Vlaanderen
is supercomputing

Software Development for Researchers

High Performance Python

Engelbert Tijskens

Please, feel free to interrupt me at any time

There are no dumb questions, just dumb answers

Software development for researchers

- Is software development in research different from other software development?
- It is …
  - the goal is to solve a scientific problem which is stated in rather general terms
  - no clear specifications of what exactly to achieve and how to achieve it
  - many open questions
  - subject to progressive insight (shooting at a moving target)
  - flexibility needed
  - long development cycles are problematic
  - performance is usually an issue
  - researcher often not trained in software development
- you need a strategy that can cope with these issues

Software development for researchers

- Software development is a many-headed monster
  - language syntax
  - dependency management
  - working in different environments
    - local
    - cluster
  - ensuring correctness
  - documenting your code
  - version control
  - collaboration
  - sharing code with other researchers
  - documentation
  - performance issues
    - understanding hardware
    - parallelization

Software development for researchers

My answer:

- (not necessarily THE answer, but based on >30 years of scientific software development in a research context)
- is Python-centric
- uses tools to relieve you from the burden of administrative tasks and let you focus on the job - solving the science problem
  - et_micc (project management, binary extension modules)
  - poetry (dependency management, publishing, virtual environments)
  - git (version control, collaboration)
  - sphinx (documentation)
- without these tools I find myself looking up how to do administrative tasks far too often, and not doing them because of time constraints
- They have proven useful for small scale developments as well as for large projects for academia as well as for industry

why Python?

- Python is **easy to Read, Learn and Write**
  - Python is a **high-level programming language** that has English-like syntax.
    - This makes it easier to read and understand the code (reuse)
    - Python is really easy to **pick up** and **learn**, that is why a lot of people recommend Python to beginners (flat learning curve)
- Python is **a very productive language**
  - Due to the simplicity of Python, developers can focus on solving the problem
  - You don't need to spend too much time in understanding the **syntax** or **behaviour** of the programming language
  - a productive language reduces the development cycle

But then …

- I once wrote a Python method to compute the Verlet list of all atoms in a Molecular Dynamics application
- I found it annoyingly slow
- I replaced it with a C++ version
- It was 1200x faster (no typo!)

- Why would we be interested in Python, if it can be that slow?

Python vs C/C++/Fortran

- Interpreted language
+ command line, smallest executable unit is a line, immediate feedback
+ edit/run cycle
+ very easy to learn/develop
+ very terse and readable code
    + Python enforces indentation
+ script is flexible
+ dynamically typed
- overhead from interpreting
- very little runtime optimization done

- Compiled language
- smallest executable unit is (sub)program, feedback is later and for a larger unit
- edit/build/run cycle
- Learning curve is steeper, and longer (C++)
- more verbose code
    - free layout
- program is static, rigid (input parsing)
- statically typed
+ compiler minimizes the overhead
+ good optimization (automatic vectorization)

productiveness, flexibility ← **We want both!** → performance

What is Python good at?

https://docs.python.org/3/faq/general.html#what-is-python-good-for

- high-level general-purpose programming language
- large standard library The Python Standard Library
- wide variety of third-party extensions: the Python Package Index (PyPI)
  - Many packages with HPC in mind, built on top of HPC libraries
- functionality of standard library and extension packages is exposed easily as

  `import module_name`
- installing packages is easy
- `pip install module_name`
- high quality Python distributions (Intel, Anaconda), Windows/Linux/MACOS
- open source
- very well documented
- large community, used in most scientific domains
- …

What is Python good at?

- The use of **modules** is so practical and natural to Python that researchers do not so often feel the need to reinvent wheels
  - The number of novices that have written their own (inefficient) linear algebra routines in Fortran/C/C++ approaches infinity.
  - Fortran/C/C++ tutorials and books usually focus on syntax, not on using third party libraries. Using libraries in Fortran is a matter of the linker, not a language feature. Python is very different in that respect.
  - Python impregnates you with the idea that you need modules to get things done and by using them you usually get things done efficiently!

The wheel was invented ~8000 years ago. A lot of very clever people have put effort in it and It is **pretty perfect** by now.

Reinventing it will most probably not result in improvement.

What is Python good for?

