that combines remarkable power
                             with very clear syntax.
```

Why use Python? Of course the answer depends on the scientist, but let's see if we can find a common theme.

Paraphrasing an online essay by physicist Steve Byrnes, http://sjbyrnes.com/python/:

- Free and open source
- Cleaner syntax compared to MATLAB
- General-purpose programming language, with extensive examples already on the internet on how to do *almost anything a computer can do*
- Very high abstraction level, but easy(-ish) tools available also for writing performance-critical parts
- Python is a valuable career skill (true for MATLAB, too)
- Rising trend; more and more of your current and future colleagues know Python
- Easy to learn; many learning resources available

From a blog post by data scientist Steve Tjoa, https://stevetjoa.com/305/:

- Completely free
- General–purpose language
- Beautiful syntax
- Inherently object-oriented
- Features that make easy many tasks that are difficult in other languages
  - See e.g. https://stackoverflow.com/questions/101268/hidden-features-of-python
- High level, easy to learn, fast to develop in (true for MATLAB, too)
- Popular and has a great community
- Libraries for almost everything already exist
- Specifically in signal processing, can do nearly everything MATLAB does
- Demand for Python programmers is increasing

From another essay, by Ph.D. student Hoyt Koepke, http://www.stat.washington.edu/∼hoytak/blog/whypython.html:

- Holistic language design
- Readability
- Balance of high level and low level programming
- Language interoperability
- Documentation system
- Hierarchical module system
- Data structures
- Available libraries
- Testing framework

Summarizing the above, Python is a **clear, general-purpose, high-level, well-designed, complete programming solution** that is **easy to learn**, has **extensive libraries** already available, and is **free and open source**. Note that *complete* includes scientific computing.

Also, as Python core developer Nick Coghlan writes (emphasis mine):

*It is important to keep in mind that CPython already has a significant user base (sufficient to see Python **ranked by IEEE Spectrum in 2014 as one of the top 5 programming languages in the world**), [...]*

http://python-notes.curiousefficiency.org/en/latest/python3/multicore_python.html#multicore-python

1 Practical things first

`hello.py`

```
print("Hello, world!")
```

Minimal *hello, world!* in Python 3. To run it:

```
python3 hello.py
```

Let's make note of some practical things:

- Two major versions of the Python language exist at the time of this writing. In this material, we will (almost everywhere) ignore the legacy version, and concentrate only on the current one, namely Python 3.
- Rigorously speaking, "Python" is a language *specification*, just like "C" or "Fortran".
  - Different implementations of this specification exist, including CPython, Jython, IronPython, and PyPy.
  - *CPython* is however the de facto standard implementation; all the others occupy only small niches. CPython is what you get when you install Python, and also what people almost always mean when talking about Python, the programming language.
  - Do not confuse CPython with *Cython*, which is a compiler for performance-critical code (more later).
- Some code examples online "`from mymodule import *`".
  - This injects everything from `mymodule` into the current namespace.
  - This is useful in an interactive session, where constantly typing in module paths gets in the way of your actual work.
  - But in a program, resist the temptation. Namespacing is a highly useful practical feature for improving code readability.
- Organization of this material:
  - Sections 0–3 cover practicalities and background.
    - Section 1 for tools, installation and such.
    - Section 2 for similarities and differences between Python, and MATLAB and Fortran. Library recommendations for scientific use are also listed here.
    - Section 3 places Python in the broad context of programming languages.
  - Section 4 for Python hands-on. Links to tutorials, and a collection of information useful for beginners that I have not seen gathered into one place in other beginner materials.
  - Section 5 is the "Part II" of this course, containing a selection of advanced topics. Feel free to browse and pick what you need. The final subsection on the very basics of software engineering is recommended.

1.1 Apps and tools

- At the bare minimum, you will want to bring at least your favorite terminal and shell (command prompt), and a syntax highlighting text editor (e.g. Notepad++, gedit, or TextWrangler for Windows, Linux, or OS X, respectively).

- You will likely want also *IPython*, which adds several helpful features to the interactive session, such as command history (via GNU readline), tab completion, and colored syntax highlighting in the interactive session. This is highly useful for quick interactive prototyping (think "MATLAB console for Python"). IPython is also integrated into many IDEs (integrated development environments).

  https://ipython.org/

- IPython has a graphical cousin, *Jupyter Notebook*, which nowadays supports many programming languages (still including Python). "Notebook" here refers to the style of interactive math scripting, with inline plotting, that was popularized, among other programs, by Mathematica.

  At least in Linux, Jupyter installs as a command-line application. Invoking "`jupyter notebook`" (note the space) in the terminal starts the local server, and pops up the GUI in your default web browser. You can also play around with Jupyter online, without installing it, at:

  https://try.jupyter.org/

  This may be convenient for interactive experimentation while learning Python. If you use try.jupyter.org and want to save your work, be sure to "File ▷ Download as ▷ Notebook (.ipynb)" it when you finish.

- You may also want an IDE (Integrated Development Environment), especially the one called **Spyder**.
  - *Spyder — The Scientific PYthon Development EnviRonment*:

    https://github.com/spyder-ide/spyder

    This is currently the closest to the MATLAB IDE (in a good way) you will get in Python. Spyder strikes a balance between interactive, scientific use oriented and software development oriented features, along with sufficient support for keeping multi-file projects together. It also provides a Jupyter console.
  - *PyCharm* is a commercial alternative aimed at professional software development. The freeware community edition is fine, but advanced users (and users who aim at becoming advanced in the future) should be aware that Cython features in PyCharm are only available via a commercial support license, which can be relatively expensive.
  - *Eclipse* with the *PyDev* add-on is focused on the development of large traditional (i.e. not scientific computing) applications. Advanced profiling requires a license for an add-on called *pyvmmonitor*.
  - *Pyzo* is aimed at lightweight development for scientists, but lacks management for multi-file projects.
  - *IDLE* comes with Python, but is very spartan, difficult to use, and seriously outdated. It is only mentioned here because the name may turn up in online conversations and documentation.

1.2 Installing Python

- An all-in-one scientific Python distribution is an easy option regardless of the OS platform.
  - *Anaconda* [Windows, Linux, OS X]:

    https://www.anaconda.com/
  - *Enthought Python Distribution* [Windows, Linux, OS X]:

    https://www.enthought.com/product/enthought-python-distribution/
  - *Python(x,y)* [Windows; possibly no longer maintained, latest release 2015]:

    https://python-xy.github.io/

- Scientific Python distributions typically have pre-installed the most common scientific libraries (e.g. NumPy, SciPy, Matplotlib) and all their dependencies (e.g. BLAS and LAPACK).

- In Linux and OS X, the OS itself comes with Python pre-installed. Some programmers however frown (for good reason) on using the system Python installation for software development, because it is then theoretically possible that, if you install system-level library upgrades due to the needs of your own project, this may break the OS.

  (...but very unofficially — and strictly at one's own risk — one can often get away with the system Python.)

1.3 Installing and uninstalling Python packages (libraries)

If you use Anaconda:

- Use the `conda` package manager.
- User manual:

  https://conda.io/docs/user-guide/tasks/manage-pkgs.html
- As the user manual notes, some Python packages are not available in the Anaconda repository, and in these cases it is ok to use `pip` (for general instructions, see below). But before you do, be sure to read this very short section of the user guide:

  https://conda.io/docs/user-guide/tasks/manage-pkgs.html#installing-non-conda-packages

If you use a separately installed Python on *nix (Linux, OS X):

- Use the `pip` package manager (included with Python):
  - `pip install packagename --user`
  - This installs the Python package called `packagename` just for you, without touching the OS. Administrative privileges are not required.
  - If you install things that include command-line apps, such as Cython, you may need to to modify your `PATH` (on the OS level) to look for binaries in the appropriate place (which depends on the OS).
  - When, as above, no explicit URL is specified, `pip` will install from PyPI, the *Python Package Index*:

    https://pypi.python.org/pypi

    Most Python-based projects push their releases here, since this is the one place that pretty much everyone in the Python community is aware of.
  - `pip` automatically manages dependencies at the Python package level; any required other packages are automatically installed. However, note that `pip` cannot manage low-level (C or Fortran) dependencies; you still have to install libraries such as BLAS and LAPACK by other methods.
- If the package is not on PyPI, check if the project has a page on GitHub.
  - The README often contains installation instructions.
  - Many packages are made using `setuptools`; in that case:

    ```
    python setup.py install --user
    ```
- To uninstall, use `pip`:
  - `pip uninstall packagename`
  - No need to specify `--user` for `uninstall`; `pip` will automatically look there first.
  - `pip` is able to uninstall most packages, including any that use `setuptools`, even if `pip` itself did not install the package. Basically, only packages created using legacy `distutils` contain no metadata, and thus cannot be uninstalled automatically.
  - If you have accidentally installed several versions of the same package (mainly a concern with installs not managed by `pip`), you can just run the uninstall command several times until `pip` says that the package is not installed (it will uninstall the different versions one by one).
- `pip list` shows the package names of all currently installed packages (along with version numbers).
- `pip show packagename` gives a short description of an installed package, including version number, install location, and homepage URL.

1.4 Language versions: Python 3 (current) vs. Python 2 (legacy)

- Python 3.x is the current major version of the language — **use it!**
  - Also known unoffficially as *Python 3000* or *py3k*.
  - Depending on your Python installation, as of late 2017, you may still need to invoke Python 3 explicitly as `python3`, if bare `python` for some reason still points to an installation of Python 2.7. (For example, some Linux distributions are still in the process of migrating to Python 3, or have only recently migrated.)
  - `python --version` to make sure.
  - Some Python 3 tips for scientists in particular, but mainly aimed at those migrating from Python 2:

    http://python-3-for-scientists.readthedocs.io/en/latest/
- Python 2.7 is the *final* legacy version (released 2010)
  - Legacy support will be completely dropped by the year 2020; **do not use** Python 2 any more!
    - (...unless you absolutely have to, and you know what you are doing.)
  - There will be *no* official Python 2.8, as stated in the official *Python 2.8 Un-release Schedule*:

    https://www.python.org/dev/peps/pep-0404/

    The official upgrade path for existing Python 2 codes is to port them to Python 3.
  - All major scientific projects — and *almost* all libraries, in general — have already ported.

    http://www.python3statement.org/
- To a very large extent, Python 3 and Python 2 are the same language, but there are some important differences, and a major break in backward compatibility that was required to support Unicode properly.
- The Python language developers have promised not to break backward compatibility again, until something really major happens in IT (of a magnitude similar to the popularization of the internet in the late 1990s, which eventually led to Unicode). See:

  Nick Coghlan: Why Python 4.0 won't be like Python 3.0
  http://www.curiousefficiency.org/posts/2014/08/python-4000.html

  Nick Coghlan: The transition to multilingual programming [**human**, not computer, languages!]
  http://www.curiousefficiency.org/posts/2014/08/multilingual-programming.html

  Brett Cannon: Why Python 3 exists:
  https://snarky.ca/why-python-3-exists/
- Just like Fortran or C, the Python language is still evolving. Just to give a taste, some exciting recent developments include:

  Literal string interpolation (f-strings) (3.6+):
  https://www.python.org/dev/peps/pep-0498/

  Additional unpacking generalizations (3.5+):
  https://www.python.org/dev/peps/pep-0448/

  Coroutines with async and await syntax (3.5+):
  https://www.python.org/dev/peps/pep-0492/
- PEP stands for *Python Enhancement Proposal*. It is the primary process through which new language features are introduced to Python.

  https://en.wikipedia.org/wiki/Python_(programming_language)#Development

2 Python vs. MATLAB (and Fortran)

Let us look, from a pragmatic viewpoint, at how Python compares to a long-time favorite in scientific computing, namely MATLAB. In the final subsection, we will also list some important practical points to keep in mind when working with Python, coming from a background in MATLAB and/or Fortran.

There are things that both Python and MATLAB do well, and then there are aspects where one of them is a better choice. We will now look at each of the three categories in detail.

By necessity, the following list is incomplete, but hopefully sufficient as a first overview.

2.1 Things both Python and MATLAB do well (and where to look in Python)

After fundamental core functionality, in no particular order. In MATLAB, some of the following may require toolboxes.

Takeaway message: **NumPy, SciPy, Matplotlib**. Add others as needed.

- Array-based ("vectorized") math → NumPy
  - `import numpy as np`
    - Pretty much everyone renames the `numpy` namespace to `np` upon loading; even the official NumPy documentation uses this convention.
  - `np.array`: similar to MATLAB's matrices, but based on cartesian tensors.
  - See also → Numexpr, which can eliminate temporaries, giving slightly higher performance and (potentially much) lower memory use in cases where this matters. Numexpr also gives multithreading.
- Linear equation system solvers, eigenvalue solvers → NumPy
  - Online manual:

    https://docs.scipy.org/doc/numpy/

    E.g. for linear algebra, go to NumPy Reference ▷ Linear algebra:

    https://docs.scipy.org/doc/numpy/reference/routines.linalg.html
- Sparse matrices, sparse solvers → SciPy
  - `import scipy.sparse`
  - Online manual:

    https://docs.scipy.org/doc/scipy/reference/

    For example, sparse linear algebra:

    https://docs.scipy.org/doc/scipy/reference/sparse.linalg.html
  - As of late 2017, bleeding edge: → PyRSB add-on for fast multithreaded sparse matrices
    https://github.com/michelemartone/pyrsb
- Support for MATLAB's `.mat` file format → SciPy
  - `import scipy.io`
  - `scipy.io.loadmat()`, `scipy.io.savemat()`
- Special functions (Airy, Bessel, Γ, ...) → SciPy
  - `import scipy.special`
- Numerical integration, ODE (IVP) solvers → SciPy
  - `import scipy.integrate`
- Signal processing (sampled time series, like audio) → SciPy
  - `import scipy.signal`

- Plotting → Matplotlib
  - `import matplotlib.pyplot as plt`
    - MATLAB-style procedural API: `figure()`, `plot()`, `xlabel()`, `axis()`, `title()`, `savefig()`, etc.
  - How to Matplotlib, with code examples:
    https://matplotlib.org/gallery.html
  - Matplotlib is the Swiss army chainsaw of plotting, and if you want maximum tweakability for publication-quality graphics, this is where you want to go. However, it was mainly designed as a plotter for NumPy, and sometimes can be tedious to use. For example, in data analysis, there are use cases in which Matplotlib is not very convenient. Many things have significantly improved in Matplotlib 2.0 (released 2017), but the following online essay from 2012 is still at least partly current.

    Karl Vogel: Will it Python? Machine Learning for Hackers Chapter 1, Part 5: Trellis graphs.
    http://slendermeans.org/ml4h-ch1-p5.html
  - Look also at other libraries in case those better suit your particular needs:
    - ggplot: Grammar of Graphics based plotter for Python; professional plots in very few lines of code
    - VisPy: new contender for high-performance scientific visualization using OpenGL.
      - Aims to handle 10^7 primitives in realtime.
      - Luke Campagnola: VisPy — Harnessing The GPU For Fast, High Level Visualization:
        https://www.youtube.com/watch?v=_3YoaeoilFI
        http://conference.scipy.org/proceedings/scipy2015/pdfs/luke_campagnola-vispy.pdf

- Graphical debugger → Spyder
- Interactive session → IPython, Jupyter Notebook, Spyder
- Documentation → `help(object)`
  - Also, `dir(object)` to see what attributes it has.
  - `help(object)` prints `object`'s *docstring*. Python software, just like the MATLAB library, is often extensively documented.
  - In IPython, `object?` usually also works, but in case of builtins such as `list`, might not show the full help. IPython's help screen (to open, just `?`) says that `object?` shows details about `object`, and `help` invokes Python's own help system.
  - For those interested in keyboard commands, the help reader in terminal IPython is effectively GNU `less` (as in `less` is `more` and all that).
    - The most useful are `/`, `n`, `N` (search) and `Esc`, `u` (turn off search match highlighting — less visual noise once you've found what you need). Note that the search is case sensitive. Cheat sheet:
      https://en.wikipedia.org/wiki/Less_(Unix)#Frequently_used_commands
  - The Spyder IDE has a graphical window for displaying documentation in pretty-rendered form, as long as the docstrings in the code conform to the NumpyDoc standard (e.g. NumPy, SciPy and Matplotlib all do).
    - For the curious, the standard is described at:
      https://github.com/numpy/numpy/blob/master/doc/HOWTO_DOCUMENT.rst.txt
    - See also Docstring Conventions:
      https://www.python.org/dev/peps/pep-0257/
    - It will in any case display the docstrings; just pretty-rendering is only available for NumpyDoc.
  - Also, online documentation:
    https://docs.python.org/3/
    https://docs.python.org/3/library/index.html

  - Tip for **new IPython users**:

    ```
    ipython profile create
    ipython locate profile
    ```

    This first generates a default configuration file, and then shows where it is stored.

    At that location, there should now be a file called `ipython_config.py`. Open it in a text editor, change the value of `c.TerminalInteractiveShell.autocall` to 1, and save. Restart IPython if already open.

    After this is done once, then in any interactive session, you can `help object` without the parentheses, which is much faster to type.

- Low-level interface for performance-critical code (like MATLAB's MEX) → Cython, Numba, CFFI, f2py. Here the options cover a spectrum of different approaches:
  - *Cython* is a superset of the Python language, mainly for writing code operating on arrays at native ("C", "machine") speed. It is a compiled language that produces Python modules. Cython can also be used to interface with existing C libraries, but this is not its main use in a numerical context.
  - *Numba* is a JIT (just-in-time) compiler for a subset of the Python language, based on LLVM.
  - *CFFI* is the *C Foreign Function Interface*, meant for interfacing Python with existing C code.
  - Similarly, for Fortran users, *f2py* is the *Fortran to Python interface generator*.
- Parallel computing → Cython (OpenMP), mpi4py (MPI), `concurrent.futures.ProcessPoolExecutor`, `multiprocessing`, Numexpr (easy multithreaded NumPy expressions), ZeroMQ
- Symbolic math → SymPy
  - http://docs.sympy.org/latest/index.html
  - Also has a code generator for C, C++, Fortran, Julia, Rust, Octave/MATLAB. http://docs.sympy.org/dev/modules/utilities/codegen.html
- Hypergeometric functions (Gauss, Kummer, ...) → mpmath
- Arbitrary-precision floating point (like MATLAB's VPA) → mpmath
  - http://mpmath.org/doc/current/
- GPGPU → Theano
  - GPGPU: https://en.wikipedia.org/wiki/General-purpose_computing_on_graphics_processing_units
  - Theano allows computing with NumPy arrays on both CPU and GPU with the same code.
- Convex optimization → cvxopt
- Data analysis for relational and labeled data → pandas
- B-splines → bspline (for low-level work); → SciPy (`scipy.interpolate` for simple fitting)
- Image processing → scikit-image
  - The package name is scikit-image, but the actual loadable Python module is called `skimage`; i.e. to load this library in Python, `import skimage`. This naming convention is common to the scikits.
- Machine learning → scikit-learn
- Conformal mapping → cmtoolkit

- Wrappers for SuiteSparse routines:
  - CHOLMOD sparse solver → scikit-sparse
    - In Python, to load it, `import sksparse`.
    - Be sure to use the new `sksparse`; an old, now obsolete version was called `scikits.sparse`.
  - SPQR sparse solver → sparseqr
    - This is the (very robust) "QR solver" that also MATLAB uses for sparse "`x = A \ b`".
  - Note that both of these:
    - Provide Python interfaces to functionality implemented in the SuiteSparse C++ library: http://faculty.cse.tamu.edu/davis/suitesparse.html
    - Play nicely with SciPy's sparse matrices.
- PDE solvers (with frameworks to build solvers for custom PDEs):
  - *The FEniCS computing platform* (developed at KTH): http://www.fenicsproject.org/
  - *Kratos Multiphysics* (developed at CIMNE): http://www.cimne.com/websasp/kratos/
  - *SfePy: Simple Finite Elements in Python*: http://sfepy.org/doc-devel/index.html
  - Mesh import, e.g. → GmshTranslator
- Performance profiler GUI, with clickable reports that take you to the relevant line of code.
  - Spyder IDE includes profiling integration.
    - Function-level profiling (Python's cProfile) is supported out of the box.
    - Line-level profiling (like MATLAB's) is available as an official add-on: https://github.com/spyder-ide/spyder-line-profiler
    - Memory-use profiling is available as an official add-on: https://github.com/spyder-ide/spyder-memory-profiler
- Both Python and MATLAB run on all major OS platforms (Windows, Linux, OS X).
  - Python is also available in many low-resource environments, such as smartphones and Raspberry Pi.
- In calls to library routines, no need to specify low-level details such as array sizes (contrast C, Fortran).

2.2 MATLAB advantages

- Commercial-product end-user convenience.
  - Python is the bazaar to MATLAB's one-stop shop.
    - In Python, sometimes you may need to install a library even to get a specific feature (e.g. SPQR).
    - Installation is usually trivial, but it's one more step to take, and one more line for the Dependencies section of your README.
    - Regardless of whether you intend to publish your code, it's a good idea to write down which libraries it needs — so that you can still run your code on a different computer five years from now.
  - Python does not guess.
    - To write performance-optimal code, you choose details such as sparse matrix storage formats and solver algorithms yourself — just like in Fortran.
    - This is by design; the KISS principle is part of the philosophy of the Python community.
- Community focused specifically on numerics.
  - If it's not built-in, you can find it on MATLAB File Exchange.
  - Many people working in numerics already use Python and publish libraries, but every esoteric thing developed by researchers writing their codes in MATLAB might not be (readily) available for Python.
    - For example, in kernel density estimation, the automatic bandwidth estimation method of Botev, Grotowski and Kroese, where the original code was written in MATLAB, is only available for Python in the library PyQt-Fit, which is old and has not been ported to Python 3.
- Highly polished, complete IDE for scientists.
  - Python is getting there (Spyder), but some features such as automated refactoring are not available.
  - Different Python IDEs offer different features (incl. refactoring); but Spyder is overall best for scientists.
- Interactive figure editor.
  - Matplotlib provides only very basic interactivity for exploration (zooming, panning, rotation for 3D).
  - Not a big deal; script your annotations, too. Or export to `.svg` (or `.pdf`), and annotate in Inkscape.
- Surprisingly clean 3D plotting API.
  - For historical reasons, in Python can't just `plot3()` anywhere, but must create a 3D axis object: http://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html
  - Start with `import mpl_toolkits.mplot3d.axes3d.Axes3D as Axes3D`, and see code examples.
- Maybe some specific algorithms built into MATLAB not yet available in Python?
  - Usually there are alternatives.
- SimuLink.
  - No equivalent alternative exists. This is a textbook example of a software niche that is a more likely target for commercial products than open-source projects.
  - If you really need this functionality in Python, and are comfortable with scripting, you may want to look at the Python Block Model Simulator.

    https://github.com/masfaraud/BMSpy
  - For a few more alternative suggestions, see:

    https://www.quora.com/When-will-there-be-a-Simulink-like-alternative-for-Python

2.3 Python advantages

- Anything outside scientific computing that your code may also want to do, such as:
  - Graphical user interfaces (GUI) → Kivy, PyQt, PyGTK
    - Plotting in a GUI → PyQtGraph
  - Command-line interfaces (CLI) → `argparse` in the standard library
  - Web apps → Flask, Django
  - Databases → `sqlite3`, MySQL Connector/Python, python-sql, TinyDB, redis-py, python-memcached
  - Computer science, such as natural language processing (NLP) → nltk
  - Scripting at the OS level (a very powerful replacement for shell scripts/bat files)
    - Integration (as in making existing software packages talk to each other)
  - Scripting inside apps that support Python scripts
    - Blender3D was one of the first popular apps to include a Python API (added in v2.25 in 2002).
- Cleanness of code and design.
  - Python has been jokingly described as "executable pseudocode".
    - https://en.wikipedia.org/wiki/Pseudocode
  - Clarity and simplicity are guiding principles, both in the language itself, and in the Python community. The philosophy mirrors that of:
    - *It seems that perfection is attained not when there is nothing more to add, but when there is nothing more to remove.*

      –Antoine de Saint Exupéry
    - *Everything should be made as simple as possible, but not simpler*.

      –Albert Einstein? (Maybe not in these words: https://quoteinvestigator.com/2011/05/13/einstein-simple/*)*
  - For example, aggressive namespacing:
    - You will almost always immediately know which module or library each function call comes from.
    - If no explicit module path in the call itself, look for a `from mymodule import foo` in the file (importing just that function directly into the current namespace)
    - If the module path does not ring any bells for you, look for an `import a.b.c as mymodule` in the file, to see if it was renamed locally.
    - Failing that, internet. With very little detective work, you now already have two pieces of information: not only the function name, but also its (original) module path, which together are sufficient to identify the function uniquely.
    - Contrast C, Fortran; where finding where a function comes from may require major detective work.
- A mature full software ecosystem with automatic dependency management, and tools to package and publish your own software in this ecosystem.
- Free of cost.
  - Unlimited instances, excellent for supercomputers and clusters.
  - Also convenient for small businesses and home users.
- Open source.
  - Open toolchain improves transparency and reproducibility of computational science.
    - Although to be fair, MATLAB is not typical proprietary software; its documentation often states which algorithms it uses.
  - Ensures quality of code at least in widely used projects, as reputation is important.
  - Practically all libraries are also open source. Sometimes a library does 99% of what you need...

2.4 NumPy basics, and key practical differences between Python, and MATLAB or Fortran

To translate your MATLAB knowledge into Python, read this:

https://docs.scipy.org/doc/numpy-dev/user/numpy-for-matlab-users.html

...and for matrices, use `np.array`, as indeed recommended there. No one uses `np.matrix` or the `matlib` wrapper module, but as of late 2017, that part of the documentation has still not been removed.

Python is case-sensitive, unlike MATLAB and Fortran. This also applies to names (variables); `A` and `a` are different.

NumPy is based on multidimensional arrays, instead of matrices like MATLAB. Multidimensional arrays can be thought of as rank-n **cartesian tensors** for general $n \in \mathbb{N}$. Matrices are a special case of rank-2 tensors. Vectors are usually treated as rank-1 tensors; there is no need to distinguish between row and column vectors (unlike in the matrix formalism). Following tensor terminology, N-dimensional arrays are called *rank-N arrays*. To get a feel for how to think in NumPy, see for example the documentation for `np.dot()` and `np.einsum()`.

Fundamental arithmetic operators +, -, *, /, ** always operate elementwise, like MATLAB's `.+`, `.-`, `.*`, `./`, `.^`. **Exponentiation uses **** (two stars), not ^ like in many other languages. Matrix and vector operations are mainly performed by calling functions or methods. However, for the particularly common use case of matrix multiplication, the operator "@" (`operator.matmul`) was introduced in Python 3.5, released in 2015.

("A dedicated infix operator for matrix multiplication": https://www.python.org/dev/peps/pep-0465/)

All matrices in NumPy are general; symmetricity is not checked. Special routines (or flags) may exist for the symmetric case, but not always. Under the hood, for dense matrices NumPy and SciPy use LAPACK, but not all routines have a Python interface. In the rare case where this matters, advanced users programming in Cython may want to know about the module `scipy.cython_lapack`, which provides direct low-level access to LAPACK.

Sparse matrix storage format in SciPy must be specified explicitly when creating the matrix. Usually, a good strategy is to use COO (coordinate format) when creating the matrix; and then convert it by one of the methods `tocsr()` and `tocsc()` before handing it in to a solver. See example below. Unlike in Fortran, using the wrong storage format won't break anything, but it will significantly decrease performance, because an extra conversion then occurs internally each time the sparse matrix is passed into a function that expects a different format.

For those new to the technical details of sparse matrix storage formats, Wikipedia provides an overview:

https://en.wikipedia.org/wiki/Sparse_matrix#Storing_a_sparse_matrix

Like in MATLAB, **explicit looping in Python is very slow** (compared to C and Fortran). Unlike in MATLAB, in Python the loop will *not* automatically accelerate after running for a while. Thus, when programming with NumPy, vectorized operations ought to be preferred, even more so than in MATLAB.

In MATLAB, the automatic acceleration of loops is due to a JIT (just-in-time) compiler, which is automatically invoked based on run time heuristics about code performance. By design, Python does not guess; but there is an explicit, easy-to-use JIT compiler called Numba, that can be used for acceleration.

(On JIT in MATLAB, see https://hips.seas.harvard.edu/blog/2013/05/13/jit-compilation-in-matlab/)

Another option for performance-critical, inherently serial parts of the code is to use Cython (roughly, the Python equivalent to MATLAB's MEX), to compile those parts of the project into native machine code. Cython can natively access data in NumPy arrays.

NumPy is single-threaded by design, for process-based parallelization. However, it uses BLAS for implementing things such as `np.dot()`, so a parallel BLAS makes NumPy parallel. For easy multi-threaded evaluation of NumPy expressions, see Numexpr. For more control of parallelization, see advanced topics at the end of these notes.

Python uses 0-based indexing everywhere, including NumPy. **NumPy uses C storage order by default** — i.e., for contiguous memory access, the *last* index changes the fastest — unless you explicitly request otherwise. Contrast MATLAB and Fortran, which use 1-based indexing and Fortran storage order (the first index changes the fastest).

In Python, the **indexing operator** is `[]`; e.g. `A[1,3]`. The `()` operator denotes a function call; e.g. `A(1,3)` calls the function "`A`" with the given argument values.

In NumPy, you can specifically request Fortran storage order by setting the optional keyword argument `order` to `'F'`, i.e. `order='F'`. This is accepted by many functions that create new arrays. Note that this does not affect your

indexing, but only the *internal memory layout* of the array; e.g. in a 2D array, the first index always refers to the row, regardless of storage format. The `order` argument is especially useful with `scipy.cython_lapack`. In general, the main use case for `order='F'` is interfacing with low-level external libraries that expect Fortran arrays; almost all of the time, there is no need to touch this.

Array slicing in Python is `start:end:step`, where `end` is non-inclusive; contrast MATLAB's `start:step:stop`. In Python, negative indices index from the end of the array (-1 = last element). If omitted, start defaults to beginning, stop to one-past-end, and step to 1.

Unlike MATLAB, in Python the slice syntax can only be used within the indexing operator `[]`. In NumPy, see `np.linspace()` and `np.arange()` for the other common use case of slicing for MATLAB-ists. NumPy also provides an indexable dummy object `np.r_` that generates 1D arrays from slice syntax à la MATLAB, but the other options usually look more readable.

Examples of slicing:

- From the second element onward: `A[1:]`
- Up to but not including the last element: `A[:-1]`
- Every other element, starting from the fifth element: `A[4::2]`
- All elements: `A[:]`
  - Main use case:

    `A[:] = B`

    forces a write into array `A`, whereas `A = B` overwrites the *name* `A`, pointing it to the same object as `B`.
- All elements in reverse order: `A[::-1]`
  - Here the default start and stop take into account the negative step.

In NumPy, the magic index `...` (literally, three dots; `Ellipsis`) means *insert as many ":" here as necessary*.

https://stackoverflow.com/questions/118370/how-do-you-use-the-ellipsis-slicing-syntax-in-python

Or, in other words:

The dots (`...`) represent as many colons as needed to produce a complete indexing tuple. For example, if x is a rank 5 array (i.e., it has 5 axes), then

- `x[1,2,...]` *is equivalent to* `x[1,2,:,:,:]`,
- `x[...,3]` *to* `x[:,:,:,:,3]` *and*
- `x[4,...,5,:]` *to* `x[4,:,:,5,:]`.

http://scipy.github.io/old-wiki/pages/Tentative_NumPy_Tutorial#Indexing.2C_Slicing_and_Iterating

Linear equation systems are solved by `x = np.linalg.solve(A, b)`. This is an explicit function call to a library, like in Fortran; but the objects `A` and `b` themselves keep track of their size metadata, like in MATLAB. For sparse systems, use `x = scipy.sparse.linalg.spsolve(A, b)`, or `x = sparseqr.solve(A, b)` if you use `sparseqr`.

The **rank of a matrix**, in the sense of linear algebra, is obtained by `np.linalg.matrix_rank(A)`. The function `np.rank(A)` returns the *array rank* (tensor rank), which is the number of array dimensions (number of indices).

It is possible to use Python procedurally, in a manner similar to MATLAB scripts. However, **Python is object-oriented**. Often, object instances will have a method that does the same thing as some procedural call, e.g. `np.dot(u, v)` is equivalent to `u.dot(v)`, and in Python 3.5 and later, also to `u @ v` (this will internally translate to `u.matmul(v)`).

Sometimes this matters; for example, in sparse matrix multiplication, the `dot()` method of the object instance gives much better performance than `np.dot()`, because the latter is designed for dense matrices.

Arrays are created explicitly (examples below). Beside the usual options, NumPy provides also an option to create an *empty* array, where the contents will be uninitialized. This is useful, if you are going to immediately write into all elements anyway, because then no time is wasted initially filling the array with values (e.g. zeroes or ones) that will never be used.

The most commonly used **data types** (`dtypes`) for NumPy arrays are `np.float64` (IEEE-754 double; C: double; Fortran: DOUBLE PRECISION i.e. REAL*8) and `np.complex128` (C: double complex; Fortran: COMPLEX*16). These are the same as the default real and complex types in MATLAB.

For boolean arrays, set dtype to `bool`. For integer arrays, to be explicit about the integer size, use `np.int32` and `np.int64`. As of this writing: the dtype `np.intc` is an alias for `np.int32`. Trying to set the dtype of an array to `int` (which, strictly speaking, represents *Python's* integer type), actually sets the dtype to `np.int64`.

Once you get some experience with NumPy, you may also want to know about **broadcasting rules** for array shapes:

https://docs.scipy.org/doc/numpy/user/basics.broadcasting.html

These rules govern which array shapes (beside exactly matching ones) are compatible to be used in the same elementwise operation, and what exactly NumPy does in the nontrivial cases. It's a pretty short read.

There is also another (rather old) document on this, with pictures:

http://scipy.github.io/old-wiki/pages/EricsBroadcastingDoc

Also, note that there are ready-made functions for common indexing tricks, especially `np.ravel()` and `np.unravel()` for meshgrids, and `np.ix_()` for MATLAB-style subarrays given index vectors along each dimension.

Examples:

