Vlaanderen
is supercomputing

Scientific Python

Engelbert Tijskens

Geert Jan Bex

VLAAMS
SUPERCOMPUTER
CENTRUM

*Innovative Computing
for A Smarter Flanders*

vscentrum.be

Please, feel free to interrupt me at any time

There are no dumb questions, just dumb answers

Why Python?

- In a research context
- In a HPC context

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

What is Python good for?

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 numpy
```

- High quality Python distributions (Intel, Anaconda), Windows/Linux/MACOS
- Open source
- Very well documented,
- Large community, used in most scientific domains
- …

Python vs C/C++/Fortran

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

- Compiled language
  - \- smallest executable unit is (sub)program, feedback is later and for a larger unit
  - \- fortran/C harder to learn, C++ hard
  - \- more verbose code
  - \- edit/build/run cycle
  - \- program is static, rigid (input parsing)
  - \+ compiler minimizes the overhead
  - \+ good optimization (automatic vectorization)

good user experience ← **We want both!** → efficient

What is Python good for?

- 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”

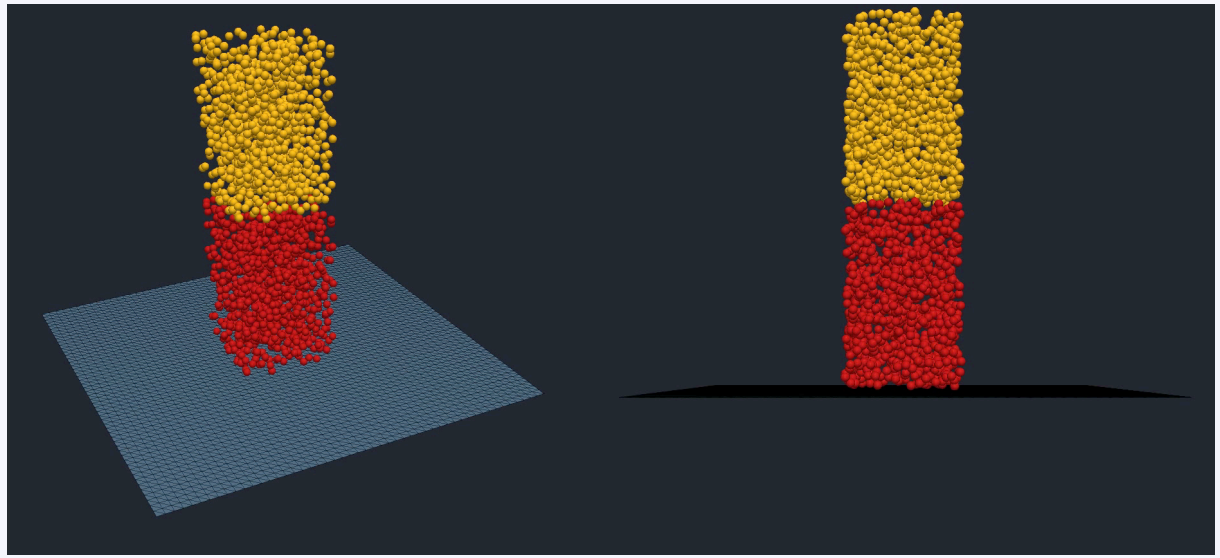
Interesting (if not indispensable) Python modules

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 inter face 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 for?

Python is extremely useful as a *glue* language

Scripting language, or *prototyping* language vs *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

Still we are stuck on efficiency: as a *programming* language Python is (far) 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

Profiling Python

Before we start to cure perfor–mance issues, we must locate them

- which sections of a code take most compute time?

Only optimize code that needs optimization

- all other optimizations are a waste of time and tax money
- 2x wasted since optimization typically makes code less readable and thus harder to maintain

- A profiling tool is an application that runs your code and gathers statistics about the fraction of cpu time that is spent in each part of your code
  - line based
  - function call based
- the parts that take a lot of cpu time are your first candidates for optimization

- profiling python
  - Intel Advisor (only Intel Python distribution)
  - cProfile module (function call based)
  - kernprof (line based)

cProfile