In many ways Python gently pushes you in the right direction

Pleasant programming experience

“The principle of the least surprise”

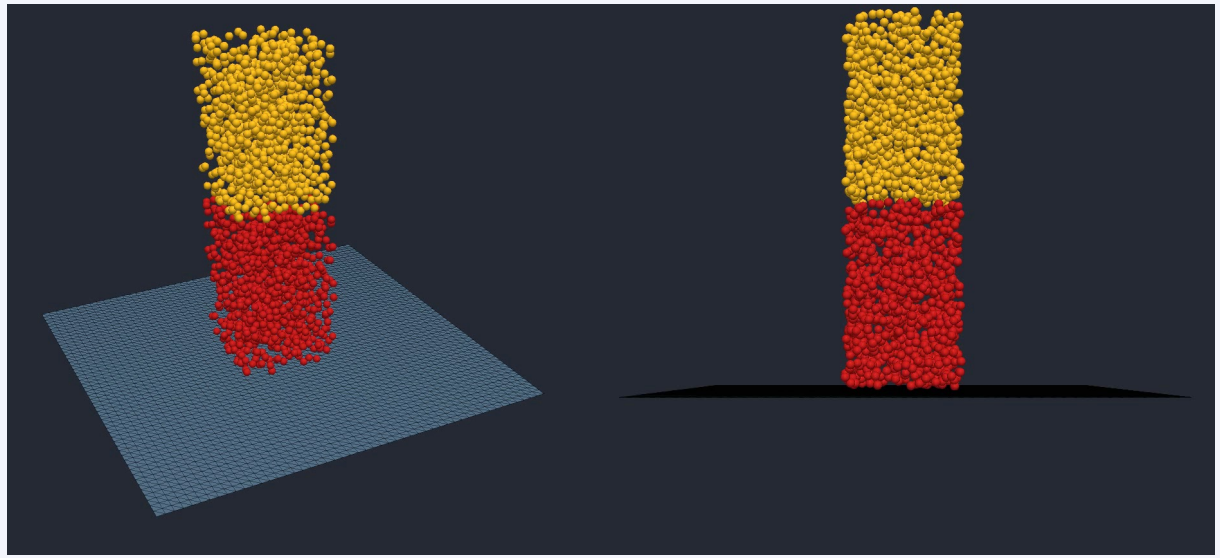
Interesting (if not indispensable) Python modules

NumPy	fast arrays / matrix operations (BLAS-like) / Fast Fourier Transform / mathematical functions defined on arrays / pseudo-random number generation to initialize arrays / simple statistics
SciPy	more mathematical functions / mathematical & physics constants / numerical integration / ordinary differential equations / optimization / interpolation / signal processing / dense and sparse linear algebra
Pandas	data science
Mpi4py	MPI message passing between Python processes
Dask	parallel computing in Python
matplotlib	2D and 3D graphics à la MATLAB
sympy	symbolic mathematics
scikit-image	image processing
h5py	hdf5 portable file format for (large) scientific datasets
…	

many of these modules build on each other

(their developers did not reinvent wheels)

MPacts



- granular dynamics code in C++
  - Grains (3D shape) instead of atoms
  - Force range relative to particle size is much shorter than in MD
  - Dissipative forces (friction)
- executable reads an input file
- adding new features became painful due to the complexity of input file parsing
- we wrapped the program's functionality in a Python module
- the input file became a Python script and the Python interpreter is the input parser
- adding features was no longer problematic
- flexibility and user friendliness x 10
- many codes today have a Python wrappers
  - for a good reason

The flexibility could even have been better, had the code been designed he other way around:

- start out with a high level Python interface and fill in the details in Fortran/C/C++
- In many cases the advantages of Python were discovered after the application program gained popularity

What is Python good at?

Python is extremely useful as a

- *glue* language
- *scripting* language, or
- *prototyping* language
- *programming* language

- your program becomes programmable
- input script vs input file
- immediate interface with all available Python packages
  - flexible pre- and post-processing
  - flexible composition of a solution strategy

Stil we are stuck on efficiency: as a *programming* language Python can be too slow

What are our options to improve performance?