```
import numpy as np

# empty: uninitialized memory
#
# size:
#
# 1D: (n,)
# 2D: (m, n): rows, cols
# 3D: (k, m, n)
# ...
#
A = np.empty((5,5), dtype=np.float64)
A[:,:] = 3  # fill the whole array with threes

# array filled with zeroes
#
B = np.zeros((5,5), dtype=np.float64)

# array filled with ones
#
C = np.ones((5,5), dtype=np.float64)

# array initialized from Python lists
#
M = np.array( [ [1, 2, 3],
                [4, 5, 6],
                [7, 8, 9] ], dtype=np.float64 )

# to convert the other way, NumPy array to Python list:
#
N = M.tolist()
```

```
# can also initialize a NumPy array from an existing NumPy array
#
K = np.array( M )

# the same, maybe more readable
#
L = M.copy()

# DANGER: this makes just an alias for the same object instance
#
V = M

# linear view into M (a.k.a. linearly indexed view, one-subscript view)
#
# "view" means that writing into W will write into M
#
W = M.reshape(-1)  # -1 = count number of elements automatically

# arange is sometimes useful: like range(), but creates 1D np.array
#
R = np.arange(9)  # [0,1,2,3,4,5,6,7,8]

# this is valid, because the input is a scalar
#
M[:] = 4  # fill M with fours

# but this is an error due to shape mismatch:
#
M[:] = R  # ValueError!

# solution a), use the linear view:
#
W[:] = R  # OK!

# solution b), reshape the input:
#
M[:,:] = np.reshape(R, M.shape)  # OK!

# inserting/suppressing "singleton dimensions" (see NumPy documentation)
# to unify indexing in code that must support both NumPy arrays and scalars:
#
P = np.atleast_1d( 42.0 )  # --> 1D array of one element
p = np.squeeze(P)          # --> back to scalar

# multidimensional indexing to get a subarray: difference to MATLAB:
#
I = np.array( (0,1,2), dtype=int )  # rows
J = np.array( (1,2), dtype=int )    # columns
M[I,J]          # IndexError!
M[np.ix_(I,J)]  # OK! (array containing columns 1,2 from rows 0,1,2 of M,
                #      like MATLAB's M(I,J))
```

```
# indexing a multidimensional array by vectors means something different in NumPy:
#
# M[I[0],J[0]], M[I[1],J[1]], ...
#
I = np.array( (0,2), dtype=int )  # rows
J = np.array( (1,2), dtype=int )  # columns
M[I,J]  # OK! -->[ M[0,1], M[2,2] ]

# identity matrix
#
# note: no shape tuple, always 2D and square, so just the size
#
I = np.eye(5, dtype=np.float64)  # size 5x5

# diagonal matrix
#
D = np.diag( np.arange(10) )

# tridiagonal matrix (from the 1D laplacian)
#
n = 10
d = -2 * np.ones( (n,), dtype=np.float64 )
s = np.ones( (n-1,),  dtype=np.float64 )
d = np.diag(d)      # main diagonal
u = np.diag(s, +1)  # upper subdiagonal
l = np.diag(s, -1)  # lower subdiagonal
T = l + d + u       # summing arrays elementwise

# sparse matrix
#
# (dr[k],dc[k],data[k]) is the kth nonzero matrix element.
#
import scipy.sparse
dr   = (0, 1, 4)  # row
dc   = (3, 2, 4)  # column
data = (1, 2, 3)
S = scipy.sparse.coo_matrix( (data, (dr,dc)), shape=(5,5), dtype=np.float64 )
S = S.tocsr()
print(type(S))  # it's now a scipy.sparse.csr.csr_matrix
print(S)
```

2.5 Linear algebra at the bazaar

Sometimes, you may notice that at first glance, an identical routine exists in both NumPy and SciPy. Historically, this has indicated that they have been developed independently, and either one could be better; either for your particular application, or possibly just strictly better, depending on the library versions.

However, recently there has been a push toward NumPy to provide a "basic" routine, and SciPy an "advanced" one. From the SciPy tutorial, https://docs.scipy.org/doc/scipy/reference/tutorial/linalg.html:

scipy.linalg contains all the functions in numpy.linalg, plus some other more advanced ones not contained in numpy.linalg.

Another advantage of using scipy.linalg over numpy.linalg is that it is always compiled with BLAS/LAPACK support, while for numpy this is optional.

Therefore, the scipy version might be faster depending on how numpy was installed.

Therefore, unless you don't want to add scipy as a dependency to your numpy program, use scipy.linalg instead of numpy.linalg.

Let's look at some examples.

Least-squares solvers, `numpy.linalg.lstsq()` vs. `scipy.linalg.lstsq()`.

- NumPy dispatches to LAPACK's *GELSD.
- Historically, SciPy used to dispatch to *GELSS, but as of August 2017, the driver has been made switchable (in SciPy 0.17.0), and now uses *GELSD by default.
- *GELSD is (much) faster, but uses more memory, when compared to *GELSS.
- *GELSY is also available via SciPy; may be slightly faster than *GELSD.

The conclusion is that:

- NumPy used to be much faster, but used more memory.
- Now both use the same driver (NumPy always, SciPy by default).
- SciPy now also allows switching the driver, so it offers the same performance by default, but is also more flexible; it is the "advanced" routine, while NumPy's is a "basic" one.
- See:

  https://stackoverflow.com/questions/29372559/what-is-the-difference-between-numpy-linalg-lstsq-and-scipy-linalg-lstsq

  https://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.lstsq.html

  https://docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.lstsq.html

Linear equation system solvers, `numpy.linalg.solve()` vs. `scipy.linalg.solve()`.

- NumPy always dispatches to LAPACK's *GESV.
- SciPy has an option to specify that the matrix `A` is generic, symmetric, hermitian or positive-definite; dispatching to *GESVX, *SYSVX, *HESVX and *POSVX, respectively.
  - As was mentioned, NumPy arrays are always generic (e.g. no special type for symmetric matrices). Testing a general dense data array for symmetricity is expensive, hence the use of an explicit option.
  - The *X version also provides some error analysis, the "X" stands for "expert".
- Here too, SciPy provides an advanced routine, NumPy a basic one.
- See:

  https://source.ggy.bris.ac.uk/wiki/LinAlgebraPacks

  https://scicomp.stackexchange.com/questions/10980/can-numpy-linalg-solve-use-back-substitution-when-possible

  https://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.solve.html

  https://docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.solve.html

Eigenvalue solvers, `numpy.linalg.eig()` vs. `scipy.linalg.eig()`.

- NumPy solves the standard linear eigenvalue problem $\boldsymbol{Ax} = \lambda \boldsymbol{x}$, using LAPACK's *GEEV.
- SciPy solves either the standard **or the generalized** linear eigenvalue problem $\boldsymbol{Ax} = \lambda \boldsymbol{Bx}$.
- See:

  https://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.eig.html

  https://docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.eig.html

3 Understanding Python

Now, let's delve into Python. This section provides an overview, mainly from a rather basic theoretical viewpoint.

3.1 The big-picture overview

Let us start by asking: **What kind of programming language is Python?**

The guiding principles of Python's design are embodied in the 20 one-line aphorisms — of which 19 have been written down — of *The Zen of Python*, by Tim Peters:

https://www.python.org/dev/peps/pep-0020/

In its entirety, the text reads:

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Flat is better than nested.

Sparse is better than dense.

Readability counts.

Special cases aren't special enough to break the rules.

Although practicality beats purity.

Errors should never pass silently.

Unless explicitly silenced.

In the face of ambiguity, refuse the temptation to guess.

There should be one— and preferably only one —obvious way to do it.

Although that way may not be obvious at first unless you're Dutch.

Now is better than never.

Although never is often better than *right* now.

If the implementation is hard to explain, it's a bad idea.

If the implementation is easy to explain, it may be a good idea.

Namespaces are one honking great idea — let's do more of those!

The text is also available as an easter egg in Python itself (`import this`).

In a more concrete sense, condensed into a list of bullet points, Python is:

1. Imperative
2. Interpreted
3. Object-oriented
4. Dynamically scoped
5. Duck-typed
6. Call by sharing (call by object)
7. Reflective

Let's now go through each of these in detail:

1. ***Imperative***
   - Or to borrow a physics term, classical. C, C++, Fortran, Java, MATLAB are also imperative languages. Imperative programs consist of an explicit sequence of elementary steps for the computer to perform.
   - Contrast *declarative programming* (Prolog), and its subcategory *functional programming* (Haskell, Scheme)
   - https://en.wikipedia.org/wiki/Programming_paradigm
   - Peter van Roy: Programming Paradigms for Dummies: What Every Programmer Should Know: https://www.info.ucl.ac.be/~pvr/VanRoyChapter.pdf
2. ***Interpreted***
   - Contrast *compiled*, such as C and Fortran.
   - Although strictly speaking, Python is a mix of both: at load time, Python source is compiled to *bytecode*, which runs on the Python *virtual machine* (VM).
     - This kind of hybrid design is common in many modern interpreted languages (also Java).
   - This is not as slow as it may at first sound. The VM runs pretty fast, and compiled bytecode is kept on disk, to avoid unnecessary re-compiling if the source code has not changed. Libraries ship with their bytecode pre-compiled.
   - Compiled bytecode can be *disassembled* for inspection, which advanced users may occasionally find useful. (I've needed this feature exactly once — actually, while preparing these notes.)
   - https://en.wikipedia.org/wiki/Interpreted_language
     https://en.wikipedia.org/wiki/Compiled_language
3. ***Object-oriented***
   - Everything is an object (yes, also functions)
     - Even *types of objects* are objects
   - Technically, at the top of the class hierarchy, everything *inherits from* `object`
     - Nowadays (Python 3) this is implicit when you define a class, and don't explicitly inherit it from something else (e.g. from one of your own classes).
     - When reading legacy codes (Python 2), and discussions on Python, keep in mind that in legacy versions, inheritance from `object` had to be specified manually. Classes inheriting from `object` used to be called *new-style classes*, to distinguish them from "old-style classes" that didn't.
   - Python supports *multiple inheritance*.
   - Python has no `struct`, by design. To create a data structure with named fields, make a class.
     - Classes can be defined anywhere, also inside functions.
   - Python is not as strict about object-oriented programming as e.g. Java, where you cannot even have a `main()` function (where program execution starts) without placing it inside a class. *Practicality beats purity.*
   - Contrast *procedural*; although Python supports that, too.
     - Python is not as strict about the conventions there, either. In Python, just like in MATLAB, you don't even *need* a `main()` function unless you *want* to have one. If all you need is a short script, feel free to write just the commands; that is valid Python, too.
   - https://en.wikipedia.org/wiki/Object-oriented_programming
     https://en.wikipedia.org/wiki/Procedural_programming

4. ***Dynamically scoped***

   - Contrast *lexically scoped* (a.k.a. *statically scoped*), such as the C language.
   - https://en.wikipedia.org/wiki/Scope_(computer_science)#Lexical_scoping_vs._dynamic_scoping
   - This is important. Since not many mainstream languages are dynamically scoped, let's be explicit. The linked Wikipedia page says (emphasis mine):

     *In **lexical** scoping, if a variable name's scope is a certain function, then its scope is **the program text of the function definition**: within that text, the variable name exists, and is bound to the variable's value, but outside that text, the variable name does not exist.*

     *By contrast, in **dynamic** scoping, if a variable name's scope is a certain function, then its scope is **the time-period during which the function is executing**: while the function is running, the variable name exists, and is bound to its value, but after the function returns, the variable name does not exist.*

     It is easy to see that this has implications as to which names exist during a function call. Consider a piece of a program calling a function that is defined somewhere else in the program, and refer to the above definitions.
   - The smallest unit of scope in Python is the function.
     - Unlike in C, a `for` loop does *not* have its own internal scope. The loop counter will overwrite the same name in the surrounding scope!
     - Further reading by Eli Bendersky (advanced):

       http://eli.thegreenplace.net/2015/the-scope-of-index-variables-in-pythons-for-loops/

       "In this writeup I want to explore why this is so, why it's unlikely to change, and also use it as a tracer bullet to dig into some interesting parts of the CPython compiler."
     - In Python 3, *list comprehensions* do not "leak" their loop counter this way. Example, as a `for` loop:

       ```
       i = 100
       L = []
       for i in range(5):
           L.append(i)
       print(i)  # 4
       ```

       The same example, using a list comprehension:

       ```
       i = 100
       L = [i for i in range(5)]
       print(i)  # 100
       ```
   - Name lookup starts from the current innermost scope, and proceeds outward in the surrounding scopes until a match is found. "Surrounding" is defined by the above rule of dynamic scoping. If the name does not exist anywhere in the current hierarchy of scopes, `NameError` is raised.
     - To be technically accurate, the hierarchy is the *scope*, and its parts are *namespaces*, but often these terms are used interchangeably even in textbooks.

       https://softwareengineering.stackexchange.com/a/273507
     - LEGB rule: **L**ocal, **E**nclosing, **G**lobal, **B**uilt-in.

       https://stackoverflow.com/a/292502
     - Sebastian Raschka: A Beginner's Guide to Python's Namespaces, Scope Resolution, and the LEGB Rule:

       http://sebastianraschka.com/Articles/2014_python_scope_and_namespaces.html

       (Otherwise very informative, but contains a small mistake in the introduction: `for` loops do not create their own namespace — as the author himself notes in the warning at the very end.)

5. ***Duck-typed***

- Definition:
  - *"If it walks like a duck and it quacks like a duck, then it must be a duck."*
  - Any object may be used in any context, up until it is used in a way that it does not support.
    - Validity of a particular usage is *only discovered at run time* (contrast *compile time*). This is one kind of *dynamic type checking*.
  - https://en.wikipedia.org/wiki/Duck_typing
- *No variables!*
  - Types are attached to *values* (i.e. *object instances*), which are referred to by untyped *names.*
  - *Names* are just labels. At any time, a name can be reassigned to point to a different object instance, of any type.
  - Contrast C, where variables have a type, and accept only values of that type.
  - Sometimes terminology is used loosely, and names are called "variables". This can lead to confusion, as these concepts imply different behavior.
  - A good (beginner to intermediate) reference is:

    Ned Batchelder: Facts and myths about Python names and values

    https://nedbatchelder.com/text/names.html
- Sometimes it is said that Python is both *dynamically typed* and *strongly typed*.
  - *Dynamically typed*: long story short, from https://en.wikipedia.org/wiki/Type_system:

    The main purpose of a type system is to reduce possibilities for bugs in computer programs by defining interfaces between different parts of a computer program, and then checking that the parts have been connected in a consistent way. This checking can happen ***statically*** *(at compile time),* ***dynamically*** *(at run time), or as a combination of static and dynamic checking.*

    [...]

    Programming languages that ***include dynamic type checking but not static type checking*** *are often called "****dynamically typed*** *programming languages".*

  - *Strongly typed*: there is no consensus on an exact definition, but often the use of the term implies things such as "no implicit type conversion". This means that for example, a string containing only digits does not automatically become a number (unlike in Perl!).

  - In https://stackoverflow.com/a/11328980, StackOverflow user Fred Foo writes:

    The strength of the type system in a dynamic language such as Python is really determined by how its primitives and library functions respond to different types. E.g., "+" is overloaded so that it works on two numbers or two strings, but not a string and a number. This is a design choice made when "+" was implemented, but not really a necessity following from the language's semantics.

  - Static type checking is making some inroads:
    - Type hints (3.5+):
      https://www.python.org/dev/peps/pep-0484/
    - Mypy, an experimental optional static type checker for Python:
      http://mypy-lang.org/

  - Finally, see:
    - https://en.wikipedia.org/wiki/Strong_and_weak_typing
    - https://stackoverflow.com/questions/11328920/is-python-strongly-typed
    - https://wiki.python.org/moin/Why is Python a dynamic language and also a strongly typed language
    - https://en.wikipedia.org/wiki/Type_system

6. ***Call by sharing*** (a.k.a. *call by object*)
   - Here Python differs from many other languages.
   - Contrast *call by value*; *call by reference*
   - In *call by sharing*, the caller and callee *share the same object instance*.
     - Specifically in Python, at the callee's end, assigning to the same name that was passed in the call, just creates a new name in the local namespace, masking (hiding) the one that was passed in. Once the call returns, the local namespace is destroyed. Thus, such "changes by assignment" will not be reflected at the caller's end. This aspect behaves like *call by value*.
     - If the passed object instance is mutable, any changes made to it by the callee will be visible at the caller's end, because both access the same object instance. This behaves like *call by reference*.
   - Useful essays:
     - http://effbot.org/zone/python-objects.htm
     - http://effbot.org/zone/call-by-object.htm
     - Paul Graham: What made LISP different
       http://paulgraham.com/icad.html
   - https://en.wikipedia.org/wiki/Evaluation_strategy#Call_by_sharing

7. ***Reflective***
   - Almost anything can be modified, at any time.

     https://en.wikipedia.org/wiki/Reflection_(computer_programming)
   - Python's extensive *introspection* facilities tie into this; it is possible to programmatically inspect things such as what attributes object instances or their classes have, or whether a given instance's class inherits from a given class.

     https://en.wikipedia.org/wiki/Introspection_(computer_science)

     http://www.learnpython.org/en/Code_Introspection
   - Reflection allows *monkey patching* (a.k.a. *duck punching*), i.e. dynamic modifications to a program's own code while it is running.

     https://en.wikipedia.org/wiki/Monkey_patch

     Much of this can be achieved using rather mild tools (`setattr()` on class objects at most).
   - For the ultimate power, see the built-in functions `compile()`, `eval()`, and `exec()`, which operate on source code provided (or generated) at run time.
     - The symbolic computation package SymPy has `lambdify()`, which takes a SymPy expression and returns a compiled function that can be used with NumPy arrays.
   - There is also an `ast` module in the standard library; Python can parse and analyze Python code.
     - AST: https://en.wikipedia.org/wiki/Abstract_syntax_tree
     - AST vs. CST: http://eli.thegreenplace.net/2009/02/16/abstract-vs-concrete-syntax-trees
     - SymPy builds on this, avoiding the need for a new mini-language, since the symbolic math expressions can be Python.
     - The static function call dependency analyzer *pyan* also uses `ast` to analyze the given source code.

       https://github.com/davidfraser/pyan/
     - In the context of packaging Python libraries, someone suggested using `ast` to pull the version number from the actual source code of the software itself, thus ensuring the version of the package always matches what is actually inside. The principle behind this approach is known as Don't Repeat Yourself.

       https://stackoverflow.com/a/12413800

       http://wiki.c2.com/?DontRepeatYourself
   - The downside is that, in principle, the high reflectivity makes Python code impossible to reason about statically.
     - However, in practice, this power is very rarely used to its full extent, so static code analyzers for Python do exist, and almost always give acceptable results. → pyan, Pyflakes, Pylint

Some other features that bear pointing out are:

- *Automatic memory management* (a.k.a. *garbage collection*)
  - No need to destroy object instances explicitly.
  - Similar to Java, MATLAB; contrast C, and `new`/`delete` of C++.
  - However, in the rare case where you *specifically want to* destroy an instance immediately (e.g. a multi-gigabyte array that you no longer need), use the `del` keyword. (This destroys the *name*; but if it was the last reference to the instance, the instance gets destroyed too.)
  - https://en.wikipedia.org/wiki/Garbage_collection_(computer_science)
- *First-class functions*
  - A function is just an object. It can be assigned to a name, returned as a return value, passed as an argument, and so on. Functions can be defined anywhere in the code (also inside other functions).
  - https://en.wikipedia.org/wiki/First-class_function
- Some *functional programming* constructs
  - Anonymous functions, keyword `lambda` (after *lambda calculus*), analogous to MATLAB's `@(x)`.
    - For the curious: https://en.wikipedia.org/wiki/Lambda_calculus
  - `functools` in the standard library
    - Partial application (of function arguments)
      https://en.wikipedia.org/wiki/Partial_application
      - Special case: currying
        https://en.wikipedia.org/wiki/Currying
    - `functools.reduce()` can be occasionally useful.
    - `reduce()` is a "friend" of the built-ins `map()` and `filter()`, but in Python, *list comprehensions* and *generators* nowadays offer a much more readable alternative for most use cases of the latter two.
    - See also decorator `@functools.wraps`; occasionally useful. (Copies metadata, importantly the docstring, from the wrapped function to the wrapper.)
- *No declaration of names*
  - Assigning to a name creates it (in the current namespace) if it does not already exist.
  - So, just like in MATLAB, be sure to avoid typos on the LHS of assignment statements.
    - Pretty much all other use cases involve looking up the undefined name, which will raise `NameError`.
    - The only exception to the rule are statements supporting the optional `as` keyword, since that effectively causes an assignment. These are `with` (*context manager*), `except` (*exception handler*), and `import` (module loader).
  - Similar to MATLAB; contrast: C, Java
- *Expressions* vs. *statements*
  - An expression produces a value, a statement does not. A statement just *does* something (i.e. causes *side effects*).
    - In some situations this matters; e.g., statements cannot be passed as function arguments.
  - This is a classical distinction, made in many imperative languages.
- *Named arguments*
  - Function *arguments can be passed also by name*, not only by position in the argument list.
  - Eliminates a common category of bugs where the arguments were given in the wrong order or off-by-one.
  - Improves readability, as less often used optional arguments can be passed by name.
  - Optional; can also use traditional positional arguments, or any mix of both.
  - https://docs.python.org/3/tutorial/controlflow.html#more-on-defining-functions
  - Python 3 added also *keyword-only arguments* (i.e. arguments that can be passed *only* by name): https://www.python.org/dev/peps/pep-3102/

- *Whitespace is semantic*
  - Code blocks are identified by their level of indentation *only.*
    - Improves readability by reducing visual noise, as there is no block terminator symbol.
  - Hence, consistent indentation within the same source file is required by the language syntax.
    - This also improves code readability.

3.2 Strings, Unicode, and why you should care

Handling text data, i.e. strings, is something programmers often have to deal with. It is mostly beyond the scope of this text, as we focus scientific computing.

Nevertheless, even numerical code typically cannot completely avoid dealing with strings. Not everything is expressible ASCII or Latin-1 (e.g. $\partial \int \alpha \equiv$ for printed messages); and in any case, in order to write conceptually correct programs — i.e. to avoid conflating unrelated concepts — it is crucial to understand the basics.

The takeaway message from this section is, as some university courses put it, the:

- **DUE process**: **D**ecode, **U**nicode, **E**ncode
  - Also known as the **Unicode sandwich**, because we:
  - **Decode input** from bytes; handle as **Unicode** string; then **encode output** to bytes.
  - The Unicode string is essentially just a sequence of integers, each representing a Unicode codepoint.
  - The popular `utf-8` *encoding* is a bytestream format (serialization format; protocol) that is used for disk storage, network transmission, input and output; it is not "Unicode" in the sense of "Unicode string".

To make sense out of that, here are some pointers:

- Unicode and strings in Python:
  - Ned Batchelder, PyCon 2012: Pragmatic Unicode, or, How do I stop the pain?

    https://nedbatchelder.com/text/unipain.html

    (Graphical, comprehensive, short, to the point; **recommended**.)
  - Nick Coghlan: Processing Text Files in Python 3:

    http://python-notes.curiousefficiency.org/en/latest/python3/text_file_processing.html
  - Differences between legacy and current:

    http://portingguide.readthedocs.io/en/latest/strings.html
- Python Unicode HOWTO:

  https://docs.python.org/3/howto/unicode.html
- On Unicode in general:
  - Most of the time, one Unicode codepoint represents one character, but not always; combined characters exist, made up of several codepoints. Some of them even have alternative single-codepoint versions available.

    https://en.wikipedia.org/wiki/Precomposed_character
  - Unicode standardizes the representation of very many characters, including over 2000 emoji:

    http://unicode.org/emoji/charts/full-emoji-list.html
  - Joel Spolsky: The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)

    https://www.joelonsoftware.com/2003/10/08/the-absolute-minimum-every-software-developer-absolutely-positively-must-know-about-unicode-and-character-sets-no-excuses/

To summarize:

An *encoding* specifies how the sequence of Unicode codepoints is represented as a stream of individual bytes. An encoding is used when the data needs to be transmitted between systems or stored. In general, different codepoints may need a different number of bytes to encode, and in fact, `utf-8` is such a *variable-length encoding*. Historically, variable-length encodings were introduced to save space, and to make the single-byte (actually 7-bit) ASCII into a proper subset of Unicode.

An example may be useful here. The first 128 Unicode codepoints, numbered 0–127, are defined as identical to ASCII, and in `utf-8`, they are also represented identically at the bytestream level.

Unicode codepoints 128–255 are defined as identical to those in the single-byte ISO-8859-1 (a.k.a. Latin-1) encoding; but at the bytestream level, `utf-8` represents these codepoints as two-byte sequences.

An identical bytestream representation for codepoints 128–255 is clearly impossible, because Latin-1 takes up a full byte, and `utf-8` needs to be able to represent also Unicode codepoints outside the range of Latin-1. Hence the first byte of each codepoint (in its `utf-8` bytestream representation) must be able to include a signal indicating whether a the codepoint takes up several bytes.

A Python 3 string is essentially a sequence of Unicode codepoints.

Once the input text is received as bytes and decoded into text, this — i.e. Unicode text — is what your Python program will deal with 99% of the time. Only at output time does the encoding come back into play. And in Python 3, even that is mostly automatic.

Just be sure to save your Python program (from your text editor) using the `utf-8` encoding.

If you find that you need details, see especially the HOWTO and N. Batchelder's presentation slides; both of these are highly informative.

4 How to Python

We will not get hands-on with *all* the basics in these notes, so let us begin with some useful external material.

The very basics of interactive use of Python (5–15 minutes, **recommended**):
https://docs.python.org/3/tutorial/introduction.html

A beginner tutorial, covering Python, Numpy and Matplotlib (**recommended**):
http://cs231n.github.io/python-numpy-tutorial/

- Matt Harrison: Learn 90% of Python in 90 minutes
  https://www.slideshare.net/MattHarrison4/learn-90
- David Goodger: Code Like a Pythonista: Idiomatic Python
  http://python.net/~goodger/projects/pycon/2007/idiomatic/handout.html
  This is a one-presentation crash course on Python that has some overlap with our present one.
- A large collection of Python tutorials:
  https://wiki.python.org/moin/BeginnersGuide/Programmers
- Official Python tutorial:
  https://docs.python.org/3/tutorial/
- Official NumPy quickstart tutorial (intermediate):
  https://docs.scipy.org/doc/numpy/user/quickstart.html
- Official SciPy tutorial:
  https://docs.scipy.org/doc/scipy/reference/tutorial/index.html
- Official Matplotlib gallery (example plots and the codes that generate them):
  http://matplotlib.org/gallery.html

4.1 Built-in types: primitives and containers

Object instances of **primitive types** in Python are immutable, i.e. cannot change value once created:

- `str`: string, Unicode in Python 3
- `int`: integer, no size limit

  https://stackoverflow.com/questions/13795758/what-is-sys-maxint-in-python-3

  (Answer: there isn't.)

  - If, for algorithmic reasons, you need a value that is larger than any number, just use ∞:

    ```
    inf = float('+inf')
    ```

    and don't worry that it's a float; it will compare correctly to integers, too. Also available:

    ```
    minf = float('-inf')
    nan  = float('nan')
    ```

- `float`: floating point number
  - In practice, always IEEE-754 double precision (C: double; Fortran: DOUBLE PRECISION i.e. REAL*8)
  - `import sys; print(sys.float_info)` on your machine for details:
    https://docs.python.org/3/library/sys.html#sys.float_info

- `complex`: complex number
  - `j` denotes the imaginary part (as often in engineering), but only as a suffix of a numeric constant
  - When you want just $\sqrt{-1}$ on its own, use `1j` (to avoid any font issues here, that is "one jay")
  - The trick `a+0j`, where `a` is a float, is useful when you want to force Python to interpret the datatype of your real-valued constant as `complex`
- `bytes`: arbitrary binary data

There are also abstract base classes for the numeric types, meant for things like `isinstance(x, numbers.Integral)`.

Among the *built-in types*, beside the primitives, there are **containers**.

These are mutable (with the exception of `tuple`, below), i.e. elements can be inserted, deleted and replaced after the creation of the container object instance.

- `list` (ordered sequence): `L = [1, 2, 3]`
  - To create an empty list, `L = []`
  - Important methods:
    - `append()` an item
      ```
      L.append(4)
      ```
    - `extend()` by another list (technically, by any *iterable*)
      ```
      L.extend(K)
      L += K       # this is equivalent to L.extend(K)
      ```
    - `insert()` an item before given index
    - `pop()` remove the item at given **index** (default last) and return it
      - Why "pop"? A list can be used to implement a *stack*, where the standard operations are called *push* and *pop*.
        https://en.wikipedia.org/wiki/Stack_(abstract_data_type)
    - `remove()` given **item** (will search for it in the list)
    - `sort()` in-place
    - `reverse()` in-place
  - `help(list)` for more
- `set` (unordered collection of unique elements): `S = {1, 2, 3}`
  - To create an empty set, `S = set()`
    - The notation `{}`, instead, refers to an empty dictionary. This is both for historical reasons (backwards compatibility), and because dictionaries are in practice used more often than sets.
  - The elements must be *hashable*; `set` uses a *hash table* to provide very fast lookup.
    - Only *immutable* objects can be *hashable*, as the hash value is related to contentwise equality, and Python assumes that the hash value for any given object instance remains constant.
    - Primitive types are *hashable*.
  - Important methods:
    - `add()` an item
    - `intersection()`, `union()`, `difference()` between `set`s
    - `remove()` given item
  - Testing element membership:
    ```
    if a in S:
        print("yes, object a is in set S")

    if b not in S:
        print("no, object b is not in set S")
    ```
    This works for the other containers, too.
    - For dictionaries, it will search the keys.
    - For lists, the search will be slow (linear time; lists are not meant for searching).

- `dict` (unordered mapping from keys to values): `D = {1 : 42, 'banana' : 'fruit', 'π' : 3.14}`
  - To create an empty dict, `D = {}`
  - Yes, that's a Unicode key. But actually keys can be arbitrary *hashable object instances*, not just primitives, as the dictionary uses a *hash table* for the keys. Values can be any object instances.
  - To insert or get an item, index the dictionary by the key:

    ```
    D[k] = v  # insert (or replace existing)

    D[k]  # get the value corresponding to key k
    ```

    The assignment will overwrite if key `k` already exists in `D`.

    Trying to retrieve by a key that does not exist in `D` will raise `KeyError`.
  - Important methods:
    - `pop()` remove item having given key, return its value (or an optional default value if not there)
    - `update()` by another `dict` (will insert any new keys, and overwrite items with matching keys)
    - Element membership, now in the set of keys, is tested using the `in` keyword just like for `sets`:
      ```
      if k in D:
          print("key k found in dictionary D")
      ```
    - Iteration by `keys()`, `values()`, `items()`. Examples:
      ```
      # get (key,value) pairs:
      for k,v in D.items():
          print("%s -> %s" % (k, v))

      # just the keys
      for k in D.keys():
          print(k)

      # just the values (if you don't care which key each belongs to)
      for v in D.values():
          print(v)
      ```
  - Creating a dictionary for reverse lookup (if the values of the original are hashable):

    ```
    R = { v: k for k,v in D.items() }
    ```

    Here the RHS is a *dictionary comprehension*.
- `tuple` (*immutable* version of `list`): `T = (1, 2, 3)`
  - To create an empty tuple, `T = ()`
    - Technically, Python implements `()` as a *singleton*, i.e. there is only ever one instance of it. This is because the empty tuple is immutable and unique; it would make no sense for Python to create multiple copies of it.
  - To create a one-item tuple, `T = (youritem,)`
    - The trailing comma tells Python that this is meant as a tuple, not as a parenthesized expression.
    - This is convenient when a function expects an iterable as input, and you want to give it just one item (e.g. array shape when creating 1D arrays).
  - Using a tuple (where applicable) instead of a list eliminates bugs by preventing accidental edits.
    - Remember *call-by-sharing*.
  - Tuples are *hashable*, so can be used as keys in dictionaries (and added to sets).
- `frozenset` (*immutable* version of `set`): `F = frozenset( (1, 2, 3) )`
  - The parameter to the constructor must be an iterable.

See *built-in types* in the documentation:

https://docs.python.org/3/library/stdtypes.html

4.2 Tricks for containers, iteration, data structures

- Negative numbers index from the end. For example, `L[-1]` is the last element of `L` regardless of its length. Similarly, `A[-1,:]` refers to the last row of the 2D array `A`.
- NumPy arrays behave like (although are not!) lists of lists. So if you need to, you can iterate over rows like:

```
# given a 2D NumPy array A:
for v in A:
    # v = A[0,:], A[1,:], ...
```

  To iterate over columns, transpose first:

```
# given a 2D NumPy array A:
for v in A.T:
    # v = A[:,0], A[:,1], ...
```

- *List comprehensions* for processing lists:
  - Usage:

    `B = [ dostuff(x) for x in A if condition(x) ]`

    where `dostuff()` and `condition()` are functions, `A` is the original list to be processed, and `B` is the result. Some of the parts are optional; these are also OK:

    `B = [ dostuff(x) for x in A ]`

    `B = [ x for x in A if condition(x) ]`
  - Similarly, *set comprehensions* and *dict comprehensions:*
    - `S = { i**2 for i in range(10) }`
    - `D = { i :  i**2 for i in range(10) }`
  - http://python-3-patterns-idioms-test.readthedocs.io/en/latest/Comprehensions.html
  - http://www.secnetix.de/olli/Python/list_comprehensions.hawk (old, Python 2)
  - https://en.wikipedia.org/wiki/List_comprehension
- *Generator expressions* are a cousin of list comprehensions, creating the elements *lazily*, i.e. as they are needed.
  - In general, a generator instance can be iterated over only once; the elements are not saved.
  - Example:

```
g = ( dostuff(x) for x in A if condition(x) )
for x in g:
    print(x)
```

  - Generators are not limited to these expressions; more complex generators can be created by writing functions that use the `yield` keyword.
- For more magic, see `itertools`:

  This module implements a number of iterator building blocks inspired by constructs from APL, Haskell, and SML. [...] Together, they form an "iterator algebra" making it possible to construct specialized tools succinctly and efficiently in pure Python.

  https://docs.python.org/3/library/itertools.html
- See `collections` in the standard library for some more containers. For example, `collections.OrderedDict` provides a dictionary that remembers the order items were inserted; and `collections.deque` a double-ended queue.

- If you need more advanced data structures that perform some particular task well, it is likely someone has already implemented them. For example:
  - SciPy provides a kd-tree as `scipy.spatial.cKDTree`.
    - This gives fast nearest-neighbor searches against a set of points in k space dimensions.
    - The "c" prefix just means it's the fast C implementation of SciPy's KDTree.
  - The standard library provides a min-heap as `heapq`. This is useful as a *priority queue*.
    - A priority queue is a queue where you can insert items with an associated priority. When you `heappop()` an item off the queue, you will automatically get the item that currently has the highest priority. Python's `heapq` uses a min-heap, so a smaller number means higher priority.
    - https://en.wikipedia.org/wiki/Heap_(data_structure)
- If you need named fields, define a class. It's simple:

  At the bare minimum, a class doesn't need much beside the fields (instance or static members, depending on what you want):

```
class MyData:
    z = 42  # static member, common for all instances of MyData

    def __init__(self, x, y):  # constructor
        # instance members
        self.x = x  # RHS is the x passed as argument, LHS names the field to store it in.
        self.y = y
```

  And done. Usage:

```
# store some data
m1 = MyData(2, 3)
m2 = MyData(x=5, y=17)  # passing by name is also fine (then order does not matter)

print(MyData.z)  # 42
print(m1.z)      # also 42, since a single copy of z is shared across all instances

print(m1.x)      # 2
print(m2.x)      # 5
print(MyData.x)  # AttributeError, does not exist since x is an instance member
```

  By convention, in Python class names are capitalized in CamelCase.

  The first argument to all instance methods is passed automatically, and always refers to the current object instance. The standard name for it is `self`.

- As a curious aside, to see what generators can do at the extreme — up to and including implementation of an academic multitasking operating system — see the presentation slides (advanced):

  David Beazley: A Curious Course on Coroutines and Concurrency:

  http://www.dabeaz.com/coroutines/

  Note, however, that although extremely well written, the material is old; support for asynchronous tasks has since been improved in Python 3.5, and coroutines have been decoupled from generators (to help programmers avoid wrong intuitions). See:

  https://www.python.org/dev/peps/pep-0492/

  https://www.python.org/dev/peps/pep-0380/

  https://docs.python.org/3/library/asyncio-task.html

4.3 * and **: unpacking tuples, lists and dictionaries

Unpacking is best explained by example:

```
# given:
def f(a, b, c):
    return a+b+c
data = range(1,4)  # (start,one-past-end): we get [1, 2, 3]

# we can:

a,b,c = data    # tuple unpacking; must have correct number of names on the LHS
a,*rest = data  # but this is also OK: data[0] -> a, (data[1], ...) -> rest
*rest,c = data  # and similarly, this too

s = f(a,b,c)

# we can also unpack in the call:
s = f(*data)  # data[0] -> first arg, data[1] -> second arg, ...
```

The *star operator* works also in the argument list. By convention, `args` is the standard name, but it can be anything:

```
def h(*args):
    # any positional arguments passed in are now
    # available as args[0], args[1], args[2], ...
    pass

h(a,b,c)  # a,b,c will be collected into args
```

Dictionary unpacking works similarly, but with a double star:

```
def f(**kwargs):
    # kwargs is now a dict containing named arguments
    for k,v in kwargs.items():
        print( "%s -> %s" % (k,v) )

f(a=2, b='coffee')  # a and b will be packed into kwargs
```

The "inverse" of the previous example is also available:

```
def f(a, b):
    # almost anything can display itself as a string, so %s is a safe choice.
    print("a = %s" % a)
    print("b = %s" % b)

# The dictionary keys here must match the named arguments of f()
# so that Python knows what to pass in where.
D = { a : 42, b : 'qwerty' }
f(**D)  # equivalent to f(a=42, b='qwerty')
```

See
https://stackoverflow.com/questions/2921847/what-does-the-star-operator-mean

Extended iterable unpacking (3.0+):
https://www.python.org/dev/peps/pep-3132/

Additional unpacking generalizations (3.5+):
https://www.python.org/dev/peps/pep-0448/

4.4 Python math vs. NumPy math

Due to immutability of the primitive types, scalar math expressions in pure Python essentially work by creating new object instances at each step.

NumPy arrays are a different story. For them, the ***size* is immutable, but the content is mutable**. This is because NumPy essentially makes C-level arrays accessible from Python. Note that NumPy arrays typically use C-compatible numeric types such as `np.float64` (C "`double`") and `np.int64` instead of Python's own types, although they *can* store arbitrary Python objects if desired.

Be aware that the Python and NumPy semantics do not always match down to the last detail, as the Python language itself and the numerics community have different design goals. For example, if you create a NumPy array out of literal integers without manually specifying a dtype (data type), you will get *an array with an integer dtype*. To avoid that, either force your numbers to `float` (e.g. `5.0` or equivalently `5.`, instead of just `5`), or specify `dtype=np.float64` when you create the array.

Incidentally, the immutability of the size of NumPy arrays gets rid of a common MATLAB anti-pattern, namely that of continually resizing arrays on the fly, simply by making it inconvenient to implement. (Such unnecessary resizing often causes a performance hit in carelessly written MATLAB code.)

To change the size of a NumPy array, a new array must be explicitly created, and the data copied over. Instead, if the array is immediately allocated at its final size, it needs to be created only once. Recall above (from NumPy basics) `np.zeros()`, `np.ones()`, `np.empty()`.

Although the size is immutable, **it is possible to change the *shape*** (e.g. create a view with a linear index; or transpose/roll axes), as long as `prod(shape)` (i.e. the total number of elements) stays constant.

Finally, note also that the `math` module of the standard library operates only on scalars; e.g. in `math.sin(x)`, the argument `x` must be scalar (not an array).

Vectorized math functions for NumPy arrays live in the NumPy namespace. For example, `np.sin(x)` computes `sin` elementwise for all elements of the NumPy array `x`. This is efficient, as then NumPy performs the looping internally in C.

The Python standard library includes some number types for exotic situations:

- A software implementation of base-10 fixed point and base-10 floats:
  https://docs.python.org/3/library/decimal.html
- Exact fractions:
  https://docs.python.org/3/library/fractions.html

4.5 Useful construct: for/else

Python's `for` loops have a little-known feature that sometimes comes in handy: `for/else`. Consider this rather artificial example of looking for an item in a tuple by explicitly walking over it:

```
T = tuple(range(10))

found = False
for i in T:
    if i == 42:
       found = True
       break

if not found:
    print("no 42 in T")
```

With `for/else`, the example reduces to

```
T = tuple(range(10))

for i in T:
    if i == 42:
       break
else:
    print("no 42 in T")
```

which is, disregarding the line that creates `T`, 2/7 lines $\approx$ 28.6% shorter (obviously this is the best case). It also avoids polluting the local namespace with a temporary flag variable (that the classical version leaves lying around), which would be a potential source of bugs, if the same function needs to do this in two different places and the programmer forgets to reset the flag.

This particular example is artificial, because in Python, normally one would just

```
T = tuple(range(10))

if 42 not in T:
    print("no 42 in T")
```

but the `for/else` feature is useful also in real-world use cases that exhibit a similar pattern.

Mnemonic: "`for/break/else`"; the `else` block runs if no `break` occurred.

4.6 Error handling: try/except

As many object-oriented languages, in error handling Python follows the *EAFP paradigm*: it is ***E**asier to **A**sk for **F**orgiveness than **P**ermission*. In contrast to requiring the caller to explicitly check the return value for an error code each time a function is called, like in C, errors are signaled by raising *exceptions* (like in C++, Java, and MATLAB).

Many OO languages call the signaling of an error "*throwing* an exception"; Python prefers the term "*raising*". The language keyword to signal an error is `raise`, as in

```
raise ValueError('got %s, but valid values are {"sandwich", "coffee"}.' % (s))
```

This example also demonstrates a convention: in the Python community, it is considered polite to include as much useful information about the error into the error message as reasonably possible. For example, if it's string-representable, showing the offending data value is often useful, so that the programmer can spot that they actually requested a "snadwich" from this routine (thus making the bug easier to find).

Example of dealing with exceptions:

```
import os

try:
    result = os.stat("this_file_does_not_exist.txt")
except FileNotFoundError as e:
    pass  # ignore the error (we could also do something else here)
```

The `try` block is self-explanatory; we *try to do this*, in case no errors occur. Errors are caught by *exception handlers* represented by `except` blocks, specific to each exception type (or optionally, a tuple of types).

The full construct follows the pattern:

```
try:
    ...
except ExceptionType1 as e:
    ...
except (ExceptionType2, ExceptionType3) as e:
    ...
else:
    ...
finally:
    ...
```

Similar to `for/else`, the `else` block runs after the `try` block is done, if no exception occurred. The `finally` block runs last, whether or not an exception occurred — and even if the code in the `try` block encountered a `return`. It is mainly useful for freeing any resources (such as closing open files) that were allocated in the `try` block (but see the `with` keyword below).

Note that the `except` and `else` blocks are also allowed to raise exceptions!

This may be useful:

https://stackoverflow.com/questions/6051934/purpose-of-else-and-finally-in-exception-handling

An exception does not always indicate an error. Exceptions are sometimes used for program flow control. For example, an iterable raises `StopIteration` when it is exhausted; and pressing `Ctrl+C` in the terminal, to stop the running Python program, will raise `KeyboardInterrupt`.

Finally, just like everything else, exceptions are objects; they may carry additional information about the particular error that occurred.

Built-in exception types are listed and explained in the documentation, along with what data they contain:

https://docs.python.org/3/library/exceptions.html

4.7 The context manager: with

Any allocated resources must be freed correctly, no matter how the code block terminates (i.e. whether by falling through normally, `return`, or an exception).

Since memory management in Python is automatic, this mainly concerns things such as open files and database connections. Enter the context manager.

Strictly speaking, it is possible to do this using `try`, but getting all the corner cases correct is tedious and error-prone. Thus, already in Python 2.5, a new syntax element was introduced to simplify this common use case.

To make sure that the file `f` is closed when the processing is done, use the `with` keyword:

```
with open("my_utf8_text_file.txt", "rt", encoding="utf-8") as f:
    for line in f:
        print(line.strip())
```

And that's it.

4.7.1 Advanced: Adding context manager support to your own class

Your own classes can also support `with`. Example (performance benchmarking):

```
import time

class SimpleTimer:
    def __init__(self, label="", n=None):
        self.label = label
        self.n     = n      # number of repetitions (for averaging)

    # this runs when the "with" block is entered
    def __enter__(self):
        self.t0 = time.time()
        return self  # this return value goes into the "as s" below

    # this runs when the "with" block exits
    def __exit__(self, errtype, errvalue, traceback):
        dt         = time.time() - self.t0
        identifier = ("%s" % self.label) if len(self.label) else "time taken: "
        avg        = (", avg. %gs per run" % (dt/self.n)) if self.n is not None else ""
        print( "%s%gs%s" % (identifier, dt, avg) )

def main():
    print("Benchmarking do_stuff():")

    with SimpleTimer(label="    done in ")) as s:
        do_stuff()

    reps = 100
    with SimpleTimer(label="    %d reps done in " % (reps)), n=reps) as s:
        for k in range(reps):
            do_stuff()

main()
```

This example is again somewhat artificial; for this use case, IPython has the `%timeit` magic command; one could just import this module and `%timeit do_stuff()`.

Takeaway message: to support `with`, an object just needs to have the methods `__enter__()` and `__exit__()`, and generally, when `__enter__()` finishes, it should `return self`.

4.8 Saving and loading NumPy arrays

There are a few different ways to do this.

All the methods for saving and loading arrays have one thing in common. When loading a data file, no loader will inject variables from the loaded data into the current namespace à la MATLAB; and to force Python to do that is hard if not impossible. This is by design; Python prefers to keep things clean and predictable.

Instead, each loader gives you a Python dictionary, from which you should grab what you need.

4.8.1 Optional: The Bunch pattern

Manually grabbing each array can quickly get tedious when the file contains a lot of arrays. Consider:

```
A = data["A"]
B = data["B"]
C = data["C"]
[...]
Z = data["Z"]
```

and surely, the alternative "do nothing", i.e. just refer to the arrays as `data["A"]` etc. everywhere they are used, is not a practical option.

Thus, is there a way to automate this, to reduce the amount of both typing and repetitive-looking code?

It turns out that, although it is difficult to inject names into the current namespace, automatically creating and populating *fields of an object* is easy, via what is known in the programming community as *the Bunch pattern*. A Bunch takes in a dictionary, and results in an object that has the dictionary keys as the names of its fields.

Variant A1 — minimalist, *namespace injection by setattr()*

```
class Bunch:
    def __init__(self, **kwargs):
        for key in kwargs:
            setattr(self, key, kwargs[key])

# usage (data is a dict)
b = Bunch(**data)
print(b.A)  # assuming there was a data["A"]
```

Variant A2 — minimalist, *directly manipulating self.__dict__*

The takeaway message here is really that `self.__dict__` stores the mapping, from an object's attribute names, to the actual attributes. It is just a dictionary like any other.

```
class Bunch:
    def __init__(self, adict):
        self.__dict__.update(adict)

# usage (data is a dict)
b = Bunch(data)
print(b.A)
```

Variant B — comprehensive

This one is `print()`able (i.e. knows how to make a string representation of itself), deep-copyable (the `copy()` method copies the data, does not just make a reference), and can convert back to `dict`. Source:

http://stackoverflow.com/questions/2597278/python-load-variables-in-a-dict-into-namespace

```
class Bunch(object):
    def __init__(self, adict):
        self.__dict__.update(adict)

    def __str__(self):
        return str(self.__dict__)

    def as_dict(self):
        return self.__dict__

    def copy(self):
        return Bunch(self.__dict__)

# usage (data is a dict)
b = Bunch(data)
print(b.A)
```

Variant C — use a library

If you prefer, install → bunch from PyPI, which gives you a ready-made Bunch and some utility functions. (But will it remain maintained? This is simple, so perhaps better to roll your own to avoid introducing dependencies.)

Now, let's look at the options for saving and loading NumPy arrays.

4.8.2 MATLAB .mat files

If you need to interact with MATLAB (or MATLAB users), this is convenient.

```
import scipy.io

# saving
data = { 'A' : A }  # dict; name --> object
scipy.io.savemat('my_datafile.mat', mdict=data)

# loading
data = scipy.io.loadmat('my_datafile.mat')  # --> dict
A = data["A"]  # get matrix "A" from the dict
```

4.8.3 NumPy .npz files

NumPy's own binary format, gzip compressed.

```
import numpy as np

# saving
data = { 'A' : A }  # dict; name --> object
np.savez_compressed('my_datafile.npz', **data)

# loading
with np.load('my_datafile.npz') as data:
    A = data['A']
```

The documentation for `np.load()` states that an `.npz` file must be closed when done, so we use a `with` block (which `np.load()` supports for `.npz` files).

https://docs.scipy.org/doc/numpy/reference/routines.io.html

Note that since in Python the smallest unit of scope is the function, the name `A` created inside the `with` block will remain alive outside of it (which is of course what we want).

4.8.4 Serialization (pickling)

In Python, almost any object instance can be *serialized*; or in plain English, converted into a bytestream, which can be saved to disk, or sent over the network, and then converted back later.

This is a huge benefit of Python, and of high-level languages in general. Traditionally, serialization has involved a lot of tedious manual work toward supporting it from the part of the developer; essentially, writing routines to save all relevant data from a given type of object, and to read it back in and populate a new object instance with it.

This can be used for saving NumPy arrays; either directly in the case of a single array, or if multiple arrays, by storing them into a dictionary (which can also keep their names, as keys) and serializing that. Serialization is also useful for inter-process communication (IPC) in parallel computing; incidentally, this is how mpi4py transmits Python objects.

Python calls serialization *pickling*, after the `pickle` module (in the standard library) that implements it.

If a particular type of object cannot be pickled, its documentation (or the documentation of the library that object belongs to) will usually state so. This is sometimes the case for objects implemented in native extension modules, where supporting serialization may require large amounts of extra work from the library developer.

Note that the pickle format has no file header, or even a magic identifier, so the file type will not be recognized by the `file` command on *nix systems (Linux, OS X), or by graphical file managers (Windows Explorer). Thus, if a user has a pickle file, but doesn't know that it's a pickle file, there's no way to tell for sure (short of trying to load it in Python).

So this file format is the opposite of self-documenting. With that warning aside:

```
import pickle

# saving
# data: dict, name --> object
with open('my_datafile.bin', 'wb') as f:
    pickle.dump(data, f, protocol=pickle.HIGHEST_PROTOCOL)

# loading
with open('my_datafile.bin', 'rb') as f:
    data = pickle.load(f)
A = data["A"]   # etc.
```

https://stackoverflow.com/questions/13939913/how-to-test-if-a-file-has-been-created-by-pickle
(Short answer: you can't.)

https://en.wikipedia.org/wiki/Serialization

4.8.5 Text files

For ultimate interoperability: uncompressed, human-readable text files. Can only handle a single array per file.

```
import numpy as np

# saving
np.savetxt('A.txt', A)

# loading
A = np.loadtxt('A.txt')

# loading alternative, for importing data:
A = np.genfromtxt('A.txt')
```

See the help pages for optional arguments; e.g. `np.genfromtxt()` has `delimiter`, which is useful for importing data in `.csv` (comma-separated values) format.

4.9 Tips and tricks

- By returning a `tuple`, a function may pack an arbitrary number of outputs into the return value. Output arguments in the sense of C or Fortran are not needed.
- If you don't need a particular output, dummy it out:

  ```
  a,b,c = do_stuff()

  _,b,c = do_stuff()  # _ is a dummy
  a,_,c = do_stuff()
  a,*_  = do_stuff()  # tuple unpacking into _ to discard all after "a"
  ```

  Standard names for the dummy variable are `_` (single underscore) and the explicit `dummy`.
- The module name of a Python module is always the filename (without the .py). Unlike Java, the module name is not declared separately. Module path, roughly speaking, corresponds to the directory path.
- Useful especially in looping: `range(n)` is `[0, 1, ..., n-1]`. See `help(range)` for more options.
- Chained comparisons: `1 < x < 2`. Does exactly what a mathematician would expect, and evaluates each term at most once.

  ("At most": The comparison will start from the left, and short-circuit if a term evaluates to `False`.)

  https://docs.python.org/3/library/stdtypes.html#comparisons
- To continue an expression to the next line, the line continuation symbol in Python is backslash "\", like in C. It is only needed if there is no parenthesis (, bracket [or brace { currently open (unmatched). On a continued line, indentation is not semantic, so it can be used for maximal clarity:

  ```
  s = a \
    + b
  ```
- Exponentiation is denoted by `x**y` (two stars). Works also for float exponents. Same as `pow(x,y)`.
- The division operator / always produces a float (in Python 3!) — as it mathematically should, considering that integers alone do not form a division ring. To force integer division, use // (two slashes).
- The primitive logic operators are bare lowercase English words: `and`, `or`, `not`.
- Exclusive OR is obtained by testing for inequality: `a != b`.
- Pythonic swap:

  ```
  a,b = b,a
  ```

  No temporary. The general case of this is built on the unpacking mechanism. Internal workings (advanced):

  https://stackoverflow.com/a/21047622
- *Escape sequences* in strings (for a table, scroll down by one screen or so). Mostly the same as in C.

  https://docs.python.org/3/reference/lexical_analysis.html#string-and-bytes-literals

  https://en.wikipedia.org/wiki/Escape_sequence#Programming_languages

- Raw strings, i.e. how to disable *escape sequences*. Prefix the string with `r`, just outside the opening quote.