```
#file primes.py
from primes import primes
result = primes(1000)
```

```
$ python -m cProfile -s time primes.py

 2914 function calls (2878 primitive calls) in 0.261 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.250    0.250    0.251    0.251 primes.py:6(primes)
        1    0.002    0.002    0.002    0.002 {built-in method loads}
     1194    0.001    0.000    0.001    0.000 {'append' of 'list'}
       43    0.001    0.000    0.001    0.000 {'join' of 'str'}
```

nearly all time is spent in function primes (which lives in module primes

Line profiler kernprof

https://github.com/rkern/line_profiler

```
$ kernprof  -l  -v  primes_lprof.py 1000
Timer unit: 1e-06 s

Total time: 1.01724 s
File: /home/gjb/Documents/Projects/training-material/Python/Profiling/primes_lprof.py
Function: primes at line 4

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     4                                           @profile #decorate function to profile
     5                                           def primes(kmax):
     6         1            2      2.0      0.0      max_size = 1000000
     7         1        72903  72903.0      7.2      p = array('i', [0]*max_size)
     8         1            4      4.0      0.0      result = []
     9         1            2      2.0      0.0      if kmax > max_size:
    10                                                   kmax = max_size
    11         1            1      1.0      0.0      k = 0
    12         1            0      0.0      0.0      n = a2
```

profiling Python

cProfile and kernprof:

- relatively simple tools to expose performance bottlenecks
  - cProfile for larger Python code with function calls
  - kernprof for Python script without a lot of function calls

Intel Advisor

- more complicated
- much more detailed
  - roofline model
  - info on expected cause of bottlenecks
  - advice for curing

- First option to cure performance bottlenecks is replacing our code with calls to functions in HPC modules
- that is, obviously, not always possible

what are our options to write efficient Python functions?

timing microbenchmarks

a microbenchmark is a small piece of code for which you implement several versions and measure their execution time with the purpose of optimization

- ipython command line: use magic `%time` or `%timeit`

```
In [1]: from primes import primes
In [2]: %timeit result = primes(1000)
10 loops, best of 3: 172 ms per loop
```

timing result

- Command line: use `timeit` module

module to use

statements to execute, a string per line

```
$ python -m timeit 'from primes import primes' 'primes(1000)'
10 loops, best of 3: 174 msec per loop
```

second option: 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

Motivating example: timings

```
In [1]: import primes_p

In [2]: import primes_n

In [3]: %timeit primes_n.primes(1000)
5.56 ms ± 226 µs per loop (mean ± std. dev.
    of 7 runs, 1 loop each)

In [4]: %timeit primes_p.primes(1000)
301 ms ± 3.25 ms per loop (mean ± std. dev.
    of 7 runs, 1 loop each)
```

54 × faster

numba implementation is much faster!

- how much work to get there?
- how complicated is it?

Motivating example: code

```
import numpy as np

def primes(kmax):
    p = np.zeros(1000)
    result = []
    if kmax > 1000:
        kmax = 1000
    k = 0
    n = 2
    while k < kmax:
        i = 0
        while (i < k and
               n % p[i] != 0):
            i = i + 1
        if i == k:
            p[k] = n
            k = k + 1
            result.append(n)
        n = n + 1
    return result
```

primes_p.py

```
import numpy as np
from numba import jit

@jit #decorator
def primes(kmax):
    p = np.zeros(1000)
    result = []
    if kmax > 1000:
        kmax = 1000
    k = 0
    n = 2
    while k < kmax:
        i = 0
        while (i < k and
               n % p[i] != 0):
            i = i + 1
        if i == k:
            p[k] = n
            k = k + 1
            result.append(n)
        n = n + 1
    return result
```

primes_n.py

That was *easy!*

Does it always work?

