UNDERSTANDING NUMBA
THE PYTHON AND NUMPY COMPILER

Christoph Deil & EuroPython 2019

Slides at https://christophdeil.com

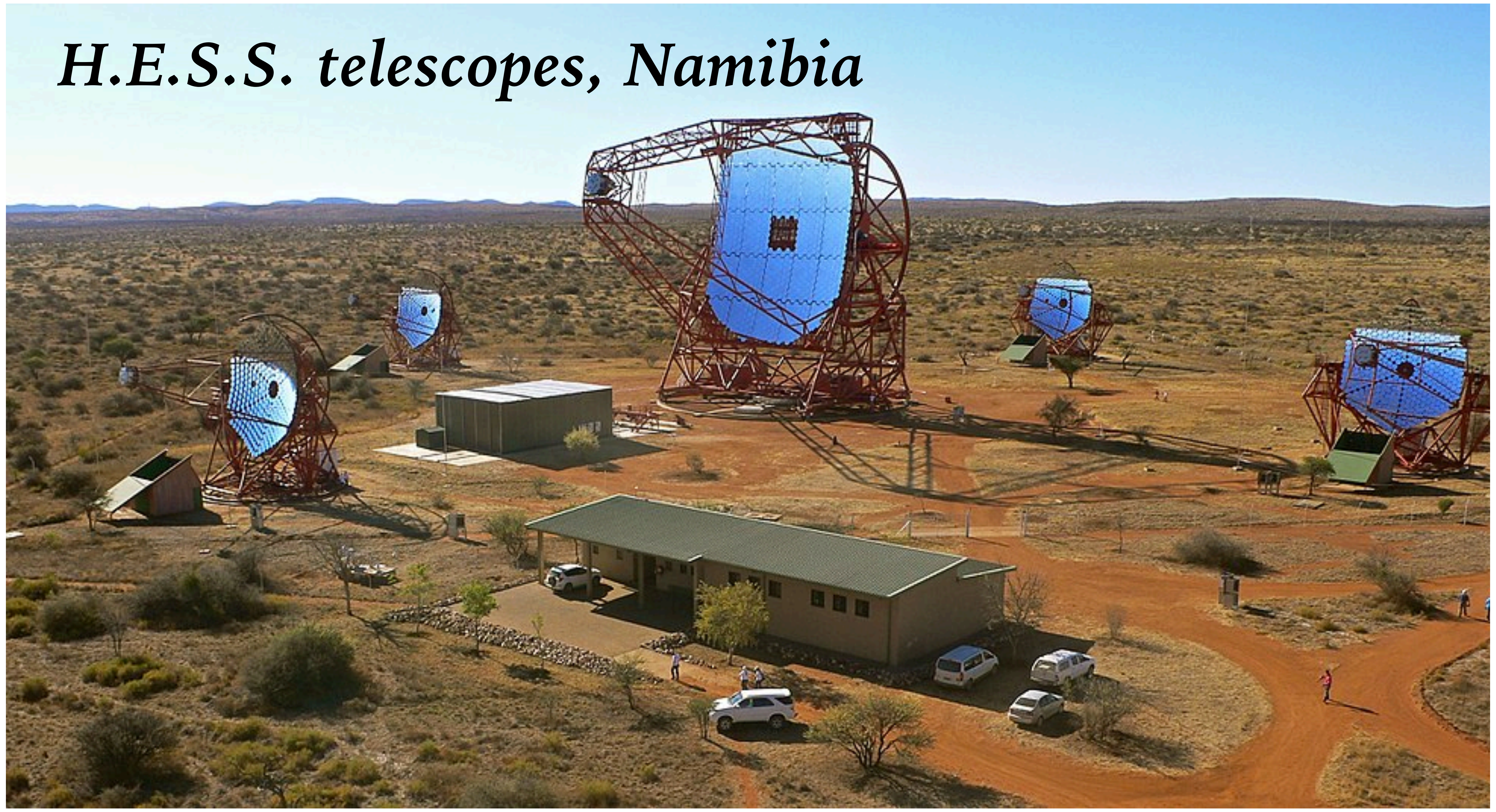
DISCLAIMER: I DON'T UNDERSTAND NUMBA!

ABOUT ME

- Christoph Deil, Gamma-ray astronomer from Heidelberg
- Not a Numba, compiler, CPU expert
- Recently started to use Numba, think it's awesome.
  This is an introduction.

MAX-PLANCK-INSTITUT
FÜR KERNPHYSIK

WHY USE NUMBA?



GAMMA-RAY ASTRONOMY

- Lots of numerical computing: data calibration, reduction, analysis
- Need both interactive data and method exploration and production pipelines.
- Software often written by astronomers, not professional programmers

TWO APPROACHES TO WRITE SCIENTIFIC OR NUMERIC SOFTWARE

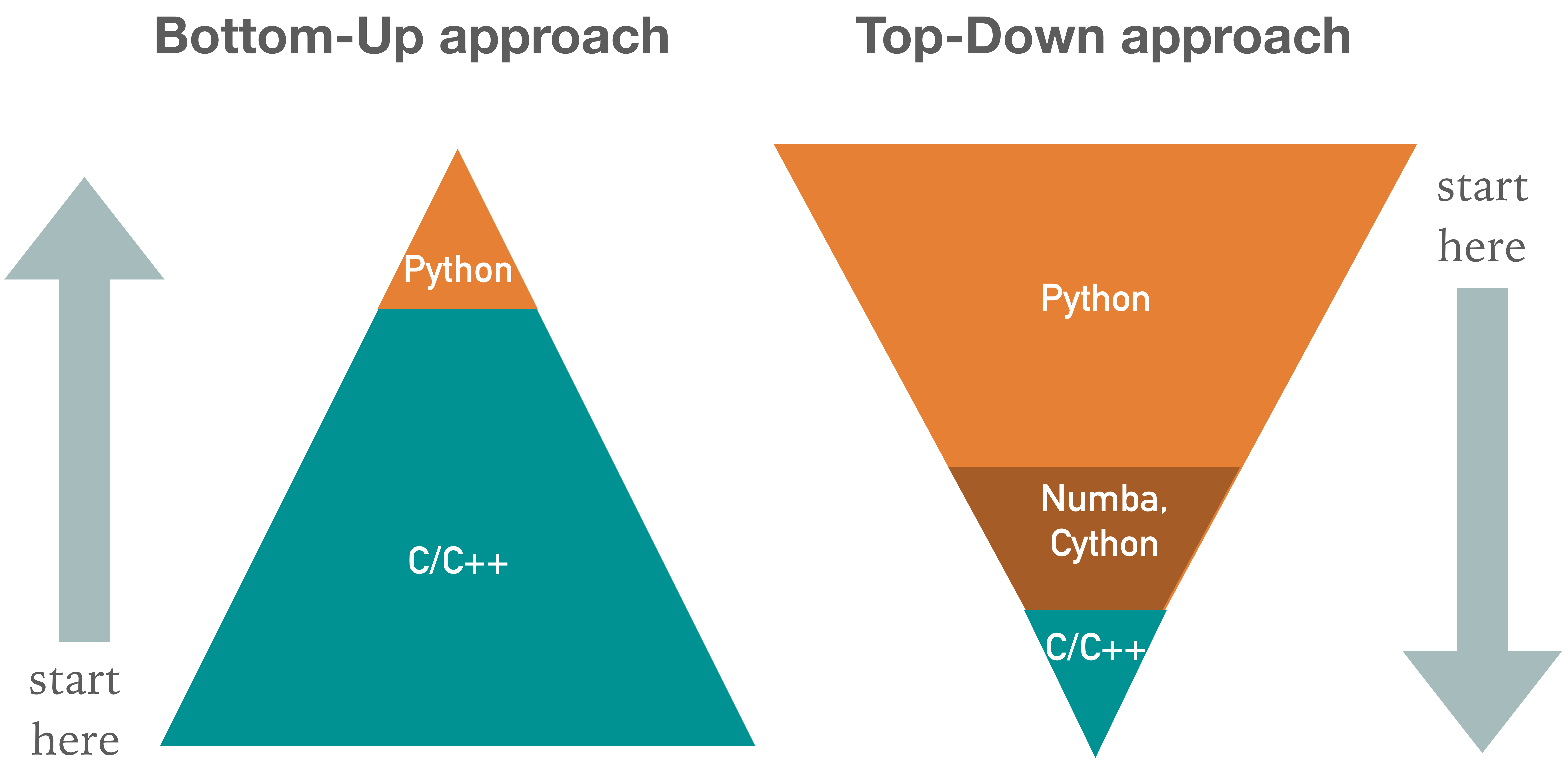


Image credit: Karl Kosack

CTA SOFTWARE

- Prototyping the Python first approach
- Use Python/Numpy/PyData/Astropy
- Use Numba/Cython/C/C++ for few % of performance-critical functions

