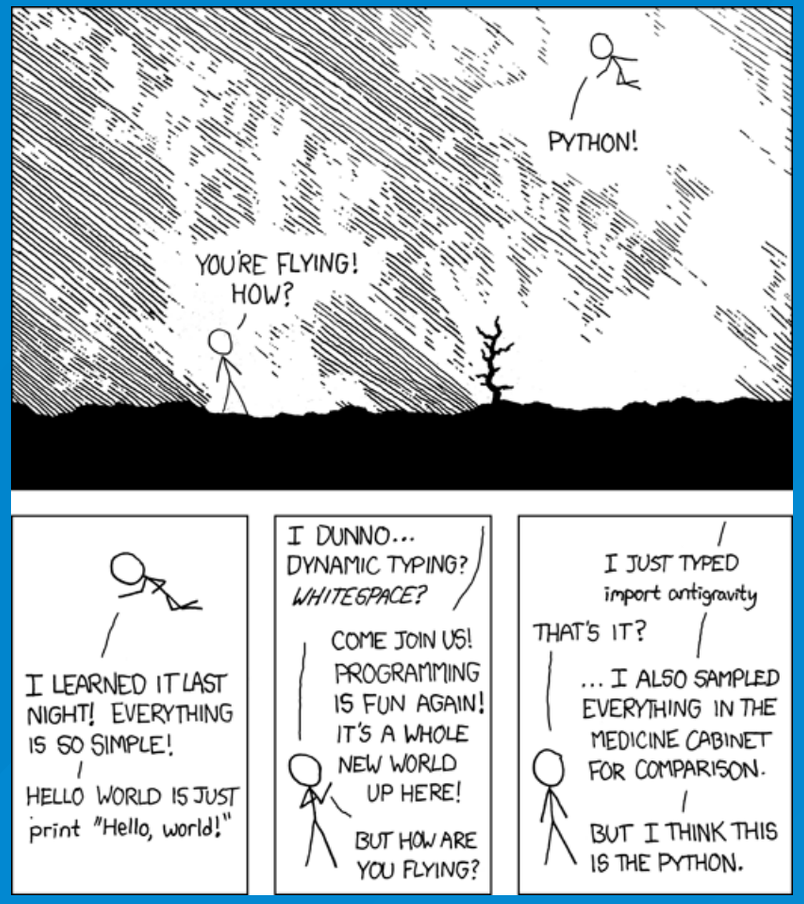
Python in HPC

NIH High Performance Computing Group

staff@hpc.nih.gov

Wolfgang Resch

Why python?



https://xkcd.com/353/

Because Python is simple

```
print("Hello world")
```

```
>>> import this

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
...
```

Because Python is fully featured

Python comes with a full set of **basic data types**, **modules**, error handling and accomodates writing code in **procedural** or **object oriented** style. Also includes some **functional** elements, comprehensions, and advanced features.

Because Python is readable

```
def get_at_content(dna):
    """
    return the AT content of a DNA string.
    The string must be in upper case.
    The AT content is returned as a float
    """
    a_count = dna.count('A')
    t_count = dna.count('T')
    at_content = float(a_count + t_count) / len(dna)
    return at_content
```

Because Python is extensible

- C (C API, cython, ctypes, cffi)
- C++ (boost)
- Fortran (f2py)
- Rust (ctypes, cffi, rust-cpython)