  ```
  s = r"$\theta$"
  ```

  gives LaTeX for "θ", instead of the nonsensical string "`$<tab>heta$`". Useful for plot labels and such.

  When disabling is not an option (maybe you need newlines, "`\n`"?), the classic trick from C, of escaping each backslash, works in Python too.

  ```
  s = "$\\theta$"   # <-- no r"..."
  ```

  also gives LaTeX for "θ".

- Matplotlib supports a subset of LaTeX in all labels, and autodetects if you pass in LaTeX math. No configuration needed. Internal layout engine, very fast, does not call LaTeX (unless explicitly told to).
  - Particularly, `\text{}` is not available; use `\mathrm{}` to de-italicize locally; e.g. the d in $\int f(x)\,\mathrm{d}x$, which is usual, but also the "if" in $H(x) = 1$ if $x > 0$, where `\text{}` would be semantically more appropriate.
  - Forcing Matplotlib to render using external LaTeX: https://matplotlib.org/users/usetex.html

- `str.join()`. Example:

  ```
  L = [str(x) for x in range(5)]  # ["0", "1", "2", "3", "4"]
  s = ",".join(L)   # mnemonic: separator.join(list_of_items)
  print(s)  # "0,1,2,3,4"
  ```

- For file paths, always use `os.path.join()`:

  ```
  import os

  mypath = os.path.join( 'dir', 'subdir', 'file.txt' )
  ```

  instead of writing *either* `dir\subdir\file.txt` *or* `dir/subdir/file.txt`, because *both are wrong*.

  Windows uses "`\`" as path separator, whereas *nix (Linux, OS X) uses "`/`". The `os.path` module detects which OS it is running on, and uses the correct path separator.

- *Regular expressions* for string matching and substitution: see the `re` module in the standard library, especially `re.findall()` and `re.sub()`. Delving further into this topic is beyond the scope of this material.

  https://en.wikipedia.org/wiki/Regular_expression

- Python's *ternary if* syntax (equivalent to C's "`b?a:c`"):

  ```
  x = a if b else c
  ```

  This is essentially an *expression form* of the `if` *statement*. This can be used anywhere an expression is accepted. The `else` part is mandatory; essentially, because an expression must always produce a value.

- Two kinds of equality: `==` tests *equality* (in terms of content), whereas `is` tests *object identity*:

  ```
  # Given:
  a = 23
  b = 23
  c = a

  # We have:
  a is b  # False
  a == b  # True
  a is c  # True (both names a and c point to the same object instance)
  ```

- Multiplying a string by a positive integer `n` repeats it `n` times. For example, to print a horizontal line:

```
print(79 * "=")
```

- Multiplication works also for lists. Note that it doesn't copy elements, but only creates new references:

```
a = ['item'] * 3              # ['item', 'item', 'item']
a[0] is a[1] is a[2]          # True ...they're all the same item!
a = [] * 3                    # [], since the list had no items to repeat
a = [ [] ] * 3                # 3 references to the *same* empty list
a = [ [] for j in range(3) ]  # 3 *independent* empty lists
```

- Practical guide to *string formatting*, i.e. how to e.g. `"%d" % (myvariable)`:

  https://pyformat.info/

  Disclaimer: in this text, we use the old "C printf style" string formatting. As of late 2017, the standard Python 3 way is to use the `format()` method of `str` instances. For now, pick whichever suits you best.

  In my personal opinion as a long-time programmer, `format()` has always felt clunky to use. Apparently, other programmers have felt the same: in Python 3.6, a new, third style for string formatting called f-strings was introduced. It uses the same formatting mini-language as `format()`, but it is now cleaner to feed in variables, and inline code for calculating values is allowed.

  Once Python 3.6 becomes more common, the new style will likely become the dominant style for string formatting. For documentation, see:

  f-strings:
  https://docs.python.org/3/reference/lexical_analysis.html#formatted-string-literals

  Both f-strings and format():
  https://docs.python.org/3/library/string.html#formatspec

  C printf style:
  https://docs.python.org/3/library/stdtypes.html#printf-style-string-formatting

- You can freely mix and match different argument passing styles in the same function. This is a pythonic way to implement optional arguments. Example:

```
def f(a, b, *args, **kwargs):
    print("a = %s" % a)
    print("b = %s" % b)
    for x in args:
        print(x)
    for k,v in kwargs:
        print("%s -> %s" % (k,v))
    if "foo" in kwargs:
        print("foo was given, its value is %s" % kwargs["foo"])

# a and b are matched to formal argument names in the definition of f().
# f() does not have a named argument "c", so c goes into kwargs.
f(a=2, b=3, c=5)

# The first two arguments go into a and b. Again, c goes into kwargs.
f(2, 3, c=5)

# The first two arguments go into a and b; the third goes into args.
f(2, 3, 5)
```

- Default argument values can be specified in the style of C++ and Java:

```
def f(a=42):
    pass  # do whatever

f()  # uses default value for "a", since not given
```

- But beware; defaults should almost never be mutable objects. Do *not* use an empty list as a default! (Use an empty tuple instead.) Because if you do, you'll find that:

```
def f(L=[]):  # don't do this!
    L.append(1)
    return L

print(f())  # [1]
print(f())  # [1, 1]
print(f())  # [1, 1, 1]
```

  What happened? The function definition, when treating default arguments, *binds to object instance*, not to value. The definition of `f()` creates an empty list — exactly once, when Python executes the line with the `def` — and uses *this list instance* as the default value for `L`.

  There are use cases where this is actually desirable. (Can you think of one?)

  http://docs.python-guide.org/en/latest/writing/gotchas/

- To specify "not given" as a default value for an explicitly named argument (like `L` above), it is standard to use the special value `None`.
  - This helps toward writing *self-documenting code*.
  - When the user asks for `help(f)`, Python automatically shows the *function signature* (essentially, the `def` up to the colon) above the docstring.
  - Thus any explicitly named arguments will show up automatically (making the user aware of their existence), while any arguments handled via `*args` or `**kwargs` will not; those must be documented manually.

- `None` is a *singleton*; or in plain English, there is only ever one instance of it. So the standard check for none-ness is to test the object identity against `None`:

  ```
  x is None
  ```

  To check for non-none-ness:

  ```
  x is not None
  ```

  Yes, literally in English. (Also note the capital N in `None`; it is grammatically a proper noun, as (arguably) expected for a singleton.)

- Some Python programmers use the pattern

  ```
  y = x or my_default
  ```

  which sets `y` to `x` if `x is not None`, and otherwise to `my_default`. This is an abbreviation for

  ```
  y = x if x is not None else my_default
  ```

  Explicit is better than implicit, but *readability counts*.

  Strictly speaking, the abbreviation works only if we make some generally reasonable assumptions about `x`. For the exact rules on how objects behave in logical expressions, see:

  https://docs.python.org/3/library/stdtypes.html#truth-value-testing

- In Python, `for` loops (roughly speaking) iterate directly over the items of a collection; also `range()` is just a collection.

  But sometimes you absolutely need the index number in a sequence of items. In this case, `enumerate()` it:

  ```
  for i,x in enumerate(L):   # (0,L[0]), (1,L[1]), ...
      print( "element %d of L is %s" % (i, x) )
  ```

  This is mainly useful for printing progress messages in your programs, calculating the progress percentage as `100 * ((i+1) / len(L))`.

  If you need to refer to an element of another sequence at the same index, for this use case there are much better design patterns: use a class to group related items (if the grouping is relevant across the whole program); or `zip()` the containers for the loop (for temporary throwaway grouping).

- *Zipping* (think of a zipper, the physical object). Example:

  ```
  # given lists a,b,...
  zip(a,b)     # (a[0],b[0]), (a[1],b[1]), ...

  zip(a,b,c,d) # any number of inputs is fine

  zip(*L)      # where L is a list of lists.
               # We get (L[0][0], L[1][0], ...), (L[0][1], L[1][1], ...), ...
  ```

  Zipping in loops:

  ```
  for x,y in zip(a,b):  # (a[0],b[0]), (a[1],b[1]), ...
      do_something(x,y)
  ```

  Pairs of adjacent elements (bigrams), pythonic way:

  ```
  s = "Python"  # a string is just an iterable consisting of characters
  for a,b in zip(s[:-1], s[1:]):
      print("%s%s" % (a,b))
  ```

  This prints `Py`, `yt`, `th`, `ho`, `on`. We may even simplify this to

  ```
  s = "Python"
  for a,b in zip(s, s[1:]):  # <-- only change: no [:-1]
      print("%s%s" % (a,b))
  ```

  because `zip()` terminates on the shortest input; see `help(zip)`.

  For how to generalize this to tuples of three or more adjacent elements, see:

  Scott Triglia: Elegant n-gram generation in Python (intermediate):
  http://locallyoptimal.com/blog/2013/01/20/elegant-n-gram-generation-in-python/

- There is no "`unzip()`", because `zip()` is its own inverse. Example:

  ```
  x   = zip(a,b)  # output: (a[0],b[0]), (a[1],b[1]), ..., (a[-1],b[-1])
  c,d = zip(*x)   # zip(*x) = zip( (a[0],b[0]), (a[1],b[1]), ..., (a[-1],b[-1]) )
                  #         = ( (a[0], a[1], ..., a[-1]), (b[0], b[1], ..., b[-1]) )
  a == c  # True
  b == d  # True
  ```

  https://stackoverflow.com/questions/13635032/what-is-the-inverse-function-of-zip-in-python

- Pretty-printer. From its help page: *"Very simple, but useful, especially in debugging data structures."*

```
from pprint import pprint  # this imports function "pprint()" from module "pprint"
pprint(myobj)
```

- The symbolic math package SymPy also comes with a pretty-printer, for symbolic math expressions:

```
import sympy

# "sympify" = convert given string into symbolic math expression
s = sympy.sympify("phi0 + 2*pi")

sympy.pprint(s)
```

  This prints $\varphi_0 + 2 \cdot \pi$.

- Some notes on the help system:
  - `help(import)` is a syntax error (since `import` is a *keyword*, not an object), but `help("import")` will show you its built-in documentation. Similarly for other language keywords (e.g. `with`, `try`).
  - See also `help("topics")` for a list of general topics. These topics are viewed as, e.g. `help("ASSERTION")`.
  - For some things it is quicker to search the internet, but sometimes Python's help system (including docstrings of library functions) may have crucial details not found elsewhere. (This is because Python programmers — just like MATLAB programmers — actively use the help system.)

- `Ellipsis` is not very widely known. It has been around at least since Python 2.4 (2004), but essentially it was implemented for the benefit of NumPy's slicing; after all this time, no other major libraries use it.

  In Python 3, `...` is valid anywhere, also outside slicing syntax. Hence, some programmers use it to denote "add implementation here" TODOs. Others use the keyword `pass`, which is Python for "do nothing".

  https://stackoverflow.com/a/6189281
  https://docs.python.org/3/library/constants.html#Ellipsis

  Historical: https://docs.python.org/2.4/lib/bltin-ellipsis-object.html

- *Shallow copy* vs. *deep copy*, i.e. copy references only, or make copies of referred objects, too? This distinction is important in any system that uses references. In Python, see the `copy` module that can do both:

  https://docs.python.org/3/library/copy.html

  Note that NumPy arrays have a `copy()` method. Python lists can be shallow-copied simply by using the copy constructor: `L2 = list(L1)`. Similarly, to copy dicts and sets, `D2 = dict(D1)` and `S2 = set(S1)`.

  In general, copying is a deep issue — in programming there exist objects, such as handles to open files, which cannot be meaningfully copied. For the sake of argument: forget implementation, and consider the following question theoretically: if you copied an object instance that happened to be in the middle of writing to a file, or sending data over the network, what would you expect the copy to do?

- [Somewhat advanced] *Decorators* wrap functions to modify their behavior. Many Python libraries provide decorators (e.g. `@profile`, `@numba.jit`). In the right hands, creating custom decorators is a very powerful technique. As Wikipedia points out, the established name "decorator" is a misnomer, as it easily gets confused with the design pattern of the same name; this Python feature would be properly called *advice*.

  https://realpython.com/blog/python/primer-on-python-decorators/
  https://www.thecodeship.com/patterns/guide-to-python-function-decorators/
  https://en.wikipedia.org/wiki/Python_syntax_and_semantics#Decorators
  https://en.wikipedia.org/wiki/Advice_(programming)

- For some more tips and tricks:

  http://python-3-for-scientists.readthedocs.io/en/latest/python3_user_features.html
  https://stackoverflow.com/questions/101268/hidden-features-of-python

  but take the second link with a grain of salt; some of the advice is firmly tongue-in-cheek.

4.10 A scientist's checklist for Python 2 to Python 3 conversion

If you find yourself in the unenviable — and increasingly more rare — position that the software (or an online code example) you need was written years ago and only works in Python 2, don't worry.

If you only need to convert a short piece of code, this list should cover most of it:

- `print ...` → `print(...)`
- `xrange(...)` → `range(...)`
  - In Python 3, `range()` creates a `range` object, which behaves much like any generator, except that you can iterate over it multiple times (each time starting from the beginning).
- `range(...)` → `tuple(range(...))`
  - This first creates the `range` object, then feeds it to the `tuple` constructor, which iterates over the `range`, extracts the output, and stores it into a new `tuple`.
  - Only `tuple()` a range if you have to. In most cases, this just wastes memory and makes the code slower; this is the practical reason the `range` object was introduced in Python 3.
- `filter( condition, L )` → list comprehension: `[x for x in L if condition(x)]`
- `map( func, L )` → list comprehension: `[func(x) for x in L]`
- `reduce(...)` → `functools.reduce(...)` (and import functools!)
- Integer division: `k / 2` → `k // 2`
- `import cPickle as pickle` → `import pickle`
- Some exceptions are now more descriptive; adjust any `except` statements as needed.
  - E.g. both `open()` and `os.remove()` now raise `FileNotFoundError`.
- If you want the code to remain compatible also with Python 2.7, add this magic import to the start:

  ```
  from __future__ import division, print_function, absolute_import
  ```

  For pure Python 3 code, this is not needed. For cross-compatible code, you may also want to look into → six.

  Since cross-compatibility prevents from using features that only exist in Python 3, and Python 2 has already been almost phased out, my personal recommendation is to just go with Python 3.

Conversion tools

- http://python-future.org/automatic_conversion.html
- https://python-modernize.readthedocs.io/en/latest/
- https://pypi.python.org/pypi/tox

Documentation

- Howto: https://docs.python.org/3/howto/pyporting.html
- Online book: http://python3porting.com/

4.11 Python 3 source code template

It may be convenient to use a template such as the following, when creating a new program or module:

`template.py`

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""Hello world 2.0"""

def main():
    """Print a message."""
    print("Hello, world!")

if __name__ == '__main__':
    main()
```

There are several things to note here:

- The *hashbang* (`#!`), a.k.a. *shebang*, is optional, but if included, it must be the first thing in the source file.
  - This tells Linux and OS X (and other *nix systems) which program to run this code with, if the source file itself is marked as executable.
  - Change `python` to `python3` if necessary.
  - The term "hashbang" has nothing to do with hashing or hash tables; it simply comes from the colloquial names of the characters "#" (hash) and "!" (bang) used as the *magic*.
    - https://en.wikipedia.org/wiki/Magic_number_(programming)#Magic_numbers_in_files
    - https://en.wikipedia.org/wiki/Shebang_(Unix)
- The *magic comment* (`# -*- ... -*-`) specifies, for Python, the character encoding used in the source file.
  - Optional, but if included, must be on the first or the second line. (Hence, second line, if a hashbang is also included.) If omitted, Python will treat the file as `utf-8`.
  - Mainly useful for us Finns to write printed messages and docstrings in Finnish, while ensuring that scandinavic characters show up correctly when the program runs or a user calls `help()`.
  - However, it is legal (if not often seen in practice) to use Unicode identifiers: "$\alpha = 42$" is valid Python 3, creating a name called α, and binding it to a new `int` object instance having the value 42.
    - This is limited to, roughly speaking, "alphabetical" characters. Alas, $\int$ is not a valid function name!
      https://stackoverflow.com/questions/17043894/what-unicode-symbols-are-accepted-in-python3-variable-names
    - Nevertheless, with a LaTeX input method for inputting Unicode math symbols:
      https://github.com/clarkgrubb/latex-input
      this *could* be used to write mathematical Python code in a rather natural way.
- The keyword `def` defines a function. In this example, the function takes no arguments.
- When a function returns without encountering an explicit `return` statement, the default return value is the special singleton object `None`.
  - Keep in mind that "the value of the last expression" is *not* automatically stored in Python; i.e., no "ans".
  - However, in an interactive session, many frontends such as *IPython* capture and store it into a variable called "_" (a single underscore), as well as print it.
    - But keep in mind that a *statement* produces no value.
    - Especially, in Python assignment is a statement, so e.g. `a = b*c` will only update `a`; it will not evaluate to a value (and hence, IPython will not update `_`). By itself, `b*c` obviously does evaluate to a value (making IPython update `_`).
  - By convention, the name `_` is also often used for dummying out unwanted components in a returned tuple, as a shorter alternative to the explicit standard name `dummy`.

- The magic part at the end is a standard "conditional main" pattern.
  - It checks whether the name of the module being currently executed is "`__main__`"; if it is, then this file knows it is being run as the top-level main program.
  - This allows `importing` this file from another Python module, without causing the `main()` defined here to run.
  - The name of the actual function (here `main()`) is arbitrary, but "`main()`" is perhaps the most self-documenting choice.
  - This pattern is often used for testing, or in the case of small libraries, for providing usage examples or a small demo inside the module itself.
  - I.e. this file might not be the actual main program, but a utility module whose main purpose is to provide reusable functions for other modules; it just includes also a `main()`, so it can be run directly for unit testing or demo purposes.
  - This is actually included in the language documentation for the magic module `__main__`:

    https://docs.python.org/3/library/__main__.html
  - For more, see:

    https://stackoverflow.com/questions/419163/what-does-if-name-main-do
- If you want to add a command-line interface to your program (so that you can call it easily from outside Python while passing it different argument values):
  - Look into the `argparse` module in the standard library, and look for usage examples on the internet. This topic will be skipped here.
  - The old `optparse` module did the same thing, but has now been replaced by `argparse`.
  - For backwards compatibility reasons, obsolete modules are usually not removed from the standard library, but only *deprecated*; they are no longer developed further, and their documentation is updated to mention what has replaced them.
    * Any new code is expected to use the new module; the obsolete one exists solely for backwards compatibility with existing code that already uses it.
    * Depending on the software project, deprecated features may be actually removed in a later version, after a transition period has elapsed.
    * https://en.wikipedia.org/wiki/Deprecation

5 Advanced topics

This section contains some additional information on selected topics. The final section on the basics of software engineering is recommended. All others are optional and can be read in any order.

5.1 Why object-oriented programming?

Object-oriented programming (OOP) is hardly a new idea. It was invented at the turn of the 1960s, and early implementations were seen in LISP, ALGOL, and Simula 67. OOP has since spread to many programming languages, including C++, Java, and Python. A brief history at Wikipedia:

https://en.wikipedia.org/wiki/Object-oriented_programming#History

At the time of this writing, OOP has been for a very long time well-established in the programming community, but is still underrated in scientific computing — except in larger software projects, where the developers come from a programming background.

So, as a scientist, why use OOP?

The answer is that OOP solves a practical problem: in programming, there are many situations in which it makes a lot of sense for data and algorithms to go together. Specifically in the context of scientific computing, global variables are nasty, and applying some very basic OOP is an easy way to avoid them.

Consider a solver that has a global variable `alpha`, or some internal state that must be preserved between function calls. What if we want to:

- Run two instances simultaneously, in the same process, with different values of `alpha` (e.g. for a parametric study)?
- In this solver, re-use parts from another solver, that also defines a global variable `alpha` for some different purpose?

Obviously, only one copy of a global variable (of the same name) can exist at one time: *global variables pollute the global namespace*.

The OOP solution is to, instead, hold the state and parameters of the computation in *instance attributes* (instance members) of an object instance. Obviously, multiple independent instances of the same object type can exist at the same time.

OOP also conveniently packages the related data and functions (now methods) into the same place (the class), keeping them together also in the source code. This is known as *encapsulation* (there is also another distinct meaning for the word, but that is not important here).

https://en.wikipedia.org/wiki/Encapsulation_(computer_programming)

Of course, like any other programming paradigm, OOP is no silver bullet. In some situations, a procedural approach makes more sense. For example, linear equation system solvers are fine as procedural functions, to which the uses passes in the relevant data as arguments: `solve(A,b)` looks clearer (for historical/cultural reasons) than `A.solve(b)`. But from the viewpoint of the developer of the library, keep in mind the above note about internal state...

To give a minimal taste of very basic OOP:

```
class Solver:
    def __init__(self, alpha):
        self.alpha = alpha
        self.x = None  # no solution computed yet

    def solve(self):
        # do something here
        self.x = 42 * self.alpha
        return self.x
```

```
def main():
    results = []
    for a in [1., 10., 100.]:
        s = Solver(alpha=a)
        results.append(s.solve())

    print(results)

main()
```

This example does not actually use multiple `Solver` instances simultaneously, but perhaps it gets the main point across (about keeping the data and algorithms together).

As a final example to give a taste of slightly more advanced OOP, intermediate and advanced users may find useful the *mixin pattern*, which uses *multiple inheritance* to inject functionality, while still allowing the class to inherit from something else (i.e. semantically speaking, to belong to a different part of the class hierarchy).

https://en.wikipedia.org/wiki/Mixin

Paraphrasing from one of my own projects:

```
# Base class.
#
class Geometry:
    pass  # (implementation goes here)

# Gmsh file loader.
#
class GmshGeometry(Geometry):  # <-- inherits from Geometry
    pass  # (use the GmshTranslator library to load stuff)

# Another, orthogonal class hierarchy was needed here for technical reasons,
# to perform problem type specific initialization of the geometry.
#
# As a consequence, this branch knows nothing about loading files.
#
class GenericProblemTypeSpecificGeometry(Geometry):
    pass  # (implementation goes here)

# A specific problem type.
#
class EulerFlowGeometry(GenericProblemTypeSpecificGeometry):
    pass  # (do init specific to this problem type in this class)

# Finally, now for the trick:
#
class EulerFlowGmshGeometry(GmshGeometry, EulerFlowGeometry):  # <-- multiple inheritance
    pass  # (no implementation needed here)
```

In this particular project, all the necessary functionality was already implemented by the ancestors in the two branches of the inheritance graph, i.e. no additional code was needed for the final class. This is generally not the case; usually mixins are used to bring in specific pieces of functionality.

And yes, the inheritance graph has a diamond shape, as both branches start at `Geometry`. For cases like this — and especially for more complex ones — you will want to know that for determining the *method resolution order* (MRO), i.e. which ancestors are tried first when looking up methods of the derived class, Python applies what is known as the *C3 linearization algorithm* to the inheritance graph.

https://en.wikipedia.org/wiki/C3_linearization

Unfortunately, we must stop here; how to OOP is beyond the scope of these lecture notes.

5.2 Parallel computing

Writing concurrent software is difficult; correctly, even more so. But in scientific computing, it is often worth it.

Early concurrency was based on *shared memory*, and *locking* data structures so that at any given time, only one thread could write to any given data structure, thus ensuring consistency. An important term that is still in use: functions that be safely called simultaneously from multiple threads are called *reentrant*. (Especially, documentation often warns if some particular function is *not* reentrant.)

The locking approach was prone to *deadlock*: thread `A` could start waiting to obtain access to a resource owned by thread `B`, while `B` was already waiting to obtain access to a resource owned by `A`. Once any bugs in a program were sorted out, shared memory concurrency worked up to a few simultaneous threads. But as the number of threads grew, the requirement of exclusive access became a bottleneck: at any given time, most of the threads would be waiting for a lock. Large-scale parallelism required new solutions.

The current paradigm in concurrency, especially in scientific computing, is *message passing*. In short, independent agents communicate via messages; all memory is private. The programmer must coordinate the agents so that when one is sending, there will eventually be another listening; otherwise the software will typically deadlock. Still, this is usually simpler to get right than classical shared memory concurrency.

Message passing also scales well (also to supercomputers with over 10^4 cores), until the message size becomes so large that the time taken for data transmission becomes a bottleneck. An important thing to keep in mind when working with the message passing paradigm is that any time spent communicating is time *not* spent on actually computing the solution.

A rising trend in concurrency is *transactional memory* (both in hardware and in software). This is a return to the idea of shared memory, while making deadlocks impossible, and aiming toward a simpler abstraction for the programmer using the paradigm. Transactional memory must be directly supported by the programming language; Python (especially the standard CPython interpreter) currently does not have it.

See:

https://en.wikipedia.org/wiki/Lock_(computer_science)
https://en.wikipedia.org/wiki/Race_condition (a common type of bug in concurrent programming)
https://en.wikipedia.org/wiki/Reentrancy_(computing)
https://en.wikipedia.org/wiki/Message_passing
https://en.wikipedia.org/wiki/Transactional_memory
https://en.wikipedia.org/wiki/Software_transactional_memory

On concurrency, Nick Coghlan writes:

One of the key issues with threading models built on shared non-transactional memory is that they are a broken approach to general purpose concurrency. Armin Rigo has explained that far more eloquently than I can in the introduction to his Software Transactional Memory work for PyPy, but the general idea is that threading is to concurrency as the Python 2 Unicode model is to text handling — it works great a lot of the time, but if you make a mistake (which is inevitable in any non-trivial program) the consequences are unpredictable (and often catastrophic from an application stability point of view), and the resulting situations are frequently a nightmare to debug.

http://python-notes.curiousefficiency.org/en/latest/python3/multicore_python.html#multicore-python

5.2.1 Parallel computing in Python

Regardless of the paradigm used, concerning concurrency inside a single *process* (in the operating system sense of the word), the standard Python implementation, CPython, has an important limitation: the Global Interpreter Lock (GIL). Only one Python thread can be actively executing at a time. This is a design choice that simplifies the implementation of the interpreter.

The GIL is automatically released while waiting for I/O (including user input). A large portion of time spent waiting is typical for server workloads, for which Python has been traditionally used. However, this does not help in scientific computing, where a 100% CPU load is typical.

So, how to work around the GIL?

General tips in case of NumPy and SciPy: https://scipy.github.io/old-wiki/pages/ParallelProgramming

Solution 1: process-based parallelism

- Nowadays often based on the *message passing* paradigm.
  - **Recommended**. Relatively easy to learn, also scalable to clusters and supercomputers.
  - See:
    - https://stackoverflow.com/questions/7140544/message-passing-vs-locking
    - http://www.cs.cornell.edu/courses/cs312/2008sp/lectures/lec26.html
    - http://wiki.c2.com/?MessagePassingConcurrency
- MPI → mpi4py
  - The Message Passing Interface (MPI) is the de facto standard message-passing framework in scientific computing; clusters and supercomputers usually have it. Can also be run locally on a single machine. Implementations of the MPI specification include Open MPI and MPICH.
  - Each process (MPI rank) basically runs the same code independently, until told otherwise:
    - Explicit conditional execution; can switch on MPI rank.
  - Explicit send/receive of messages between processes.
    - Unicast (point-to-point), broadcast, scatter (divide data between processes), gather (results into root process), allgather (results to all), barrier (wait until all processes reach this point in the code)
  - http://pythonhosted.org/mpi4py/
    - Can send both almost arbitrary Python objects, and NumPy arrays. NumPy arrays are transmitted at native speed using buffers.
    - Almost arbitrary: the object must be pickleable (*serializable*). See `pickle` in the standard library.
- ZeroMQ
  - Message passing with both fine-grained control and advanced abstractions, meant for an arbitrary set of communicating, otherwise fully independent processes.
  - http://zeromq.org/
- `multiprocessing`
  - *[In the standard library,] multiprocessing is a package that supports spawning processes using an API similar to the threading module.*
  - https://docs.python.org/3/library/multiprocessing.html
- `concurrent.futures.ProcessPoolExecutor` (introduced in Python 3.2)
  - Easy-to-use high-level abstraction that builds on `multiprocessing`.
  - https://docs.python.org/3/library/concurrent.futures.html

Solution 2: native extension module

- *Native*: compiled, machine-language.
- *Extension module*: an external library that has a Python interface, so that Python modules can `import` it.
- Program your low-level parts in Cython, and inside native code, release the GIL when a long-running computation starts.
  - Many libraries, such as NumPy, do exactly this internally.
  - Getting this right in your own Cython code may require some wizardry. For example, "`with nogil:`" blocks must not be nested — anywhere in the dynamic scope the program happens to be running in! (Design carefully.)
- Not scalable beyond a single machine; but in that environment, definitely helps.