A **Python** package for **gamma-ray** astronomy

ctapipe

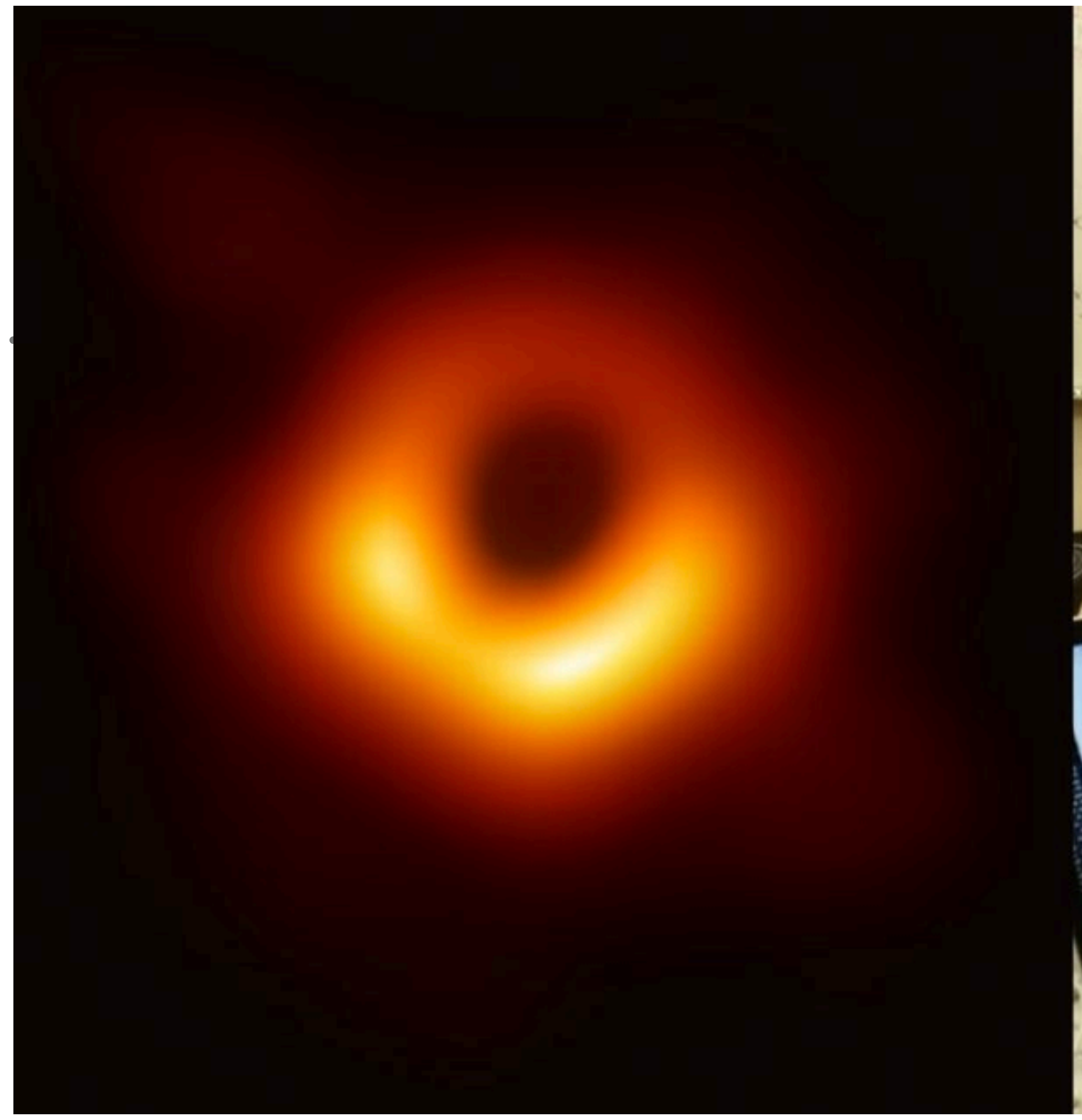
The Astropy Project

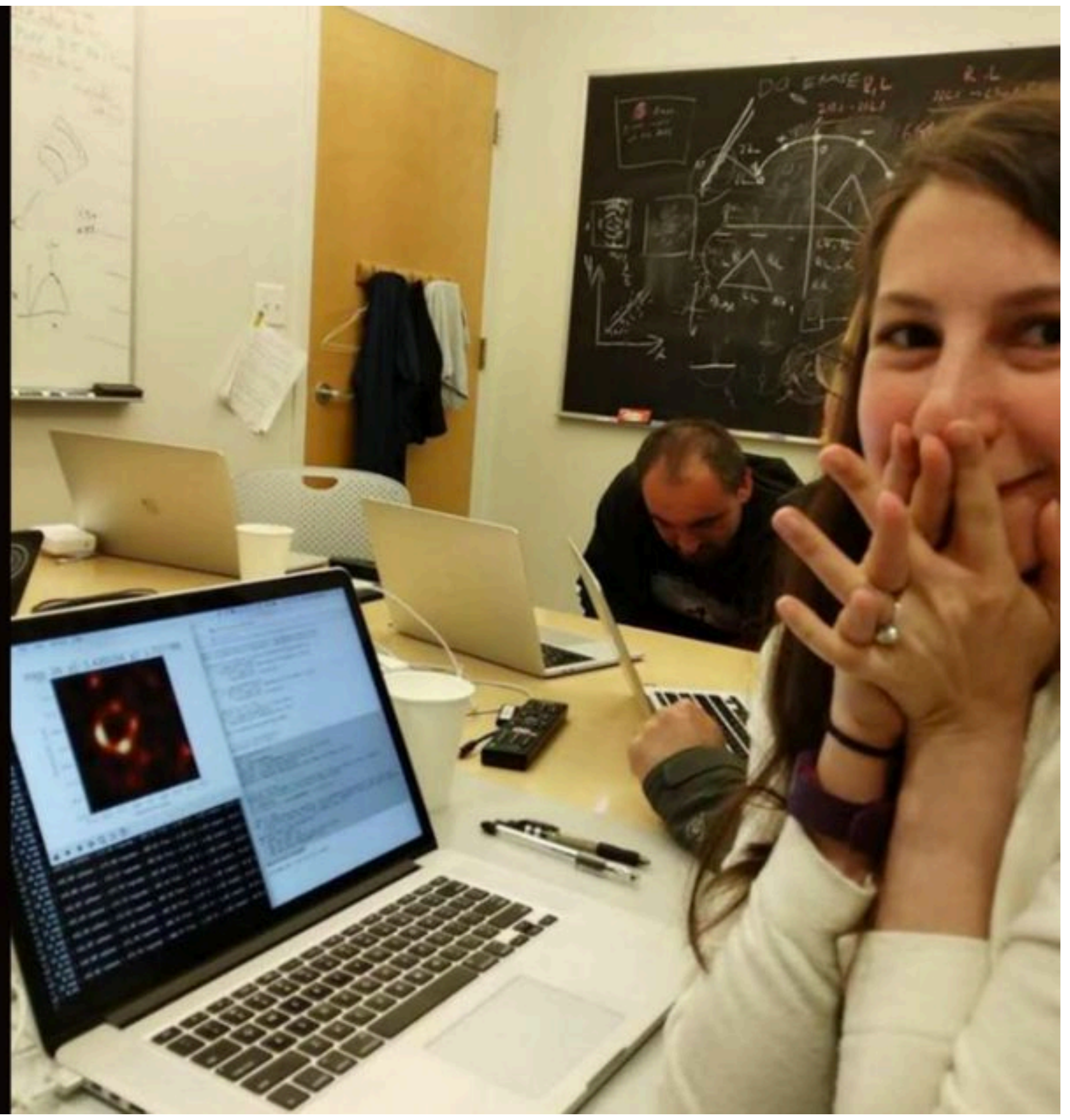
PyData

python™

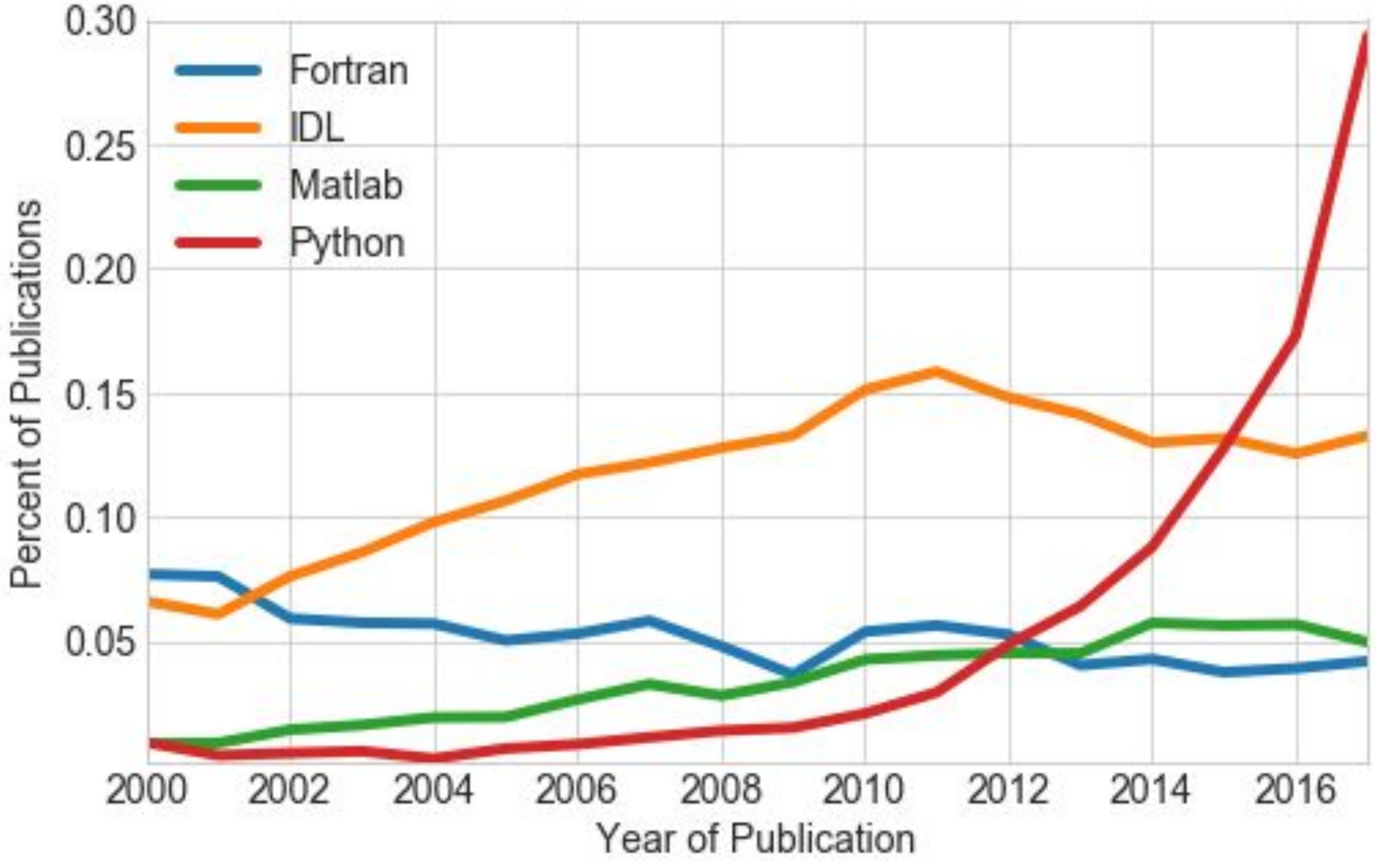
PYTHON IN ASTRONOMY

- *"Python is a language that is very powerful for developers, but is also accessible to Astronomers."*
  — Perry Greenfield, STScI, at PyAstro 2015