Because Python has many third party libraries, tools, and a large community

Because Python is ubiquitous, portable, and free

- every linux distribution
- most (every?) cluster
- Windows, OS X

A quick word about Python 3 vs 2

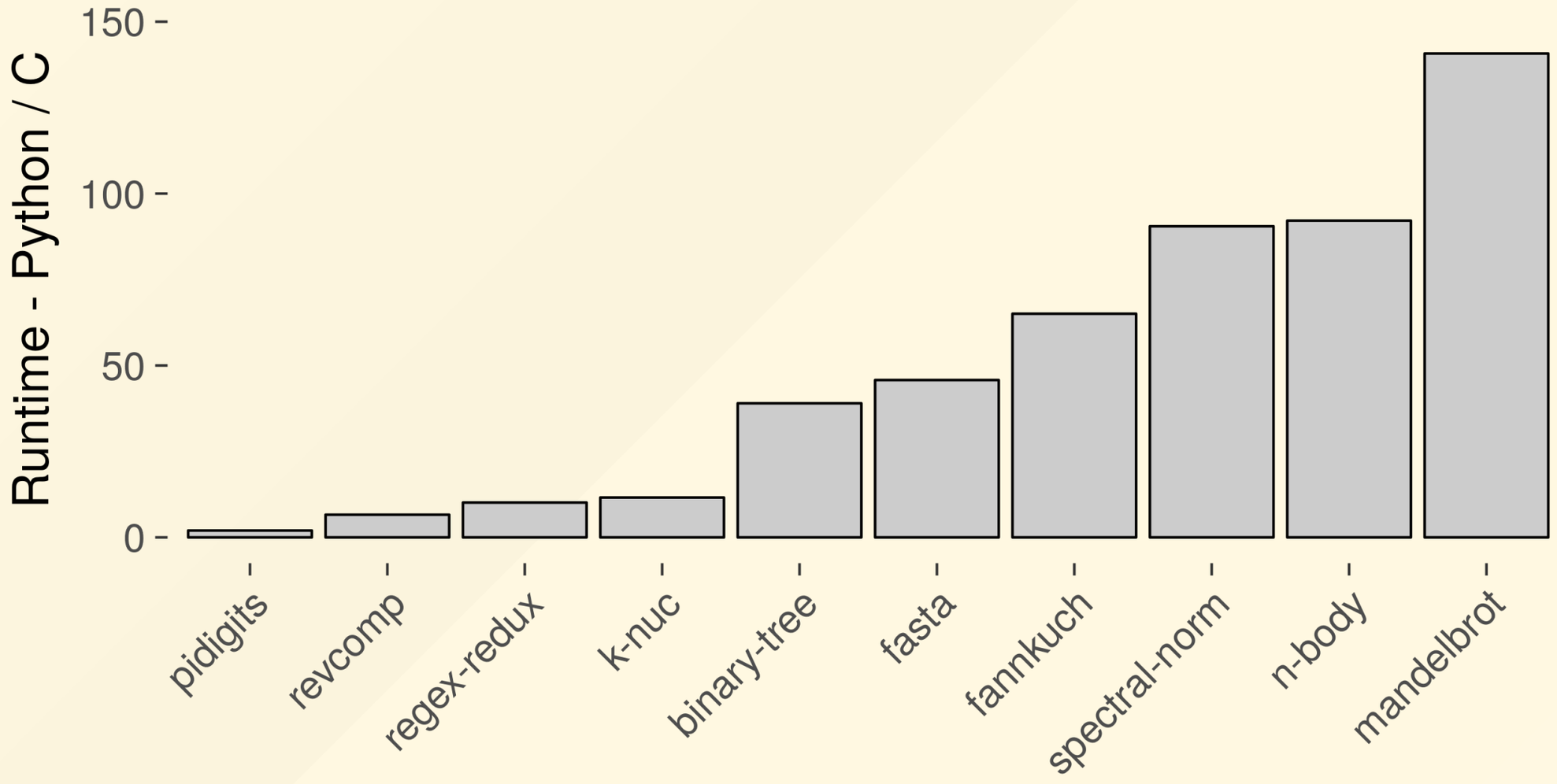
Python 3 made **backwards incompatible** changes (print, exceptions, division, unicode, comprehension variables, open(), ...).

There used to be reasons to write new code in Python 2, but they are getting less compelling.

Soon-ish support for Python 2 by the core Python team and major packages (e.g. numpy) will end.

But Python is slow.
Right?

Well - kind of



data from http://benchmarksgame.alioth.debian.org/u64q

But for a lot of tasks

it doesn't really matter

How do you decide if it matters?

- Is my code fast enough to produce the results I need in the time I have?
- How many CPUh is this code going to waste over its lifetime?
  - How inefficient is it?
  - How long does it run?
  - How often will it run?
- Does it cause problems on the system it's running on?
- How much effort is it to make it run faster?