```
import numpy as np
from numba import jit

@jit
def primes(kmax):
    p = np.zeros(1000)
    result = []
    if kmax > 1000:
        kmax = 1000
    k = 0
    n = 2
    while k < kmax:
        i = 0
        while (i < k and
               n % p[i] != 0):
            i = i + 1
        if i == k:
            p[k] = n
            k = k + 1
            result.append(n)
        n = n + 1
    return result
```

primes_n.py

Minor change

```
from array import array
from numba import jit

@jit
def primes(kmax):
    p = array('i', [0]*1000)
    result = []
    if kmax > 1000:
        kmax = 1000
    k = 0
    n = 2
    while k < kmax:
        i = 0
        while (i < k and
               n % p[i] != 0):
            i = i + 1
        if i == k:
            p[k] = n
            k = k + 1
            result.append(n)
        n = n + 1
    return result
```

primes_na.py

Does it always work: timings?

```
In [1]: import primes_pa

In [2]: import primes_na

In [3]: %timeit primes_na.primes(1000)
81.9 ms ± 2.18 ms per loop (mean ± std. dev.
    of 7 runs, 1 loop each)
In [4]: %timeit primes_pa.primes(1000)
99.3 ms ± 878 µs per loop (mean ± std. dev.
    of 7 runs, 10 loops each)
```

1.2 × faster

numba is just slightly faster,
there be dragons...

Eager JIT

912 × faster than Python

```
from numba import jit

@jit
def julia_set(domain, iterations, max_norm, max_iters):
    for i, z in enumerate(domain):
        while (iterations[i] <= max_iters and
               z.real*z.real + z.imag*z.imag <= max_norm*max_norm):
            z = z**2 - 0.622772 + 0.42193j
            iterations[i] += 1
```

julia_numba.py

2.4 ×

```
from numba import jit, void, int32, float64, complex128

@jit(void(complex128[:], int32[:], float64, int32))
def julia_set(domain, iterations, max_norm, max_iters):
    for i, z in enumerate(domain):
        while (iterations[i] <= max_iters and
               z.real*z.real + z.imag*z.imag <= max_norm*max_norm):
            z = z**2 - 0.622772 + 0.42193j
            iterations[i] += 1
```

Function signature specification

julia_numba_eager.py

2195 × faster than Python

Python to numba type mapping

Python type	numba type
`None`	`void`
`int`	`int8, uint8, int16, uint16, int32, uint32, uint64, int64`
`float`	`float32, float64`
`complex`	`complex64, complex128`
1D array	e.g., `float64[:]`
2D array	e.g., `float64[:,:]`

Note: no maximum `int` in Python 3, numba can overflow!

numpy ufunc

- numpy ufunc
  - element-wise on numpy arrays
  - supports reduction, accumulation, broadcasting
  - can be written in C/Cython
    - cumbersome
- numba
  - @vectorize: create ufunc
  - @guvectorize: create generalized ufunc

ufunc example

1680 × faster than Python

Return type

```
from numba import guvectorize, void, int32, float64, complex128

@guvectorize([void(complex128[:], float64[:], int32[:], int32[:])],
             '(n),(),()->(n)')
def julia_set(domain, max_norm, max_iters, iterations):
    for i, z in enumerate(domain):
        iterations[i] = 0
        while (iterations[i] <= max_iters[0] and
               z.real**2 + z.imag**2 <= max_norm[0]**2:
            z = z**2 - 0.622772 + 0.42193j
            iterations[i] += 1
```

julia_ufunc.py

Don't forget!