The Astropy Project

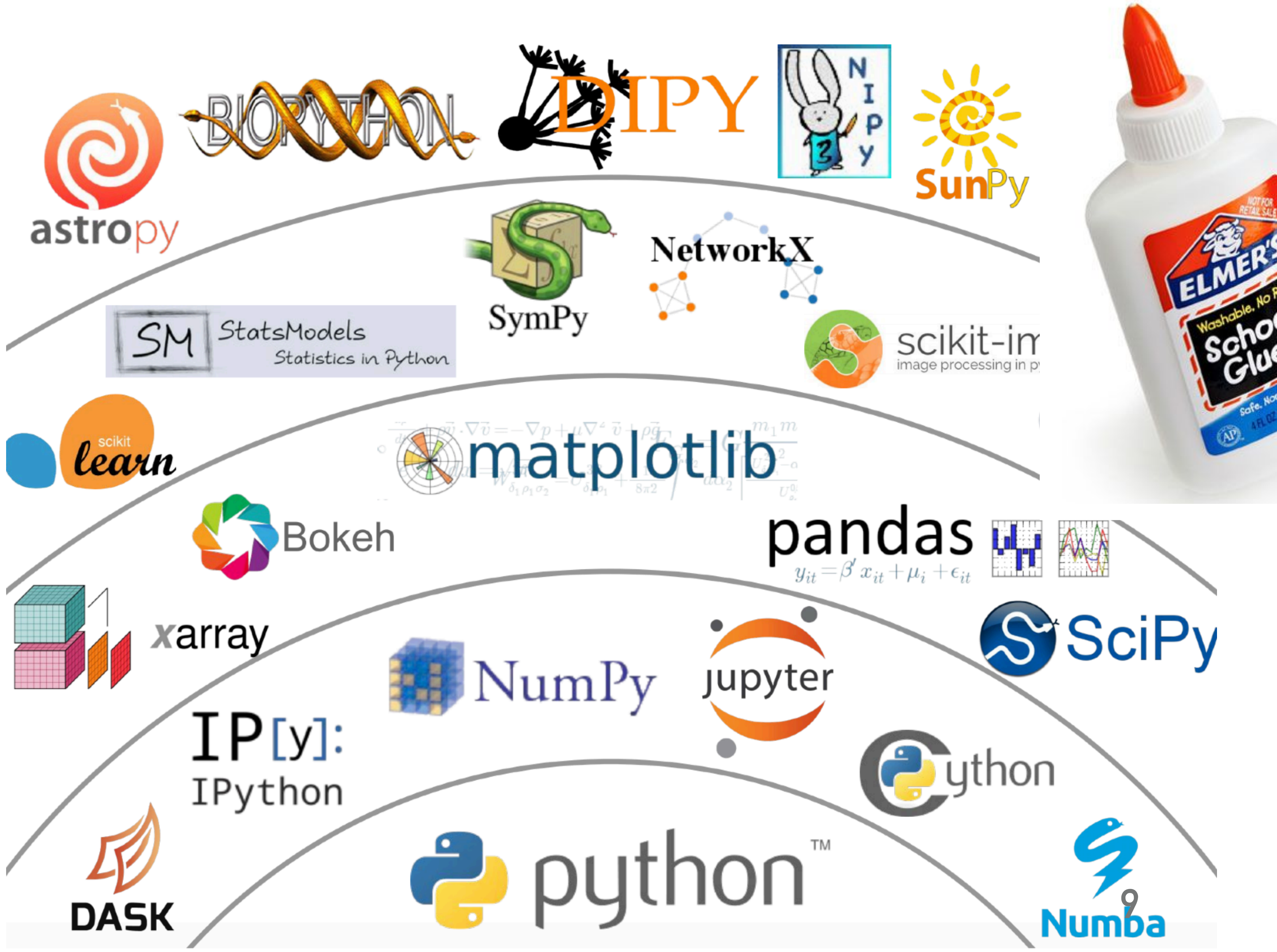


Compiled from NASA ADS (code).

Thanks to Juan Nunez-Iglesias, Thomas P. Robitaille, and Chris Beaumont.

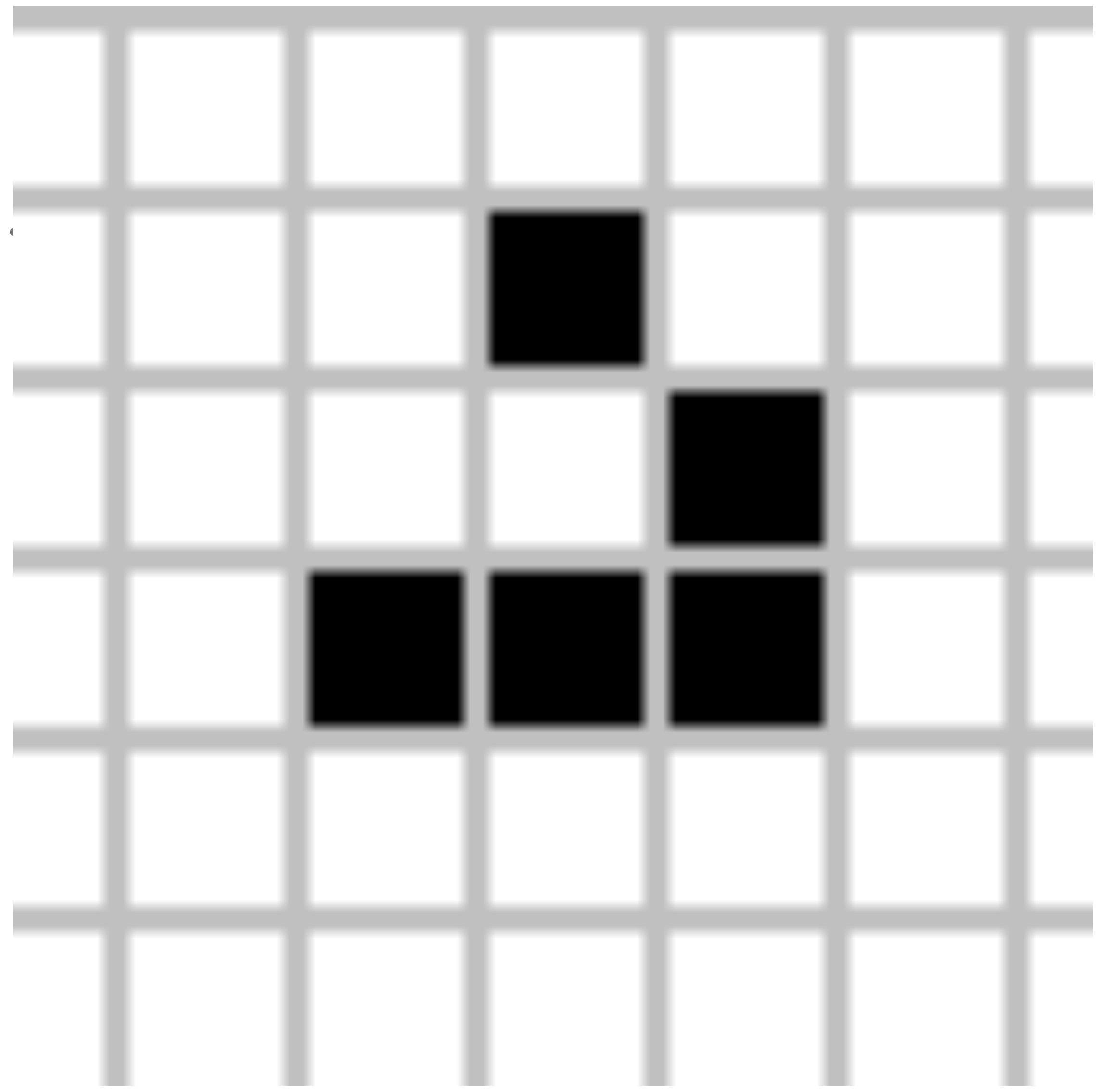
THE UNEXPECTED EFFECTIVENESS OF PYTHON IN SCIENCE

- Keynote PyCon 2017 by Jake VanderPlas
- *"For scientific data exploration, speed of development is primary, and speed of execution is often secondary."*
- *"Python has libraries for nearly everything … it is the glue to combine the scientific codes"*



WHY DO WE NEED NUMBA?

- Some algorithms are hard to write in Python & Numpy.
- Example: Conway's game of life
  See https://jakevdp.github.io/blog/2013/08/07/conways-game-of-life/
- Writing C and wrapping it for Python can be tedious.



```python
def life_step(X):
    """Game of life step using generator expressions"""
    nbrs_count = sum(np.roll(np.roll(X, i, 0), j, 1)
                     for i in (-1, 0, 1) for j in (-1, 0, 1)
                     if (i != 0 or j != 0))
    return (nbrs_count == 3) | (X & (nbrs_count == 2))
```

"Don't write Numpy Haikus. If loops are simpler, write loops and use Numba!"

— Stan Seibert, Numba team, Anaconda

INTRODUCING NUMBA

WHAT IS NUMBA? — HTTPS://NUMBA.PYDATA.ORG

Numba

Numba makes Python code fast

Numba is an open source JIT compiler that translates a subset of Python and NumPy code into fast machine code.

Learn More | Try Numba »

WHAT IS NUMBA?

“Numba” = “NumPy”+ “Mamba”

Numba crunching in Python, fast like Mambas.