Luckily, if it does matter for your code

it can often be made faster

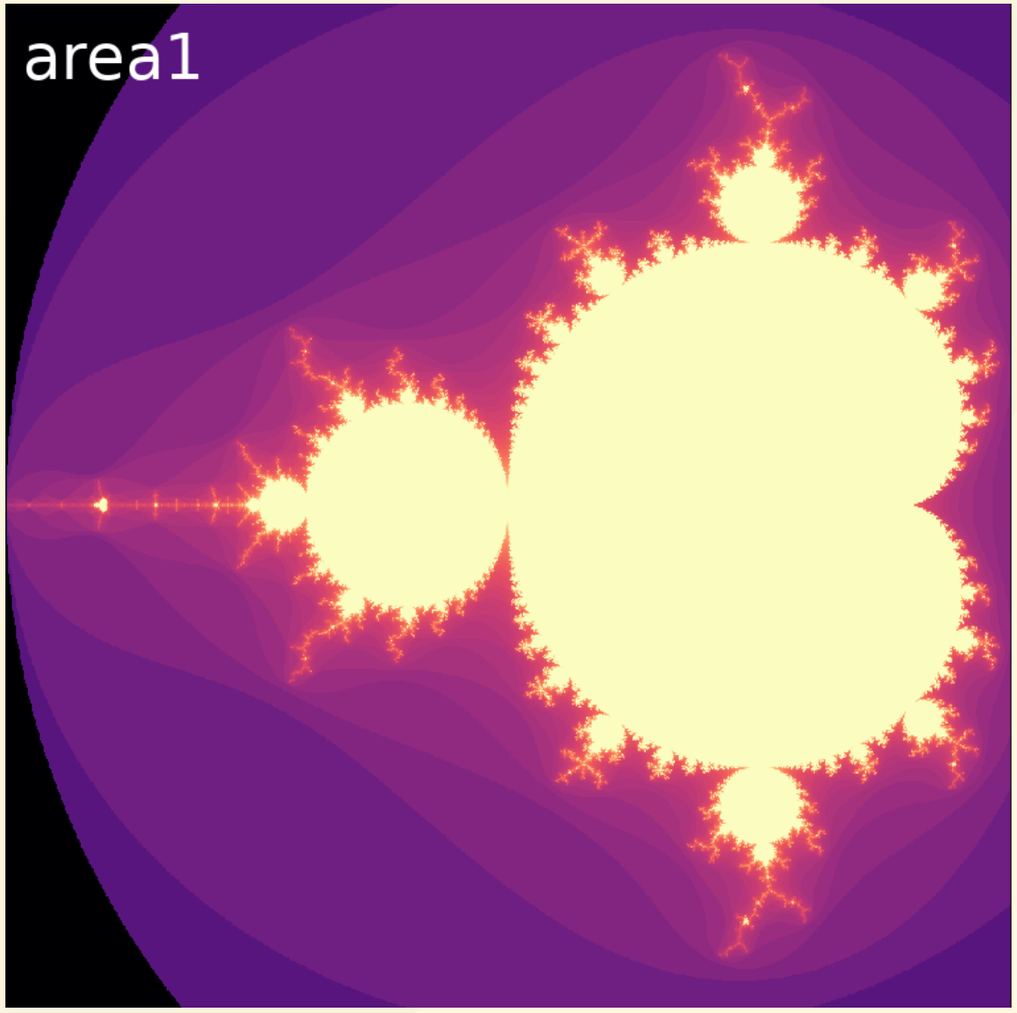
Make it go faster

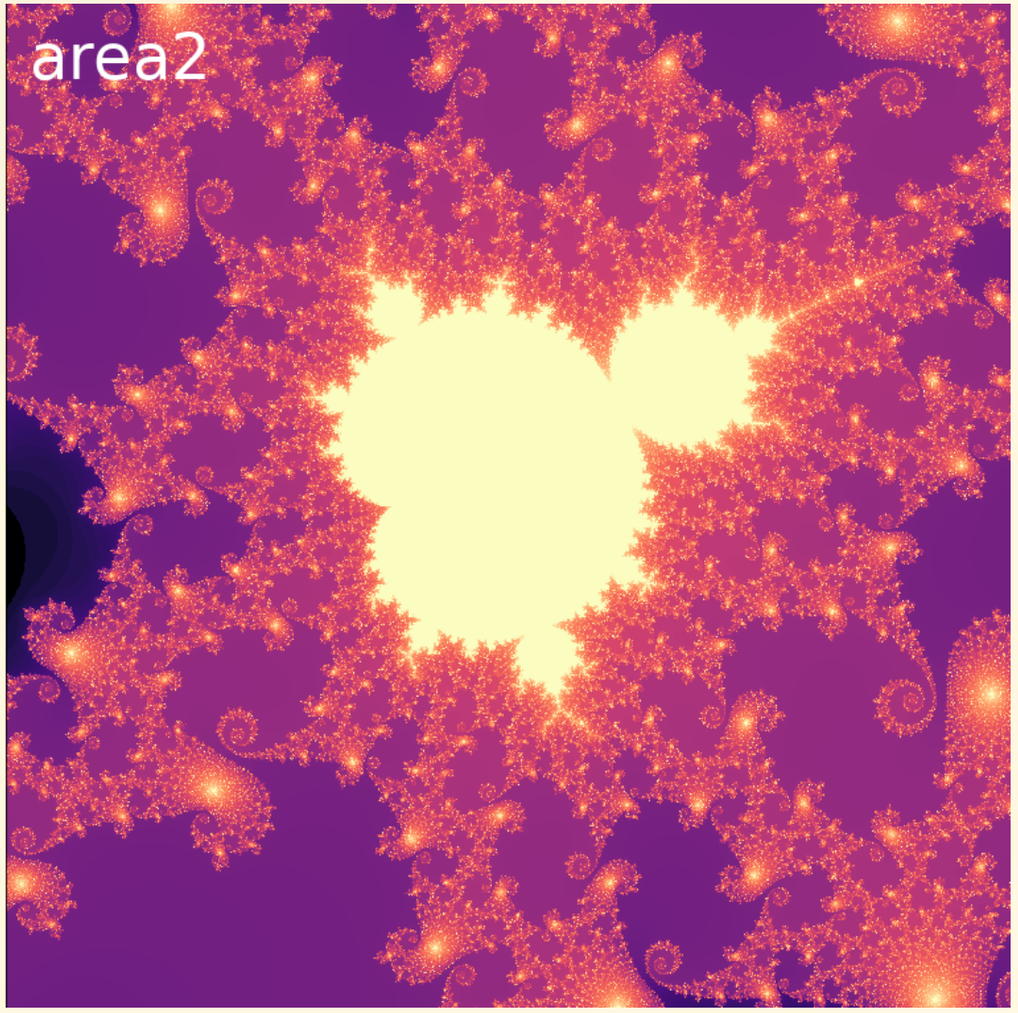
Profile

It's necessary to know what sections of code are bottlenecks in order to improve performance.

Measure - don't guess

Example: Mandelbrot set





https://github.com/NIH-HPC/python-in-hpc

```
def linspace(start, stop, n):
    step = float(stop - start) / (n - 1)
    return [start + i * step for i in range(n)]

def mandel1(c, maxiter):
    z = c
    for n in range(maxiter):
        if abs(z) > 2:
            return n
        z = z*z + c
    return n

def mandel_set1(xmin=-2.0, xmax=0.5, ymin=-1.25, ymax=1.25,
                width=1000, height=1000, maxiter=80):
    r = linspace(xmin, xmax, width)
    i = linspace(ymin, ymax, height)
    n = [[0]*width for _ in range(height)]
    for x in range(width):
        for y in range(height):
            n[y][x] = mandel1(complex(r[x], i[y]), maxiter)
    return n
```

Baseline timing with `%timeit` ipython magic:

```
In [4]: %timeit mandel_set1()
7.77 s ± 26.8 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Equivalently, on the command line:

```
$ python -m timeit -s 'import mandel01' 'mandel01.mandel_set1()'
10 loops, best of 3: 7.81 sec per loop
```

So ~8s to calculate *area1*.

Profiling with `%prun` ipython magic:

```
In [4]: %prun -s cumulative mandel_set1()
```

or the equivalent command line below. This, however, requires an executable script.

```
$ python -m cProfile -s cumulative mandel01.py
```

Yields the following results:

```
25214601 function calls in 12.622 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000   12.622   12.622 {built-in method builtins.exec}
        1    0.013    0.013   12.622   12.622 mandel01.py:1(<module>)
        1    0.941    0.941   12.609   12.609 mandel01.py:13(mandel_set1)
  1000000    9.001    0.000   11.648    0.000 mandel01.py:5(mandel1)
 24214592    2.647    0.000    2.647    0.000 {built-in method builtins.abs}
```

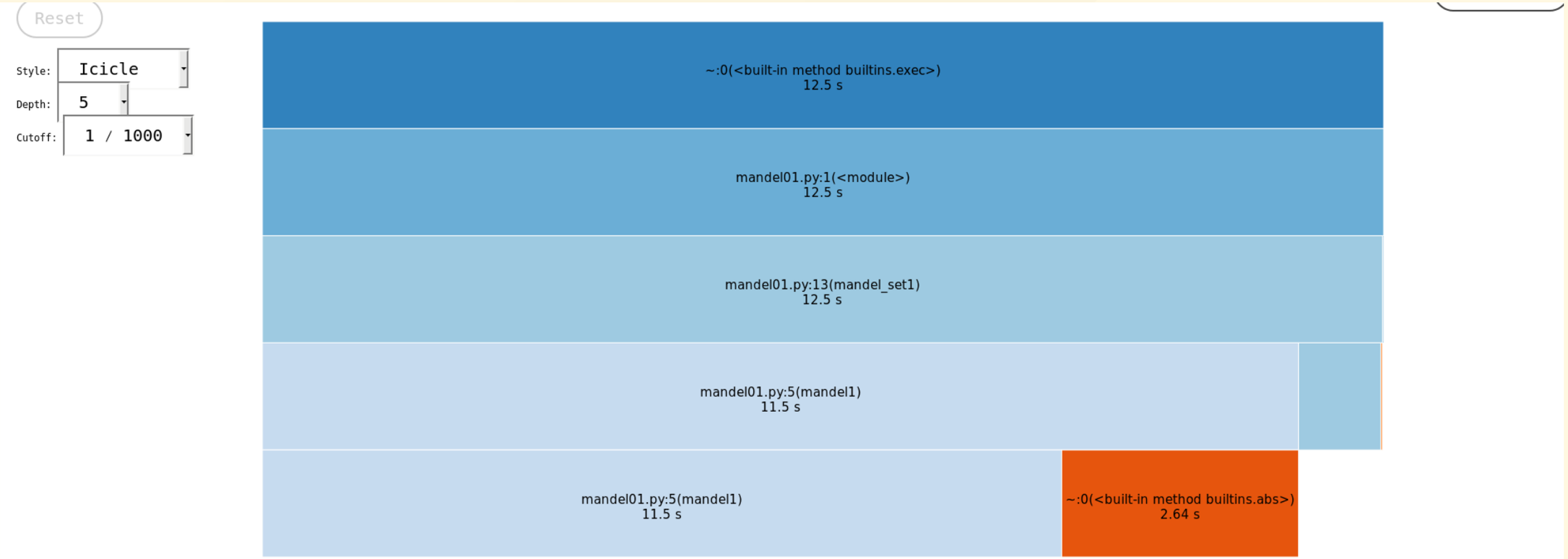
- Most time is spent in the `mandel1` function
- profiling introduces some overhead (runtime of 12s vs 8s)

Profiling results can be visualized with SnakeViz:

```
$ python -m cProfile -o mandel01.prof mandel01.py
$ snakeviz --port 6542 --hostname localhost --server mandel01.prof
```

Which starts a web server on port 6542.

Snakeviz generates an interactive visualization and sortable table:



ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.01213	0.01213	12.49	12.49	mandel01.py:1(<module>)

Most the time is spent in the `mandel1()` function. Use the `line_profiler` package to profile this function line by line.

Using `%lprun` ipython magic:

```
%load_ext line_profiler
%lprun -f mandel1 mandel_set1()
```

To do the equivalent on the command line, import the `line_profiler` package and decorate the function(s) to profile with `@profile`:

```
import line_profiler

@profile
def mandel1(c, maxiter):
    z = c
    for n in range(maxiter):
        if abs(z) > 2:
            return n
        z = z*z + c
    return n
...
```

Then, on the command line:

```
$ kernprof -l -v mandel01.py
```

The line-by-line profile returned by either method:

```
Total time: 42.2215 s
File: mandel01.py
Function: mandel1 at line 7

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     7                                           @profile
     8                                           def mandel1(c, maxiter):
     9   1000000       441285      0.4      1.0      z = c
    10  24463110     11668783      0.5     27.6      for n in range(maxiter):
    11  24214592     16565164      0.7     39.2          if abs(z) > 2:
    12    751482       345196      0.5      0.8              return n
    13  23463110     13081688      0.6     31.0          z = z*z + c
    14    248518       119431      0.5      0.3      return n
```

There are some algorithmic improvements possible here, but let's first try the simplest thing we can do.

Improve sequential performance

Using the `numba` just in time (jit) compiler 2

```
from numba import jit

@jit
def mandel2(c, maxiter):
    z = c
    for n in range(maxiter):
        if abs(z) > 2:
            return n
        z = z*z + c
    return n
```

No changes to the code - just decorate the function in the tight loop with a `@jit` results in a **7-fold** speedup when calculating *area1*

```
In[4]: %timeit mandel_set2()
1.13 s ± 23.7 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Converting mandel_set to numpy arrays 3

Now mandel_set is the bottleneck. Since it uses nested lists, `numba` can't jit compile it. Can we speed it up by converting to numpy arrays?

```
def mandel_set3(xmin=-2.0, xmax=0.5, ymin=-1.25, ymax=1.25,
                width=1000, height=1000, maxiter=80):
    r = np.linspace(xmin, xmax, width)
    i = np.linspace(ymin, ymax, height)
    n = np.empty((height, width), dtype=int)
    for x in range(width):
        for y in range(height):
            n[y, x] = mandel3(complex(r[x], i[y]), maxiter)
    return n
```

