Praxisgrundlagen der Informatik

Performance: Cython – C-Extensions for Python

Prof. Dr. Eduard Kromer
University of Applied Sciences Landshut

Cython: C-Extensions for Python

Optimize what needs optimizing

From python.org:

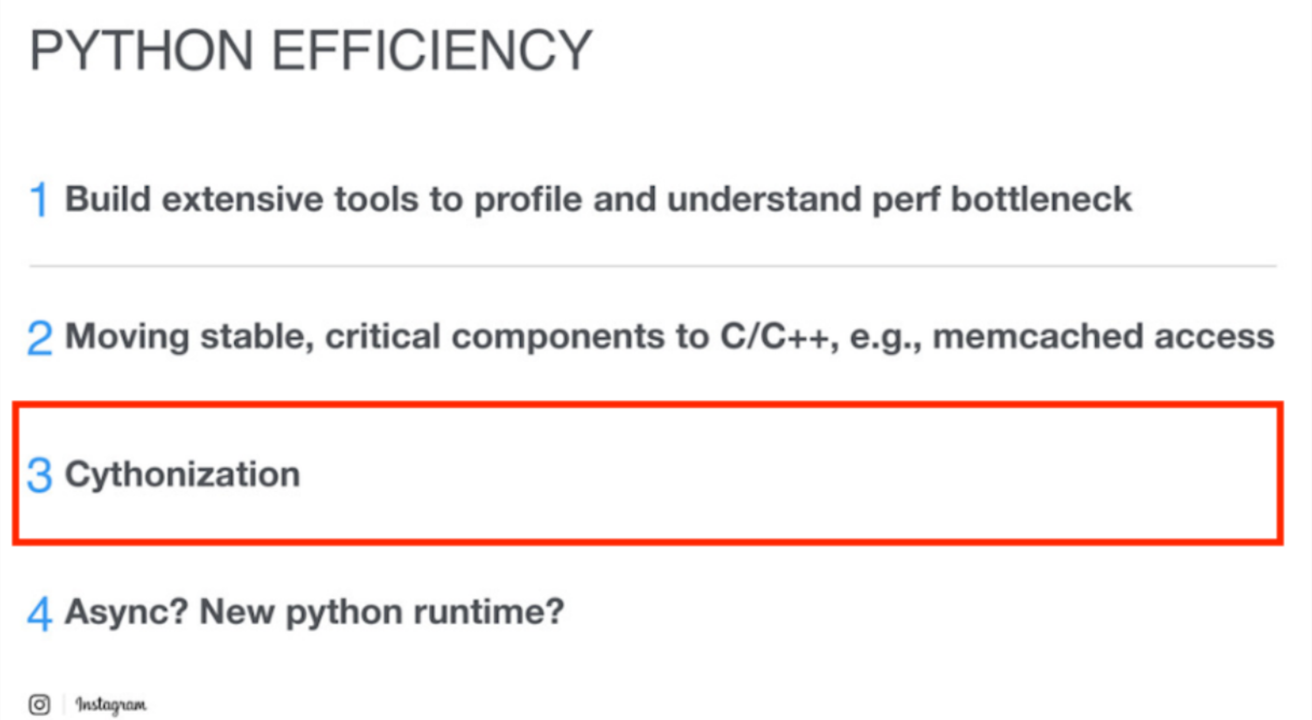
1. Get it right.
2. Test it's right.
3. Profile if slow.
4. Optimize.
5. Repeat from 2.

Prerequisites

- to understand all the technical details in this lecture you are required to have worked through the C tutorial at https://www.w3schools.com/c/ (including C functions and C structures)
- by using the corresponding web-based editor from w3schools.com you don't even have to install anything on your machine to experiment with C-code examples
- the Cython documentation provides you with all the necessary information to get started with Cython

Python efficiency at Instagram

From a PyCon 2017 talk:



Installing Cython

For the most recent Cython-version on conda-forge:

```
conda install cython -c conda-forge
```

For Jupyter notebooks:

- first, you need to load the Cython extension:

```
%load_ext cython
```

- to compile and load the Cython code inside a code cell you need:

```
%%cython  #cell-magic
```

What is Cython?