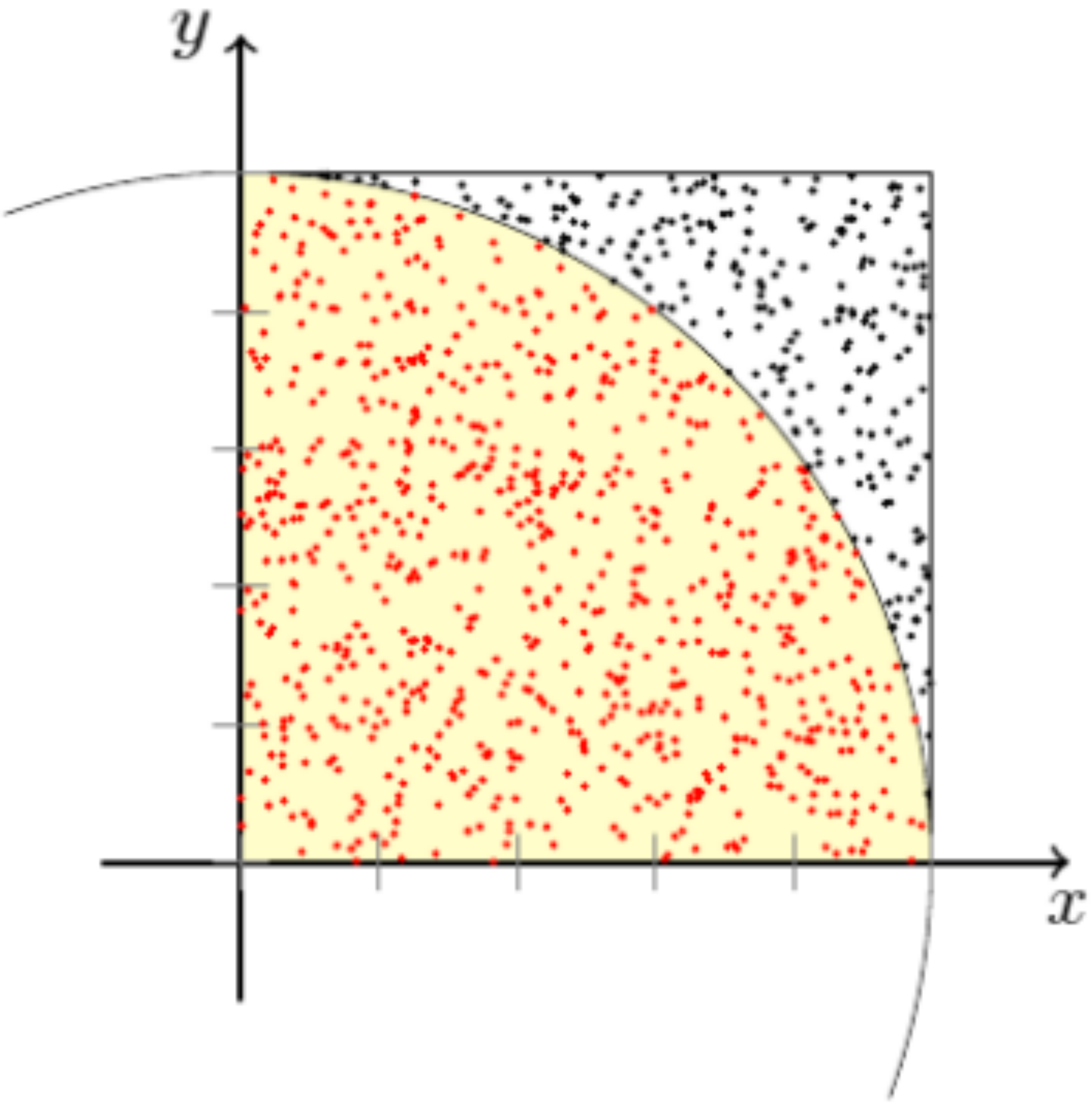
Numba logo (https://numba.pydata.org)

NUMBA ACCELERATES NUMERICAL PYTHON FUNCTIONS

```
import random


def monte_carlo_pi(nsamples):
    acc = 0
    for i in range(nsamples):
        x = random.random()
        y = random.random()
        if (x ** 2 + y ** 2) < 1.0:
            acc += 1
    return 4.0 * acc / nsamples
```

```
%timeit monte_carlo_pi(1_000_000)
```

400 ms — very slow



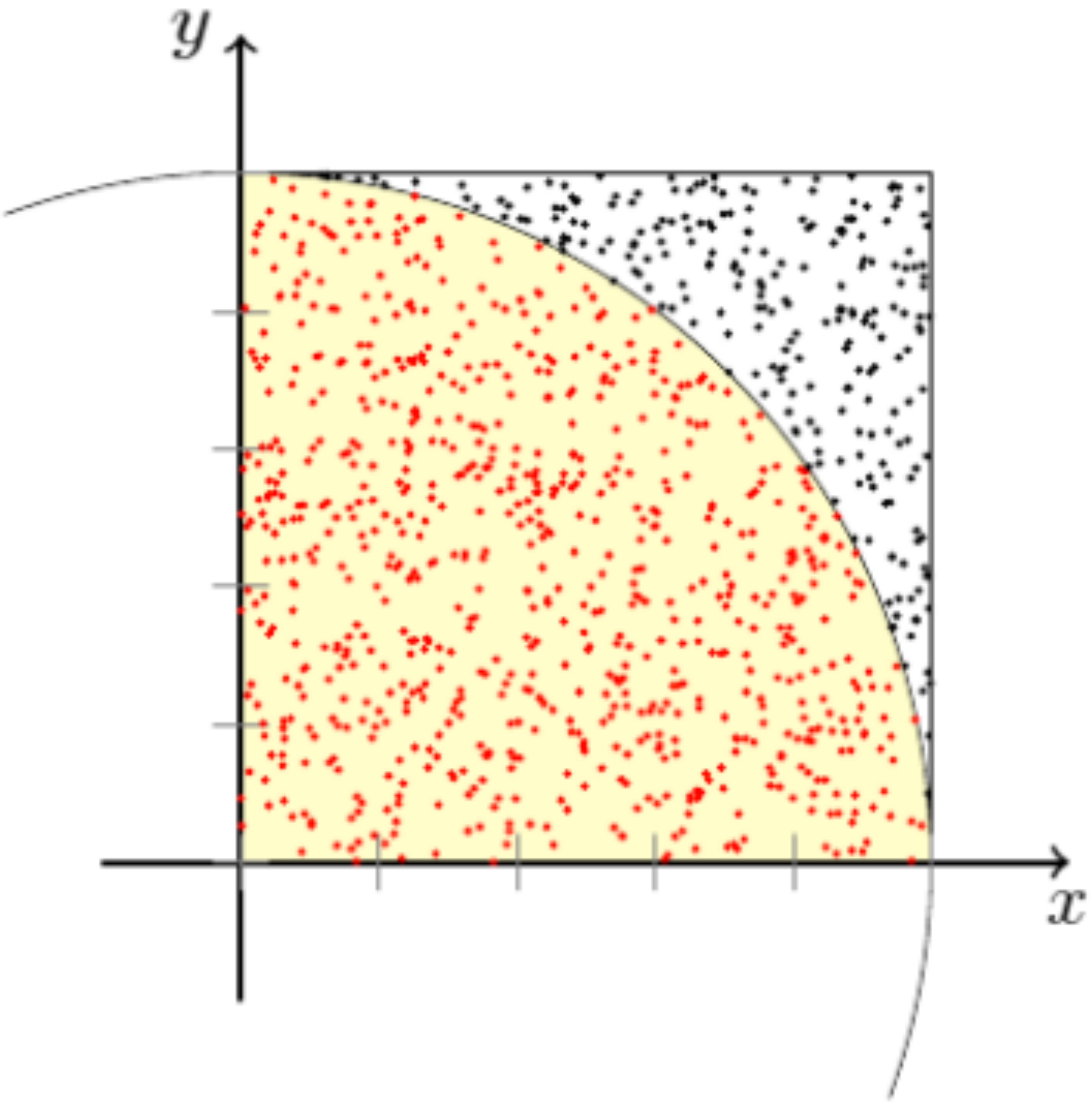
NUMBA ACCELERATES NUMERICAL PYTHON FUNCTIONS

```
import random
import numba

@numba.jit
def monte_carlo_pi(nsamples):
    acc = 0
    for i in range(nsamples):
        x = random.random()
        y = random.random()
        if (x ** 2 + y ** 2) < 1.0:
            acc += 1
    return 4.0 * acc / nsamples
```

Tell Numba to JIT your function

```
%timeit monte_carlo_pi(1_000_000)
```

13 ms — Numba/Python speedup: 30x



NUMBA UNDERSTANDS NUMPY

```
import numpy as np
x = np.random.random(1_000_000)
y = np.random.random(1_000_000)
```

```
def monte_carlo_pi(x, y):
    acc = np.sum(x ** 2 + y ** 2 < 1)
    return 4 * acc / len(x)

%timeit monte_carlo_pi(x, y)
```

```
4.07 ms
```

```
@numba.jit
def monte_carlo_pi(x, y):
    count = np.sum(x ** 2 + y ** 2 < 1)
    return 4 * count / len(x)

%timeit monte_carlo_pi(x, y)
```