5.2.2 Amdahl's law

Regardless of your method of parallelization, keep in mind **Amdahl's law**, which is an upper bound for scalability:

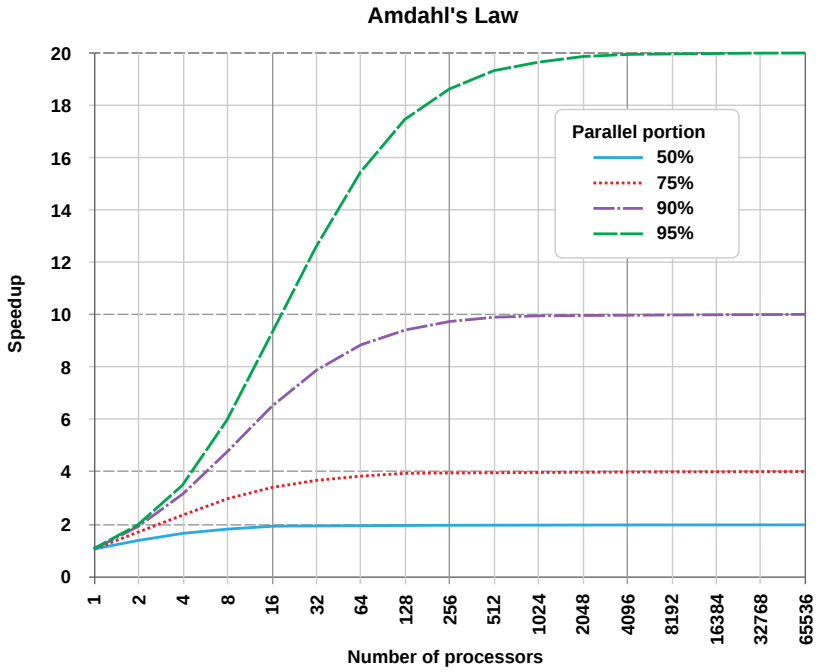
$$S(s) = \frac{1}{(1-p) + p/s} \,.$$

https://en.wikipedia.org/wiki/Amdahl%27s_law

Roughly speaking, S is the theoretical speedup factor of the whole program (as more processors are thrown at it), p is the proportion of time spent in parallelized parts of the program, and s is the speedup factor of the parallelized part. In the unrealistic, theoretical ideal case with perfect scalability, s is the number of processors. We have

$$\lim_{s \to \infty} S(s) = \frac{1}{1-p} \,,$$

which illustrates that the most important factor is $1 - p$, i.e. the portion of serial code in the whole program. Wikipedia provides this helpful graph (courtesy of user Daniels220, CC-BY-SA-3.0):



Especially note that for a serial portion of only 5%, the maximum speedup that can be obtained for the execution of the whole program *even with an infinite number of processors* is just $20\times$.

Further reading (beginner to advanced):

Nick Coghlan: Efficiently Exploiting Multiple Cores with Python
http://python-notes.curiousefficiency.org/en/latest/python3/multicore_python.html

David Beazley (USENIX 2009): An Introduction to Python Concurrency
http://www.slideshare.net/dabeaz/an-introduction-to-python-concurrency

Bartosz Milewski: Beyond Locks and Messages: The Future of Concurrent Programming:
https://bartoszmilewski.com/2010/08/02/beyond-locks-and-messages-the-future-of-concurrent-programming/

5.3 Numexpr

Numexpr is an easy-to-use, fast evaluator for elementwise expressions operating on NumPy arrays. It essentially works by reducing the number of temporary copies, when compared to evaluating the same expression in NumPy. Hence, expressions evaluated using Numexpr may need less memory. An orthogonal advantage is that Numexpr supports multithreading right out of the box, speeding up evaluation. Typical speed-ups are 2–4×.

It is important to keep in mind that in computers, memory latency is high, and memory bandwidth is much more (≈ 10–15×) expensive than math, so this is especially important for large arrays. (That is, the processor may have time to complete 10 to 15 math instructions in the time it takes to perform one memory access.)

It is possible to avoid temporaries in pure NumPy by using the `out=...` argument, but this typically makes the code much less readable. On the other hand, Numexpr code must be wrapped into strings. This may also decrease readability, as editors do not syntax highlight code inside strings.

As of late 2017, in recent versions of NumPy, there have been some improvements to automatic identification of temporaries, but Numexpr retains its advantage of multithreading.

Hence, using Numexpr is an easy way to reduce memory use and increase speed.

Simple example:

```
import numpy as np
import numexpr

b = np.random.random(5)
c = np.random.random(5)
d = np.random.random(5)

# Names are captured automatically from the enclosing scope.
# This is also automatically multithreaded.
#
numexpr.evaluate("a = b + c*d")
```

See:

- Official site:

  https://github.com/pydata/numexpr

- User manual:

  http://numexpr.readthedocs.io/en/latest/

  - Note especially:

    The initial number of threads will be set to the number of cores detected in the system or 8, whichever is *lower*.

    http://numexpr.readthedocs.io/en/latest/user_guide.html#general-routines

  - To adjust, `numexpr.set_num_threads(nthreads)`.

- A slightly more detailed description of how Numexpr works:

  http://numexpr.readthedocs.io/en/latest/intro.html

5.4 Numba

Numba is essentially a JIT (just-in-time) compiler for a subset of Python, with some limited support also for traditional ahead-of-time (AOT) compilation. Numba uses LLVM to compile Python code to machine code, as well as exploits vector operations provided by the CPU when appopriate.

As compile targets, Numba supports single-core CPU, parallel CPU, and NVIDIA GPU (CUDA). A CUDA simulator is available for debugging CUDA code.

"Subset of Python" means, in practice, that Numba supports almost all of Python, but is missing some of the high-level features such as list/dict/set comprehensions, lambdas, and nested function definitions. Generators are only partially supported. But it can still do a lot.

The idea of Numba is to compile each function when it is called for the first time. This introduces a very high overhead for the first call, but subsequent calls will be very fast, as they use the compiled code. Note that the function is re-compiled, if the data types of the arguments change.

Numba is ideal for JITing small performance-critical snippets in its *nopython* mode. It can also lift low-level loops in its *object* mode.

In any case, it has automatic type inference, so there is no need to annotate the data types used (int64, float64, etc.). It is however possible to do so for the function arguments and the return value, if precise control over input and output data types is desired.

Basic usage is very easy. Silly example:

```
import numba   # <--- add this

@numba.jit     # <--- and this
def f(x,y):
    return x+y
```

Now you can use `f()` just as usual.

```
f(5, 3)            # int64
f(1.414, 3.14159)  # float64
```

And that's it.

- Documentation:

  http://numba.pydata.org/numba-doc/0.30.1/index.html

- Especially, lists of supported features:

  http://numba.pydata.org/numba-doc/0.30.1/reference/pysupported.html
  http://numba.pydata.org/numba-doc/0.30.1/reference/numpysupported.html

  Check your code against these if you get a nonsensical-looking error message from Numba's JIT.

- The only practical concerns are:
  - Installation; at least at some point in history, Numba has been rather particular about the versions of LLVM it supports.
  - Distribution of projects using Numba to end users, since compilation occurs at runtime (and hence Numba and LLVM are needed as runtime dependencies).

5.4.1 NumPy ufuncs and gufuncs in Numba

Numba allows implementing custom NumPy *ufuncs* (universal functions) and *gufuncs* (generalized universal functions) in Python, without manually writing a C extension (which is the usual way to extend NumPy itself, in the rare case where that is needed).

As the user manual points out:

You might ask yourself, "why would I go through this instead of compiling a simple iteration loop using the @jit decorator?". The answer is that NumPy ufuncs automatically get other features such as reduction, accumulation or broadcasting.

http://numba.pydata.org/numba-doc/0.30.1/user/vectorize.html

- *ufunc*: scalar in, scalar out, applied automatically elementwise.
- *gufunc*: arrays in, arrays out, has a specific function signature (NumPy term) / input and output layout (Numba term).
  - In NumPy, a *function signature* is what Numba calls the *layout string*. This is the number of elements on each axis of an array, e.g. for a matrix product, we have

    (m,n),(n,p)->(m,p)

    This is pretty much Einstein notation (compare `np.einsum`).
  - In Numba on the other hand, a *function signature* indicates data types, including the number of array dimensions. For example,

    `void(float64[:,:], float64[:,:], float64[:,:])`

    is a signature: the function takes in three 2D float64 arrays, and returns `void` i.e. nothing (realistic if it operates on the data in-place).
- See:

  http://numba.pydata.org/numba-doc/0.30.1/reference/jit-compilation.html?#numba.vectorize

  http://numba.pydata.org/numba-doc/0.30.1/reference/jit-compilation.html?#numba.guvectorize

  https://docs.scipy.org/doc/numpy/reference/c-api.generalized-ufuncs.html#details-of-signature
- Importantly: ufuncs and gufuncs *can operate on NumPy arrays*
- This also leads to appropriate separation of concerns / modularity
  - The calling code becomes simple, short, and to the point. The loop is an implementation detail.
  - The function becomes generic, helping its reusability in future projects.
- https://en.wikipedia.org/wiki/Separation_of_concerns
  - Cf. https://en.wikipedia.org/wiki/Single_responsibility_principle (for modules, classes)

5.5 Cython

To properly cover Cython would require a separate course, so we will give only an overview, and some practical pointers.

Cython is a compiled language that produces Python modules — it is, essentially, a Python compiler. It is a classical ahead-of-time (AOT) compiler, like compilers for Fortran and C. However, the real point of Cython is that it facilitates manual optimization of implementation details (especially compute-heavy inherently serial algorithms), and allows connecting to existing C libraries; not that it also happens to be a static compiler for Python.

Stefan Behnel: A static Python compiler? What's the point?
http://blog.behnel.de/posts/indexp241.html

Thus, in a scientific computing context, we may say that the main use case of Cython is that it is the Python equivalent of MATLAB's MEX. For an example of using Cython in numerics to connect to existing C libraries, see scikit-sparse, which provides Python bindings to CHOLMOD via Cython.

From a language point of view, Cython is essentially *Python with C datatypes*. It introduces static type checking, and type annotations. Cython also supports C++ to some extent, but that is beyond the scope here.

Importantly, the static types are optional; any valid Python code is also valid Cython code. The way this works is that, when generating the C code, Cython translates any dynamically typed Python code into the equivalent calls to Python's C API (application programming interface). This typically leads to a speedup of approximately 30%, as the Python interpreter is bypassed.

But much larger speedups, in the order of $10^2\times$, can be obtained in serial algorithms that only crunch numbers without accessing Python objects. Any Cython code that has the static types, and does not access Python objects (accessing data in NumPy arrays is allowed), is translated directly into C code — including any `for` loops. Single-core performance of this is similar to Numba.

5.5.1 Good to know

Cython is being actively developed, and new versions are released regularly. At the time of this writing (September 2017), the current version is Cython 0.27.

The Python installer `pip` knows how to install Cython, as a special case — that *is* special enough to break the rules — although Cython is not (only) a Python package. This is probably because many libraries, especially in the scientific Python software stack, depend on Cython to provide high performance. Practicality beats purity, again.

Incidentally, `pip install cython --upgrade --user` is the best practical way to stay up to date.

Cython source files have the file extension .pyx, after Pyrex, an earlier project Cython is based on.

Cython provides parallel for loops via OpenMP, in code that does not access Python objects; just like the "parallel" target of Numba.

Cython code is first compiled to C (using Cython itself), which is then compiled to machine code (typically using GCC, the GNU C Compiler) and placed into a dynamically linked library (.dll, .so). This library can then be imported into Python just like any Python module.

In projects based on Cython, binaries can be easily distributed. Once the code is compiled, the produced binaries are independent of Cython.

Also the generated C source code does not strictly require Cython. However, for purposes of archiving the project source code, it is better to treat the C file strictly as a generated file (does not need to be stored in version control!), as it is not even meant to be human-editable.

The fact that Cython's *intermediate representation* happens to be text, and in a programming language also humans use, is just an implementation detail.

https://en.wikipedia.org/wiki/Intermediate_representation

5.5.2 Tuning the C compilation step

Binaries are compiled for a generic architecture such as `x86_64`. It is possible to use processor family specific options such as `-msse2` in the C compilation step. For example, I have used the following GCC options for my modules involving math functions:

```
-march=native -O2 -msse -msse2 -mfma -mfpmath=sse
```

In scientific computing, **do not use** `-ffast-math`, unless you know exactly what you are doing; because if you do, the compiler is then allowed to make approximations that, in many non-scientific contexts, do not usually affect the result significantly.

Here `-mfma` is relatively new; it allows the compiler to use the processor's *fused-multiply-add* (FMA) instruction, if it has one. This is an efficient daxpy (double precision $ax + y$) which rounds only once, so it is usually good for both speed and accuracy.

https://en.wikipedia.org/wiki/Multiply%E2%80%93accumulate_operation

5.5.3 Viewing the generated C source

Although the generated C source is not meant to be human-*editable*, it is certainly human-*readable*. For debugging and performance-tuning purposes, it is sometimes useful to examine the generated code, as it allows you to see exactly what Cython has done. To make the most use of it, search **the generated C source file** for (parts of) lines of your actual Cython program — it **includes your original Cython code in the comments**.

You may be interested also in the command

```
cython -a myfile.pyx
```

The `-a` stands for "annotate". This takes in your Cython source `myfile.pyx`, and produces `myfile.html`, which can be opened in a web browser, and similarly contains both the Cython source and the generated C source, but in an interactive form.

In the .html file, Cython source lines can be clicked to show the corresponding generated code, and the shading of each line of Cython shows how much code it generated. This allows a quick visual inspection of things such as whether a loop has become an efficient C loop as intended — and if not, what has gone wrong.

For example, the generated C code shows which arrays are still accessed using the memoryview API, instead of C-style pointer arithmetic. This, in turn, tells you that those arrays are likely missing a static type declaration, or have been assumed (maybe implicitly!) to have a memory layout that does not allow calculating memory offsets of elements directly (e.g. `cython.view.general`). Or maybe a loop counter, or some other integer, is missing a static type declaration, and is being treated as a Python object.

5.5.4 cimport vs. import

Beside .pyx source files, there are also .pxd files, which are Cython's *header files*, somewhat analogous to .h files in the C language. However, .pxd files are only needed for declaring the C-level public API; they are only used to export functions (and constants) to other Cython modules.

To import functions from a .pxd, Cython uses the keyword `cimport`, to distinguish this from loading Python modules with the standard keyword `import`.

Or in other words: `cimport` brings in compile-time information, essentially for things visible at the Cython level — i.e. things that are not visible to regular Python modules. It essentially works like the `#include` directive of C. The regular `import` brings in Python modules, as usual.

The same module may be both `imported` and `cimported`, either `as` the same name or `as` different names; just keep in mind that these do different things. To reiterate:

- `cimport` loads the .pxd file (Cython header file)
- `import` loads a Python module

As an example, consider the following two ways to create a parallel `for` loop in Cython. This is equivalent to MATLAB's `parfor`. First, using `cimport`:

```
cimport cython.parallel

def f():
    cdef int j
    with nogil:  # <-- GIL is now released, and re-acquired when this block exits.
       # (no Python objects allowed in this block)
       # OpenMP parallel loop
       for j in cython.parallel.prange(20):
            do_stuff(j)
```

Alternatively, we may use a Python-level `import`:

```
import cython.parallel

def f():
    cdef int j
    # prange(..., nogil=True) instructs Cython to "with nogil" the loop internally.
    for j in cython.parallel.prange(20, nogil=True):
        # (no Python objects allowed in this loop)
        do_stuff(j)
```

On parallel computing with OpenMP in Cython, see:

http://cython.readthedocs.io/en/latest/src/userguide/parallelism.html

5.5.5 Cython-level compiler directives

For optimizing the perfomance of Cython code, it may be useful to turn off support for negative indices (wraparound) and memory access safety (bounds checking) — once you are sure that your code is bug-free — because then, any invalid indexing will lead to silent data corruption, or in the best case, a crash.

Also to improve performance of Cython code, it is useful to change the division operator `/` to use C semantics. This leads to integer division if both inputs are `int`, and also disables Python's division-by-zero checking.

Per function, by decorator:

```
cimport cython

@cython.wraparound(False)
@cython.boundscheck(False)
@cython.cdivision(True)
def f():
    pass  # do whatever here
```

Per file, as magic comments:

```
# Set Cython compiler directives. This section must appear before any code!
#
# cython: wraparound  = False
# cython: boundscheck = False
# cython: cdivision   = True
```

For a full list of available directives, and their explanations, see:

http://docs.cython.org/en/latest/src/reference/compilation.html

5.5.6 Interacting with NumPy arrays in Cython

NumPy arrays are a special case in that accessing data in them does not require Python object manipulation. The data is accessed through *typed memoryviews*. User manual (very comprehensive):

http://cython.readthedocs.io/en/latest/src/userguide/memoryviews.html

For the curious, on the Python side:

https://docs.python.org/3/library/stdtypes.html#memoryview

Previously, there was a buffer interface specific to NumPy arrays, which has now been replaced by the newer, and more general, typed memoryviews. The old interface is *deprecated*, i.e. is now obsolete, and may be removed in the future.

As of late 2017, any documentation that instructs you to `cimport numpy` and use **syntax like**

`np.ndarray[np.float64_t, ndim=2]`

is outdated, and refers to the deprecated API.

The cimport is no longer needed, and the **current syntax** looks like a hybrid between C, and Fortran for allocatable arrays:

`double[:,:]`

(There is still a lot of documentation and code examples using the old API.)

If you want to use some advanced features of memoryviews, you may need to:

`from cython cimport view`

Especially `view.generic` is useful to specify that you want your function to accept an arbitrary sliced axis. See the Cython manual section on memoryviews, linked above.

The new API has brought with it one further important technical detail. If you return a memoryview object from a Cython function, Python will see it as a memoryview object, not as a NumPy array — even if it actually points to an underlying NumPy array. This is due to the static type declaration: the type of the variable is memoryview.

To make your Python code see the NumPy array, `return np.asanyarray(v)` instead (where `v` is your memoryview object). A silly example:

```
import numpy as np

# The "step of 1" is special Cython syntax to specify the memory layout.
# See the Cython manual section on memoryviews.
#
# Here the array A is C-contiguous.
#
def matrix_vector_product(double[:,::1] A, double[::1] b):
    # LHS: Cython memoryview syntax; RHS: pure Python
    #
    # Note that memoryviews have "shape" and "ndim".
    #
    # shape can be subscripted: shape[0], shape[1], ..., shape[ndim-1].
    #
    # Note also that as long as we do not need to allocate memory
    # inside a "with nogil" block, we can essentially use NumPy
    # as an allocator for dynamically sized arrays. Hence, usually
    # there is no need to bother with the traditional malloc()/free().
    # As a bonus, NumPy arrays use Python's automatic memory management.
    #
    cdef double[:] x = np.zeros(b.shape, dtype=np.float64)
```

```
# Everything here has a static type, so this will be automatically
# compiled into an efficient C loop.
#
# Our A is C-contiguous, so we make the inner loop run over columns.
#
cdef int n = A.shape[0]
cdef int m = A.shape[1]
cdef int i, j
for i in range(n):
    for j in range(m):
        x[i] += A[i,j] * b[j]

# x is a memoryview; return a NumPy array.
#
# This will create a new np.array object, but without copying the data;
# the new np.array will point to the same memory as the original one.
#
return np.asanyarray(x)
```

5.5.7 Compiling Cython programs

In practice, the easiest way to do compile Cython programs is to control Cython via a `setuptools`-based `setup.py`.

We will skip the details here, and just note that on GitHub, there are `setuptools`-based `setup.py` templates for Cython projects:

https://github.com/thearn/simple-cython-example
https://github.com/Technologicat/setup-template-cython/

which can be quickly customized for your own project.

Maybe also useful:

- The Cython manual includes material on using the older `distutils` (which `setuptools` is based on and extends):

  http://docs.cython.org/en/latest/src/reference/compilation.html

  http://docs.cython.org/en/latest/src/userguide/source_files_and_compilation.html#distributing-cython-modules

- Search path for include files (.pxd):

  http://cython.readthedocs.io/en/latest/src/userguide/sharing_declarations.html

- http://stackoverflow.com/questions/4505747/how-should-i-structure-a-python-package-that-contains-cython-code
- https://github.com/cython/cython/wiki/PackageHierarchy
  - Slightly outdated, should use `include_path` option of `cythonize()` instead.

5.6 Virtual environments

We will not go into details; this is just something to be aware of. *Virtual environments* are an approach to software installation that allows to:

- Isolate the Python environment from the OS.
  - Hence, install anything to it while avoiding conflicts with system-level Python on *nix (Linux, OS X).
- Install a Python-based app and libraries, *and leave it be*.
  - Installing upgrades to libraries could, in theory, break the app. This may be a problem if you need those upgraded libraries for another app, or for your own programming work.
- Run several different versions of Python.
  - This is useful for library and application developers, as it helps testing against different configurations.
- virtualenv is a popular tool for this: https://pypi.python.org/pypi/virtualenv
- Python 3.3 and later have the `venv` module: https://docs.python.org/3/library/venv.html#module-venv
- virtualenv, pyenv, virtualenvwrapper, pipenv, ... For a list and descriptions, see:

  https://stackoverflow.com/a/41573588

5.7 Hash table: note on Python dictionaries and sets

Python dictionaries can, effectively, use (almost) arbitrary objects for indexing. How is this possible?

Short answer: *hash table*.

Long answer:

In information technology, a *hash* is a kind of checksum (like CRC32, MD5, SHA-1). Essentially, it is a *one-way function* that maps arbitrary-length data into a number, usually of much shorter length (in terms of required data storage) than the data.

Dictionary keys — and items in `sets` in Python — must be *hashable*. In programming languages, and in Python in particular, *hashable* means that the object has the ability to create a hash value (integer) from the data contained in it. Essentially, it implements a `__hash__()` method. Java programmers are likely familiar with this.

Object instances that compare equal (`==`) must have the same hash value.

Only immutable objects can be hashable, because Python assumes that the hash value for any given object instance remains constant.

A *hash table*, then, is a data structure that stores key-value pairs for fast ($O(1)$ time) lookup.

By the *pigeonhole principle* — or in plain English, because the space of hashes is (much) smaller than the space of data — there will be *hash collisions*, i.e. several different data will inevitably hash to the same value.

However, a well-designed hash function *distributes* well, so that collisions for sensible real-world data are rare. Nevertheless, to make hash tables robust, they must in practice include a mechanism to be able to handle hash collisions.

Incidentally, Linux uses hashing for storing user passwords — i.e. the actual passwords are never written anywhere. The user is authenticated by comparing the hash of the user-entered password to the stored one. This is very secure as long as the password is not easy to guess, as (given a good hash function) entering an incorrect password gives practically no information about the correct one.

For more, see:

https://docs.python.org/3/reference/datamodel.html#object.__hash__

https://stackoverflow.com/questions/2909106/whats-a-correct-and-good-way-to-implement-hash

https://en.wikipedia.org/wiki/Hash_table

https://wiki.python.org/moin/TimeComplexity

5.8 Loop counters, lambdas, and functional programming in Python

Consider this code:

```
funcs = [ lambda x: i*x for i in range(5) ]
```

What happens when we apply the created functions and print the result?

```
y = [ f(2) for f in funcs ]  # apply each func to x=2
print(y)
```

In this example, we get `[8, 8, 8, 8, 8]`. Why?

This is because the `i` appears inside the body of the `lambda` (which is a function definition). Technically speaking, what here gets saved to the list `funcs` is not a bare function, but a *closure*. In plain English, it contains information about the scope it had when it was created (including information about names that were visible at that time).

Python then captures the *name* `i` instead of its value. Or, in other words:

Python's closures are late binding. *This means that the values of variables used in closures are looked up at the time the inner function is called.*

http://docs.python-guide.org/en/latest/writing/gotchas/
https://en.wikipedia.org/wiki/Closure_(computer_programming)

Thus all the closures that were saved to the list captured the same `i`.

How to get around this? There are at least three ways to persuade Python to bind early, i.e. to pass in the value instead of the name.

The first solution is to exploit the fact that, as we already saw above, default arguments grab the value at function definition time:

```
funcs = [ lambda x,i=i: i*x for i in range(5) ]
```

With this modification, we get `[0, 2, 4, 6, 8]`, as expected.

In the lambda, the `i` on the LHS of the assignment (and in the body) is the formal argument name, while the one on the RHS of the assignment is the loop counter. Or in other words, we could as well write:

```
funcs = [ lambda x,j=i: j*x for i in range(5) ]
```

For clarity, we will below use `j` to indicate where it is independent of `i`.

If that solution feels unnecessarily hacky (it's a matter of aesthetics, but I think it does: what if someone passes in another value for `j`?), another way is to use the *factory pattern*.

A *function factory* — as the name suggests — is a function that creates functions. Roughly speaking, you call it with a value for `j`, and it returns a function (of `x`) that uses this value. Applying the idea to the present example:

```
def make_f(j):
    return lambda x: j*x

funcs = [ make_f(i) for i in range(5) ]
```

This works, because the loop calls the function factory, *passing in the current value* of `i`. Also, now there is no room for a bug, because the created functions have only one argument, namely `x`.

The code can be slightly shortened by using the lambda notation for the function factory. Now the outer `lambda`, when called, returns the inner `lambda`:

```
make_f = lambda j: lambda x: j*x   # same as lambda j: (lambda x: j*x)
funcs  = [ make_f(i) for i in range(5) ]
```

This can be further compacted onto one line:

```
funcs = [ (lambda j: lambda x: j*x)(i) for i in range(5) ]
```

which you may sometimes see in the wild. This one-liner creates an *anonymous* function factory, calls it, and — because it then falls out of scope having no name associated with it — discards it when the loop is done.

The third solution is to bind explicitly using `functools.partial` (partial application of function arguments). For example:

```
from functools import partial

def f(i,x):
    return i*x

funcs = [partial(f, i) for i in range(5)]
```

In this particular (rather silly) example, we may even use the fact that our `f()` is nothing but a multiplication, which is already in the standard library, which saves us the trouble of defining `f()`:

```
from functools import partial
from operator import mul  # mul(a,b) is the same as a*b

funcs = [partial(mul, i) for i in range(5)]
```

See:

https://docs.python.org/3/library/functools.html#functools.partial

https://docs.python.org/3/library/operator.html#mapping-operators-to-functions

https://en.wikipedia.org/wiki/Partial_application

5.9 Order-preserving uniqification (discarding duplicates) of a list of items

It is possible to use `set` to uniqify (hashable) items stored in an iterable. Let `L` be a list.

If the ordering of the items does not matter, this is trivial:

```
L = list(set(L))
```

But what to do if we want to preserve the ordering?

The obvious idea is to use a `set` to keep track of which values we have seen so far, and then just append each unique item into another `list` in a loop, like this:

```
tagged = set()
tmp = []
for x in L:
    if x not in tagged:
        tmp.append(x)
        tagged.add(x)
L = tmp
```

But that's a bit long. Using a list comprehension, this can be compacted to:

```
tagged = set()
def tag(x):  # set.add() returns None, so make an expression version of add.
    tagged.add(x)
    return x
L = [ tag(x) for x in L if x not in tagged ]
```

which should also run faster, since using a list comprehension instead of an explicit loop gives shorter bytecode (as we will see). But there is now the overhead of a function call to `tag()`, whereas the previous version performed the operations directly in the loop.

Playing code golf (and maybe pushing the limits of readability), we may also express the same idea as:

```
tagged = set()
L = [ tagged.add(x) or x for x in L if x not in tagged ]
```

which gets rid of the extra function call (and gets us a birdie).

How does this version work? The item part of the list comprehension must tag the value of `x` as seen as a side effect, but it must also evaluate to `x` as an expression, because the result of this expression is what gets appended to the list being built. However, `help(set)` tells us that `set.add()` returns `None`.

The "`or x`" fixes this problem. The truth value of `None` is `False`; so the `or` must evaluate also its second argument, which will be left standing as the value of the expression.

Following Python's strong typing, the mere presence of `or` does not convert the result to `bool`; instead, Python tests the truthiness of the object. As was already noted in the section on tips and tricks, for the exact rules on how objects behave in logical expressions in Python, see:

https://docs.python.org/3/library/stdtypes.html#truth-value-testing

5.10 Graphical user interfaces

The history of graphical user interfaces (GUIs) can be traced back to the CAD program Sketchpad in 1963. The mouse was introduced in the late 1960s. Some readers may have heard of the early GUI research at Xerox PARC in the 1960s and 1970s.