Cython is two closely related things:

- *Cython* is a programming language that blends Python with the static type system of C and C++
- *cython* is a compiler that translates Cython source code into efficient C or C++ source code. This source code can then be compiled into a Python extension or module or a standalone executable.

Why use Cython?

- Python is a high-level, dynamic, flexible and easy to learn language
  - but it can be orders of magnitude slower than *statically typed compiled* languages
- C is very low level and very powerful, but it can be very difficult to learn and use and it does not have many safeguards in place
- Cython: it combines Python's expressiveness and dynamism with C's bare-metal performance
  - it still feels (mostly) like Python
  - many Python libraries are performant because they are partially implemented in lower level languages

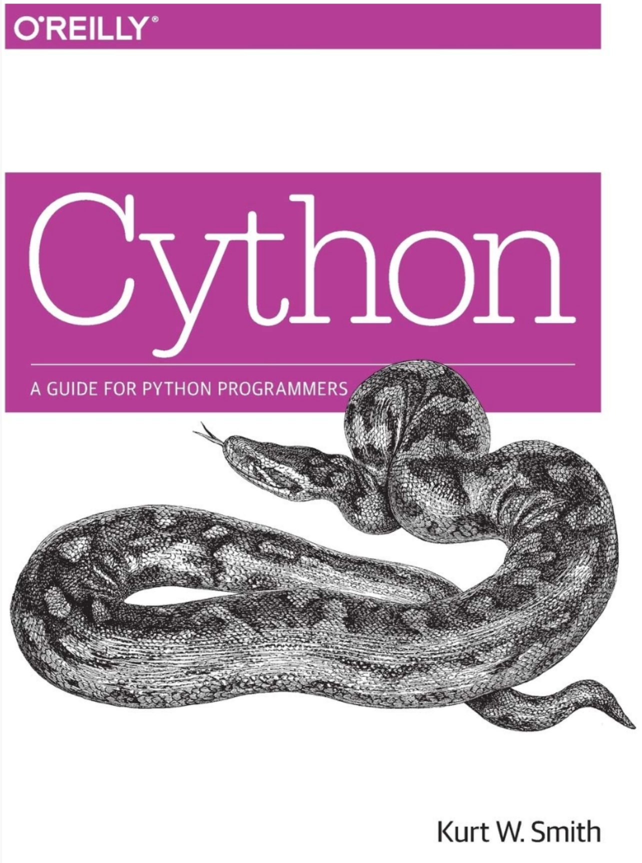
Why use Cython?

Dr. Stefan Behnel, Cython core dev since 2007:

> Write C without having to write C

Why use Cython?

- Cython can wrap existing C, C++, and Fortran libraries efficiently and easily (easily accessible to Python via NumPy arrays)
- Memory- and CPU-bound Python computations perform much better when translated into a statically typed language
- when dealing with large data sets, having control over the precise data types and data structures at a low level can yield efficient storage and improved performance

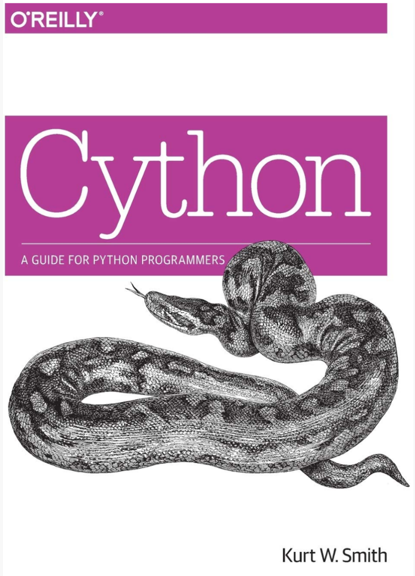


Dynamic vs. Static Typing

- an important difference between high-level languages like Python or JavaScript and low-level languages like C or C++ is that the former are *dynamically typed* and the latter are *statically typed*
- statically typed languages require the type of a variable to be fixed at *compile time*
  - advantages: compilers use static typing to generate fast machine code that is tailored to that specific type
- dynamically typed languages place no restrictions on a variables type
  - advantages: dynamically typed languages are typically easier to write

Python runtime evaluation

> What happens when the Python runtime evaluates `a + b`:
>
> 1. interpreter inspects the Python object referred by `a` for its type
> 2. interpreter asks the type for an implementation of the addition method
> 3. if the method in question is found, the interpreter then has an actual function it can call, implemented either in Python or in C
> 4. addition function extracts the necessary internal data from `a` and `b`; if successful, only then it can perform the actual operation that adds `a` and `b` together
> 5. the result must be placed inside a (perhaps new) Python object and returned; only then the operation is complete



What happens with C?