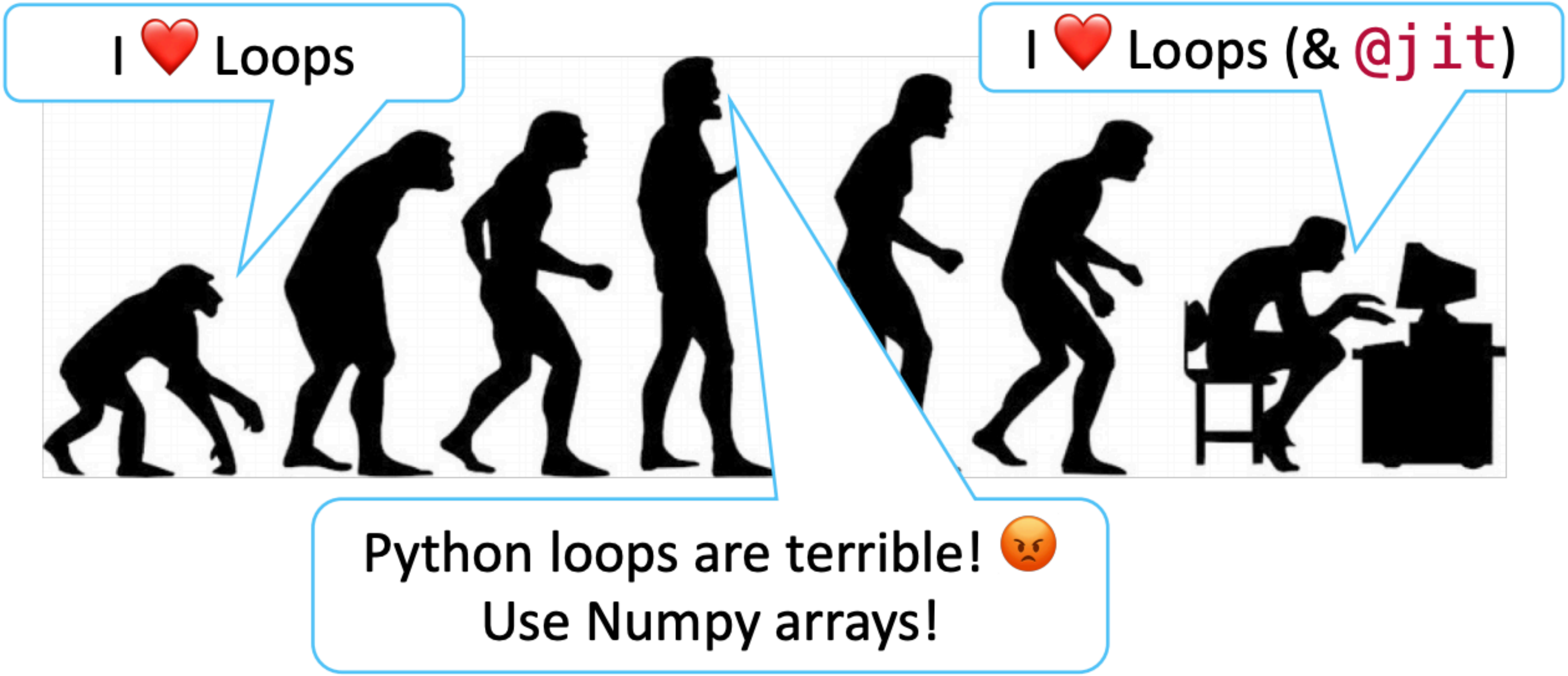
```
1.01 ms
```

- Use Numpy if you want!
  Use Python for loops if you want!
- Numba will compile either way to optimised machine code

```
@numba.jit
def monte_carlo_pi(x, y):
    acc = 0
    for i in range(x.shape[0]):
        if (x[i] ** 2 + y[i] ** 2) < 1:
            acc += 1
    return 4.0 * acc / x.shape[0]

%timeit monte_carlo_pi(x, y)
```

```
692 µs
```

EVOLUTION OF A SCIENTIFIC PROGRAMMER COMING TO PYTHON



Credit: Jason Watson (PyGamma19)

NUMBA LIMITATIONS

```python
def spam(n):
    return n * ["spam", 42]

spam(3)
```

```
['spam', 42, 'spam', 42, 'spam', 42]
```

```python
@numba.jit(nopython=True)
def spam(n):
    return n * ["spam", 42]

spam(3)
```

TypingError: Failed in nopython mode pipeline

- **Numba compiles individual functions.** Not whole programs like e.g. PyPy
- **Numba supports a subset of Python.** Some dict/list/set support, but not mixed types for keys or values
- **Numba supports a subset of Numpy.** Ever growing, but not all functions and all arguments are available.
- **Numba does not support pandas or other PyData or Python packages.**

```python
@numba.jit
def spam(n):
    return n * ["spam"]

spam(3)
```

NumbaWarning: Compilation is falling back to object mode
['spam', 42, 'spam', 42, 'spam', 42]

```python
@numba.jit(nopython=True)
def spam(n):
    return n * ["spam"]

spam(3)
```

TypingError: Failed in nopython mode pipeline

NUMBA.JIT MODES

- @numba.jit has a fallback **"object" mode**, which allows any Python code.
- This "object" mode results in machine code, but with PyObject and Python C API calls, and same performance as using Python directly without Numba
- Not what you want 99% of the time
- To get either the desired **"nopython" mode**, or a TypingError you can use **@numba.jit(nopython=True)** or the equivalent **@numba.njit**

NUMBA.OBJMODE CONTEXT MANAGER

- To call back to Python there is **numba.objmode** (rarely needed)
- Can be useful in long-running functions e.g. to log or update a progress bar

```
@numba.njit
def foo():
    x = np.arange(5)
    with numba.objmode(y='intp[:]'):  # annotate return type
        # this region is executed by object-mode.
        y = np.asarray(list(reversed(x.tolist())))
    return y
```

UNDERSTANDING NUMBA

(A LITTLE BIT)

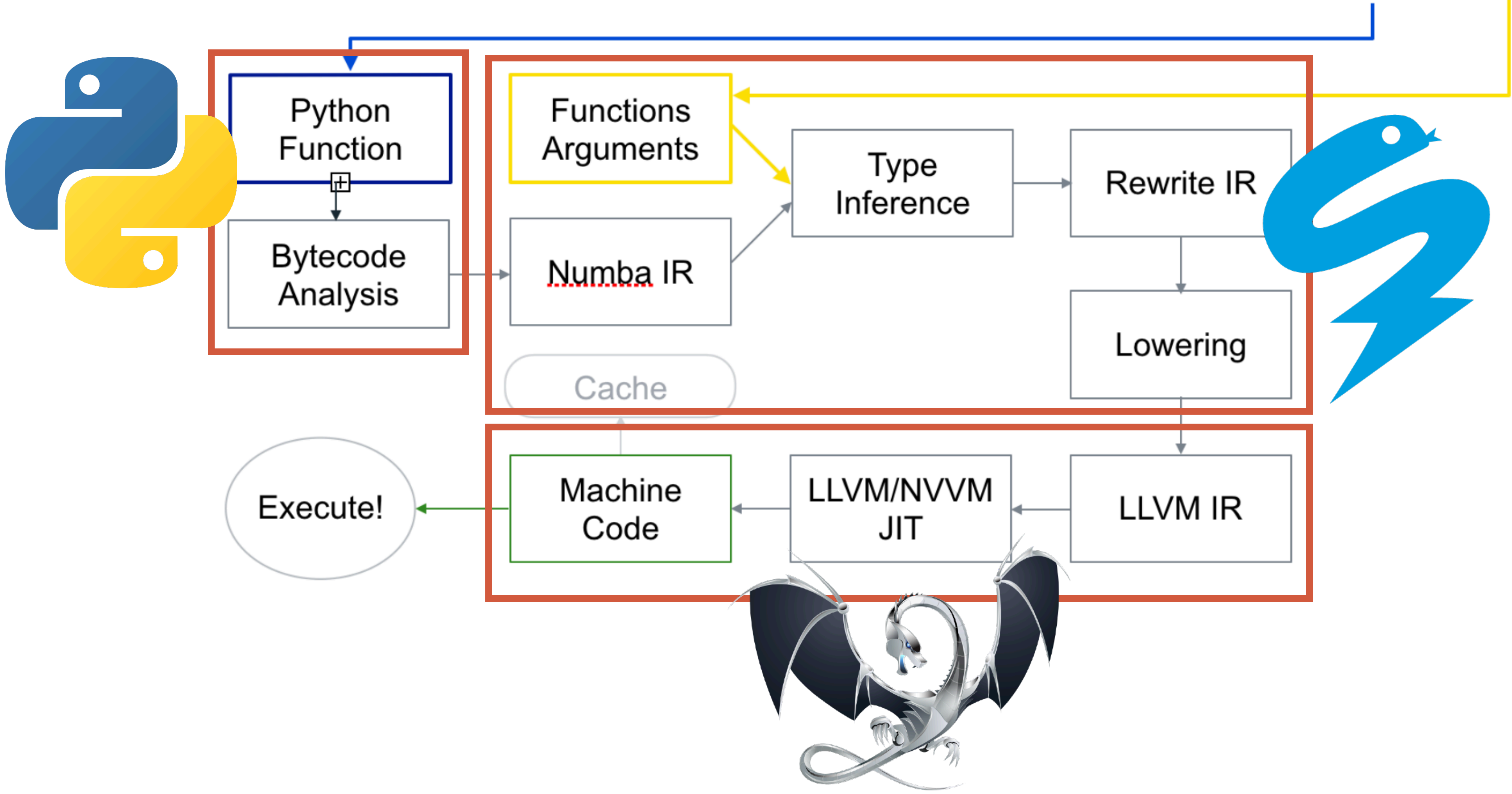
UNDERSTANDING NUMBA

"Numba is a type-specialising JIT compiler from Python bytecode using LLVM"

https://youtu.be/LLpIMRowndg

PYTHON & NUMBA & LLVM

```
@jit
def do_math(a, b):
    …
>>> do_math(x, y)
```



PYTHON

- Python compiler starts with source code, parses it into an Abstract Syntax Tree (AST), then transforms it to Bytecode
- Happens on import of a module
- Bytecode for a function is attached to the Python function object (code=data)

```
>>> def cond():
...     x = 3
...     if x < 5:
...         return 'yes'
...     else:
...         return 'no'
...
```