The main mode of mainstream human–computer interaction, for well over 20 years, have been graphical user interfaces in one form or another. Hence, just for general education, we will very briefly explain the two main technical ideas behind GUIs.

The first thing to know is that **GUIs are event-driven**. When an application starts, its `main()` function typically first initializes the GUI layout, either by a series of function calls that instantiate the GUI elements, or by loading the layout from a file.

It then calls the *event loop*, provided by the GUI toolkit. At this point the control flow of the main thread becomes non-linear (event-driven), dependent on the user's actions. For example, when the user clicks a button, the event loop calls the *event handler* (a function in the application code) that in that particular GUI is attached to that button.

Another typical kind of GUI event is a timer, which triggers after a set period of time has elapsed. Timers are used for both single-shot and periodic cases. Timer events may occur late, if the main thread was doing something else at the exact time the timer should have triggered.

Animated GUIs, perhaps due to popular influence of OS X, are becoming more common. This improves the user experience especially for onlookers, because they lack the added context of the "driver's seat". Typically, GUI animations work by instantiating an animation object, and asking the GUI toolkit to schedule it to run. This is somewhat automated in some GUI toolkits.

Important properties to note are that:

- While the app is running, control of the main thread returns to application code *only during event handlers*.
  - When the user quits the application, the event loop returns (terminates). The rest of `main()` performs any app-specific shutdown operations (unless an event handler for the quit event already did that).
- The GUI blocks the main thread from doing anything else. Conversely, if the main thread computes something (in an event handler), this blocks the GUI, making the application appear unresponsive to the OS until the computation finishes.
  - Therefore, any lengthy computation in a GUI application should be performed in a separate thread. When finished, the thread should *send an event* indicating that the computation is done. The main thread can then trigger on this event to display the results in the user interface.
- Operations interacting with GUI elements are typically safe to invoke only from the main thread.

The other thing to know is the **model–view–controller (MVC) pattern**. This is a modular design that prevents the application code from becoming spaghetti.

- The **model** is *data and operations on the data*. Essentially, your solver code goes here.
- The **view** contains the *GUI representation* that displays the data, and the user interface that allows the user to interact with it. Several views to the same data may exist simultaneously.
  - This is convenient to e.g. provide an overall view and a zoomed-in view at the same time, while making sure that both views correctly reflect any changes to the data.
- The **controller** *coordinates operations between the model and the view*. The controller responds to the user's actions on the view, sending commands to the model. Also conversely, the controller responds to changes in the model, sending commands to the view.

The MVC is an early design pattern. Nowadays more sophisticated variants are available; but perhaps this is enough for general education.

https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

https://en.wikipedia.org/wiki/Graphical_user_interface

5.10.1 GUI libraries

If you want to play around with writing simple GUI apps, you can use at least one of:

- PyQt [Windows, Linux, OS X, Android, iOS]
  - Qt: classical, widely used for the desktop. Started in 1995.
  - As an example, the Spyder IDE is based on PyQt.
  - For plotting in a Qt GUI, possible to use PyQtGraph. (Runs on PyQt5.)
  - Be sure to use the current PyQt5. The old PyQt4 is legacy; Qt4 is no longer supported.
  - PySide also used to provide Qt for Python, but PyQt seems more actively maintained.
  - https://riverbankcomputing.com/software/pyqt/intro
    https://github.com/pyqtgraph/pyqtgraph
    https://wiki.qt.io/PySide
    https://en.wikipedia.org/wiki/Qt_(software)
- PyGTK [Windows, Linux, OS X]
  - GTK+: classical, widely used for the desktop. Started in 1998.
  - http://www.pygtk.org/
    https://en.wikipedia.org/wiki/GTK%2B
- Kivy [Windows, Linux, OS X, Android, iOS]
  - Main target smartphones, but applications can run unmodified also on PCs. Started in 2011.
  - Multitouch support out of the box, great for mobile apps.
    - Matplotlib integration available, but only on PC. VisPy for mobile.
  - https://kivy.org/
    https://en.wikipedia.org/wiki/Kivy_(framework)
- Intimidating full list of Python GUI toolkits (unfortunately, partly out of date):
  https://wiki.python.org/moin/GuiProgramming

5.10.2 Kivy

In principle, being able to run the same code on PC and smartphones sounds promising. Let's quickly try Kivy.

On the project website https://kivy.org/ , there is this hello-world example:

```
from kivy.app import App
from kivy.uix.button import Button

class TestApp(App):
    def build(self):
        return Button(text='Hello World')

TestApp().run()
```

- `TestApp` is the class that represents the application. Its ancestor `App` defines a method `run()`, which will call `build()` and then enter the event loop.
- The method `build()` is implemented in the application code, and it defines the GUI layout. In this extremely simple example, we get just one button that covers the whole window, and does nothing (no event handler).
- Add-on for Matplotlib integration:
  - https://github.com/kivy-garden/garden.matplotlib
  - The documentation is embedded in `backend_kivy.py`.
  - To install, basically

    ```
    pip install kivy-garden --user
    garden install matplotlib
    ```

  - Usage examples provided in `examples/` subfolder at the GitHub page.
  - As of late 2017, minor issues:
    - The figure settings window is not implemented, but the toolbar button itself comes from Matplotlib and cannot be (easily) hidden.
    - The toolbar icons are dark and not very well visible in dark GUI themes (which are common, especially on mobile).
    - Maybe not a problem if the intention is to just show a non-interactive plot, without the toolbar.
- What if we wanted to get this onto an Android smartphone?
  - Kivy has a packaging tool called `buildozer`, which integrates your application code, a Python interpreter and any necessary libraries into an Android application package (.apk).

    https://en.wikipedia.org/wiki/Android_application_package
    - "Hello world" becomes 30 MB... but at least you can code it in Python!
  - How to install `buildozer`:

    http://buildozer.readthedocs.io/en/latest/installation.html
  - How to write specifications for your app metadata:

    http://buildozer.readthedocs.io/en/latest/specifications.html
  - Binary extension modules require special care; `buildozer` needs to know details on how to handle each specific module.

  - Cython extension modules are possible, but require writing custom *recipes* (packaging specifications). Existing recipes:

    https://github.com/kivy/python-for-android/tree/master/pythonforandroid/recipes

    Instructions on adding new recipes:

    http://buildozer.readthedocs.io/en/latest/contribute.html

    The documentation is mainly targeted at adding support for open-source libraries not yet supported, but is also applicable for your own custom extension modules.

- Numerical library status on Android, as of late 2017:

  - NumPy is supported, but an old version (1.9.2; current is 1.12.1), and only on Python 2. There was an initial attempt in 2016 to make it work on Python 3 with the Crystax NDK, but it didn't work immediately, and it seems there is no currently active effort to fix this.

    https://github.com/kivy/python-for-android/issues/882
    https://github.com/kivy/python-for-android/issues/1074
    https://www.crystax.net/android/ndk

  - SciPy is not supported, and might be difficult to add due to many Fortran dependencies. Some versions of gfortran exist for Android, but nothing has been integrated into `buildozer` yet.

    https://github.com/kivy/python-for-android/issues/874

  - VisPy is supported, but only on Python 2. It just might work simply by patching `__init__.py` to allow also `python3crystax` (see the above issue 882 on NumPy) — or it might not.

    https://github.com/kivy/python-for-android/tree/master/pythonforandroid/recipes/vispy

  - Matplotlib is not supported. On the other hand, maybe not needed in a mobile app; VisPy is for realtime visualization, whereas Matplotlib is for publication-quality printable graphics.

    https://github.com/kivy/python-for-android/issues/520
    https://stackoverflow.com/questions/30295886/matplotlib-compilation-recipe-for-python-for-android
    https://github.com/kivy/python-for-android/issues/1090

  - It's pretty safe to assume this won't ever play well with Numba, since LLVM is unlikely to be easily packageable for a smartphone. (Smartphone processors are more than capable; this is more a matter of practical software logistics.)

In conclusion, as of late 2017, as for making Python-based scientific computing apps for smartphones:

The edges are still a bit rough, and not everything is supported. This is maybe a good target for advanced developers, if the parts that exist already cover 99% of what you need. But for now, Python on Android is not a production-ready environment for arbitrary codes.

5.11 Bytecode disassembly

```
def f():
    return (x for x in range(10))

import numba
g = numba.jit(f)  # <--- crash!
```

```
AttributeError:  'DataFlowAnalysis' object has no attribute 'op_MAKE_FUNCTION'
```

Excuse me, *the what* does not have *what*? Considering it sounds like a VM opcode, let's have a look at `f()`, like this:

```
import dis  # <-- Python bytecode disassembler
dis.dis(f)
```

We have

```
  2           0 LOAD_CONST               1 (<code object <genexpr> [...] )
              3 MAKE_FUNCTION            0
              6 LOAD_GLOBAL              0 (range)
              9 LOAD_CONST               2 (10)
             12 CALL_FUNCTION            1    # <--- this calls range(10)
             15 GET_ITER                      # <--- and grabs the iterator to the range
             16 CALL_FUNCTION            1    # <--- this calls the generator,
                                              #      which will return an iterator
             19 RETURN_VALUE                  # <--- which is returned to the caller here
```

Legend: source code line number (in this example just one, namely "2"), bytecode offset, OPCODE NAME, raw `arg` number, human-readable representation of arg.

Usually `arg` is 2 bytes, unless `op=EXTENDED_ARG`. See the source code for dis.disassemble(); it's Python. (To display the source code in IPython, `dis.disassemble??`, with two question marks.)

The Python VM is a stack machine; `RETURN_VALUE` pops the topmost value off the stack.

What the error message tried to tell us is that the Python virtual machine opcode `MAKE_FUNCTION`, which this example needs, is not implemented by Numba. (Indeed, Numba documentation confirms that generator expressions are not supported.)

Let's examine what happens if we change `f()` to use a list comprehension instead. Updated example:

```
def f():
    return [x for x in range(10)]  # <--- changed just ( ) to [ ]

import numba
g = numba.jit(f)  # <--- still crash!
```

```
AttributeError:  'DataFlowAnalysis' object has no attribute 'op_LIST_APPEND'
```

Ok, still no go. Now the disassembly reads

```
  2           0 BUILD_LIST               0
              3 LOAD_GLOBAL              0 (range)
              6 LOAD_CONST               1 (10)
              9 CALL_FUNCTION            1
             12 GET_ITER
        >>   13 FOR_ITER                12 (to 28)
             16 STORE_FAST               0 (x)
             19 LOAD_FAST                0 (x)
             22 LIST_APPEND              2
             25 JUMP_ABSOLUTE           13
        >>   28 RETURN_VALUE
```

The arg for loop opcodes is usually the offset from the current position after the arg; e.g.

```
13 FOR_ITER 12 (to 28)
```

means *at offset 13, op FOR_ITER, target offset 12 bytes. This is to (13 start) + (1 op + 2 arg) + (12 offset) = 28.*

The >> markers, generated by the disassembler, also show visually the beginning and the end of the loop.

So, the opcode LIST_APPEND is not implemented by Numba. But wait, isn't that a rather critical feature to have?

Let's see what happens if we spell out the example explicitly. Surely, this is what the previous example must do internally?

```
def f():
    L = []
    for x in range(10):
        L.append(x)
    return L

import numba
g = numba.jit(f)  # <--- ok!

g()  # --> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9], as expected
```

Now it works, so something must have changed! Disassembling, we have

```
  2           0 BUILD_LIST               0
              3 STORE_FAST               0 (L)

  3           6 SETUP_LOOP              33 (to 42)
              9 LOAD_GLOBAL              0 (range)
             12 LOAD_CONST               1 (10)
             15 CALL_FUNCTION            1
             18 GET_ITER
        >>   19 FOR_ITER                19 (to 41)
             22 STORE_FAST               1 (x)

  4          25 LOAD_FAST                0 (L)
             28 LOAD_ATTR                1 (append)
             31 LOAD_FAST                1 (x)
             34 CALL_FUNCTION            1
             37 POP_TOP
             38 JUMP_ABSOLUTE           19
        >>   41 POP_BLOCK

  5     >>   42 LOAD_FAST                0 (L)
             45 RETURN_VALUE
```

which contains no opcode that is not supported by Numba.

The empty list is created using BUILD_LIST, and list.append() is invoked using CALL_FUNCTION!

Quoting Eli Bendersky:

As a reminder, LOAD_FAST and STORE_FAST are the opcodes Python uses to access names that are only used within a function. Since the Python compiler knows statically (at compile-time) how many such names exist in each function, they can be accessed with static array offsets as opposed to a hash table, which makes access significanly faster (hence the _FAST suffix).

http://eli.thegreenplace.net/2015/the-scope-of-index-variables-in-pythons-for-loops/

(The args to LOAD_FAST and STORE_FAST are, apparently, these static array offsets.)

As a concluding note, this analysis also tells us that the list comprehension or generator expression versions *will likely run faster*, since they result in shorter bytecode.

Let's test this. IPython:

```
def f1():
    return [x for x in range(10)]

def f2():
    L = []
    for x in range(10):
        L.append(x)
    return L

%timeit f1()
1000000 loops, best of 3: 522 ns per loop

%timeit f2()
The slowest run took 5.86 times longer than the fastest. This could mean that an intermediate
result is being cached.
1000000 loops, best of 3: 854 ns per loop
```

Comparing the measured average run time and program bytecode length ratios,

$$\frac{t(f_1)}{t(f_2)} = \frac{522}{854} \approx 0.611 \,, \quad \frac{\ell(f_1)}{\ell(f_2)} = \frac{12}{19} \approx 0.632 \,,$$

we observe that they are within 4% of each other. So indeed, list comprehensions are faster than explicit loops, and rather close to the number expected by just examining the disassembled bytecode.

This is of course an academic example, because the body of the loop does almost no work. The proportion of overhead from the looping mechanism, out of the total bytecode length (in both variants of the code), will be smaller in real-world use cases, where more work is done per item.

Finally, intuition tells us that Numba, or indeed any compiler, cannot do much here, as the loop operates on Python objects. Trying it out:

```
import numba
g = numba.jit(f2)  # prepare f2() for JIT compilation, store the result into "g"

%timeit g()  # compile at first call
The slowest run took 78260.53 times longer than the fastest. This could mean that an
intermediate result is being cached.
1000000 loops, best of 3: 511 ns per loop
```

(What the large spread in runtimes actually means is that the JIT compiler was invoked on the first run.)

We now get 511 ns vs. the earlier 522 ns; in practice nothing happened.

But as a bonus, this tells us that the JIT compilation step took

$$t_{\mathrm{JIT}} \approx 78620.53 \cdot 511\,\mathrm{ns} \approx 0.0402\,\mathrm{s} \,.$$

5.12 Publishing and distribution

Sometimes, you may want to publish your Python-based project. There are de facto standard ways of doing this, which we will very briefly look into here.

5.12.1 Online code repositories

The first step in publishing a Python-based project is to publish the codes on the internet. GitHub and BitBucket are popular social media for source code version control and publishing. If you would like others to find your project, it is recommended to at least upload the code to one of these services.

Due to the simple reason that I'm personally more familiar with GitHub, most of the tips below will focus on it.

- Both GitHub and BitBucket work by hosting version control repositories. Roughly, a repository is a place to store codes from one project. A user can have multiple repositories (e.g. one for each project).
- GitHub was originally based on `git`, whereas BitBucket was based on Mercurial (`hg`). Nowadays they both support multiple different backends.
- Specifically on GitHub, for public (open-source) projects, a free account is fine.
- 10-minute tutorial to getting started with GitHub:
  https://guides.github.com/activities/hello-world/
- It is considered polite to supply at least a minimal README, to tell readers what the project is all about, how to install it, and basic usage. See existing projects for examples.
  - On GitHub, READMEs are usually written in Markdown, which is a minimal markup language (think: like HTML, but much less verbose). To learn it in 5 minutes:
    https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet
  - GRIP (Github Readme Instant Preview):
    https://github.com/joeyespo/grip
- Tagging your repository with keywords is also recommended, as this makes it easier to find by topic.
  - GitHub has auto-completion of keywords. E.g. if you start typing "lin", it will suggest (among other possibilities) "linear", and if you then continue to type "linear-eq", it will suggest "linear-equations".
  - This is meant to unify the spelling of keywords entered by different users for the same topic, thus improving searchability.
  - Hence, try first if any of the existing keywords suits your purposes, and only if nothing matches, then add your own (just type it).
- Once your code is in what you think is a reasonable state, *tag* it, i.e. assign a unique version number to that state — and then, create a *release*.
  - This ensures that users (and you yourself) can, at an arbitrary later time, return to that known state of the code, if needed.
    - E.g. in scientific use, it may be useful to assign a version number to the version of the code that produced the figures for a published article.
    - Then, if the figures underwent corrections in the referee process, the final version of the code should have a new version number.
  - How to number versions?
    - Different software projects use different versioning schemes.
    - *Semantic versioning* is a scheme that standardizes the meaning of each component of the version number:

      http://semver.org/
- GitHub has many hidden (but documented) keyboard shortcuts, which accelerate the workflow for advanced users. See:

  https://help.github.com/articles/using-keyboard-shortcuts/

5.12.2 Packaging: distribution to other Python programmers

If you want to go a step further — especially, if your project is somehow generic or reusable — consider *packaging* the project to allow other Python programmers to install it conveniently.

It must be said that packaging can be rather technical, and there are several ways to do it wrong, so this is perhaps a world you'll want to dive into only after obtaining a certain degree of familiarity and experience with Python.

With that warning aside, the main points:

- Packaging guide (**recommended**):

  https://packaging.python.org/

  - The Hitchhiker's Guide to Python also has a section on packaging, but with a different focus (e.g. tips on setting up a private server to publish proprietary applications):

    http://docs.python-guide.org/en/latest/shipping/packaging/

- The Python packaging process is still evolving, so detailed packaging advice may outdate quickly. The following is valid as of late 2017.
- The recommended packaging software is `setuptools`.
  - `setuptools` comes with Python.
  - Instead of a plain configuration file, `setuptools` uses a Python script to control the software (standard name `setup.py`). This script typically makes some definitions specific to the project, and then invokes `setuptools` to do the actual work.
  - PyPA (Python Packaging Authority) maintains an example project on how to use `setuptools`:

    https://github.com/pypa/sampleproject/
  - You will mainly want to create an `sdist`, i.e. a source distribution.
    - For most scientific projects, an `sdist` is enough; even if you use Cython, scientific users are not scared by an automatic compile step, and will likely have the tools for that already installed anyway.
  - The recommended binary format is "wheel". Pure Python projects can be built into *universal wheels*, which are independent of the OS they are running on.
    - If you use Cython, it is possible, but possibly also much more involved, to create binary distributions. These are *platform wheels*, i.e. OS-specific binaries.
  - Including data files into the package — correctly, so that the resulting package works in any environment — can be rather technical. The main distinction is between what goes into the source distribution, and what gets installed. See here (for Cython, but the same notes apply to pure Python projects):

    https://github.com/Technologicat/setup-template-cython/#packaging-your-data-files
- Upload your package to PyPI to make it available for `pip install`.
  - This helps also you, if you have more than one computer.
  - When you publish your first project on PyPI, you will need to create a user account, which then remains valid indefinitely for all of your current and future projects.
  - The packaging guide covers publishing on PyPI, too:

    https://packaging.python.org/tutorials/distributing-packages/#uploading-your-project-to-pypi
- About other tools:
  - An overview of different (also historical, obsolete) tools:

    http://stackoverflow.com/questions/6344076/differences-between-distribute-distutils-setuptools-and-distutils2
    - A brief history of Python packaging, up to 2015:

      https://www.pypa.io/en/latest/history/
  - Tools operating at the next higher abstraction level, to provide a simplified interface to Python packaging, already exist; but no single dominant software has emerged yet. There are at least Flit and pbr.

    https://github.com/takluyver/flit

    https://docs.openstack.org/pbr/latest/

5.12.3 Freezing: distribution to end users

On the other hand, if you have developed an application for end-users, they do not likely care that it is written in Python; they just want something that runs. This is the territory of *freezing* (contrast *packaging* for other programmers).

Technically speaking, freezing an app packages both your own code and a Python interpreter into a single install package, which has everything that is needed to run your app. Python's license allows doing this regardless of whether your app is open-source or proprietary.

- The Hitchhiker's Guide to Python has a section on freezing:

  http://docs.python-guide.org/en/latest/shipping/freezing/

- Tools:
  - PyInstaller [Windows, Linux, OS X]

    http://www.pyinstaller.org/
  - py2app [OS X]

    https://py2app.readthedocs.io/en/latest/
  - py2exe [Windows]

    https://pypi.python.org/pypi/py2exe/

- An alternative on Linux is to make .deb packages, but this is not recommended, because they would depend on the system Python.

  http://docs.python-guide.org/en/latest/shipping/packaging/#packaging-for-linux-distributions-ref

- For the curious: Python's LICENSE:

  https://docs.python.org/3/license.html

5.13 Crash course in software engineering

This section very briefly, and selectively, covers some of the very basics of software engineering. This is to raise awareness for these ideas and tools in the scientific community, and also to give some basic practical pointers to improve productivity. For an overview, see Wikipedia:

https://en.wikipedia.org/wiki/Software_development

As for design, *agile software development* may be of interest. It emphasizes an iterative approach to software design, instead of trying to nail down all requirements at the beginning (contrast the classical *waterfall model*).

https://en.wikipedia.org/wiki/Agile_software_development
https://en.wikipedia.org/wiki/Waterfall_model

5.13.1 Version control

Perhaps one of the most important tools in all of software engineering is *version control*. A version control software that has already remained popular for quite many years now is `git`, so that is what we will focus on here.

Version control means that previous revisions of the code will be automatically saved for future reference. As the user, at any given time, you still see only one version (the current *working tree*) of your code — practically eliminating the mistake of accidentally making changes to the wrong version. Or in other words: no more numbered zip files; and with some very minimal effort toward writing commit messages, no more guessing which version of the code did what: version control gives you a searchable log.

Version control allows:

- Logging a human-readable summary description of each set of changes.
  - This requires a minimal amount of additional effort for writing the description, but is very convenient when returning to the code (months or years!) later.
  - An ideal log message is short, human-readable, descriptive.
  - Logs can be viewed per-file, for a set of files, or for the whole project.
  - Logs can be text-searched, which is useful for browsing or searching the project history later.
- Fully automatically, seeing what *exactly* has changed between any two versions of the code, down to the last changed line.
  - This requires no additional effort from the user.
  - *Any version* includes the version currently being worked on (i.e. also before *committing* changes to version control).
    - This is very useful for checking that no unintended changes have creeped in.
- Going back to any earlier version, and returning to the latest version.
  - Hence, no need to keep around commented-out blocks of code (that were possibly difficult to implement but not needed right now), since old stuff can be retrieved from version control.
- Tagging important versions.
  - Usually software releases.
  - In scientific computing, useful for later identifying e.g. the version that was used to produce figures for a published article.
- In larger projects, branching to test out new ideas.
  - From branches, change sets can be selectively *backported*, i.e. included into the master version under development.
  - Change sets can also be *rebased*, i.e. to be made to apply to a different version of the code. This is automatic, if the relevant parts of the "base" code have not changed too much.
  - For readers familiar with gaming, branches can be used as "savepoints" before committing changes that may break something:

    http://think-like-a-git.net/sections/experimenting-with-git/branches-as-savepoints.html

- Tracking down programming mistakes (bugs) introduced at some point of the recorded development history.
  - The first commit containing a given bug can be identified by *binary search*, i.e., by bisecting a given interval of commits.

    https://en.wikipedia.org/wiki/Binary_search_algorithm
  - The user must know one version (commit) that has the bug, and one version that does not. These are used to set the search interval endpoints.
  - The user must be able to test a given version and say whether it has the bug or not.
  - `git bisect` automates the rest.
  - A binary search completes in $O(\log n)$ steps, so even if the interval to be tested contains lots of commits, the faulty commit will be found in just a few iterations.
- Keeping track of changes (obviously).
  - No need to tag changes in the source code itself (as comments), because you can `git diff` to see what has changed.

Some further things:

- `git` is a *distributed version control system* (DVCS).
  - DVCS is the third generation of version control systems, see e.g.:

    Eric Sink: A History of Version Control:
    http://ericsink.com/vcbe/html/history_of_version_control.html
  - This means that unlike early version control systems such as PVCS, CVS or Subversion (SVN), `git` does not need a centralized server to host the *repository* that contains the version history.
    - In other words, a copy of the *whole version history* of the project is always present *right there on your computer*.
  - You can sync your local repository with that on a server by `pull`ing changes made by others (or by you on other computers), and by `push`ing your own changes to the server.
    - A `pull` is a combined `fetch` and `merge`. Here `fetch` just updates the local *repository*; `merge` incorporates the fetched changes into the current *working tree*.
    - Most of the time, your project will have only one remote repository (standard name `origin`), but it is possible to have several.
      - This is convenient especially when you collaborate on someone else's project on GitHub. Another remote, `upstream`, may point to the original source (so that you can `pull` it easily), while `origin` points to your own copy on your own GitHub account (so that you can `push` your own changes, and then make a *pull request (PR)* on GitHub to the author, asking to incorporate your changes).
- `git` communicates with remotes using an encrypted SSH connection.

  The user is typically authenticated with *key-based authentication*, which does not require typing a password, and is also more secure. On this topic, see e.g.:

  https://help.ubuntu.com/community/SSH/OpenSSH/Keys
  https://en.wikipedia.org/wiki/Secure_Shell

  Read-only access, i.e. `git pull` and `git clone`, also work over HTTPS (encrypted, but no user authentication).

- Proper use of version control requires some self-discipline.
  - Each commit should be related to one thing only.
    - Even a simple whitespace cleanup should go into a separate commit.
  - And we really mean it! Even though, in practice, not everyone does it all the time.
    - If you already modified several unrelated things in the same file, don't worry: `git add --patch` lets you tell `git` which changes belong and which don't.
    - And if you commit, but immediately notice something was broken, just undo it (as long as the broken commit only exists on your own computer):

      https://stackoverflow.com/questions/2845731/how-to-uncommit-my-last-commit-in-git
- Version control has evolved out of the programming community, and consists mostly of command-line tools.
  - Basic everyday tasks, after a short initial learning curve, are much faster from the command line.
  - For more rarely encountered tasks, a graphical frontend may be useful.
    - GitEye [Windows, Linux, OS X]:

      https://www.collab.net/products/giteye
      - Free, based on Eclipse (JGit + EGit), prepackaged for ease of use for this particular task.
      - Instructions for the version control component: http://wiki.eclipse.org/EGit/User_Guide
      - Also contains ALM (application lifecycle management), based on Mylyn. (We will skip this.)
      - Downsides: doesn't support GPG signed tagging (we will skip this); big download (100 MB).
    - TortoiseGit [Windows]:

      https://tortoisegit.org/
  - Specifically for visualizing the commit log, especially in projects that have branches:
    - gitg [Linux]

      https://wiki.gnome.org/Apps/Gitg/
    - GitX [OS X]

      http://gitx.frim.nl/
    - You may also want to make a shell alias (e.g. `gg`) for this command:

      ```
      git log --oneline --abbrev-commit --all --graph --decorate --color
      ```

      which prints a (rather visual) summary to the terminal (command prompt). See

      http://think-like-a-git.net/sections/graphs-and-git/visualizing-your-git-repository.html

Some useful material:

- Sam Livingston–Gray: Think Like (a) Git:
  http://think-like-a-git.net/sections/about-this-site.html
- Ryan Tomayko: The Thing about Git [is never having to say, "you should have"]:

  http://2ndscale.com/rtomayko/2008/the-thing-about-git

  This is a good explanation of `git add --patch`, or in other words, what is and how to solve *the tangled working copy problem*.
- `git` vs. Mercurial (`hg`):
  http://stackoverflow.com/questions/35837/what-is-the-difference-between-mercurial-and-git

5.13.2 Getting started with git

Let's try out the `git` version control system.

Part 1/3: Starting a new project

- Create an empty directory called e.g. `git-tutorial`.
- In this directory, create a text file called `README.txt`. Write something (anything) in it.
- In a terminal (command prompt), go into the directory you created. Give the commands:

  ```
  git init .
  git add .
  git commit -m "my initial commit"
  ```

  The first command creates an empty `git` repository. You need to do this only once per new project.

  The second command adds all files and subdirectories from the current directory into the next *commit*.

  The third command *commits* all pending changes into the repository. The `-m` adds a one-line human-readable message, which will be saved in the commit log (for future reference and searching).

- In a real project, you will want to be more selective, to avoid committing *noise* such as temporary files into version control (noise makes it harder to read logs). For example, for Python projects, you could shellglob

  ```
  git add *.py
  ```

  to add all Python source files that have changed. Also, if your code needs some data files (or computes some data files, taking a long time), it may be good to store those in version control. You can store binaries, too; only text-specific tools such as `git diff` are not available for them (as they will tell you if you try).