- the C compiler can determine at compile time what low-level operations to perform and what low-level data to pass to arguments
- at runtime, a compiled C program skips nearly all steps that the Python interpreter must perform
- for the operation `a+b` with `a` and `b` both being fundamental numerical types, the compiler generates a handful of machine code instructions to load the data into registers, add them, and store the result

Adding static types

- in Python a variable can be associated to objects of different types during the execution of the program (Python is flexible and dynamic)
- Cython extends the Python language with explicit type declarations
  - this way efficient C-extensions can be generated

Variables

- in Cython the type of the variable is declared by prepending it with `cdef` and its respective type
- variable `i` declared as 16-bit-integer

```
cdef int i
```

- multiple variable names with *optional initialization*

```
cdef double x, y = 3.14, z = 2.0
```

Dynamic typing?

```
%cython
cdef int i
i = 3.14



    cdef int i
         ^
SyntaxError: invalid syntax
```

Mixing statically and dynamically typed variables

- Cython **allows** assignments between statically and dynamically typed variables
- this is really powerful: it allows us to use dynamic Python objects for the majority of our code base, and convert them into fast, statically typed versions for the performance-critical sections

```
%%cython

cdef int a, b, c
# insert calculations involving a, b and c
tuple_of_ints = (a, b, c)
```

Casting

- certain data types are compatible – they can be converted into each other – this is called casting
- in Cython it is possible to cast between types by surrounding the destination type between `<>` brackets

```
%%cython

cdef int i = 3
cdef double d
d = <double> i
```

Functions

- you can add type information to the arguments of a Python function
- functions with those specifications will work like regular Python functions (but their arguments will be type-checked)

```
%%cython

def max(int a, int b):
    return a if a > b else b
```

- the `max` function will treat the arguments as typed variables, just like in `cdef` definitions
- but it is still a Python function incurring the respective call overhead
- `def` is used for code that will be called directly from Python code

Functions - Function call optimizations

- for function call optimizations: declare the return type of the function using a `cdef` statement

```
cdef int max_cy(int a, int b):
    return a if a > b else b
```

- functions declared this way are translated to native C functions (less overhead compared to Python functions)
- but there are drawbacks:
  - they can't be used from Python, but only from Cython (or C)
  - they are restricted to the same Cython file (unless they are exposed in a definition file)

Functions - Callable from Python and Cython

- Cython allows you to define functions that are both callable from Python and Cython
- you need to declare a function with the cpdef statement to achieve that
- then Cython will generate two versions of the function:
  - a Python version available to the interpreter
  - a (fast) C function usable from Cython

```
%%cython

cpdef int max_cy(int a, int b):
    return a if a > b else b
```

Functions - Inlining

- sometimes, the call overhead can be a performance issue even with C functions
  - e.g. function is called very often in a loop
- when the function body is small, it is convenient to add the inline keyword in front of the function definition
- the function call will then be replaced by the function body itself

```
%%cython

cdef inline int max_cy_inl(int a, int b):
    return a if a > b else b
```

Example - Fibonacci numbers

```python
# Python version
def fib(n):
    a, b = 0.0, 1.0
    for i in range(n):
        a, b = a + b, a
    return a
```

```cython
%%cython
# Cython version with static types
def fib_c(int n):
    cdef int i
    cdef double a = 0.0, b = 1.0
    for i in range(n):
        a, b = a + b, a
    return a
```

Peformance?

```
%timeit fib(500)
```

```
8.63 µs ± 77.6 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops
each)
```

```
%timeit fib_c(500)
```

```
699 ns ± 0.737 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops
each)
```

Cython Code Annotations

```
%%cython -a

def fib_c(int n):
    cdef int i
    cdef double a = 0.0, b = 1.0
    for i in range(n):
        a, b = a + b, a
    return a
```