```
…
iterations = julia_set(domain, max_norm, max_iters)
…
```

2D arrays: automatic broadcasting

numba conclusions

- numba
  - Pros
    - Very simple to use
    - Offers excellent speedups when applicable
    - Easy to create numpy ufunc
  - Cons
    - Black box
    - Requires numba install
- Features not covered here:
  - Automatic parallelization: experimental
  - CUDA code generation: requires familiarity with CUDA

Third option 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 forht 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

Motivating example: timings

```
In [1]: import primes_vanilla as primes_p

In [2]: import primes_cython. as primes_c

In [3]: %timeit primes_c.primes(1000)
100 loops, best of 3: 4.89 ms per loop

In [4]: %timeit primes_p.primes(1000)
1 loops, best of 3: 356 ms per loop
```

72 × faster

Cython implementation is much faster!
but...
- how much work to get there?
- how complicated is it?

Motivating example: code

```
from array import array

def primes(kmax):
    p = array('i', [0]*1000)
    result = []
    if kmax > 1000:
        kmax = 1000
    k = 0
    n = 2
    while k < kmax:
        i = 0
        while (i < k and
               n % p[i] != 0):
            i = i + 1
        if i == k:
            p[k] = n
            k = k + 1
            result.append(n)
        n = n + 1
    return result
```

primes_p.py

```
def primes(int kmax):
    cdef int n, k, i
    cdef int p[1000]
    result = []
    if kmax > 1000:
        kmax = 1000
    k = 0
    n = 2
    while k < kmax:
        i = 0
        while (i < k and
               n % p[i] != 0):
            i = i + 1
        if i == k:
            p[k] = n
            k = k + 1
            result.append(n)
        n = n + 1
    return result
```

primes_c.pyx

Motivating example: setup.py, building & using

```
from distutils.core import setup
from Cython.Build import cythonize

setup(
    ext_modules=cythonize('primes_c.pyx')
)
```

setup.py

```
$ python setup.py build_ext --inplace
```

Fairly painless, don't forget to build though!

```
#!/usr/bin/env python

from primes_c import primes
import sys

results = primes(int(sys.argv[1]))
print(', '.join(map(str, results)))
```

Import like any other Python module

primes.py

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

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

Python was designed to be extended by modules developed in Fortran/C/C++

In principle a Python module is nothing but a shared library
(it can also be an ordinary Python source file)

several tools are available to build shared libraries that can be used as Python modules

A low-level language like Fortran/C/C++ allows maximal code optimization.

The language in which the shared library was written is in principle immaterial.

There are, however, practical differences.

Fortran? C? C++?

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

Here's a list of arguments I often hear…

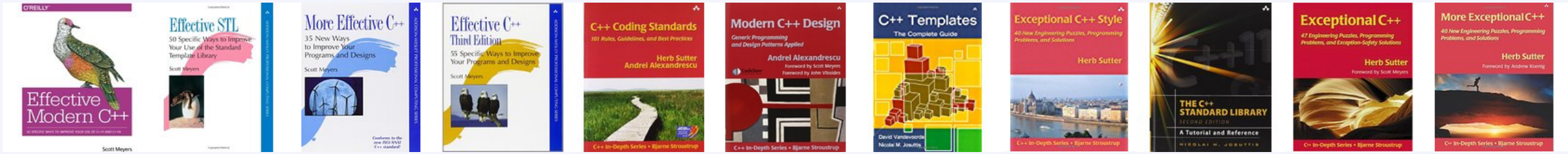
- ~~C++ is inefficient~~ Lie #1
  - Modern compilers good enough to generate efficient code
  - After all your are using the same hardware
- ~~Fortran is efficient~~ Lie #2
  - Also fortran has constructs that sometimes come in handy, but can kill performance
- But C++ has quite a bit more features which 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)

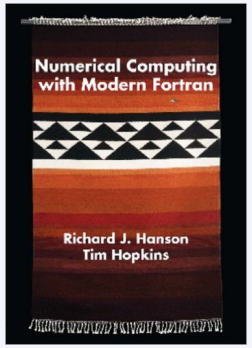


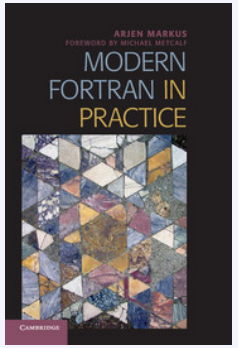
Fortran? C? C++?

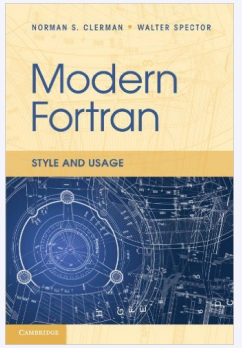
The art of choosing a programming language (for research codes)

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!

f2py example