No - it's actually slower now on *area1*.

```
In[4]: %timeit mandel_set3()
1.43 s ± 53.9 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

numpy arrays have some overhead that may hurt performace with smaller array sizes. But now the function can be jit compiled with numba by decorating it with the `@jit` decorator. 4

```
In[4]: %timeit mandel_set4()
467 ms ± 143 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Now the speedup is **17-fold**.

```
Total time: 42.2215 s
File: mandel01.py
Function: mandel1 at line 7

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     7                                           @profile
     8                                           def mandel1(c, maxiter):
     9   1000000       441285      0.4      1.0      z = c
    10  24463110     11668783      0.5     27.6      for n in range(maxiter):
    11  24214592     16565164      0.7     39.2          if abs(z) > 2:
    12    751482       345196      0.5      0.8              return n
    13  23463110     13081688      0.6     31.0          z = z*z + c
    14    248518       119431      0.5      0.3      return n
```

Algorithmic improvement 5

Based on the definitions of the absolute value and the square of a complex number, we can factor out some calculations in the `mandel` method:

```
@jit
def mandel5(creal, cimag, maxiter):
    real = creal
    imag = cimag
    for n in range(maxiter):
        real2 = real*real
        imag2 = imag*imag
        if real2 + imag2 > 4.0:
            return n
        imag = 2 * real*imag + cimag
        real = real2 - imag2 + creal
    return n
```

Which gives us a respectable

```
In[4]: %timeit mandel_set5()
104 ms ± 173b µs per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

75-fold improvement over the original pure python implementation.

Cython

Cython is a python-like language that can be compiled to a C extension for python. In Ipython/Jupyter notebooks using cython is as simple as

```
In[4]: %load_ext cython
```

The `mandel` function in cython 6

```
%%cython
import cython
import numpy as np

cdef int mandel6(double creal, double cimag, int maxiter):
    cdef:
        double real2, imag2
        double real = creal, imag = cimag
        int n

    for n in range(maxiter):
        real2 = real*real
        imag2 = imag*imag
        if real2 + imag2 > 4.0:
            return n
        imag = 2* real*imag + cimag
        real = real2 - imag2 + creal;
    return n
```

The `mandel_set` function in cython

```
@cython.boundscheck(False)
@cython.wraparound(False)
cpdef mandel_set6(double xmin, double xmax, double ymin, double ymax,
                  int width, int height, int maxiter):
    cdef:
        double[:] r1 = np.linspace(xmin, xmax, width)
        double[:] r2 = np.linspace(ymin, ymax, height)
        int[:,:] n = np.empty((height, width), np.int32)
        int i,j

    for i in range(width):
        for j in range(height):
            n[j,i] = mandel6(r1[i], r2[j], maxiter)
    return n
```

```
In[4]: %timeit mandel_set6(-2, 0.5, -1.25, 1.25, 1000, 1000, 80)
103 ms ± 2.31 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

So cython runs as fast as the numba version at the cost of changing code.

GPU with pyOpenCL 7

Using `pyopencl` to implement the `mandel` function with an NVIDIA K80 backend, the following timing was measured (for code see the notebook on GitHub):

```
In[4]: %timeit mandel_set7(-2, 0.5, -1.25, 1.25, 1000, 1000, 80)
26.9 ms ± 1.05 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

A **278-fold** decrease in runtime compared to the pure python implementation.

Fortran 8

Fortran bindings can be created with the numpy utility `f2py`. How does fortran stack up against
numba and cython?

```
$ cat mandel8.f90
subroutine mandel_set8(xmin, xmax, ymin, ymax, width, height, itermax, n)
    real(8), intent(in)   :: xmin, xmax, ymin, ymax
    integer, intent(in)   :: width, height, itermax
    integer               :: niter
    integer, dimension(width, height), intent(out) :: n
    integer               :: x, y
    real(8)               :: xstep, ystep

    xstep = (xmax - xmin) / (width - 1)
$ f2py -m mb_fort -c mandel8.f90 --fcompiler=gnu95
...
```

```
In[4]: from mb_fort import mandel8, mandel_set8
In[5]: %timeit mandel_set8(-2, 0.5, -1.25, 1.25, 1000, 1000, 80)
114 ms ± 3.11 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

about 10% slower than the numba and cython implementations.

Improve sequential performance - Summary

	Implementation	area1 time	speedup	area2 time	speedup
1	pure python	7.770s	1x	174.00s	1x
2	numba 1	1.130s	7x	11.70s	15x
3	+ numpy	1.430s	5x	11.90s	15x
4	+ numba 2	0.467s	17x	10.80s	16x
5	+ algo	0.104s	75x	2.67s	64x
8	f2py	0.114s	68x	2.67s	64x
6	cython	0.103s	75x	2.63s	66x
7	pyopencl	0.028s	228x	0.06s	3047x

Was that really all sequential?