Generated by Cython 3.0.8

Yellow lines hint at Python interaction.
Click on a line that starts with a "+" to see the C code that Cython generated for it.

```
 1:
+2: def fib_c(int n):
 3:     cdef int i
+4:     cdef double a = 0.0, b = 1.0
+5:     for i in range(n):
+6:         a, b = a + b, a
+7:     return a
```

Classes, structs and extension types

- extension types can be defined in *Cython* using the `cdef class` statement
- the corresponding attributes need to be declared in the class body

```
%%cython

cdef class Point:
    cdef double x
    cdef double y

    def __init__(self, double x, double y):
        self.x = x
        self.y = y
```

Using extension types

If you want to use the `cdef class` extension type in your code, you need to declare the type of the corresponding variable first, but you can simply use the extension type name:

```
cdef class Point:
    cdef double x
    cdef double y

    def __init__(self, double x, double y):
        self.x = x
        self.y = y

cdef double l2_norm(Point p):
    return (p.x**2 + p.y**2)**0.5
```

Accessing extension types from Python

- there are limitations in accessing extension type attributes from Python
- in order to access attributes from Python code, you need to use the `public` (read/write access) or `readonly` specifier in the attribute declaration
- without those specifiers trying to access the attributes will lead to an `AttributeError`

```
a = Point(1.5, 3.6)
a.x
> AttributeError: 'Point' object has no attribute 'x'
```

```
%%cython

cdef class Point:
    cdef public double x
```

Declaration sharing

- we want to organize our most used functions and classes in Cython in separate files to be reused in different modules
- in Cython those files are called definition files, they have a `.pxd` extension and we can `cimport` from those files
- such a file only contains the types and function prototypes we want to share with other modules

Definition files

- consider a file `fib.pxd` with (no function body):

```
cdef double fib_c(int n)
```

- and a file `fib.pyx` with the implementation

```
cdef double fib_c(int n):
    cdef int i
    cdef double a = 0.0, b = 1.0
    for i in range(n):
        a, b = a + b, a
    return a
```

- the `fib` module is now importable from another Cython module with the `cimport` statment

```
from fib cimport fib_c
```

Sharing extension types

- an extension type declaration can be split into two parts, one in a definition file and the other in the corresponding implementation file
- the definition part declares only C attributes and C methods, not pure Python methods
    - all of that type's C attributes and C methods must be declared
- the implementation part must implement all of the C methods declared in the definition part
    - adding any further C attributes not allowed
    - adding / defining Python methods allowed

Sharing extension types

points.pxd

```
cdef class Point:
    cdef double x
    cdef double y
```

points.pyx

```
cdef class Point:

    def __init__(self, double x, double y):
        self.x = x
        self.y = y

def l2_norm(Point p):
    return (p.x**2 + p.y**2)**0.5
```

- when the `points` module is compiled the two declarations from `points.pxd` and `points.pyx` are combinded into one
- in a module with `cimport points` we can refer to the `Point` type as `points.Point`

Cython, NumPy and working with arrays

Cython, NumPy and working with arrays

- you can access and work with C arrays in Cython, but for improved safety you should use NumPy arrays and typed memoryviews
- Cython allows us to bypass overhead created by the Python interpreter and act directly on the underlying memory area used by NumPy arrays
- NumPy arrays can be declared as the `ndarray` data type and used after the `cimport` of the numpy Cython module

```
cimport numpy as c_np

cdef c_np.ndarray[double, ndim=3] arr
```

Working with the `ndarray` data type

```
%%cython

import numpy as np

def numpy_bench_py():
    py_arr = np.random.rand(1000)
    cdef int i
    for i in range(1000):
        py_arr[i] += 1
```

```
%%cython

import numpy as np
cimport numpy as c_np

def numpy_bench_c():
    cdef c_np.ndarray[double, ndim=1] c_arr
    c_arr = np.random.rand(1000)
    cdef int i
    for i in range(1000):
        c_arr[i] += 1
```

Performance?

```
%%timeit

numpy_bench_py()
```

```
175 µs ± 1.94 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

```
%%timeit

numpy_bench_c()
```

```
6.12 µs ± 13.8 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops
each)
```

Typed memoryviews

- Cython provides a typed memoryview interface to all types of array data types
- a memoryview is an object that maintains a reference on a specific memory area
  - it can read and change the contents of the memory area
  - it is a view on the underlying data
- a memoryview works similarly to slicing a NumPy array (lecture on NumPy)
  - a slice does not copy the data but returns a view on the same memory area