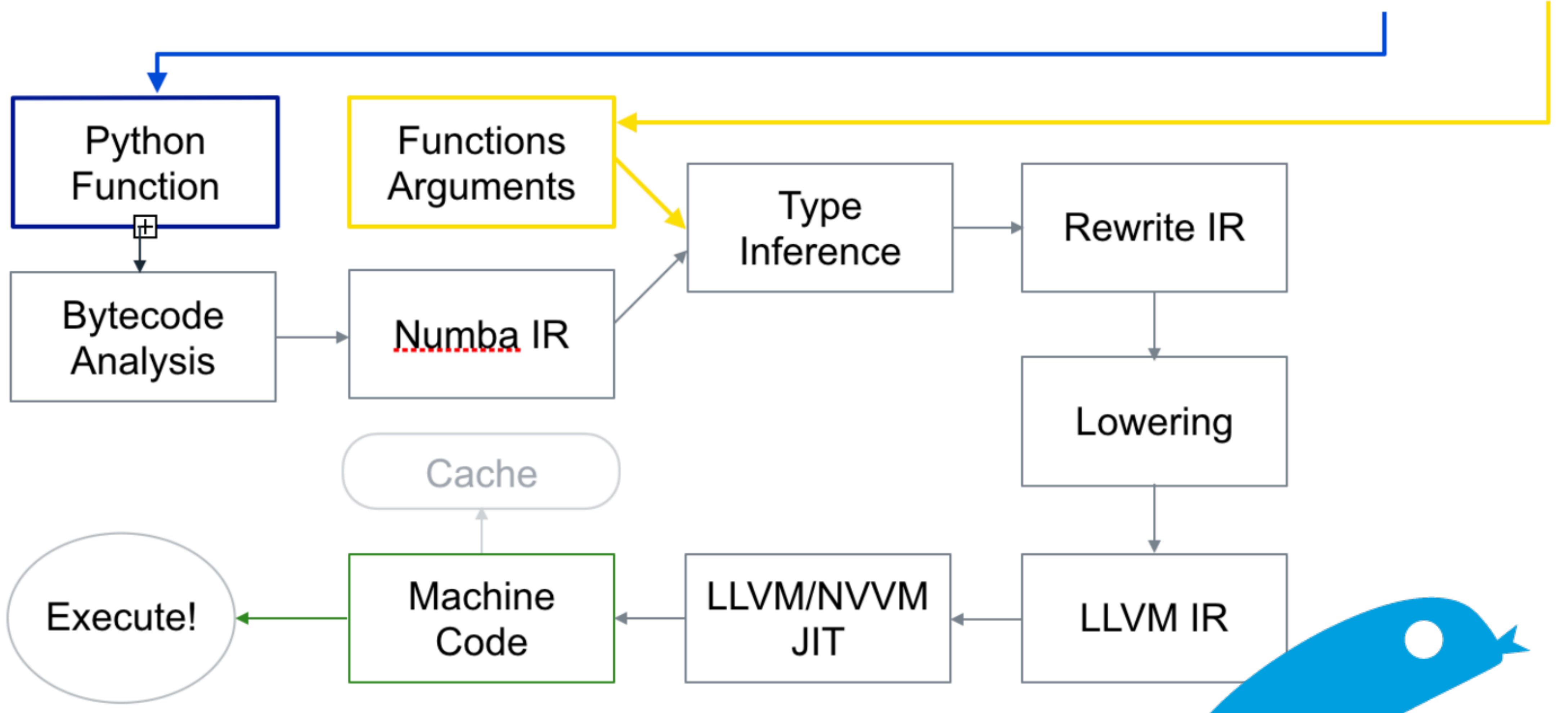
```
>>> dis.dis(cond)
  2           0 LOAD_CONST               1 (3)
              3 STORE_FAST               0 (x)

  3           6 LOAD_FAST                0 (x)
              9 LOAD_CONST               2 (5)
             12 COMPARE_OP               0 (<)
             15 POP_JUMP_IF_FALSE       22

  4          18 LOAD_CONST               3 ('yes')
             21 RETURN_VALUE

  6     >>   22 LOAD_CONST               4 ('no')
             25 RETURN_VALUE
             26 LOAD_CONST               0 (None)
             29 RETURN_VALUE
```

```
>>> cond.__code__.co_code  # the bytecode as raw bytes
b'd\x01\x00}\x00\x00|\x00\x00d\x02\x00k\x00\x00r\x16\x00d\x03\x00Sd\x04\x00Sd\x00
   \x00S'
>>> list(cond.__code__.co_code)  # the bytecode as numbers
[100, 1, 0, 125, 0, 0, 124, 0, 0, 100, 2, 0, 107, 0, 0, 114, 22, 0, 100, 3, 0, 83,
 100, 4, 0, 83, 100, 0, 0, 83]
```

NUMBA

- On **@numba.jit** decorator call, Numba makes a CPUDispatcher proxy object.
- On function call, Numba will:
  - JIT compile Bytecode to LLVM IR exactly for the input types
  - Manage LLVM compilation
  - Execute compiled function

```
@jit
def do_math(a, b):
    …
>>> do_math(x, y)
```




```
@numba.jit
def compute(n):
    return 2 * n
```

```
compute
```

```
CPUDispatcher(<function compute at 0x6261a0e18>)
```

```
compute.overloads
```

```
OrderedDict()
```

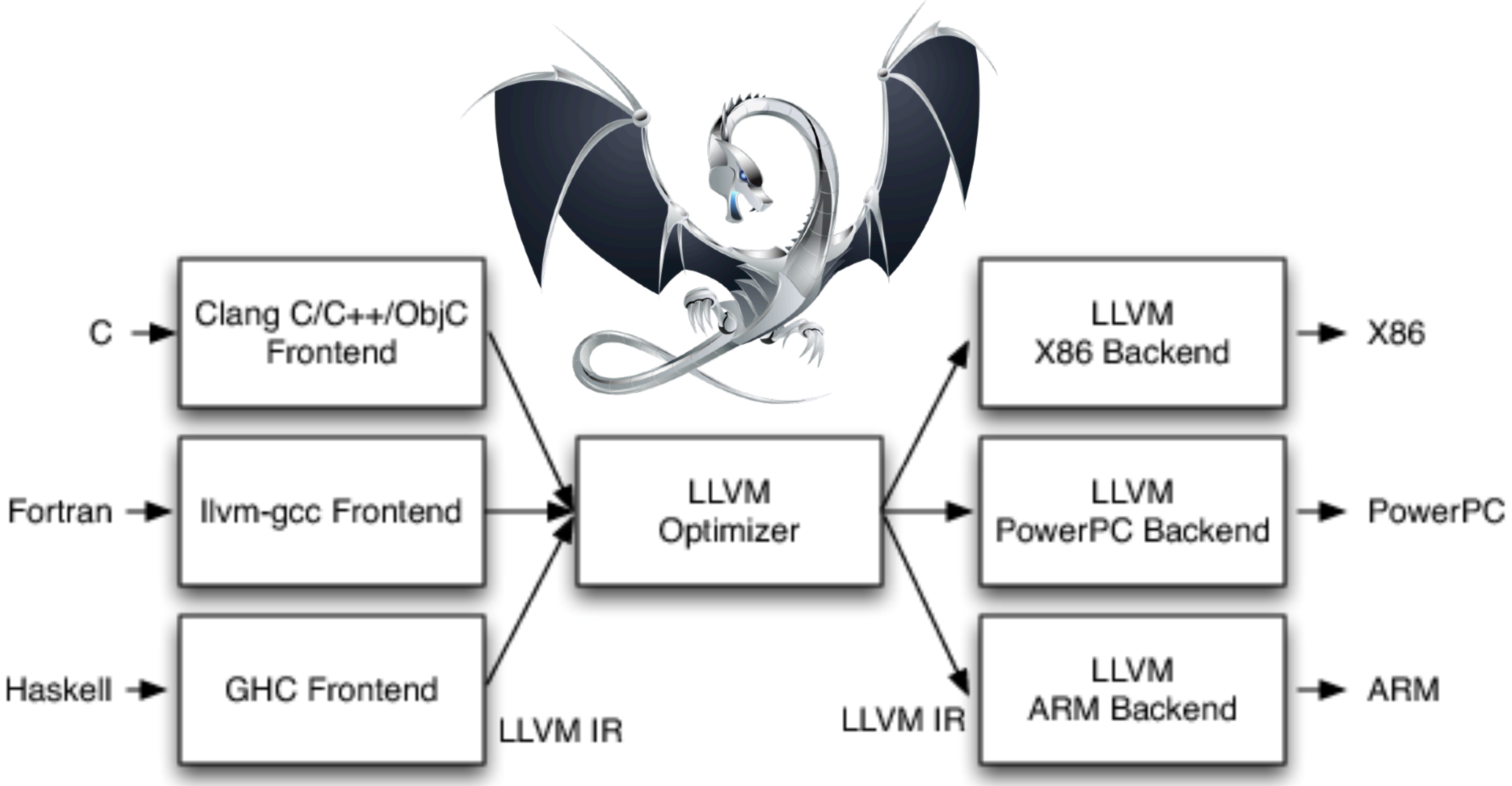
```
compute(3)
```

```
6
```

```
compute.overloads
```

```
OrderedDict([((int64,),
              CompileResult(typing_context=<numba
```

LLVM



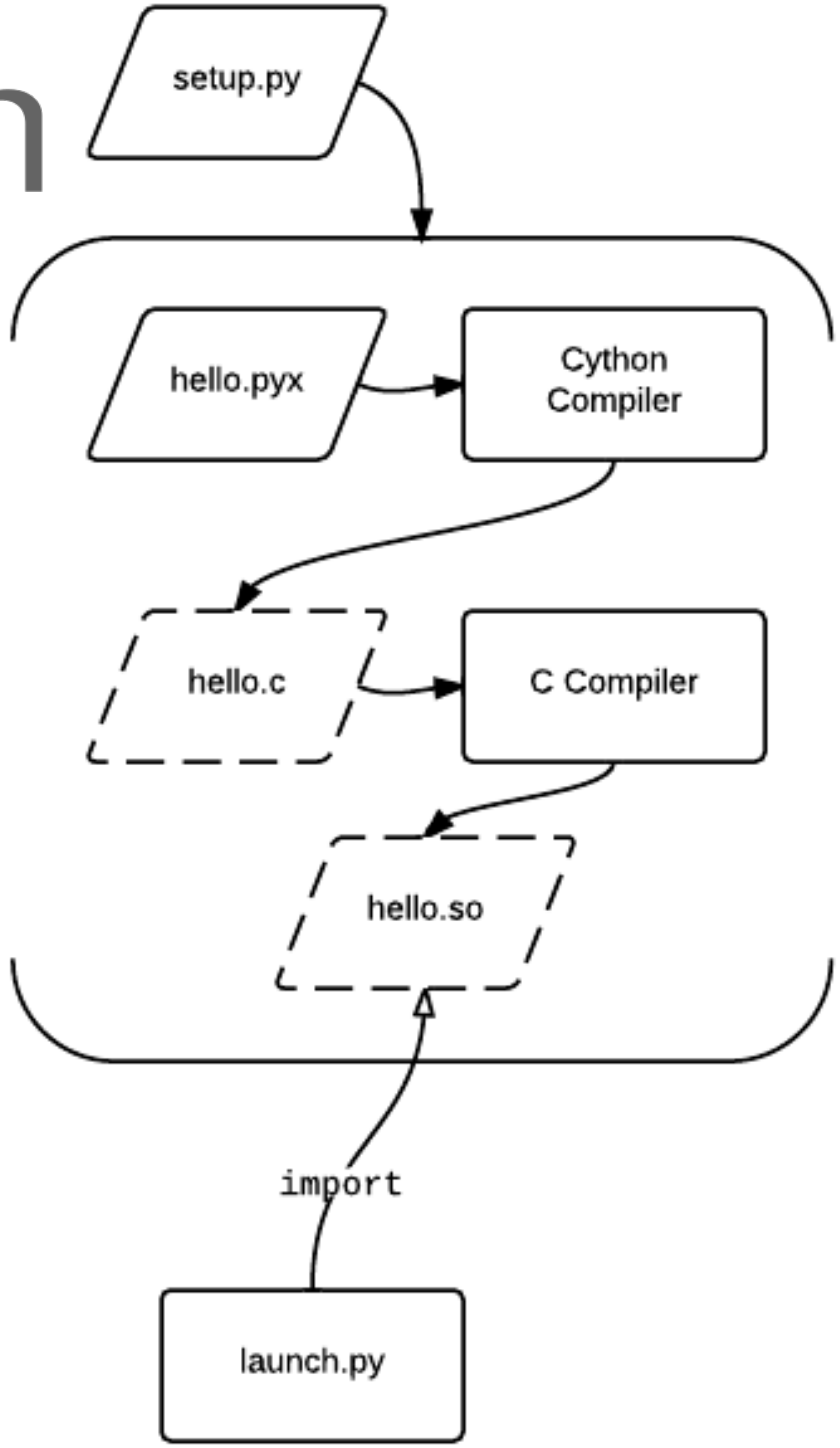
LLVM intermediate representation (IR) example:

```
define i32 @add1(i32 %a, i32 %b) {
entry:
  %tmp1 = add i32 %a, %b
  ret i32 %tmp1
}

define i32 @add2(i32 %a, i32 %b) {
entry:
  %tmp1 = icmp eq i32 %a, 0
  br i1 %tmp1, label %done, label %recurse
```

