Numba:
A JIT Compiler for Scientific Python

Stan Seibert [@seibert]
(on behalf of the entire Numba team)

Continuum Analytics
SciPy 2015

The Need for Speed

- For many programs, the most important resource is *developer time*.
- The best code is:
    - Easy to understand
    - Easy to modify
- But sometimes execution speed does matter. Then what do you do?
- Go find a compiler!

A Python Compiler?

- Takes advantage of a simple fact:
  - *Most functions in your program only use a small number of types.*

➡ Generate machine code to manipulate only the types you use!

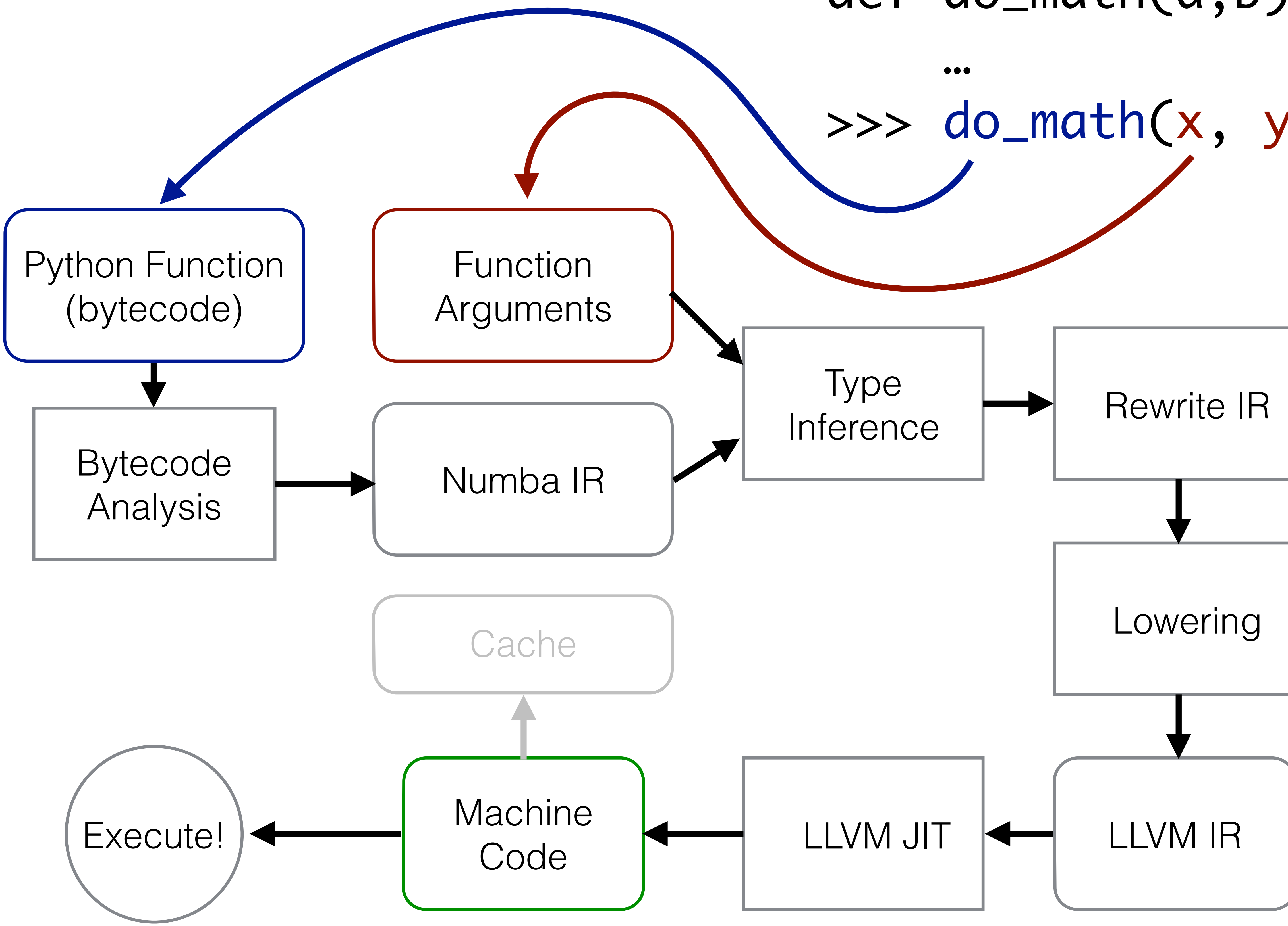
- LLVM library handles the compiler backend for us