Numpy can be compiled against different backends. Some of them, like MKL and OpenBlas, implement implicit parallelism for some operations. We use Anaconda python which now uses MKL, so some of the numpy code could have been implicitly parallel.

```
import numpy
numpy.show_config()
```

```
lapack_opt_info:
    define_macros = [('SCIPY_MKL_H', None), ('HAVE_CBLAS', None)]
    library_dirs = ['/usr/local/Anaconda/envs/py3.5/lib']
    include_dirs = ['/usr/local/Anaconda/envs/py3.5/include']
    libraries = ['mkl_intel_lp64', 'mkl_intel_thread', 'mkl_core', 'iomp5', 'pt
...
```

Parallelize - within a single machine

A word about the Python interpreter

Python only allows a singly thread to execute Python bytecode at any one time. Access to the interpreter is enforced by the **Global Interpreter Lock** (GIL).

While this is sidestepped by I/O, it does prevent true parallelism with pure Python threads.

However, compiled extension modules can thread and other paradigms for parallelism have developed.

numba.vectorize 9

`numba.vectorize` and `numba.guvectorize` are convenience decorators for creating numpy ufuncs that can be single threaded, parallel, or use GPU for computation. Parallel computation uses threads.

```
@vectorize([int32(complex64, int32)], target='parallel')
def mandel9(c, maxiter):
    nreal = 0
    real = 0
    imag = 0
    for n in range(maxiter):
        nreal = real*real - imag*imag + c.real
        imag = 2* real*imag + c.imag
        real = nreal;
        if real * real + imag * imag > 4.0:
            return n
    return n

def mandel_set9(xmin=-2.0, xmax=0.5, ymin=-1.25, ymax=1.25,
                width=1000, height=1000, maxiter=80):
    r1 = np.linspace(xmin, xmax, width, dtype=np.float32)
    r2 = np.linspace(ymin, ymax, height, dtype=np.float32)
    c = r1 + r2[:,None]*1j
    n = mandel9(c,maxiter)
    return n
```

	Implementation	area2 time	speedup	efficiency
1	pure python	174.00s	1x	NA
5	numba + algo	2.67s	64x	NA
9	vectorize, 1 thread	2.72s	64x	100%
	vectorize, 2 threads	1.54s	113x	88%
	vectorize, 4 threads	1.08s	161x	63%
	vectorize, 8 threads	0.60s	290x	57%
	vectorize, 14 threads	0.42s	420x	47%

With one thread (set with `NUMBA_NUM_THREADS`) it matches the best sequential implementation. More threads improve upon this, but scaling isn't great.

multiprocessing

Multiprocessing uses subprocesses rather than threads for parallelism to get around the GIL

Each child inherits the state of the parent

After the fork data has to be shared explicitly via interprocess communication

Spawning child processes and sharing data have overhead

The multiprocessing API is similar to the threading API

multiprocessing - things to watch out for

If each child is also doing (implicit) threading, care has to be taken to limit $\frac{nthreads}{child} \times children$ to the number of available CPUs

Don't use `multiprocessing.cpu_count()` - it returns all CPUs on the node

Make children ignore `SIGINT` and parent handle it gracefully

Script's main should to be safely importable - less important on linux

multiprocessing the Mandelbrot set 🔟

Create one to many subprocesses and execute CPU bound computations on each independently. In this example we'll see a `multiprocessing.Pool` of worker processes each processing one row of the Mandelbrot set.

```
@jit
def mandel10(creal, cimag, maxiter):
    real = creal
    imag = cimag
    for n in range(maxiter):
        real2 = real*real
        imag2 = imag*imag
        if real2 + imag2 > 4.0:
            return n
        imag = 2 * real*imag + cimag
        real = real2 - imag2 + creal
    return n

@jit
def mandel10_row(args):
    y, xmin, xmax, width, maxiter = args
    r = np.linspace(xmin, xmax, width)
    res = [0] * width
    for x in range(width):
        res[x] = mandel10(r[x], y, maxiter)
    return res
```

```
def mandel_set10(ncpus=1, xmin=-2.0, xmax=0.5, ymin=-1.25,
                 ymax=1.25, width=1000, height=1000, maxiter=80):
    i = np.linspace(ymin, ymax, height)
    with mp.Pool(ncpus) as pool:
        n = pool.map(mandel10_row, ((a, xmin, xmax, width, maxiter) for a in i)
    return n
```

How is the performance on *area2*?