  You can do several `git add`s if needed. The command writes the changes into the *staging area*, which roughly means *the next upcoming commit*.

- If this is your first time using `git` (on this computer), now `git` may ask you to tell it who you are. This makes it possible for `git` to automatically mark your commits as yours, in case you later collaborate with someone on the same (usually remote) repository. To configure, invoke the commands

  ```
  git config --global user.name "John Doe"
  git config --global user.email "john.doe@example.com"
  ```

  obviously using your own name and email.

  This needs to be done only once (per computer); all future commits, for any repository, will use this information.

- If you want to fix the author details in the one commit that was already made, run the command

  ```
  git commit --amend --reset-author
  ```

  Note that in everyday usage, this is not needed.

  If, at this point, `git` has decided to be mean and expose you to `vim` just because you needed to write a short log message, you may want to know that `i` enters *insert mode* (the mode where you actually write), `Esc` returns to *command mode*, and the sequence of commands `:wq` (and `Enter`) saves changes and exits. To exit discarding changes, `:q!`.

  To avoid pain in the future, `git` can be configured to invoke a different text editor; see part 3 below.

- Your first commit has been saved to version control! Try the command

  ```
  git log
  ```

  to see your commit log so far. Just like for Python's help system, the reader is GNU `less`; pressing `q` exits. Again, cheat sheet for keyboard commands:

  https://en.wikipedia.org/wiki/Less_(Unix)#Frequently_used_commands

  Try also

  ```
  git status
  ```

  which should report that right now, nothing is staged for the next commit, and that the working directory is *clean*. Clean means here that the directory contains no files that are not in version control, and no changes have been made to files that are in version control.

- Terminology: unlike many other (especially earlier) version control systems, if you start collaborating on someone else's project (copying the code from their existing repository), this is not a *checkout*, but a *clone*:

  ```
  git clone https://github.com/someuser/someproject
  ```

  In `git`, the term `checkout` has a different meaning; we will return to that in part 3 below.

Part 2/3: Working on an existing project

Continuing the previous example:

- Create another file in the same directory with your readme, for example `test.txt`. Write something into it and save.
- Open your `README.txt`, write something there too, and save. (The point is to modify our existing, already version-controlled file.)
- Now that we have made some changes to the working tree, let's:

  ```
  git status
  ```

  You will see something like this:

  ```
  On branch master
  Changes not staged for commit:
    (use "git add <file>..." to update what will be committed)
    (use "git checkout -- <file>..." to discard changes in working directory)

          modified:   README.txt

  Untracked files:
    (use "git add <file>..." to include in what will be committed)

          test.txt

  no changes added to commit (use "git add" and/or "git commit -a")
  ```

  In a real project, this gives you an overview of which files you have changed or added.

- Let's see what exactly has changed:

  ```
  git diff
  ```

  You will see output similar to

  ```
  diff --git a/README.txt b/README.txt
  index 9290ee6..52cdf61 100644
  --- a/README.txt
  +++ b/README.txt
  @@ -1 +1,3 @@
   Testing how to create a git repo.
  +
  +Testing some more.
  ```

  This tells us that in `README.txt`, we have added (+) a blank line, and the line saying "Testing some more.".

- If you have made a lot of changes, a file level overview may be useful. For this:

  ```
  git diff --stat
  ```

  We get:

  ```
   README.txt | 2 ++
   1 file changed, 2 insertions(+)
  ```

  Note that `git diff` only accounts for files which are already in version control! The file `test.txt` does not show up, because it is *untracked* (not in version control).

- Let's add `test.txt` to version control, in our next commit:

  ```
  git add test.txt
  ```

- Let's also add all already tracked files *that we have changed in the working tree*:

  ```
  git add -u
  ```

  The `-u` stands for *update*. This variant of `git add` is very useful in daily work, because you don't have to list (or shellglob) the filenames, and no noise files will be accidentally added.

  But when you use this, keep in mind that when you need to add a new file to version control, that must be done separately (as above).

- Now that we're done with this change set, our next task is to write a commit message.

  But what if a lunch break occurred in between, and we have forgotten what we changed? If we try

  ```
  git diff
  ```

  we get no output, because the changes are already in the staging area. However, `git diff` has a flag for exactly this use case:

  ```
  git diff --cached
  ```

  This gives a report of the changes:

```
diff --git a/README.txt b/README.txt
index 9290ee6..52cdf61 100644
--- a/README.txt
+++ b/README.txt
@@ -1 +1,3 @@
 Testing how to create a git repo.
+
+Testing some more.
diff --git a/test.txt b/test.txt
new file mode 100644
index 0000000..a579d6c
--- /dev/null
+++ b/test.txt
@@ -0,0 +1 @@
+Here is another test file.
```

Now our new file `test.txt` is also shown, because it has been added to the upcoming commit (so if committed, it will be tracked), and there is no corresponding file yet in version control. In the output, the previous revision of `test.txt` has the special path `/dev/null` to indicate this.

- Naturally, a summary is available also for the `--cached` diff:

  ```
  git diff --cached --stat
  ```

  which produces

  ```
  README.txt | 2++
  test.txt   | 1+
  2 files changed, 3 insertions(+)
  ```

- Finally, we commit our changes to version control:

  ```
  git commit -m "made some test edits"
  ```

- Your second commit is done! From here on, just follow the same pattern.

Part 3/3: Advanced

- If your local repository tracks a remote repository (if there is just one, usually called `origin`), you can upload your changes by

  ```
  git push
  ```

  GitHub has a useful help page on `git push`: https://help.github.com/articles/pushing-to-a-remote/

- To download changes from the remote (uploaded by your collaborators, or by you from a different computer), and merge those changes into your working tree:

  ```
  git pull
  ```

  Roughly speaking, this will, essentially,

  ```
  git fetch origin
  git merge origin/master
  ```

  Note that `git merge` always merges *into* whatever branch is active in your working tree at the moment; you just tell it where to merge *from*.

- *Changing the text editor*. Here:

  ```
  git config --global gui.editor gedit
  ```

  Replace `gedit` with whatever command you want `git` to invoke as the text editor.

- *Tagging*. If, for reasons discussed above, you would like to tag the current state of your working tree with a unique version number, this is done as follows:

  ```
  git tag -a "my_version_name"
  ```

  This tells `git` to add an *annotated* (`-a`) tag with the name `my_version_name`. The tag applies to the version currently in the working tree. Typically the name is a version number, such as `v1.0.0`, but it can be anything you want.

  When you do this, `git` will open a text editor (possibly again `vim` unless you have configured it), where you can type in a short message describing this version. If you have written release notes, you can paste those into the message.

- To see the messages of existing tags:

  ```
  git tag -l -n3
  ```

  where the number specifies up to how many lines of each tag message to show. To view a particular tag:

  ```
  git tag -l my_version_name -n50
  ```

- To *check out* an earlier version, i.e. to replace the content of your current working tree with that earlier version:

  ```
  git checkout my_version_name
  ```

  where `my_version_name` is the tag you gave earlier. (It is best to commit any changes or discard them before doing this.)

  The same command, `git checkout`, also allows you to return to any untagged version. Look at the commit log (`git log`), and instead of a tag name, give the *commit ID* (or a long enough part to be unique, from its beginning) from the log. Obviously, this allows you to add tags to old versions later, as long as you can identify the correct commit from the log.

- To return to the most recent version (replacing the content of the working tree with it):

  ```
  git checkout HEAD
  ```

  If you have an older version checked out for some reason, be sure to do this before continuing your development work. Otherwise you many have to create a branch, and then backport its changes onto the master branch. (To avoid mistakes, it is best to return to the most recent version as soon as possible.)

- The `checkout` command has also another use — to discard changes to a particular file:

  ```
  git checkout my_erroneously_changed_file.py
  ```

- To see the current values of `git`'s settings — especially useful, if you once long ago changed something, but can no longer remember what it was — invoke this:

  ```
  git config -l
  ```

  For clarity, that is a lowercase ell (standing for *list*).

  Useful settings include `gui.editor`, `gui.historybrowser`, `merge.tool` and `diff.tool`.

- `.gitignore` file.

  This is very useful to prevent noise files from getting into your repository by accident, even if you "`git add .`". This also has the further advantage that `git status` will not complain about ignored files being untracked.

  Create a file called `.gitignore` (note the leading dot) inside your repository's top-level directory, and write into it shellglobs (one per line) identifying which filetypes do not belong in version control. Typically, these include text editor backups (`*.*~`, `*.bak`) and compiled binaries (`*.obj`, `*.so`, `*.exe`). Paths can also be used; the root "`/`" points to the top-level directory of the repository.

  For both Python and Cython projects, the following is a useful default `.gitignore`:

```
/.project
/build
/dist
*.*~
*.bak
*.pyc
*.c
```

  This tells `git` to ignore Spyder's GUI settings (which change every time if you e.g. simply just change which files are open in the editor), anything under `build/` and `dist/` (for Cython and packaging), any text editor backup files, compiled Python bytecode, and (for Cython) any C-language source code.

- If you accidentally add the wrong file to the upcoming commit, but have not yet committed it yet, undo:

  ```
  git reset HEAD README.txt
  ```

  This does not modify your file; it only undoes the act of placing it into the staging area.

- Undoing a commit (as long as you have not pushed it to a remote):

  https://stackoverflow.com/questions/2845731/how-to-uncommit-my-last-commit-in-git

- Dealing with a tangled working copy (i.e. how to `git add --patch`):

  http://2ndscale.com/rtomayko/2008/the-thing-about-git

- For advanced work, you will want to read about `git`'s *branch* features. In `git`, branches are very, very cheap, compared to early version control systems; `git` doesn't have to do much but create a reference.

  https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging

  To create a new branch, from whatever revision is currently checked out in the working tree:

  ```
  git checkout -b my_awesome_branch
  ```

  The `-b` stands for *make a new* ***b****ranch*. To switch branches later:

  ```
  git checkout master             # switch to master
  git checkout my_awesome_branch  # switch back to the other branch
  ```

  You can also upload branches to remotes, and download branches from remotes.

  https://stackoverflow.com/a/6232535
  https://stackoverflow.com/a/9537923

- If your project is open-source, you may want to publish your code on the social version control service GitHub; this makes collaboration easier. See the section on publishing and distribution, above.

5.13.3 Comments (with a slice of very basic computer science)

There is an old programming adage:

Writing code is easy, reading code is very hard.

The worst part is that this applies also to your own code: in six months, you won't remember the details. Hence, it is a good idea to document the *why* right when you write the code, or that information will be lost, often forever (barring an exercise in reverse-engineering your own code, and re-discovering the pitfalls that led to your particular design choices).

User manuals and beginner explanations of programming languages sometimes do new programmers a disservice, by introducing comments as *something that is ignored*. Comments may very well be ignored by the computer when it executes the program, but for humans, they are extremely important!

Some useful material:

https://en.wikibooks.org/wiki/Python_Programming/Source_Documentation_and_Comments

https://en.wikipedia.org/wiki/Comment_(computer_programming)

It is often said that in comments, one should *document the why, not the how*. For example, see the online essay by software developer Jeff Atwood:

Jeff Atwood: Code Tells You How, Comments Tell You Why:
https://blog.codinghorror.com/code-tells-you-how-comments-tell-you-why/

(His recommendations of the classic book *Structure and Interpretation of Computer Programs* (often abbreviated to *SICP*) and Donald Knuth's 1984 essay on *Literate Programming* are worth noting also here.)

But what does "why, not how" mean in practice? It is perhaps most readily explained by example. Consider the following silly program:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

# recursively sum f(k-1) and f(k-2)
def f(k):
    k = int(k)
    if k < 0:
        raise ValueError("out of domain")
    if k < 2:
        return k
    else:
        return f(k-1) + f(k-2)

def main():
    numbers = [ f(j) for j in range(30) ]
    print(numbers)

if __name__ == '__main__':
    main()
```

What does the program do? The comment above `f()` is completely useless, as it focuses on the *how*, which the code itself already describes perfectly clearly — not to mention more exactly: corner cases, termination condition...

To an audience with a background in mathematics, it is probably immediately clear what this program does — it calculates Fibonacci numbers. But what about the (arguably rare) programmer who hasn't yet heard of them?

We can improve the situation easily, by instead writing down *why* `f()` does what it does:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

# compute Fibonacci numbers
def f(k):
    k = int(k)
    if k < 0:
        raise ValueError("out of domain")
    if k < 2:
        return k
    else:
        return f(k-1) + f(k-2)

def main():
    numbers = [ f(j) for j in range(30) ]
    print(numbers)

if __name__ == '__main__':
    main()
```

Changing just one comment, we have given the uninformed reader of this code a *search keyword* that can be immediately used to learn about the topic.

Proper commenting becomes even more important when the program grows more complex. Consider the same example, version 2.0:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

__version__ = "2.0"

# compute Fibonacci numbers (accelerated using memoization)
class Fibo:
    memo = {}

    def __init__(self):
        pass

    def __call__(self, k):
        if k in self.memo:
            return self.memo[k]

        if k < 0:
            raise ValueError("k must be >= 0, got %g" % (k))

        k = int(k)
        out = None
        if k < 2:
            out = k
        else:
            out = self(k-1) + self(k-2)
        self.memo[k] = out

        return out
```

```
def main():
    f = Fibo()
    numbers = [ f(j) for j in range(1000) ]
    print(numbers)

if __name__ == '__main__':
    main()
```

Improvements:

- The error message is now much more descriptive, informing the user about both the valid domain and what the routine got as input (making any bugs in calling code easier to find).
- Another search keyword (*memoization*):
  https://en.wikipedia.org/wiki/Memoization
  - Memoization is a highly useful general programming technique to speed up function evaluation at the cost of memory use. This is called a *space–time tradeoff*.
    https://en.wikipedia.org/wiki/Space%E2%80%93time_tradeoff
    * Memoization depends on the property that given any fixed values of function arguments, the output of the function is always the same.
    * This is a manual application of memoization, but compilers for functional programming languages use memoization automatically.
  - We could also use this bit of magic, which would let us memoize by simply adding a decorator:

    Daniel Miller (2010): A memoize decorator for instance methods [Python recipe]:
    https://github.com/ActiveState/code/tree/master/recipes/Python/577452_memoize_decorator_instance

    [Advanced!] If you want to *understand* the magic, especially `__get__()`, see:

    https://www.blog.pythonlibrary.org/2016/06/10/python-201-what-are-descriptors/
    http://www.ianbicking.org/blog/2008/10/decorators-and-descriptors.html
    https://krzysztofzuraw.com/blog/2016/python-class-decorators.html
- This allowed us to compute the sequence much further (1000 vs. earlier 30) in reasonable *wall time*.
  https://en.wikipedia.org/wiki/Elapsed_real_time
- For this algorithm, memoization decreased the asymptotic runtime to $O(n)$, while — at least at first glance — increasing asymptotic memory use from $O(1)$ to $O(n)$.
  https://en.wikipedia.org/wiki/Asymptotic_computational_complexity
  - But that is not the whole story. The function makes two calls to itself per invocation. Since in the naïve version, none of the results are re-used, each of these calls will spawn two further calls, until the recursion bottoms out separately in each branch of the "call tree" thus generated.
  - Since the code is single-threaded, this "call tree" will be explored *depth-first*.
    https://en.wikipedia.org/wiki/Depth-first_search
  - The deepest branch $f(n-1), f(n-2), ..., f(1), f(0)$ makes $n+1$ calls. Hence, $O(n)$ storage will be needed on the *call stack*.
    * *Call stack*: roughly speaking: in each thread that is running, each currently active function call requires some (implicit, automatically handled) storage, so that the program knows where to resume execution when the called function returns.
      https://en.wikipedia.org/wiki/Call_stack
- What was the asymptotic runtime of the original version? By writing out the first few cases manually, the previous consideration is easily seen to lead to, in total, $\text{fib}(n+1)$ internal calls to $f()$ for any $n \geq 2$ (where — like in our program — $\text{fib}(0) = 0$, $\text{fib}(1) = 1$). The original runtime was thus $O(\text{fib}(n))$.

- The Fibonacci sequence grows quickly as n increases. As a few lines of NumPy — when added to the above improved program — will tell you, $\log_{10}(\text{fib}(n+1)) - \log_{10}(\text{fib}(n)) \approx 0.209$ for any $n \gtrsim 20$.
- Contrast this with polynomial runtime. Since $\log_{10}(n^k) = k \log_{10}(n)$, we have:

$$\log_{10}((n+1)^k) - \log_{10}(n^k) = k\left[\log_{10}(n+1) - \log_{10}(n)\right] = k \log_{10}(1+1/n) \ ,$$

  which tends to zero as $n \to \infty$. Hence the Fibonacci sequence, for large enough n, grows faster than any polynomial. (In the exact mathematical sense that for any fixed $k > 0$, there exists $n_0 = n_0(k)$ such that...)
  - Although n^k is, mathematically speaking, a monomial, the standard technical term is polynomial runtime, because the $O(\dots)$ discards any asymptotically non-dominant terms.

Some tricks can also be gleaned from the improved program:

- `memo` is a static member, shared across all instances of Fibo. Thus all current and future instances (during the same run of the program) benefit when one instance saves a result to `memo`.
  - Even the internal call for $f(n-2)$ will benefit from the call to $f(n-1)$.
  - Even in a multithreaded scenario, the GIL prevents *race conditions*, i.e. roughly speaking, inappropriate simultaneous reads and writes that can have unpredictable results.
    https://en.wikipedia.org/wiki/Race_condition
- In Python, defining the magic method `__call__()` makes the object *callable*. The instances of the class can then be called as if they were functions (as `main()` does above).
  - Like "`operator()`" in C++.
  - This is of course just syntactic sugar — we could have as well defined a method called `f()` and used that — but some programmers find using the callable object syntax more aesthetic.
    https://en.wikipedia.org/wiki/Syntactic_sugar
  - In this example, `__call__()` itself calls itself: `self(k-1)` actually means `self.__call__(k-1)`. We used this to implement the recursion.
    https://en.wikipedia.org/wiki/Recursion_(computer_science)
- Python modules often define a module-level constant called `__version__`. This is the standard way to inform the user (and other Python programs) about the module version.

Let us end this topic with two final examples from real projects I've worked on.

The flip side of *document the why, not the how* is that *sometimes it is useful to document the how* — if you stick to a general enough overview, which is sometimes easily lost in the details in the actual code. The description of low-level technical details should be left to the code itself. If it is not readable, it should be rewritten more clearly.

But this is sometimes impossible. The code may already be as clear as it can be; some things are inherently complex. In those cases, especially with particularly cryptic-looking vectorized algorithms in numerics, it may be very helpful to give a functional equivalent of the code — albeit one that would run very slowly, if actually used — in the comments. It doesn't matter if the *explanation* uses nested for loops, as long as it gets the idea across.

An example:

```
    I,J,IJ = util.index.genidx( (nx, ny)  )  # see genidx() below
    K,L,KL = util.index.genidx( (nxb,nyb) )

    # loop only over rows of the equation system
    for i,j,ij in zip(I,J,IJ):
        A[nf*ij,  KL] = Du[i,K] * Av[j,L]
        A[nf*ij+1,KL] = Au[i,K] * Dv[j,L]

    b[nf*IJ]   = dpsi_dx[I,J]  # RHS for B_x
    b[nf*IJ+1] = dpsi_dy[I,J]  # RHS for lambda_xx

#    # the above is equivalent to this much slower version:
#    #
#    # equation system row
#    for j in range(ny):
#        for i in range(nx):
#            ij = np.ravel_multi_index( (i,j), (nx,ny) )
#
#            # equation system column
#            for l in range(nyb):
#                for k in range(nxb):
#                    kl = np.ravel_multi_index( (k,l), (nxb,nyb) )
#                    A[nf*ij,  kl] = Du[i,k] * Av[j,l]
#                    A[nf*ij+1,kl] = Au[i,k] * Dv[j,l]
#
#            b[nf*ij]   = dpsi_dx[i,j]  # RHS for B_x
#            b[nf*ij+1] = dpsi_dy[i,j]  # RHS for lambda_xx
```

The module containing `genidx()` is a study in informative commenting and docstringing (although maybe its module docstring is not):

```
"""Utilities for converting indices.

Created on Fri Mar 24 13:21:07 2017

@author: Juha Jeronen, juha.jeronen@tut.fi
"""

import numpy as np


# These generalize to the nD case given below.
#
#def genidx2D( nx,ny ):
#    """Generate index vectors to a 2D meshgrid and the corresponding raveled array."""
#    i = range(nx)
#    j = range(ny)
#    I,J = np.meshgrid( i,j, indexing='ij' )
#    Ilin = np.reshape(I,-1)
#    Jlin = np.reshape(J,-1)
#    IJ = np.ravel_multi_index( (Ilin,Jlin), (nx,ny) )
#    return (Ilin,Jlin,IJ)
#
#def genidx3D( nx,ny,nz ):
#    """Generate index vectors to a 3D meshgrid and the corresponding raveled array."""
#    i = range(nx)
#    j = range(ny)
#    k = range(nz)
#    I,J,K = np.meshgrid( i,j,k, indexing='ij' )
#    Ilin = np.reshape(I,-1)
#    Jlin = np.reshape(J,-1)
#    Klin = np.reshape(K,-1)
#    IJK = np.ravel_multi_index( (Ilin,Jlin,Klin), (nx,ny,nz) )
#    return (Ilin,Jlin,Klin,IJK)

def genidx( shp ):
    """Generate index vectors to an nD meshgrid and the corresponding raveled array.

Parameters:
    shp: tuple of int
        Shape of the meshgrid (nx, ny, nz, ...).

Returns:
    tuple (I_0, I_1, ..., I_{n-1}, R), where:
        I_0, I_1, ..., I_{n-1}: rank-1 arrays
             These index each dimension of the meshgrid as M[I_0, I_1, ..., I_{n-1}].

        R: rank-1 array
             Corresponding indices to the corresponding raveled array.

so that M[I_0[j], I_1[j], ..., I_{n-1}[j]] and raveled[R[j]] denote the same grid point.
"""
    if np.array(shp).ndim > 1:
        raise ValueError("shp must be a list or a rank-1 array")
```

```
# Create index vectors for each dimension of the grid.
#
# Note that
#
#    shp = (n_0, n_1, ..., n_{d-1})
#
# The kth vector has length shp[k].
#
i = [ range(nk) for nk in shp ]

# Create the meshgrid:
#
#     I = (I_0, I_1, ..., I_{d-1})
#
# where each I_k is a rank-d array of shape shp (accepting a multi-index of d dimensions),
# where the value is the grid point index on the kth axis.
#
# This is used just to generate all prod(shp) combinations of the per-axis indices.
#
I = np.meshgrid( *i, indexing='ij' )

# Flatten the meshgrid to remove grid structure, so that the tuple
#
#     pj = ( Ilin[0][j], Ilin[1][j], ..., Ilin[-1][j] )
#
# gives the indices, on each axis, of the jth grid point.
#
# Each Ilin[k]:
# - is a rank-1 array of length prod(shp).
# - takes on values in the range 0, 1, ..., shp[k]-1.
#
# Ilin is now the multi-index.
#
Ilin = [ Ik.reshape(-1) for Ik in I ]

# Generate the raveled index that corresponds to the multi-index.
#
# This numbers the grid points sequentially.
#
R = np.ravel_multi_index( Ilin, shp )

out = list(Ilin)  # just for semantic tidiness; we don't actually need a copy
                  # as the original Ilin is no longer needed
out.append( R )
return out
```

Here is a final example, incidentally also about index conversion, and here too more explanation than actual code. This obtains efficient access for updating data in a CSR (compressed sparse row) sparse matrix without rebuilding the sparsity pattern. A more detailed use case is given in the docstring of `CSR_location_of_nonzeros_as_IjIk()`.

```
def IjIk_to_CSR_index(A, Ij, Ik):
    """Convert vectors of (row,column) index pairs to raw CSR matrix data vector indices.

    Parameters:

    A: scipy.sparse.csr_matrix
        The sparse matrix in CSR format.
    Ij: rank-1 np.array
        Row indices.
    Ik: rank-1 np.array
        Column indices.

    Returns:
        rank-1 np.array of indices to A.data, with result[k]
        corresponding to (Ij[k],Ik[k]).

    Requirements (NOT checked):

    Ij and Ik must be of the same length.

    Each referred matrix element (Ij[k],Ik[k]) must already exist within
    the sparsity pattern of A.
"""
    result = np.zeros( (np.size(Ij),), dtype=int )

    # indptr contains an extra element pointing one-past-end, so diff(indptr)
    # returns the number of data elements on each row.
    #
    # It is the same as the number of entries in A.indices on that row,
    # containing the column numbers of the corresponding data elements.
    #
    dlen = np.diff( A.indptr )

    # Keep only entries for rows which we refer to:
    #
    dlen = dlen[Ij]
    imax = np.max( dlen )  # largest number of nonzeroes on one row

    # Do the actual index finding in a vectorized manner, so that we
    # never need to Python-loop more than imax times.
    #
    # Ij and Ik may have any number of elements; the only requirement is
    # that each entry (Ij[k],Ik[k]) exists in the sparsity pattern of A.

    # The last few rows may need separate handling, because in a FEM matrix
    # especially the last row is likely to have fewer nonzero elements than
    # the maximum number encountered.
    #
    # Hence, if we tried to walk it up to this maximum (which we can do on all other
    # rows to keep things vectorized), we would try to read past the end of data,
    # triggering an IndexError from NumPy.
    #
    # To prevent this, we copy the index data into a slightly larger array,
```

```
    # allowing us to read a few elements past the end of the data. Any results
    # past the end of a row will not be used, but computing them must be possible
    # in order to keep the computation fully vectorized.
    #
    # Allocating uninitialized memory is fine, since the remainder is not used
    # for anything sensible.
    #
    aimax = np.size( A.indices )
    temp_indices = np.empty( (aimax+imax,), dtype=int )
    temp_indices[0:aimax] = A.indices

    for i in range(imax):
        # Explanation of black magic:
        #   - (i < dlen) matches only for those elements where we have
        #     not exceeded the length of the data of the current row.
        #     Hence, at each row, we effectively stop looking once the
        #     length of data is exceeded.
        #
        #   - A.indices[ A.indptr[Ij] ] are the column indices of the first entry on rows Ij.
        #     Using temp_indices[] instead of A.indices[] allows (nonsensical) reads past the
        #     end of the array, so that we won't need special handling for the last few rows.
        #
        #   - Then A.indices[] at this location +1, +2, ... +(dlen-1) are the column indices
        #     of the other entries at these rows, hence at ith iteration of the loop we look
        #     at this +i.
        #
        #   - The check for equality with Ik returns True (1) only if the column index
        #     is the one we are searching for. Hence, at each element of the result,
        #     we sum something nonzero only once (as there are no duplicates; in the CSR
        #     format, each entry of A has a unique (row,column) index pair).
        #
        #   - The first two factors in the product are boolean, serving only to
        #     switch the last factor on (1*...) or off (0*...).
        #
        #   - The final multiplication by (A.indptr[Ij] + i) gets us the data we want,
        #     when the loop finds the match.
        #
        idx = A.indptr[Ij] + i
        result += (i < dlen) * (temp_indices[ idx ] == Ik) * idx

    return result

def CSR_location_of_nonzeros_as_IjIk(A):
    """Companion to IjIk_to_CSR_index().

    Returns the row and column index vectors corresponding to ALL nonzero entries
    of a CSR matrix.

    In a sense, this is the inverse operation to IjIk_to_CSR_index(),
    converting "to the other direction".


    Where this is needed: consider the following example, which takes in
    sparse CSR matrices A and B, and creates another sparse CSR matrix C.
    The sparsity patterns of A and B can be different (this is important
    for the example to make any sense).
```

```
    C = A + B
    Ij,Ik = CSR_location_of_nonzeros_as_IjIk(A)
    dataind_A = IjIk_to_CSR_index(C, Ij, Ik)
    Ij,Ik = CSR_location_of_nonzeros_as_IjIk(B)
    dataind_B = IjIk_to_CSR_index(C, Ij, Ik)

    Now "dataind_A" contains the raw CSR data vector indices of those elements in "C",
    which correspond to the contribution of the CSR matrix "A". Likewise, "dataind_B"
    contains the indices in "C" of the contribution from "B".

    If all three matrices have sorted indices (guaranteeing that the internal data storage
    order is the same), this allows us later to directly update the data of "C" with data
    from another CSR matrix, which shares its sparsity pattern with the example's original
    "A" (or "B").


    This, in turn, can be useful e.g. in some implicit time integration schemes, which contain
    expressions such as "C = A - dt*B". Here A and B have different sparsity patterns, and "C"
    must be recomputed at each timestep, or in nonlinear problems, even at each iteration.

    The idea is to create the matrix C (creating the correct sparsity pattern) only once,
    and then directly manipulate its data vector, avoiding unnecessary and expensive
    regeneration of the sparsity pattern every time the data needs to be recomputed.

    Continuing the example, the above expression can be computed efficiently as

    C.data[:] = 0.0  # clear old data
    C.data[ dataind_A ]  = A.data
    C.data[ dataind_B ] -= dt*B.data

    (To be efficient, it is assumed that dataind_A and dataind_B are computed only once
     and cached.)