The Python Compilation Space

	Ahead Of Time	Just In Time
Relies on CPython / libpython	Cython Shedskin Nuitka (today) Pythran	Numba HOPE Theano
Replaces CPython / libpython	Nuitka (future)	Pyston PyPy

How Numba Works

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



Numba Features

- Numba supports:
  - Windows, OS X, and Linux
  - 32 and 64-bit x86 CPUs and NVIDIA GPUs
  - Python 2 and 3
  - NumPy versions 1.6 through 1.9
- Does *not* require a C/C++ compiler on the user's system.
- < 70 MB to install.
- Does *not* replace the standard Python interpreter
  (all of your existing Python libraries are still available)

Numba Modes

- *object mode:* Compiled code operates on Python objects. Only significant performance improvement is compilation of loops that can be compiled in nopython mode (see below).
- *nopython mode:* Compiled code operates on "machine native" data. Usually within 25% of the performance of equivalent C or FORTRAN.

How to Use Numba

1. Create a realistic benchmark test case.
   (Do not use your unit tests as a benchmark!)
2. Run a profiler on your benchmark.
   (cProfile is a good choice)
3. Identify hotspots that could potentially be compiled by Numba with a little refactoring.
   (see rest of this talk and online documentation)
4. Apply @numba.jit and @numba.vectorize as needed to critical functions.
   (Small rewrites may be needed to work around Numba limitations.)
5. Re-run benchmark to check if there was a performance improvement.

A Whirlwind Tour of Numba Features

The Basics

In [87]:
```
@jit(nopython=True)
def nan_compact(x):
    out = np.empty_like(x)
    out_index = 0
    for element in x:
        if not np.isnan(element):
            out[out_index] = element
            out_index += 1
    return out[:out_index]
```

In [88]:
```
a = np.random.uniform(size=10000)
a[a < 0.2] = np.nan
np.testing.assert_equal(nan_compact(a), a[~np.isnan(a)])
```

In [89]:
```
%timeit a[~np.isnan(a)]
%timeit nan_compact(a)
```

```
10000 loops, best of 3: 52 µs per loop
100000 loops, best of 3: 19.6 µs per loop
```

The Basics

Numba decorator (nopython=True not required)

Array Allocation

Looping over ndarray x as an iterator

Using numpy math functions

Returning a slice of the array

In [87]:
```
@jit(nopython=True)
def nan_compact(x):
    out = np.empty_like(x)
    out_index = 0
    for element in x:
        if not np.isnan(element):
            out[out_index] = element
            out_index += 1
    return out[:out_index]
```

In [88]:
```
a = np.random.uniform(size=10000)
a[a < 0.2] = np.nan
np.testing.assert_equal(nan_compact(a), a[~np.isnan(a)])
```

In [89]:
```
%timeit a[~np.isnan(a)]
%timeit nan_compact(a)
```

```
10000 loops, best of 3: 52 µs per loop
100000 loops, best of 3: 19.6 µs per loop
```

2.7x speedup!

Calling Other Functions

In [85]:
```
@jit
def norm(vec):
    mag = 0.0
    for element in vec:
        mag += element**2
    mag **= 0.5

    ret = np.empty_like(vec)
    for i, element in enumerate(vec):
        ret[i] = element / mag

    return ret

@jit
def clamp(x):
    if x > 1.0:
        return 1.0
    elif x < -1.0:
        return -1.0
    else:
        return x

@jit
def angle_between(vec1, vec2):
    norm_vec1 = norm(vec1)
    norm_vec2 = norm(vec2)

    cos_angle = (norm_vec1 * norm_vec2).sum()
    return np.arccos(clamp(cos_angle))
```

Calling Other Functions

```
In [85]: @jit
def norm(vec):
    mag = 0.0
    for element in vec:
        mag += element**2
    mag **= 0.5

    ret = np.empty_like(vec)
    for i, element in enumerate(vec):
        ret[i] = element / mag

    return ret

@jit
def clamp(x):
    if x > 1.0:
        return 1.0
    elif x < -1.0:
        return -1.0
    else:
        return x

@jit
def angle_between(vec1, vec2):
    norm_vec1 = norm(vec1)
    norm_vec2 = norm(vec2)

    cos_angle = (norm_vec1 * norm_vec2).sum()
    return np.arccos(clamp(cos_angle))
```