```
! file my_sq.f90
  function my_sq(x)
    implicit none
  ! declare return value
    real*8 :: my_sq
  ! declare dummy arguments
    real*8 :: x
  ! function body
    my_sq = x*x
  end function
```

- there is a surprise here…
  - we squared an `integer` (10) and got back a `float` (100.0).
  - the Fortran function my_sq expects a double precision number and returns a double precision number
  - the Python wrapper of the Fortran function my_sq automatically converts the argument (if
  - This involves a copy operation and a conversion operation and may be costly, especially in the case of array arguments.
  - add “-DF2PY_REPORT_ON_ARRAY_COPY=1” to the f2py command to receive warnings when passing arguments involves copying
- Note that the return value of a function is always copied
- So, NEVER return arrays from functions
- How do we have to return arrays then?
  - use dummy arguments for modifying existing NumPy arrays

f2py surprise 2

```
! file my_sum.f90
   subroutine my_sum(the_sum,a)
   real, intent(out) :: the_sum
   real, dimension(:), intent(in) :: a
   ...
   the_sum = ...     !compute result
 end subroutine my_sum
```

f2py converts intent(out) to left hand side return value
(the wrapper behaves as a function)

```
> python
...
>>> from my_f90_tools import my_sum
>>> import numpy as np
>>> a = np.array([1,2,3])
>>> sum_a = 0.0
>>> my_sum(sum_a,a)
TypeError: my_f90_tools.my_sum() takes
at most 1 argument (2 given)
>>> sum_a = my_sum(a)
>>>
```

f2py surprise 2

```
! file my_sum.f90
   subroutine my_sum(the_sum,a)
   real, intent(inout) :: the_sum
   real, dimension(:), intent(in) :: a
   ...
   the_sum = ...     !compute result
 end subroutine my_sum
```

```
> python
...
>>> from my_f90_tools import my_sum
>>> import numpy as np
>>> a = np.array([1,2,3])
>>> sum_a = 0.0
>>> my_sum(sum_a,a)
>>>
>>> sum_a = my_sum(a)
TypeError: my_f90_tools.my_sum()
missing required argument 'a' (pos 2)
>>>
```

Specify intent(inout) to create true output arguments which do not involve copying

f2py surprise (3)

- Make sure your variables have the same precision in python and Fortran
  - if they don't they will get copied back and forth (which can make you waste a lot of cycles)
  - add -DF2PY_REPORT_ON_ARRAY_COPY=1 to f2py options to get a notice when arrays are copied

Python	Fortran
int, np.int64	integer*8
np.int32	integer*4
float, np.float64	real*8
np.float32	real*4

- Arithmetic in Fortran with *4 is faster than *8 (typically 2x)
- **But arithmetic in Python with np.int32 or np.float32 is slower!**

typical f2py script

```
#!/bin/bash
source=my_sq.f90
module=my_f90_tools
F90=`which gfortran`

rm -f ${module}.cpython-*

/Users/etijskens/miniconda3/envs/python36/bin/f2py -c \
    --build-dir f2py_build \
    --opt="-O3 -fopt-info-all" --arch="-mavx" \
    -DF2PY_REPORT_ON_ARRAY_COPY=1 \
    --f90exec=${F90} \
    ${source} -m ${module}
ls –l ${module}*so
```

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 or the Boost libraries

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