```
cdef int[:] a
cdef double[:, :] d # two dimensional memoryview of double
```

Binding memoryviews to NumPy arrays

```
%%cython

import numpy as np

cdef double[:] a
ones = np.ones(8, dtype='float64')
a = ones # binding the array 'ones' to the memoryview a
a[3] = 0.0
print(ones)
```

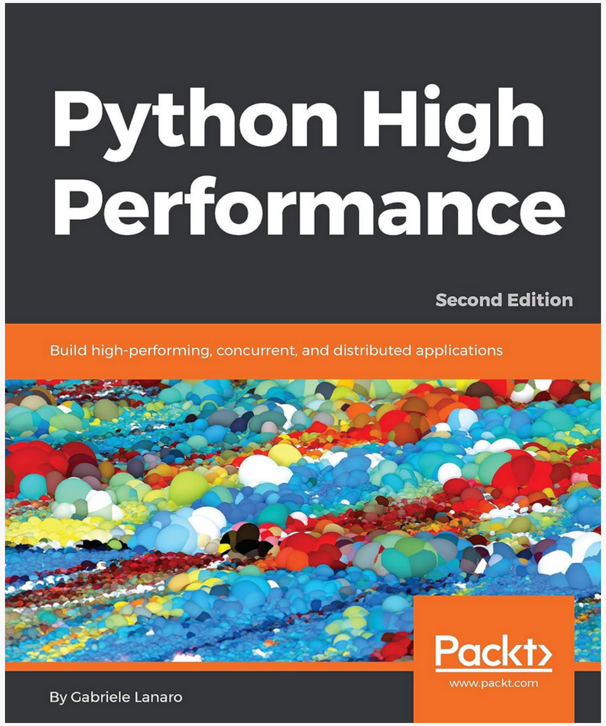
```
[1. 1. 1. 0. 1. 1. 1. 1.]
```

Slicing memoryviews

```
%%cython
import numpy as np

cdef double[:, :, :] a
a = np.ones(shape=(2,2,2))
a[0, :, :] # is a 2-dimensional memoryview
a[0, 0, :] # is a 1-dimensional memoryview
a[0, 0, 0] # is a double
```

Data copies between memoryviews

```
%%cython

import numpy as np
cdef double[:, :] b
cdef double[:] r
b = np.random.rand(10, 3)
r = np.zeros(3, dtype='float64')

b[0, :] = r # copy the value of r in the first row of b
```



Using Cython without Jupyter Notebooks

Using Cython without Jupyter notebooks

- Cython code can be compiled and run in Jupyter notebooks (from an IPython interpreter)
- it can be compiled automatically at import time
- it can be separately compiled by build tools like Python's `distutils`
- it can be integrated into standard build tools like `make`, `CMake` or `SCons`

Using `distutils` with cythonize

- the standard library includes the `distutils` package for building, packaging and distributing Python projects
  - we can use it to compile C source into an extension module
- it manages all the platform, architecture and Python-version details for us
- consider a `fib.pyx` file with

fib.pyx

```
def fib_c(int n):
    cdef int i
    cdef double a = 0.0, b = 1.0
    for i in range(n):
        a, b = a + b, a
    return a
```

- we want to generate a `fib.so` for macOS and Linux or a `fib.pyd` for Windows

Using `distutils` with cythonize

- we use a Python script `setup.py` to control the behavior of `distutils`
- it will trigger the compilation of the `fib.pyx` source file into an extension module

setup.py

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

setup(
    name='Fibonacci computation',
    ext_modules=cythonize('fib.pyx', compiler_directives={'language_level': 3}),
)
```