    At this point, it should be clear why sorted indices are needed to guarantee that
    this trick works.

    Parameters:
        A: CSR matrix (of type scipy.sparse.csr_matrix)

    Returns:
        tuple (Ij,Ik): pair of rank-1 np.arrays, each having length A.nnz.

    The matrix element stored at A.data[k] has row and column indices (Ij[k],Ik[k]).
"""
    Ij = []
    Ik = []

    dlen = np.diff( A.indptr )
    for j,rowstart in enumerate(A.indptr[:-1]):
        Ij.extend( [j for k in range(dlen[j])]  )  # [j, j, ..., j], dlen[j] copies
        Ik.extend( A.indices[rowstart:(rowstart+dlen[j])] )

    return (np.array(Ij, dtype=int), np.array(Ik, dtype=int))
```

5.13.4 Assertions

In software, as in many fields of engineering, reliability is important. If a failure occurs silently, the program state almost always becomes corrupted in some way. This will eventually lead to a visible error, but often much later, long after the context of the original failure has been lost. It is then almost impossible to find the bug that caused the error. The software engineering solution, to eliminate this situation, is to make the program *fail fast*.

A program that fails fast, immediately reports any condition that is likely to indicate failure and stops processing, rather than continuing with a possibly flawed process. When accompanied by a sufficiently informative error message, this makes the cause of the failure much easier to find, and fix early in software development.

Assertions are the key tool for the fail-fast approach. They catch when a critical assumption does not hold, halt program execution (i.e. intentionally cause a controlled crash), and display the specified error message (which can contain arbitrary information, as set by the programmer). Assertions should be made to trigger on failures of *internal* assumptions that should always hold, if the program is implemented correctly. Failures of assumptions about *external* state — such as invalid inputs, or memory allocation errors — should raise an exception instead.

https://en.wikipedia.org/wiki/Fail-fast
https://en.wikipedia.org/wiki/Assertion_(software_development)

Generally speaking, control flow in software can be complex; program state may change in nontrivial ways. Assertions make your assumptions explicit, even if you're sure that in your current code, they always hold. (If the code is good, you might reuse pieces of it much later, in a different context, with different internal assumptions.)

For example, if you have a function that generates a point inside a unit circle centered on x_0, and another (internal, not user-facing) function that then uses such a point as input, in this second function you can:

```
assert np.sum( (x - x0)**2 ) < 1, "x = %s is not inside the circle" % (x)
```

In any code following the assert (in the same function), you can then be certain that this *precondition* is fulfilled; because if not, the program will halt and alert you — the developer — to the invalid value that somehow got in.

We do not raise an exception; a failure here is an internal error. Contrast invalid inputs, which ultimately come from the user; or running out of memory, which is an external condition that may occur regardless of whether the program is implemented correctly. An exception alerts the *using* programmer; an assertion alerts the *developer*.

There are also some finer points. Quoting:

Jim Shore: Fail Fast (IEEE Software, September/October 2004, 21–25):
https://www.martinfowler.com/ieeeSoftware/failFast.pdf

When you're writing a method, avoid writing assertions for problems in the method itself. Tests, particularly test-driven development, are a better way of ensuring the correctness of individual methods.

Assertions shine in their ability to flush out problems in the seams of the system. Use them to show mistakes in how the rest of the system interacts with your method.

Finally, assertions are related to the software design approach of *contract programming* (a.k.a. *programming by contract; design by contract*):

https://en.wikipedia.org/wiki/Design_by_contract
https://www.eiffel.com/values/design-by-contract/introduction/

where they can be used to test *preconditions*, *postconditions*, and *invariants*. For Python, programming by contract was proposed, in 2003, as PEP 316:

https://www.python.org/dev/peps/pep-0316/

This was, however, deferred, and the idea has not been proposed again. While there is currently no one obvious way to do it, as of late 2017, at least one actively maintained library exists:

- PyContracts:
  https://github.com/AndreaCensi/contracts
- There is also a DIY approach in the PythonDecoratorLibrary:
  https://wiki.python.org/moin/PythonDecoratorLibrary#Pre-.2FPost-Conditions

5.13.5 Tests

Testing is an integral part of developing error-free software. Nowadays, much of testing is automated. To give some flavor of this, we will briefly introduce *unit testing*, and just mention *coverage testing* and *integration testing*.

https://en.wikipedia.org/wiki/Software_testing
https://en.wikipedia.org/wiki/Unit_testing
https://en.wikipedia.org/wiki/Integration_testing

Unit testing aims to verify the correct working of *units*, which are typically individual modules. *Test-driven development* is a methodology related to this. Often unit tests come as separate modules, one for each unit of actual software to be tested. Unit tests must be manually created by the programmer, but software frameworks exist to automate the repetitive task of running them against each version of the software. For Python, popular testing frameworks include pytest and nose.

Unit testing is particularly applicable to scientific computing software, where the control flow is (almost) trivial, there are relatively few connections between parts, and many parts are of a nature that can be easily tested individually.

The main use of unit tests is to automatically catch any *regressions*, i.e. things that have worked previously, but have become broken when the code was modified.

Roughly speaking, a single test in a unit test module is a function invocation that produces a known result. The test module calls the function, and compares the computed result to the pre-determined answer. Any mismatch then triggers an assertion failure, or error logging, with a message identifying which test failed.

Any tricky corner cases can be listed once into a test script, and then all future versions of the software can be automatically verified whether they handle these cases correctly. (For example, even vs. odd length of input in some algorithms.)

Unit testing also helps test the handling of invalid inputs. Even if in your current code, you are certain there are no invalid inputs to a given routine, it may occur that when you later reuse the code in a different context, this is no longer the case. (For example, what to do if some size argument n is negative, or of the wrong datatype.)

Why test invalid inputs? It is good practice to assume that the caller is actively trying to break your routine, especially in user-facing code, to protect the programmer using your routine (maybe you yourself) from innocent mistakes in calling code.

If the code is to be deployed for public use on the internet, things become more complicated. One must also take into account possible (likely!) black hat activity — i.e. intentional, often automated, attempts to take over control of the system hosting the code, by utilizing information security vulnerabilities in the code. For some material on such *dangerous programming mistakes*, see, for example:

http://www.infoworld.com/article/2622611/application-security/developer-error–the-most-dangerous-programming-mistakes.html
http://cwe.mitre.org/top25/index.html

Coverage testing is a special kind of meta-testing that checks that the unit tests cover everything they should. Unless the unit tests are validated with such tools, it is likely that there exist some code paths that are overlooked (not executed) by the tests. For Python, there is Coverage.py:

https://pypi.python.org/pypi/coverage

Coverage.py measures code coverage, typically during test execution. It uses the code analysis tools and tracing hooks provided in the Python standard library to determine which lines are executable, and which have been executed.

Finally, *integration testing* aims to verify that parts of the system (the units) work correctly as a whole. A recent trend in automated integration testing is *continuous integration*, which automatically runs integration tests for the most recent version of the code daily. For open-source projects hosted on GitHub, Travis CI provides an online service that can be used for this.

https://en.wikipedia.org/wiki/Continuous_integration
https://travis-ci.org/

5.13.6 Static analyzers and linting

Roughly speaking, in its most basic form, *static code analysis* is a fancy name for looking at code, and reasoning about it *statically* — i.e. without running it. (To be technically correct, *static analysis* implies that any *dynamic* modifications to the code, e.g. class attributes created only at run time, will not be seen by the analysis.)

At this point it should come as no surprise that also for static code analysis, automated tools exist. If you have used an IDE such as the MATLAB GUI, you may already be familiar with static analyzers. Many IDEs use them to alert the programmer about things such as calls to undefined functions, and assigned to but unused variables.

The main benefit of using a static code analyzer is that it can catch many common mistakes before you even run your code. Also, static analysis may catch certain types of regressions; e.g. search for the "old chestnut" in:

https://medium.com/@acboulder/type-hints-are-scary-f52d07a36a31#.yt2e91emo

(A variable was renamed, without changing the reference to it in the relevant exception handler. This broke error handling, causing the program to crash if an error occurred.)

The same tools may also perform automated quality control, such as keep track of remaining TODOs and FIXMEs, and of missing docstrings. They may also help enforce a particular *coding style*, which standardizes things such as the use of whitespace (where not semantic).

Python actually has an official coding style. You may want to look at it, if you want to go the extra mile to make your code look like "standard Python"; anyone reading your code will appreciate this.

PEP 8: Style guide for Python code:
https://www.python.org/dev/peps/pep-0008/

Code Style in The Hitchhiker's Guide to Python:
http://docs.python-guide.org/en/latest/writing/style/

The Spyder IDE for Python comes with two static analyzers, which perform different tasks: Pyflakes and Pylint.

Pyflakes is a fast and light tool to statically analyze single Python source files. In Spyder, it is integrated into the code editor, and by default runs automatically every few seconds. It produces the exclamation marks in the margin (à la MATLAB) that alert about problems such as references to undefined names. Settings can be found in Tools ▷ Preferences ▷ Editor ▷ Code Introspection/Analysis.

Pyflakes has a friend called pycodestyle that checks the coding style against PEP 8. Both are packaged into flake8. Spyder includes this, too; realtime PEP 8 code style analysis can be toggled in the aforementioned settings.

For fixing the coding style of an existing code to conform to PEP 8, see the autopep8 package on PyPI, which can automatically fix most issues detected by pycodestyle.

Pylint is more thorough, and takes some time to analyze the code when invoked. In Spyder, Pylint is found at Source ▷ Run static code analysis. This produces a report, where the individual items can be clicked to jump to the relevant line in the code.

Very minimal settings can be found in Spyder in Tools ▷ Preferences ▷ Static code analysis, but customizing which warnings to emit requires supplying a configuration file (`pylintrc`). Pylint is rather pedantic by default; for most users, I would recommend creating a configuration file to suppress any unwanted warnings, that may otherwise hide the useful ones under all the noise. For how to do this, see below.

Some links regarding static analyzers:

- The Python Code Quality Authority is the home of many static analysis tools for Python:
  https://github.com/PyCQA
- Pyflakes project page:
  https://github.com/PyCQA/pyflakes
- Pylint documentation:
  https://pylint.readthedocs.io/en/latest/
- Why static analyzers are often called some variant of *lint*:
  https://en.wikipedia.org/wiki/Lint_(software)

5.13.7 Configuring pylint

To create a `pylintrc`, first invoke

```
pylint --generate-rcfile
```

to obtain a file containing up-to-date default settings. Then, a good first approximation of useful enable/disable settings can be found here (search for "pylintrc"):

https://medium.com/@acboulder/type-hints-are-scary-f52d07a36a31

As of late 2017, remove `unidiomatic-typecheck` before pasting the rest into your config; this flag seems obsolete.

Additionally, you may want to make a few specific settings to suppress some noise in projects that use NumPy (i.e. probably all of your projects). See:

https://stackoverflow.com/a/35259944

Explicitly, **here are all the customizations** to `pylintrc` (unchanged settings not shown):

```
[MASTER]

extension-pkg-whitelist=numpy

[TYPECHECK]

ignored-classes=SQLObject,numpy
ignored-modules=numpy

[MESSAGES CONTROL]

enable=bad-indentation,mixed-indentation,unnecessary-semicolon,superfluous-parens

disable=format,design,similarities,cyclic-import,import-error,broad-except,no-self-use,
        no-name-in-module,invalid-name,abstract-method,star-args,import-self,no-init,
        locally-disabled
```

All the disable flags are to be written on the same line.

Finally, save a copy of the modified `pylintrc` somewhere safe (where you can find it later).

If you want to use the same Pylint settings for all of your projects, in *nix (Linux, OS X) you can save the configuration as `~/.pylintrc`.

If you want to customize the settings for different projects separately, make a copy of `pylintrc` into the project directory of each project you want to check using Pylint.

5.13.8 Debuggers

There are many approaches to hunting down programming mistakes. Static analyzers help somewhat; judicious use of `print()` (and plotting) helps a bit more; but there are bugs that are fastest found with an interactive debugger that allows setting breakpoints, and live inspection of program state. Many MATLAB users are likely already familiar with this.

Also in the Python world, IDEs have a graphical debugger. How to use the debugger in Spyder:

- For interactive debugging, the program must be run inside the debugger.
  - Instead of starting the program normally (Run ▷ Run), in the Spyder menu choose Debug ▷ Debug.
  - As with all developer tools that hook into the code while it runs, this will make the program run more slowly than usual.
- Program execution is paused when the debugger hits a *breakpoint* in your source code.
  - You can set breakpoints basically on any *executable line* in the code (i.e. anywhere except in comments).
    - In Spyder, breakpoints are added by double-clicking the marginal (on the left) at the source line where the breakpoint is to be set or cleared. Also in the menu as Debug ▷ Set/Clear breakpoint.
  - The debugger will pause the program just *before* executing the line that has the breakpoint.
  - In Python, setting a breakpoint on the `def` line of a function means that you want to pause when the function becomes *defined* (i.e. when the `def` line is executed). If you want to pause when the function is *entered*, place the breakpoint inside the function body.
  - A *conditional breakpoint* will pause when hit, but only if a user-given expression evaluates to `True`. The expression is evaluated each time the breakpoint is hit.
- Whenever the program is paused, you can inspect any variables in the current scope in Variable explorer.
  - User manual:
    https://pythonhosted.org/spyder/variableexplorer.html
  - Some data structures such as lists and NumPy arrays can be double-clicked for details.
  - Expressions can be entered in the console to print their values, but (as of Spyder 3.2.3) tab completion of variable names is not available while debugging.
    - This is a current limitation of Spyder, according to the developers:
      https://stackoverflow.com/questions/25912291/no-autocompletion-with-the-ipdb-prompt-when-using-spyder
    - In bare IPython+ipdb, tab completion works:
      https://stackoverflow.com/questions/15122375/get-ipython-tab-completion-for-ipdb
- In the debug menu:
  - *Debug* loads the program into the debugger, but does not start it yet.
    - Use *Continue* to start the program.
    - This is different from some graphical debuggers, where *Debug* both loads and starts the program.
  - *Continue* resumes program execution until the next breakpoint is hit, or until the program terminates.
  - *Step* executes one line and pauses.
  - *Step Into* enters into the next function called on the current line and pauses.
  - *Step Return* resumes until the current function returns, and pauses just before executing the return.
  - *Stop* terminates the program and closes the debugger. (Also, red button above console window.)
- There is no separate "Restart". To reset and debug from the beginning: *Stop*, *Debug*, *Continue*.
  - Keyboard shortcuts: Ctrl+Shift+F12 to Stop, Ctrl+F5 to Debug, and Ctrl+F12 to Continue.
  - Or use the buttons in the toolbar.

Other notes:

- `pdb` is Python's command-line debugger and Python module.
  - If you really want to go there, see this hands-on introduction:
    https://pythonconquerstheuniverse.wordpress.com/2009/09/10/debugging-in-python/
- Currently Spyder does not provide a graphical frontend for debugging Cython programs.
  - Cython comes with the command-line tool `cygdb`, which is a frontend to `gdb` (the GNU Debugger), adding a layer that understands Cython code (if the binary is compiled with debug options enabled).
  - We will not delve further into Cython debugging; for more, see the user manual:
    http://cython.readthedocs.io/en/latest/src/userguide/debugging.html

5.13.9 Profilers

Beside robustness, another obviously desirable characteristic of software is performance. In programming, there is the old adage known as the *80/20 rule*, which states that very often, 20% of the code accounts for 80% of the run time.

It also often occurs that this "20%" is not where one would intuitively expect; although in the specific context of scientific computing, one may reasonably have a faint idea. Still, developer time is much more expensive than computer time; performance profiling of the code is needed to make sure that any performance tuning effort is applied where it matters. In tuning the performance of a code, *premature optimization is the root of all evil* (D. Knuth).

(But in parallelized programs, keep in mind also Amdahl's law.)

Properly, the 80/20 rule is called the *Pareto principle*. Compare *Zipf's law*.

https://en.wikipedia.org/wiki/Pareto_principle
https://en.wikipedia.org/wiki/Zipf%27s_law
https://en.wikiquote.org/wiki/Donald_Knuth
http://wiki.c2.com/?PrematureOptimization

Python comes with a built-in function-level profiler called cProfile. It hooks into your program just like the debugger, and gathers performance statistics while the program runs. To obtain accurate results, you may want to avoid doing anything else on the computer while the program is being profiled. A report is generated when the program terminates.

The report will tell you the total time spent in each function (including any functions it calls), the local time spent in the code of that function itself, and the total number of calls to the function. Doing a test run (or a few), and sorting the report by any of the criteria quickly tells you where the potential performance problems are.

Note that performance profiling is dynamic; it will only gather statistics about the code paths that actually got executed during the profiling.

Sometimes, especially in scientific computing, where functions may sometimes be rather long, function-level information is not enough. In this case, line-level profiling helps. There is a package called line_profiler, which does this.

For profiling the memory use of your program, the package memory_profiler gives line-by-line information on this.

Both line_profiler and memory_profiler require tagging the functions you want to profile with the `@profile` decorator. The decorator does not need to be imported; it becomes magically available when the program is run under profilers that require it. Note that trying to run the program normally, while the code contains `@profile` decorators, raises an error.

Spyder integrates all three profilers into the IDE.

- Function-level profiling is available as Run ▷ Profile.
- Line-level profiling requires first installing the line_profiler Python package, and an official Spyder add-on:

  https://github.com/spyder-ide/spyder-line-profiler
- Memory-use profiling requires first installing the memory_profiler Python package, and an official Spyder add-on:

  https://github.com/spyder-ide/spyder-memory-profiler
- To download the add-ons, get the clone URL from each add-on's GitHub page, and:

  ```
  git clone url_goes_here
  ```
- Both add-ons have been packaged using `setuptools`, so to install, go to the relevant add-on's directory in the terminal (command prompt), and:

  ```
  python setup.py install --user
  ```

  or possibly

  ```
  python3 setup.py install --user
  ```

  as discussed earlier.
- Once the add-ons are installed, and Spyder is restarted, they appear in the Spyder menu as Run ▷ Profile line by line, and Run ▷ Profile memory line by line.
- When you invoke a profiler, the report is generated when your program terminates.
- You can click in the reports to sort by the different columns, and to jump to the relevant line in the code.

Information on the command-line tools:

- https://docs.python.org/3/library/profile.html
- https://github.com/rkern/line_profiler
- https://www.pluralsight.com/blog/tutorials/how-to-profile-memory-usage-in-python

5.13.10 Design patterns

In programming, a *design pattern* (or just *pattern*) is a general reusable solution for a commonly occurring problem. Patterns play a role similar to methods in numerics; compare e.g. *Galerkin method*.

https://en.wikipedia.org/wiki/Software_design_pattern

In this material, we have already seen a few patterns: *Bunch*, *Mixin*, *Function factory*, and *MVC*. The Python-specific *Conditional main* is usually called an *idiom* instead of a pattern, although it fits the definition — likely because it is language-specific:

http://wiki.c2.com/?ProgrammingIdiom

What we have seen here is just a small sample. Many different patterns exist; indeed entire books have been written on them.

The first book on patterns was published in 1994–1995, and is still considered a canonical reference: *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides; Addison–Wesley, 1995.

For some more book recommendations on patterns, see:

https://softwareengineering.stackexchange.com/a/124114

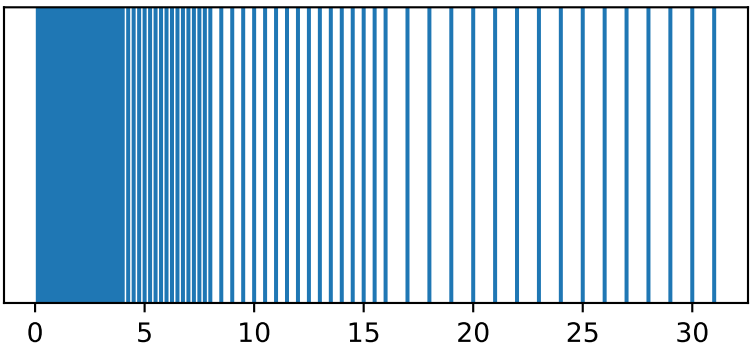
A good online source is the Portland Pattern Repository (a.k.a. C2 wiki), which may be interesting to browse.

http://wiki.c2.com/?WelcomeVisitors
http://wiki.c2.com/?PeopleProjectsAndPatterns
http://wiki.c2.com/?PatternsForBeginners
http://wiki.c2.com/?PatternIndex

This is a good place to stop. But before you go, there's one final bonus topic.

∞ The behavior of floating point numbers

The big picture:



Code:

[Based on a demo in University of Illinois, CS 357, spring 2017]

```
# Based on: University of Illinois, CS 357, spring 2017
# https://relate.cs.illinois.edu/course/cs357-s17/file-version/
#     66922aa3735332d7fb31c9fd9e8f7dde2fda1b37/
#     demos/upload/fp/Density%20of%20Floating%20Point%20Numbers.html

import matplotlib.pyplot as plt

# For illustration; see sys.float_info for real-world values.
mant_dig =  4
min_exp  = -3
max_exp  =  4

floats = []
for k in range(min_exp, max_exp+1):
    for i in range(2**mant_dig):
        mantissa = 1 + i/2**mant_dig
        floats.append(mantissa * 2**k)

plt.figure(1, figsize=(5,2))
plt.clf()

# Show where the exactly representable numbers in this floating point system are
for x in floats:
    plt.axvline(x)

# https://stackoverflow.com/questions/12998430/remove-xticks-in-a-matplotlib-plot
plt.tick_params(
    axis='y',          # changes apply to the y-axis
    which='both',      # both major and minor ticks are affected
    left='off',        # ticks along the left edge are off
    right='off',       # ticks along the right edge are off
    labelleft='off')   # labels along the left edge are off
```

Floating point numbers can be subtle in many ways:

- In the big picture, *floats are logarithmically spaced*.
  - This directly follows from the representation as $m \cdot 2^k$. For a fixed exponent k, the mantissa values m are spaced linearly. When k increases, so does the spacing.
  - The region of very small numbers around zero is spaced linearly, and consists of *subnormal numbers* (a.k.a. *denormal numbers*).
    - This is sometimes important for code performance. Even on modern PCs denormals may use a software implementation, which is very slow. Since these numbers are very, very small, it is sometimes a good strategy to truncate to zero once the result enters the subnormal range.
      - Cython and C programmers note: `fpclassify()`
    - https://en.wikipedia.org/wiki/Denormal_number
  - The *machine epsilon* ϵ is defined as the smallest floating point number such that $\mathrm{float}(1 + \epsilon) \neq \mathrm{float}(1)$. For IEEE-754 double precision, $\epsilon \approx 2.22 \cdot 10^{-16}$.
    - Roughly speaking, ϵ can be thought of as "the precision of floating point near unity", but naturally, this is not symmetric, since the exponent jumps at unity:

      ```
      import sys

      eps = sys.float_info.epsilon
      1 + eps    # 1.0000000000000002
      1 - eps/2  # 0.9999999999999999
      1 + eps/2  # 1.0 (not representable)
      ```

- Obviously, in principle floats are intended as a computer representation of $\mathbb{R}$, but any scheme using fixed-length storage to represent arbitrary numbers can only actually represent a finite subset of $\mathbb{Q}$.
  - For given size of storage, choosing this subset is inherently a tradeoff of range vs. precision.
- *Floats are base-2*. This implies e.g. that $1/10$ cannot be represented exactly:

  ```
  x = 0.1
  x.as_integer_ratio()  # (3602879701896397, 36028797018963968)

  0.1 + 0.1 + 0.1 == 0.3  # False

  sum(0.1 for j in range(100))  # 9.99999999999998

  0.1 + 0.2 == 0.3        # False
  0.1 + 0.3 == 0.4        # True
  ```

  This error source is called *representation error*. Sometimes representation errors may cancel, or end up equal after rounding.
- Comparing floats and obtaining the intended result can sometimes be difficult:

  ```
  a = 0.15 + 0.15
  b = 0.1  + 0.2
  a == b  # False
  a >= b  # False
  ```

  Adding an absolute or relative tolerance is not the full solution:

  http://floating-point-gui.de/errors/comparison/

  And in rare cases, it does make sense to compare floats for exact equality (e.g. in a termination condition for some iterative procedure that computes up to machine precision).

- *Cancellation* and *roundoff* introduce error to calculations using floating point numbers.
  - Even the summation of three or more operands is already problematic to perform accurately.
  - *Catastrophic cancellation* (a special case of *loss of significance*):
    https://en.wikipedia.org/wiki/Loss_of_significance
  - https://en.wikipedia.org/wiki/Round-off_error
- Algorithms exist to mitigate this.
  - Raymond Hettinger (2005): Binary floating point summation accurate to full precision [Python recipe]:
    https://github.com/ActiveState/code/tree/master/recipes/Python/393090_Binary_floating_point_summatiaccurate_full
  - An early, well-known algorithm for *compensated summation* is due to William Kahan (1965):
    https://en.wikipedia.org/wiki/Kahan_summation_algorithm
  - See also `math.fsum()` in the Python standard library.
  - Modern processors offer a fused multiply-add (FMA) that computes $ax + y$, rounding only once.
    http://stackoverflow.com/questions/28630864/how-is-fma-implemented
  - In the context of evaluation of polynomials, the well-known Horner's method is fast, but not accurate.
    https://en.wikipedia.org/wiki/Horner's_method
  - There exists a compensated version: Langlois, Louvet (2007): *How to Ensure a Faithful Polynomial Evaluation With the Compensated Horner Algorithm*:
    http://perso.ens-lyon.fr/nicolas.louvet/LaLo07.pdf
- Usually, applications of floating point numbers use either single precision (most non-scientific applications) or double precision (scientific applications), but recently interest has arisen in *mixed-precision algorithms*.
  - This is due to the rise of GPGPU, and the fact that cheap (gaming) GPUs nowadays come with teraflops of raw computing power, but only for single precision.
  - See e.g. the Ph.D. thesis by Dominik Göddeke (2010): *Fast and Accurate Finite-Element Multigrid Solvers for PDE Simulations on GPU Clusters*:
    http://hdl.handle.net/2003/27243
  - See also Andew Thall (2007): *Extended-Precision Floating-Point Numbers for GPU Computation*:
    http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.486.8236&rep=rep1&type=pdf
- David Goldberg: What Every Computer Scientist Should Know About Floating-Point Arithmetic (Computing Surveys, March 1991):
  http://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html
  - A recommended read, at least once. In my opinion, this title too should come with "(No Excuses!)". It's useful for any scientists writing nontrivial software that uses floating point numbers, not only for computer scientists. It's much more detailed than what is possible to fit here.
- Bruce M. Bush (1996): The Perils of Floating Point:
  http://www.lahey.com/float.htm
- http://sandbox.mc.edu/~bennet/cs110/textbook/module4_2.html
- https://docs.python.org/3/tutorial/floatingpoint.html
- https://en.wikipedia.org/wiki/Floating-point_arithmetic

Recommended book:

Press, Teukolsky, Vetterling, Flannery: *Numerical Recipes: The Art of Scientific Computing*, 3rd ed., Cambridge University Press, 2007.

This book is a delight in that does not assume exact arithmetic; algorithms making the best of finite-precision arithmetic are an important theme.

But anyone writing open-source code, beware of the code examples in the book; they are all rights reserved.

Let us conclude the course with this:

$$7/3 - 4/3 - 1 = \epsilon$$
$$4/3 - 1/3 - 1 = 0$$

...in IEEE-754 double precision floating point. Why? Observe that

```
7/3 = 1.0010101010101010101010101010101010101010101010101010101010... * 2**1    (exact)
   -> 1.0010101010101010101010101010101010101010101010101010101010011      * 2**1    (IEEE-754)

4/3 = 1.0101010101010101010101010101010101010101010101010101010101010... * 2**0    (exact)
   -> 1.0101010101010101010101010101010101010101010101010101010101010      * 2**0    (IEEE-754)

1/3 = 1.0101010101010101010101010101010101010101010101010101010101010... * 2**(-2) (exact)
   -> 1.0101010101010101010101010101010101010101010101010101010101010      * 2**(-2) (IEEE-754)
```

(IEEE-754 doubles have 52 bits after the decimal point.) Hence, matching exponents to be able to perform the subtraction, we have

```
7/3 - 4/3:

(* 2**0)

     10.0101010101010101010101010101010101010101010101010101010101010110
-     1.0101010101010101010101010101010101010101010101010101010101010101
=     1.0000000000000000000000000000000000000000000000000000000000000001
```

which is the definition of $1 + \epsilon$ (note the 1 in the least-significant bit), but

```
4/3 - 1/3:

(* 2**(-2))

    101.0101010101010101010101010101010101010101010101010101010101010100
-     1.0101010101010101010101010101010101010101010101010101010101010101
=    10.1111111111111111111111111111111111111111111111111111111111111111
->  100.0000000000000000000000000000000000000000000000000000000000000000
```

which is exactly 1.

(E.g. http://stackoverflow.com/questions/19141432/python-numpy-machine-epsilon)

Index