- LLVM is a compiler infrastructure project
- Many frontends for languages: C, C++ Fortran, Haskell, Rust, Julia, Swift, …
- Many backends for hardware: almost all CPU vendors add support and optimise
- Numba could be considered the Python front-end to LLVM
- LLVM is shipped as a Python package “llvmlite" that Numba depends on
- Numba team at Anaconda Inc. builds numba and llvmlite for conda and pip

CYTHON VS. NUMBA

```
In [1]: %load_ext Cython

In [2]: %%cython
   ...: def f(n):
   ...:     a = 0
   ...:     for i in range(n):
   ...:         a += i
   ...:     return a
   ...:
   ...: cpdef g(int n):
   ...:     cdef long a = 0
   ...:     cdef int i
   ...:     for i in range(n):
   ...:         a += i
   ...:     return a
   ...:

In [3]: %timeit f(1000000)
10 loops, best of 3: 26.5 ms per

In [4]: %timeit g(1000000)
1000 loops, best of 3: 279 µs per
```



- ➤ Like Numba, Cython is often used to speed up numeric Python code
- ➤ Cython is an "ahead of time" (AOT) compiler of type-annotated Python to C
- ➤ Cython is more widely used, easier to debug, very good at interfacing C/C++
- ➤ Numba is easier to use: no type annotations, no C compiler, but sometimes harder to debug (LLVM IR)
- ➤ Numba optimises JIT for your CPU or GPU, no need to build and distribute binaries for many architectures

Source: https://en.wikipedia.org/wiki/Cython

NUMBA ALTERNATIVES

- Many other great tools exist for high-performance computing with Python
- **Cython/C/C++/pybind11** to create Python C extensions
- **PyPy** is an alternative to CPython, that JIT-compiles the whole program
- **TensorFlow, JAX, PyTorch, Dask, …** use Python & Numpy as the language to specify computation, but then compile and execute in various ways
- How to do HPC from Python?
  Not an easy choice!

MORE NUMBA

NUMBA -S

- From the command line:
  numba -s
  numba --sysinfo
- From IPython or Jupyter:
  !numba -s
- Gives you all relevant information:
  - Hardware: CPU & GPU
  - Python, Numba, LLVM versions
  - SVML: Intel short vector math library
  - TBB: Intel threading building blocks
  - CUDA & ROC

```
$ numba -s
__Hardware Information__
Machine                                       : x86_64
CPU Name                                      : haswell
CPU count                                     : 8
CPU Features                                  :
aes avx avx2 bmi bmi2 cmov cx16 f16c fma fsgsbase invpcid lzcnt mmx movbe pclmul
popcnt rdrnd sahf sse sse2 sse3 sse4.1 sse4.2 ssse3 xsave xsaveopt

__OS Information__
Platform                                      : Darwin-18.5.0-x86_64-i386-64bit

__Python Information__
Python Compiler                               : Clang 4.0.1 (tags/RELEASE_401/final)
Python Implementation                         : CPython
Python Version                                : 3.7.3

__LLVM information__
LLVM version                                  : 7.0.0

__CUDA Information__
CUDA driver library cannot be found or no CUDA enabled devices are present.

__ROC Information__
ROC available                                 : False

__SVML Information__
SVML operational                              : True

__Threading Layer Information__
TBB Threading layer available                 : True
OpenMP Threading layer available              : True
Workqueue Threading layer available           : True
```

PARALLEL ACCELERATOR

```python
data = np.random.random(1_000_000)
```

```python
@numba.jit
def f(x):
    return np.cos(x) ** 2 + np.sin(x) ** 2

%timeit f(data)
```

```
11.3 ms
```

```python
@numba.jit(parallel=True)
def f(x):
    return np.cos(x) ** 2 + np.sin(x) ** 2

%timeit f(data)
```

```
3.51 ms
```

3.2x speedup on my 4-core CPU

- Add **parallel=True** to use multi-core CPU via threading
- Backends: openmp, tbb, workqueue
- Intel Threading Building Blocks needs **$ conda install tbb**
- Works automatically for Numpy array expressions - no code changes needed

PARALLEL ACCELERATOR

```
@numba.jit
def compute(x):
    s = 0
    for i in numba.prange(x.shape[0]):
        s += x[i]
    return s

%timeit compute(data)
```

855 µs

```
@numba.jit(parallel=True)
def compute(x):
    s = 0
    for i in numba.prange(x.shape[0]):
        s += x[i]
    return s

%timeit compute(data)
```

388 µs

2.2x speedup on my 4-core CPU

- Use **numba.prange** with **parallel=True** if you have for loops
- With the default **parallel=False**, **numba.prange** is the same as **range**.
- You can try out different options:

```
def compute(x):
    s = 0
    for i in numba.prange(x.shape[0]):
        s += x[i]
    return s
```

```
compute1 = numba.jit(compute)
```

```
compute2 = numba.jit(parallel=True)(compute)
```

FASTMATH

```python
def compute(x):
    acc = 0.0
    for item in x:
        acc += np.sqrt(item)
    return acc
```

```python
data = np.random.random(1_000_000)
```

```python
c1 = numba.jit(compute)
%timeit c1(data)
```

```
3.92 ms
```

```python
c2 = numba.jit(fastmath=True)(compute)
%timeit c2(data)
```

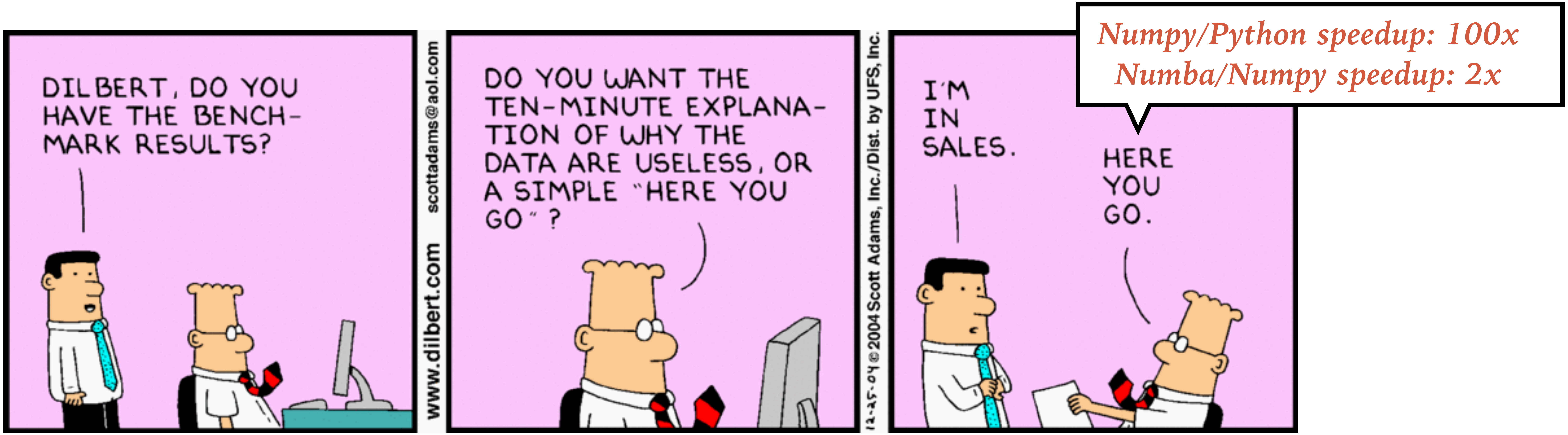
```
2.17 ms
```

- Add **fastmath=True** to trade accuracy for speed in some computations
- IEEE 754 floating point standard requires that loop must accumulate in order
- With fastmath=True, vectorised reduction is used, which is faster

- Another way to speed up math functions like sin, exp, tanh, … is this:
  $ conda install -c numba icc_rt
- If available, Numba will tell LLVM to use Intel Short Vector Math Library (SVML)

HOW FAST IS NUMBA?

- Numba gives very good performance, and many options to tweak the computation
- There is no simple answer how Numba compares to Python, Cython, Numpy, C, …
- Always define a benchmark for your application and measure!



NUMPY UFUNCS

```python
import numpy as np
```

```python
np.add(1, 2)
```

```
3
```

```python
np.add(1, [2, 3])
```

```
array([3, 4])
```

```python
np.add([[1, 2]], [[3], [4]])
```

```
array([[4, 5],
       [5, 6]])
```

```python
np.add.accumulate([2, 3, 4, 5])
```

```
array([ 2,  5,  9, 14])
```

- Numpy functions like add, sin, …
  are universal functions (“ufuncs”)
- They all support array broadcasting, data type handling, and some other features like accumulate or reduce.
- So far, you had to write C and use the Numpy C API to make your own ufunc

NUMBA.VECTORIZE

```python
@numba.vectorize("(int64, int64)")
def add(x, y):
    # Write operation for one element
    return x + y
```

```python
add(1, 2)
```

```
3
```

```python
add(1, [2, 3, 4])
```

```
array([3, 4, 5])
```

```python
add.accumulate([2, 3, 4, 5])
```

```
array([ 2,  5,  9, 14])
```

- The **@numba.vectorize** decorator makes it easy to write Numpy ufuncs.
- Just write operation for one element
- You can give a type signature, or list of types to support, and Numba will generate one ufunc on vectorize call
- If no signature is given, a DUFunc dispatcher is created, which dynamically will create ufunc for given input types on function call.