Python performance

500 × 500 matrices

Python	0.09 s
C	0.014 s
Fortran	0.012 s

Python	32 s
C	0.49 s
Fortran	0.11 s

```
def init_matrix(n):
    # represent matrix as list of lists
    m = []
    for i in range(n):
        m.append([])
        for j in range(n):
            m[i].append(random.random())
    return m

def matmul(a, b, c):
    n = len(a)
    for i in range(n):
        for j in range(n):
            c[i][j] = 0.0
            for k in range(n):
                c[i][j] += a[i][k]*b[k][j]
```

$$C = A \cdot B$$

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

NumPy example

500 × 500 matrices

numpy: 0.011 s

numpy: 0.077 s

```
import numpy as np

def init_matrix(n):
    return np.random.uniform(0.0, 1.0, (n, n))

def matmul(a, b):
    return np.dot(a, b)
```

HPC libraries

Language/library	Python	C	Fortran	Python/numpy	Fortran/BLAS
Matmul execution time [s]	32	0.49	0.11	0.077	0.060

415 ×

You can create world class applications using libraries, without having to write a lot of optimized code

Using (good) modules in Python is option 1 to avoid performance issues

- VASP is written in Fortran
- most of the cpu_time is spent in HPC libraries
  - lots of linear algebra
  - MPI

Options to improve performance

1. Replace our slow Python code with calls to HPC python modules, e.g. numpy
2. Numba, translates Python code into C, just in time compilation
3. Cython, embed C code into Python code
4. Create your own Python modules from C++ or Fortran code

numba

numba.pydata.org

Numba translates Python functions to optimized machine code at runtime using the industry-standard LLVM compiler library. Numba-compiled numerical algorithms in Python **can** approach the speeds of C or Fortran

numba.pydata.org

- Annotate Python functions with decorators
- Code (at least partially) transformed to C
  - fully automatic and transparent
  - just-in-time compilation (JIT)
- For better performance, provide type information
- simplified threading
- Automatic vectorization (SIMD)
- Can generate code for GPGPUs
  - but you'd have to know some CUDA

Options to improve performance

1. Replace our slow Python code with calls to HPC python modules, e.g. numpy
2. Numba, translates Python code into C, just in time compilation
3. Cython, a source code compiler that translates Python code to equivalent C code

cython

cython.org

Cython is an **optimising static compiler** for Python

Cython gives you the combined power of Python and C to let you

- write Python code that calls back and forth from and to C or C++ code natively at any point.
- easily tune readable Python code into plain C performance by adding static type declarations, also in Python syntax
- use combined source code level debugging to find bugs in your Python, Cython and C code.
- Interact efficiently with large data sets, e.g. using multi-dimensional NumPy arrays
- quickly build your applications within the large, mature and widely used Cpython ecosystem.
- integrate natively with existing code and data from legacy, low-level or high-performance libraries and applications.

- Annotate Python code with type information
- Code (at least partially) transformed to C
  - requires `setup.py` file
- Shared library is build

Numba vs Cython

- see
  - http://jakevdp.github.io/blog/2012/08/24/numba-vs-cython/
  - https://jakevdp.github.io/blog/2013/06/15/numba-vs-cython-take-2/
- numba takes the lead in performance and is easier to use
- but black box, if it doesn't speed up your code, you are a blind man out in the dark …

the right question is perhaps not

- how can we improve Python's performance?

but

- how does numpy get so close to Fortran/BLAS performance?

- its submodules are written in Fortran (or C/C++) and compiled into shared libraries which expose their methods to Python

Options to improve performance

1. Replace our slow Python code with calls to HPC python modules, e.g. numpy
2. Numba, translates Python code into C, just in time compilation
3. Cython, a source code compiler that translates Python code to equivalent C code
4. Create your own Python modules from C++ or Fortran code

Build your own Python modules from Fortran/C/C++ code

- Python was designed to be extended by modules developed in Fortran/C/C++
- A low-level language like Fortran/C/C++ allows maximal code optimization
- Python can use a shared library as a module
- The language in which the shared library was written is in principle immaterial
- Several tools are available to build shared libraries that can be used as Python modules

Options to improve performance

1. replace our slow Python with calls to HPC python modules, e.g. numpy
2. Numba, translates Python code into C, just in time compilation
3. Cython, embed C code into Python code
4. **Create your own Python modules from C++ or Fortran code**

   This yields

   + full control over optimisation and parallelisation approaches
   + a scalable approach
   + allows to integrate third party libraries
   - debugging these Python modules is a bit harder

   (but here is a usefull link: https://www.researchgate.net/figure/Debugging-both-C-extensions-and-Python-code-with-gdb-and-pdb_fig2_220307949)

Fortran? C? C++?