This function is not inlined

This function is inlined

9.8x speedup compared to doing this with numpy functions

Making Ufuncs

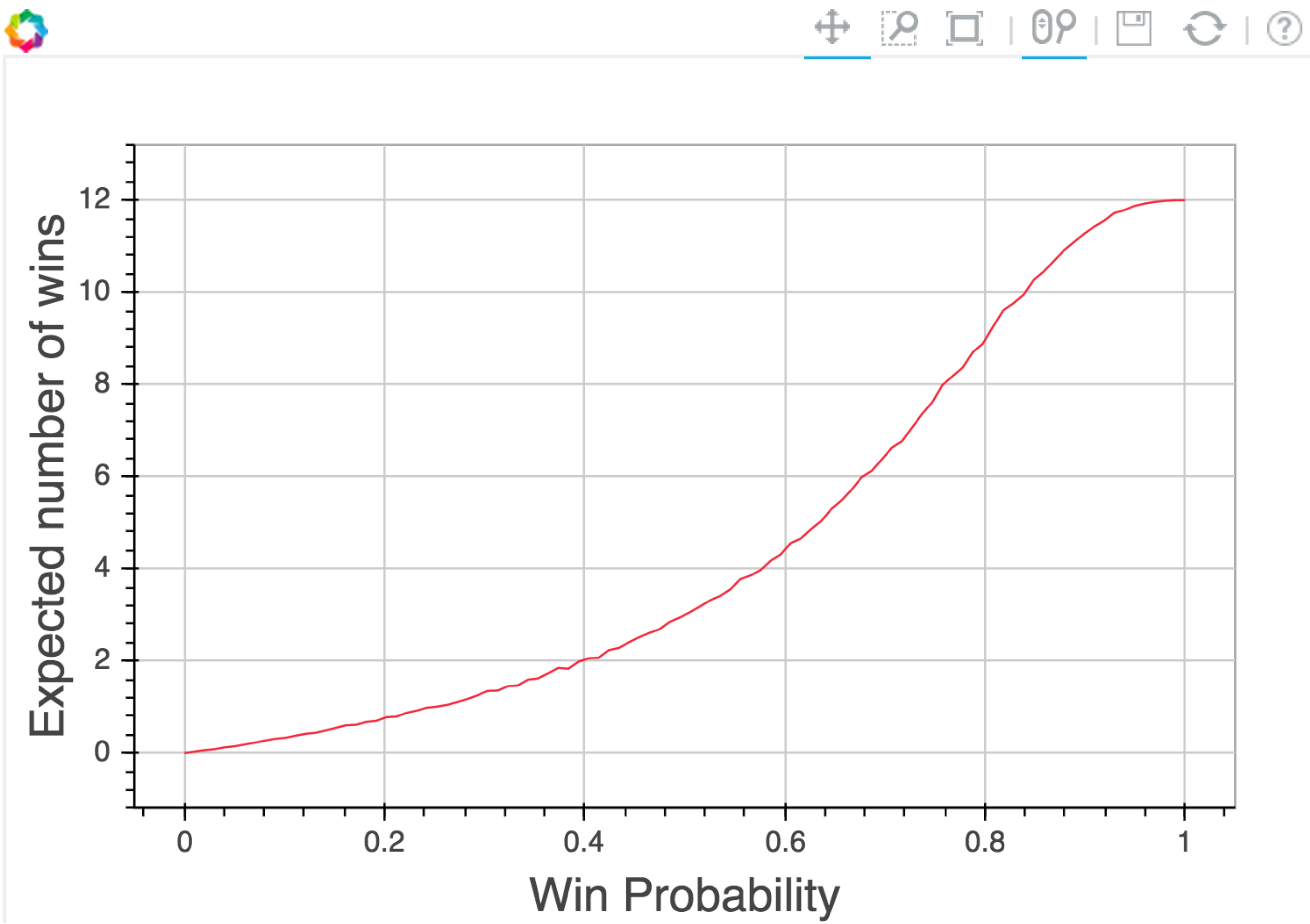
In [7]:
```
@numba.vectorize(nopython=True)
def game_wins(win_probability, max_wins, max_losses):
    wins = 0
    losses = 0
    while wins < max_wins and losses < max_losses:
        if np.random.rand() < win_probability:
            wins += 1
        else:
            losses += 1

    return wins
```