NUMBA - A FAMILY OF COMPILERS

- Numba has more compilers, all implemented as Python decorators. This was just a quick introduction, see http://numba.pydata.org/
- **@numba.jit** — regular function
- **@numba.vectorize** — Numpy ufunc
- **@numba.guvectorize** — Numpy generalised ufunc
- **@numba.stencil** — neighbourhood computation
- **@numba.cfunc** — C callbacks
- **@numba.cuda.jit** — NVidia CUDA kernels
- **@numba.roc.jit** — ARM ROCm kernels

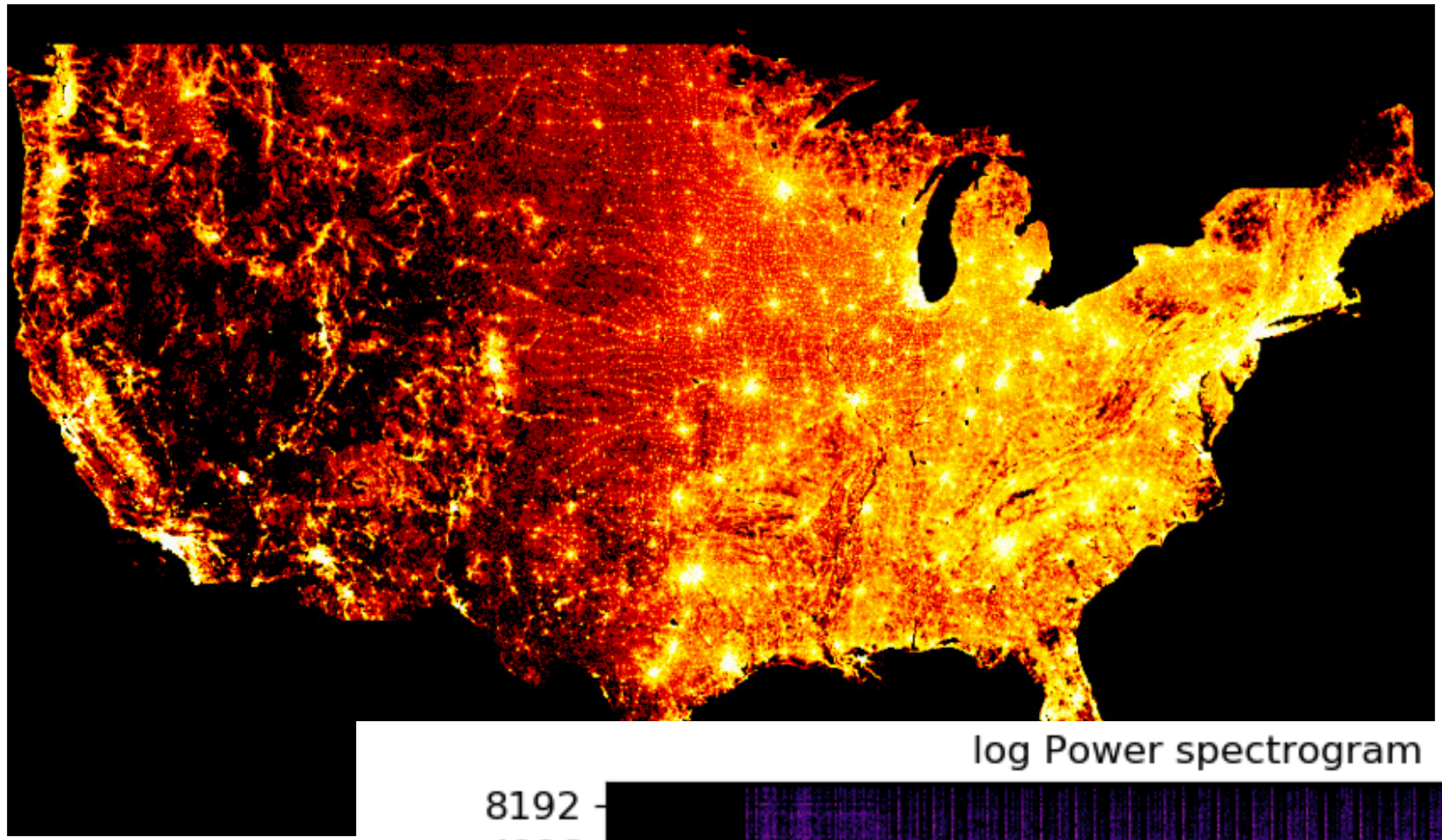
WHO USES NUMBA?

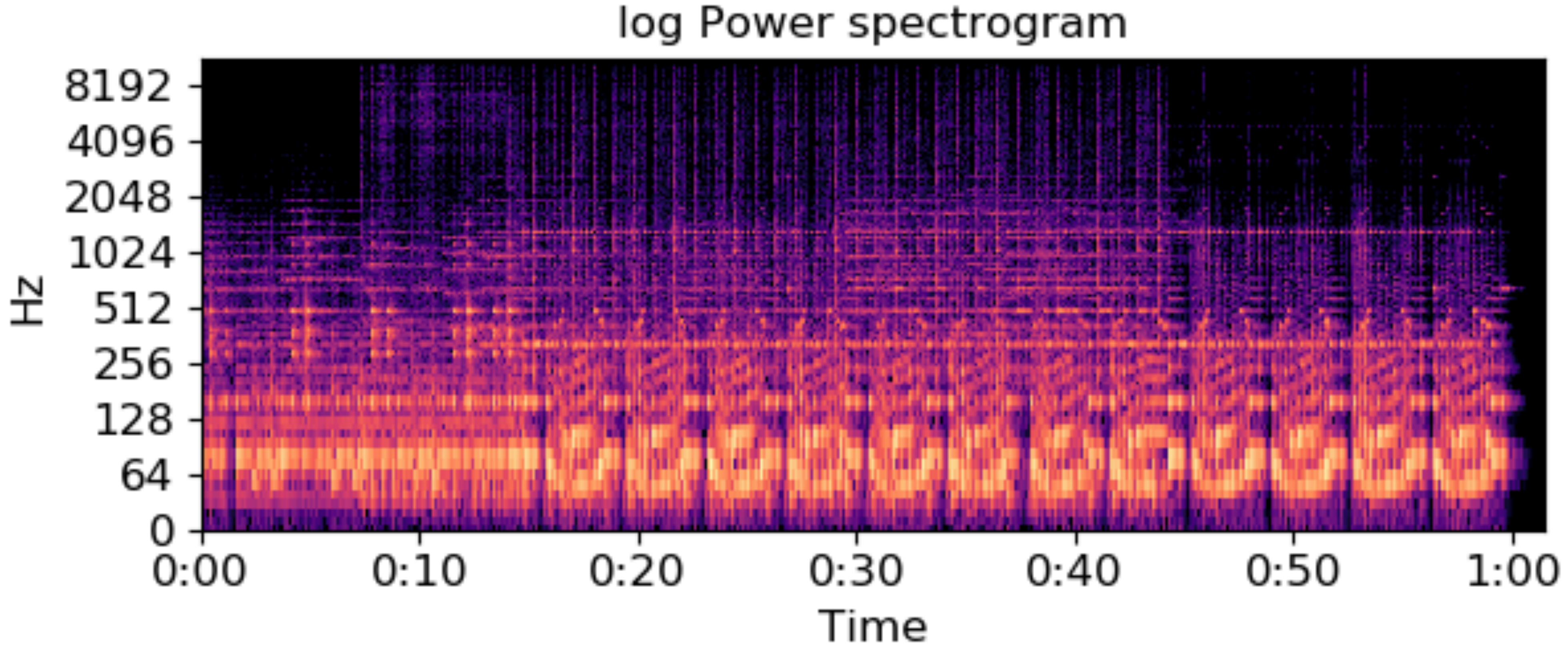
"I'm becoming more and more convinced that Numba is the future of fast scientific computing in Python."
— *Jake Vanderplas (2013)*

"The numeric Python community should consider adopting Numba more widely within community code."
— *Matthew Rocklin (2018)*

WHO USES NUMBA?

- Many people and applications use it for their work and projects
- Large libraries like Numpy, Scipy, pandas, scikit-learn, ... not yet.
- Some nice examples using Numba:
  - Datashader - large data visualisation
  - LibROSA - audio & music analysis
  - HPAT - Intel High Performance Toolkit for big data, supports pandas

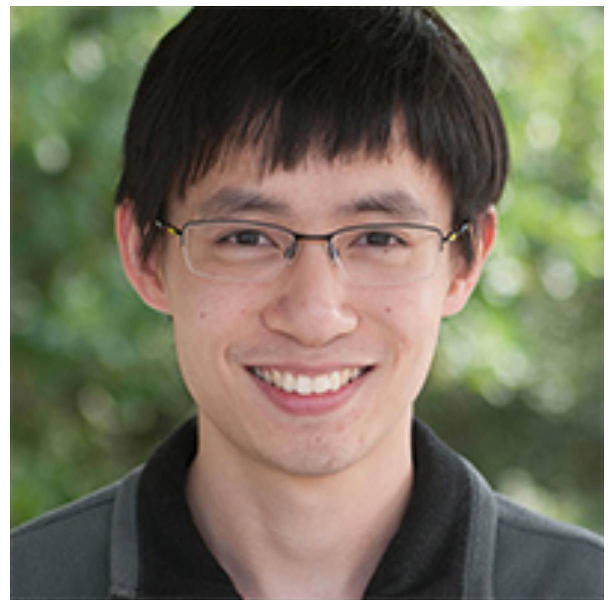





```python
@hpat.jit
def logistic_regression(iterations):
    f = h5py.File("lr.hdf5", "r")
    X = f['points'][:]
    Y = f['responses'][:]
    D = X.shape[1]
    w = np.random.ranf(D)
    t1 = time.time()
    for i in range(iterations):
        z = ((1.0 / (1.0 + np.exp(-Y * np.dot(X, w))) - 1.0) * Y)
        w -= np.dot(z, X)
    return w
```

SUMMARY & CONCLUSIONS

- Numba is a type-specialising JIT compiler from Python byte code to LLVM IR
- Started 2012, current version is v0.44, well on the road to v1.0.
- Use your CPU or GPU well, just by writing Python and adding a decorator
- Use **@numba.jit** for normal functions, and **@numba.vectorize** for Numpy ufuncs
  To check your machine & installation: **numba -s**
  Consider **parallel=True** and **fastmath=True** to run faster on the CPU
  To get Intel SVML: **conda install -c numba icc_rt**
- Thanks to the Numba devs at Anaconda, and contributions by Intel and others!!!

ANACONDA