The art of choosing a programming language (for research codes)

Here's a list of arguments I often hear…

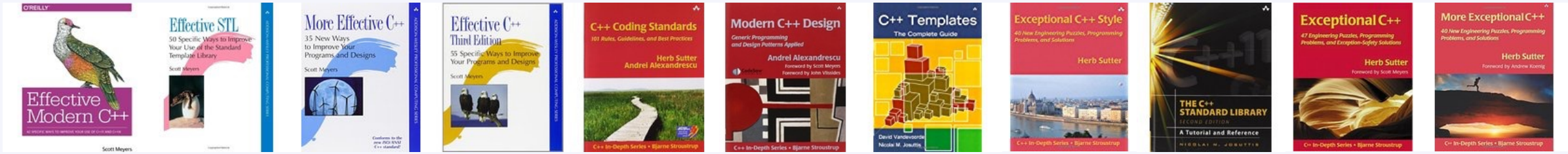
- ~~C++ is inefficient~~ Lie #1
  - Modern compilers good enough to generate efficient code
  - After all your are using the same hardware
- ~~Fortran is efficient~~ Lie #2
  - Also Fortran has constructs that sometimes come in handy, but can kill performance
- But C++ has quite a bit more features which can kill performance than Fortran.
  - Because C++ is a general purpose language and Fortran is meant for scientific computing
- Hence writing performant C++ is harder.
- Yet these features can be extremely useful if you use them wisely
  - Less critical for high level code features which carry out a lot of computation
  - For computational kernels where performance is an issue you generally need to stay close to the C subset and far away from the C++ features such as classes, inheritance, virtual functions, etc. (templates are an exception)

Fortran? C? C++?

The art of choosing a programming language (for research codes)

Here's a list of arguments I often hear…

- I'll use C++ because ~~I know it better~~ Lie #3
  - Unless you have read and understood all the C++ books by Scott Meyers, Herb Sutter, Andrei Alexandrescu, Nicolai Josuttis
  - In which case you probably also understand which C++ features can kill performance and when they should be used to your advantage
  - For number-crunching I find myself advancing faster using Fortran than using C++ (which I do know better!)
    - The only exception is when there is a need for special data structures which are not readily available in Fortran (Containers in C++ STL)

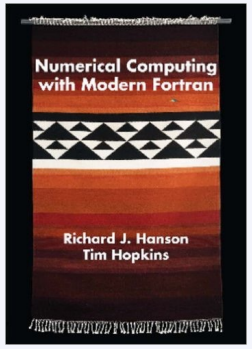


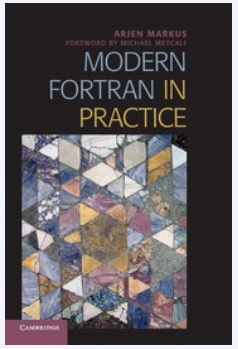
Fortran? C? C++?

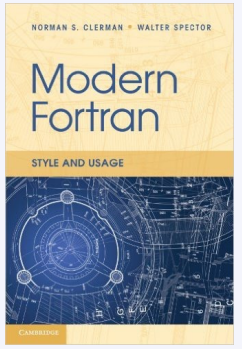
The art of choosing a programming language (for research codes)

Here's a list of arguments I often hear…

- I'll use C++ because it is better documented | Not a lie
  - There aren't many books on Fortran like the above ones on C++







  - very good material provided by Rheinold Bader https://doku.lrz.de/display/PUBLIC/Materials+-+Programming+with+Fortran?preview=/25559045/25559048/Fortran_3days.pdf
  - There is no website of the same quality as cplusplus.com or cppreference.com for Fortran (imho)
  - But still it is much harder to learn and to learn to use efficiently than Fortran
  - Not a valid argument

Fortran? C? C++?