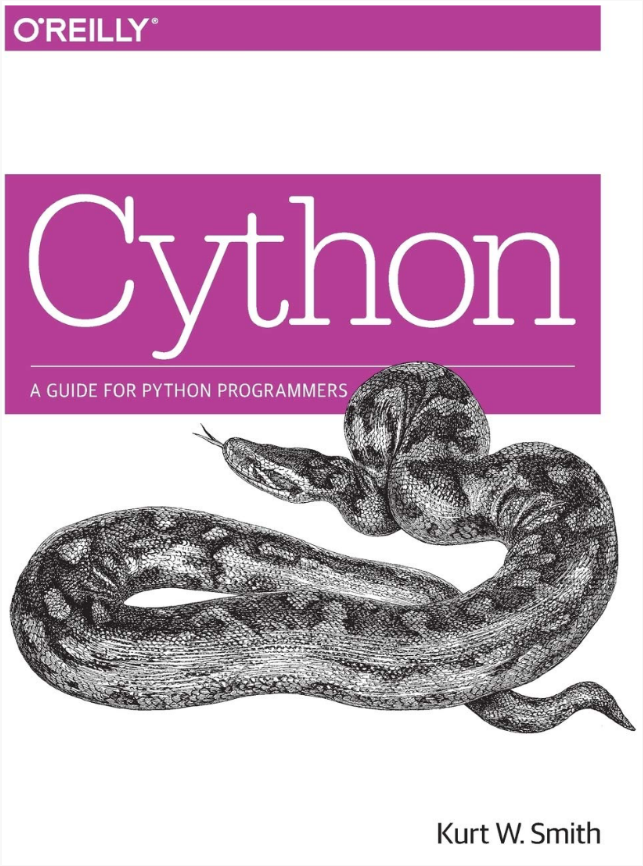
Compiling on-the-fly with pyximport

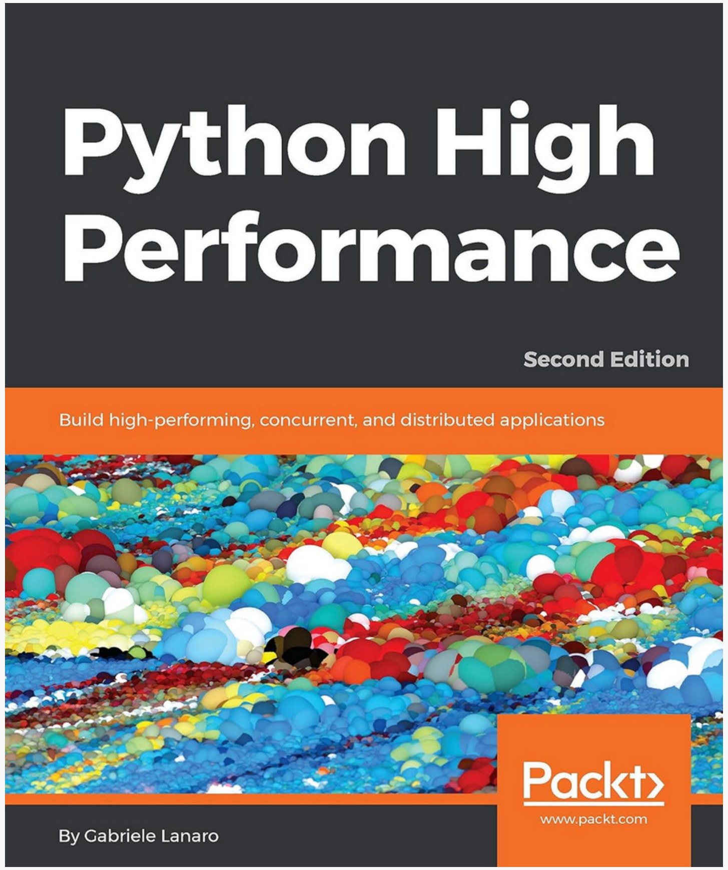
- `pyximport` retrofits the import statement to recognize `.pyx` extension modules
  - it sends them through the compilation pipeline and then imports them for use in Python
- in your module or an interactive session use

```
import pyximport
import os

pyximport.install()

import fib

print(f'Generated file: {os.path.basename(fib.__file__)}')
print(f'Result of computation: {fib.fib_c(25)}')
```

```
Generated file: fib.cpython-311-x86_64-linux-gnu.so
Result of computation: 75025.0
```

Literature


O'REILLY
Cython
A GUIDE FOR PYTHON PROGRAMMERS
Kurt W. Smith


Python High Performance
Second Edition
Build high-performing, concurrent, and distributed applications
Packt>
www.packt.com
By Gabriele Lanaro

Additional References

- D.S. Seljebotn, SciPy2009 - Fast numerical computations with Cython
- PyVideo Cython Talks/Tutorials