In [21]:
```
sim_input = np.tile(np.linspace(0.0, 1.0, 100), (5000, 1))
sim_results = game_wins(sim_input, 12, 3)
```

In [22]:
```
%timeit game_wins(sim_input, 12, 3)
```

```
10 loops, best of 3: 50 ms per loop
```

Making Ufuncs



Monte Carlo simulating 500,000 tournaments in 50 ms

Generators

```
In [3]: @jit
        def complex_grid(rmin, rmax, nr, imin, imax, ni):
            for r in np.linspace(rmin, rmax, nr):
                for i in np.linspace(imin, imax, ni):
                    yield complex(r, i)
```

```
In [3]: %%timeit
        for z in complex_grid(-2, 2, 100, -2, 2, 100):
            pass
```

```
The slowest run took 243.57 times longer than the fastest. This could mea
n that an intermediate result is being cached
1 loops, best of 3: 1.09 ms per loop
```

```
In [4]: %%timeit
        rmin, rmax, nr = -2, 2, 100
        imin, imax, ni = -2, 2, 100
        for r in np.linspace(rmin, rmax, nr):
            for i in np.linspace(imin, imax, ni):
                z = complex(r, i)
```

```
100 loops, best of 3: 5.03 ms per loop
```

Releasing the GIL

In [22]:
```
from concurrent.futures import ThreadPoolExecutor

@jit(nopython=True)
def mag2(z):
    return z.real * z.real + z.imag * z.imag

MAX_ITERS=250

@jit(nopython=True)
def mandel(c):
    z = 0j
    for i in range(MAX_ITERS):
        z = z*z + c
        if mag2(z) >= 4:
            return 255 * i // MAX_ITERS
    return 255

@jit(nogil=True, nopython=True)
def mandel_patch(args):
    rmin, rmax, nr, imin, imax, ni = args
    points = np.empty(nr*ni, dtype=np.complex128)
    values = np.empty(nr*ni, dtype=np.uint8)

    for i, c in enumerate(complex_grid(rmin, rmax, nr, imin, imax, ni)):
        points[i] = c
        values[i] = mandel(c)

    return points, values
```

Only nopython mode functions can release the GIL

Releasing the GIL

```
In [24]: %%timeit
         with ThreadPoolExecutor(max_workers=1) as executor:
             results = list(executor.map(mandel_patch, patches))

         1 loops, best of 3: 470 ms per loop
```

```
In [25]: %%timeit
         with ThreadPoolExecutor(max_workers=4) as executor:
             results = list(executor.map(mandel_patch, patches))

         10 loops, best of 3: 168 ms per loop
```

2.8x speedup with 4 cores

Too Many Things, Not Enough Time

- NVIDIA GPU Compilation!
  - Write CUDA kernels in Python
  - CUDA Simulator to debug your code in Python interpreter
- Generalized ufuncs (@guvectorize)
- Call ctypes and cffi functions directly and pass them as arguments
- Preliminary support for types that understand the buffer protocol
- "numba annotate" to dump HTML annotated version of compiled code
- See: http://numba.pydata.org/numba-doc/0.20.0/

What Doesn't Work?

(A non-comprehensive list)

- Sets, lists, dictionaries, user defined classes
  (tuples do work!)
- List, set and dictionary comprehensions
- Recursion
- Exceptions with non-constant parameters
- Most string operations
  (buffer support is very preliminary!)
- yield from
- closures inside a JIT function
  (compiling JIT functions inside a closure works…)
- Modifying globals
- Passing an axis argument to numpy array reduction functions

The (Near) Future

(Also a non-comprehensive list)

- "JIT Classes"
- Better support for strings/bytes, buffers, and parsing use-cases
- More coverage of the Numpy API (advanced indexing, etc)
- Documented extension API for adding your own types, low level function implementations, and targets.

Conclusion

- Lots of progress in the past year!
- Try out Numba on your numerical and Numpy-related projects:

  `conda install numba`

- Your feedback helps us make Numba better! Tell us what you would like to see:

  https://github.com/numba/numba

- Stay tuned for more exciting stuff this year…