The art of choosing a programming language (for research codes)

Here's a list of arguments I often hear…

- I'll use a language that interoperates nicely with Python
- Choice #1 : Fortran
  - f2py (= Fortran to Python) converts your F90 subprograms effortlessly into a Python module
  - f2py is part of NumPy and very well integrated with it
  - You can pass NumPy arrays directly to and from your F90 subprograms without copying!
    - That means you do memory management in Python – where it is easy (it is more cumbersome in Fortran)) – and computation in Fortran – where it is efficient.
  - This is by far the easiest option

Extremely good point!

Fortran? C? C++?

The art of choosing a programming language (for research codes)

Here's a list of arguments I often hear...

- I'll use a language that interoperates nicely with Python — Extremely good point!
- Choice #2 : C++
  - Achieve exactly the same with pybind11, interfaces C++ with Python, numpy and eigen
  - A little harder than Fortran, but much more powerful
  - no extra tool needed, just the compiler, and the above library
    - Pybind11 is a header-only library
  - Access to wide range of standard C++ data structures which are not readily available in Fortran
  - Automated building of Fortran (using f2py) and C++ (using pybind11) binary extensions in et-micc
  - Swig can also build Python modules from C++ code

Fortran? C? C++?

The art of choosing a programming language (for research codes)

Here's a list of arguments I often hear...

- I'll use a language that interoperates nicely with Python

Extremely good point!

- Choice #3 : C
  - Handcode Python to C interfaces (cumbersome)
  - Use swig (swig.org) (less cumbersome)
  - Take this choice only if you know C already and don't want to learn C++ or Fortran (which is a pity anyway)
  - (don't call me for helping you out...)

Conclusion for option 4 (writing your own modules in Fortran/C/C++):

use (Modern) Fortran with f2py and a good compiler suite (e.g. Intel)

unless

you are a seasoned C++ programmer and/or you need features from the C++ Standard Template Library / the Boost libraries or need to integrate third party C++ code

productiveness and performance

can be achieved with

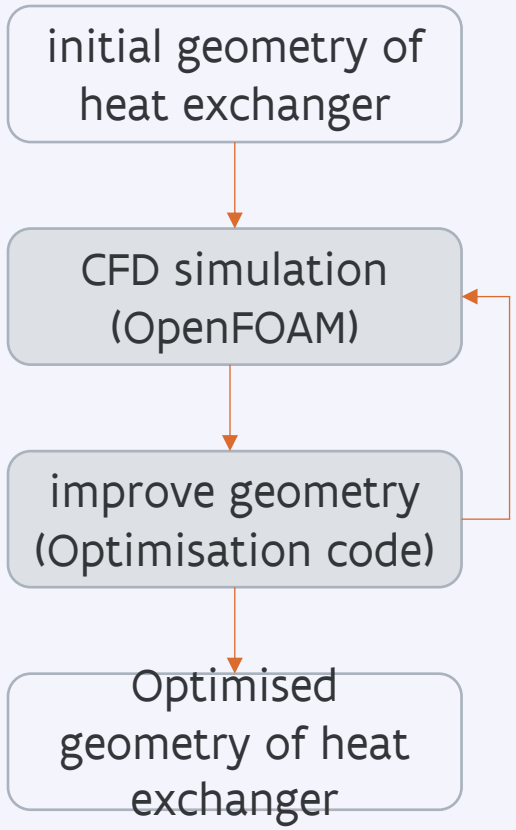
Python + C++/Fortran

That doesn't mean that it is easy …

Enters micc

- micc is a command line tool for Python project management that facilitates
  - constructing binary Python extension modules (Python modules written in C++ or Fortran)
  - constructing command line interfaces (like micc itself)
  - isolating sw development from system using virtual Python environments
  - test-driven development
  - version control
  - dependency management
  - publish your code to PyPI
  - documentation generation and publishing on readthedocs.org
  - working in different environments (local machine, VSC clusters)
  - …
- see https://micc.readthedocs.org

use case 1 - Diabatix



Originally the geometry was communicated between Open-FOAM and the Optimistion code via **files** causing **20-50%** of overhead.
This overhead was removed completely by modifying Open-FOAM so that the communication happens in **memory**.

transform OpenFOAM main() into C++ class and expose its members to Python

Use and manipulate OpenFOAM completely from Python -> tremendous flexibility

use case 2 VIPUN medical

- C++ executable to compute a measure of gastric motility from stomach pressure measurements
- The C++ main() function was converted into a class and exposed in Python
- This allows easy interfacing with
  - matplotlib for visual inspection of pressure profiles and motility index profiles
  - scipy.optimise for optimisation of algorithm parameters

use case 3: a bio-informatics PhD project

- evolutionary molecular design
- a python code and a c++ executable need to talk to each other iteratively
- initially a fragile communication mechanism was set up to make that happen.
- the C++ main function was turned in a class and exposed in a python module
- all C++ functionality can now be accessed from any python code in an intuitive way

A strategy for (research) code development that

(1) minimizes coding efforts

(2) allows for high performance

(3) provides flexible and reusable components

(research) code development strategy: Principle 1

- **Start out in Python**

- Easy and fast development
- readable code

(research) code development strategy: Principle 2

- **Start out simple**
  - as simple as possible
    - with a straightforward algorithm
    - no fancy data structures
      - stick to arrays if possible
      - SOA, no AOS
  - write as little code as possible by using existing (HPC) Python modules, e.g. NumPy, SciPy, … (use Python as glue)
    - formulate your problem in terms of mathematical domains for which Python modules exist, e,g. matrix algebra, linear algebra, …
  - certainly do not optimize/parallellize at this point
- in order to have a working code that yields correct answers as soon as possible
- this will serve for reference results to validate later improvements

(research) code development strategy: Principle 3

- **test and validate**
  - from the very beginning
  - all code is guilty until proven innocent!
    - if there is 1% chance to make an error on every change, the chance that your code is correct after 1000 changes is ~10^{-5}, which is the situation after about one week of programming!
  - write unit tests
    - Python unittest module pythontesting.net/framework/unittest/unittest-introduction/
    - nose, nose2
    - pytest
  - automate
    - rerun tests after every change, however small the change
    - integrate your tests in the build system

- a bug is always discovered too late
- the more changes you apply after before re-running your tests, the harder it becomes to locate the bug.

(research) code development strategy: Principle 4

- **[iff principles 1-3 are satisfied] improve**
  - add better algorithms
    - look for better computational complexity e.g. O(N)
    - without throwing away the reference solution, which is probably far too slow for production, but it is indispensable for validation and testing
  - still using Python
    - if anytime later you decide that for performance reasons you need to turn a Python method into a module method written in Fortran/C/C++, it will be easy to translate
    - do not throw away the Python variant
      - you need it as a reference solution (use it in your unit tests)

(research) code development strategy: Principle 5

- **[iff principles 1-4 are satisfied] profile and optimize**
  - locate performance bottlenecks
  - see what you can do with numba (or Cython).
  - verify performance relative to machine limits
    - apply the roofline model (easy with Intel Advisor)
  - study approaches for removing performance bottlenecks
    - common causes
      - vectorization prohibited
      - bad memory access pattern
  - if necessary replace the bottleneck with a Python module written in Fortran/C/C++
    - Performance programming in Fortran/C/C++ requires expertise
      - which we are happy to provide, especially if you follow this strategy
      - attend our performance programming courses

(research) code development strategy: Principle 6

- **[iff principles 1-5 are satisfied] parallellize** (if there is a need to do so)
  - when the execution time is too large
  - when one node does not provide enough memory or bandwidth
  - when your code has competitors which do parallellize
  - consider parallellization
    - mpi4py
    - dask
  - requires expertise
    (which we are happy to provide, especially if you follow this strategy)

(research) code development strategy: some missing ingredients

- versioning system
  - git
  - mercurial
- build system
  - which adjusts to your current environment
  - makefiles are rather versatile
- documentation
  - python has integrated help showing doc-strings
  - sphinx
  - smart editors can show your doc-string
- IDEs
  - eclipse with PyDev, also support for Fortran/C/C++,
  - liclipse
  - pycharm
  - Atom-2
- environment management

```
def my_fun(arg):
    """
    this is function does nothing.
    its argument arg is useless.
    """
    pass
```

```
> python
...
>>> import my_f90_tools
>>> help(my_fun)
my_fun(arg)
    this is function does nothing.
    its argument *arg* is useless.
>>>
```

- extra information (not just on Python)
  - https://github.com/gjbex/training-material
  - the Python info is in the Python directory