	Implementation	area2 time	speedup	efficiency
1	pure python	174.00s	1x	NA
5	numba + algo	2.67s	64x	NA
10	multiproc, pool(1)	3.28s	53x	100%
	multiproc, pool(2)	1.90s	92x	86%
	multiproc, pool(4)	1.30s	134x	63%
	multiproc, pool(8)	1.08s	161x	37%
	multiproc, pool(14)	1.16s	150x	20%

Slighly worse than implementation 5 with 1 CPU and not scaling well.
This problem is not very suited to multiprocessing.

Parallelize - across machines

MPI

the Message Passing Interface is a portable, and performant standard for communication between processes (tasks) within and between compute nodes. Communication can be point-to-point or collective (broadcast, scatter, gather, ...).

`mpi4py` is a Python implementation of MPI. Documentation is available on readthedocs and at the scipy mpi4py site

MPI - point-to-point communication

```
from mpi4py import MPI
import numpy
import time

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

# pass data type explicitly for speed and use upper case 'Send' / 'Recv'
if rank == 0:
    data = numpy.arange(100, dtype = 'i')
    comm.Send([data, MPI.INT], dest=1)
    print "Rank 0 sent numpy array"
if rank == 1:
    data = numpy.empty(100, dtype='i')
    comm.Recv([data, MPI.INT], source=0)
    print "Rank 1 received numpy array"
```

MPI - point-to-point communication

```
#!/bin/bash
#SBATCH --ntasks=2
#SBATCH --ntasks-per-core=1

module load openmpi/1.10.0/gcc-4.4.7
module load python/2.7
mpiexec  ./numpy_p2p.py
```

MPI - Mandelbrot set 1️⃣1️⃣

Each mpi task will process a consecutive chunk of rows using the functions jit compiled with numba.

```
from mpi4py import MPI
import numpy as np
from numba import jit

comm  = MPI.COMM_WORLD
size  = comm.Get_size()
rank  = comm.Get_rank()
```

MPI - Mandelbrot set

```
# how many rows to compute in this rank?
N = height // size + (height % size > rank)
N = np.array(N, dtype='i')  # so we can gather it later on

# what slice of the whole should be computed in this rank?
start_i = comm.scan(N) - N
start_y = ymin + start_i * dy
end_y   = ymin + (start_i + N - 1) * dy

# calculate the local results - using numba.jit **without parallelism**
Cl = mandel_set(xmin, xmax, start_y, end_y, width, N, maxiter)
```

MPI - Mandelbrot set

```
rowcounts = 0
C         = None
if rank == 0:
    rowcounts = np.empty(size, dtype='i')
    C = np.zeros([height, width], dtype='i')

comm.Gather(sendbuf = [N, MPI.INT],
            recvbuf = [rowcounts, MPI.INT],
            root    = 0)

comm.Gatherv(sendbuf = [Cl, MPI.INT],
             recvbuf = [C, (rowcounts * width, None), MPI.INT],
             root    = 0)
```

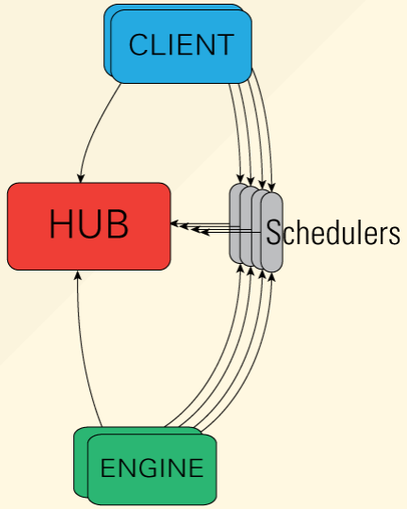
MPI - Mandelbrot set

Testing the MPI code on a higher resolution (32k x 32k) of *area2* and comparing it to the `numba.vectorize` thread parallel implementation which is limited to a single node

threads/tasks	threaded	MPI	Nodes
4	1110s	838s	1
8	620s	341s	1
16	342s	211s	4
32	179s	105s	4
64	ND	94s	10

ipyparallel

> ipyparallel enables all types of parallel applications to be developed, executed, debugged and monitored interactively



Documentation - GitHub

Spark

> Apache Spark™ is a fast and general engine for large-scale data processing.

Key idea: Resilient Distributed Datasets (RDDs) are collections of objects across a cluster that can be computed on via parallel transformations (map, filter, ...).

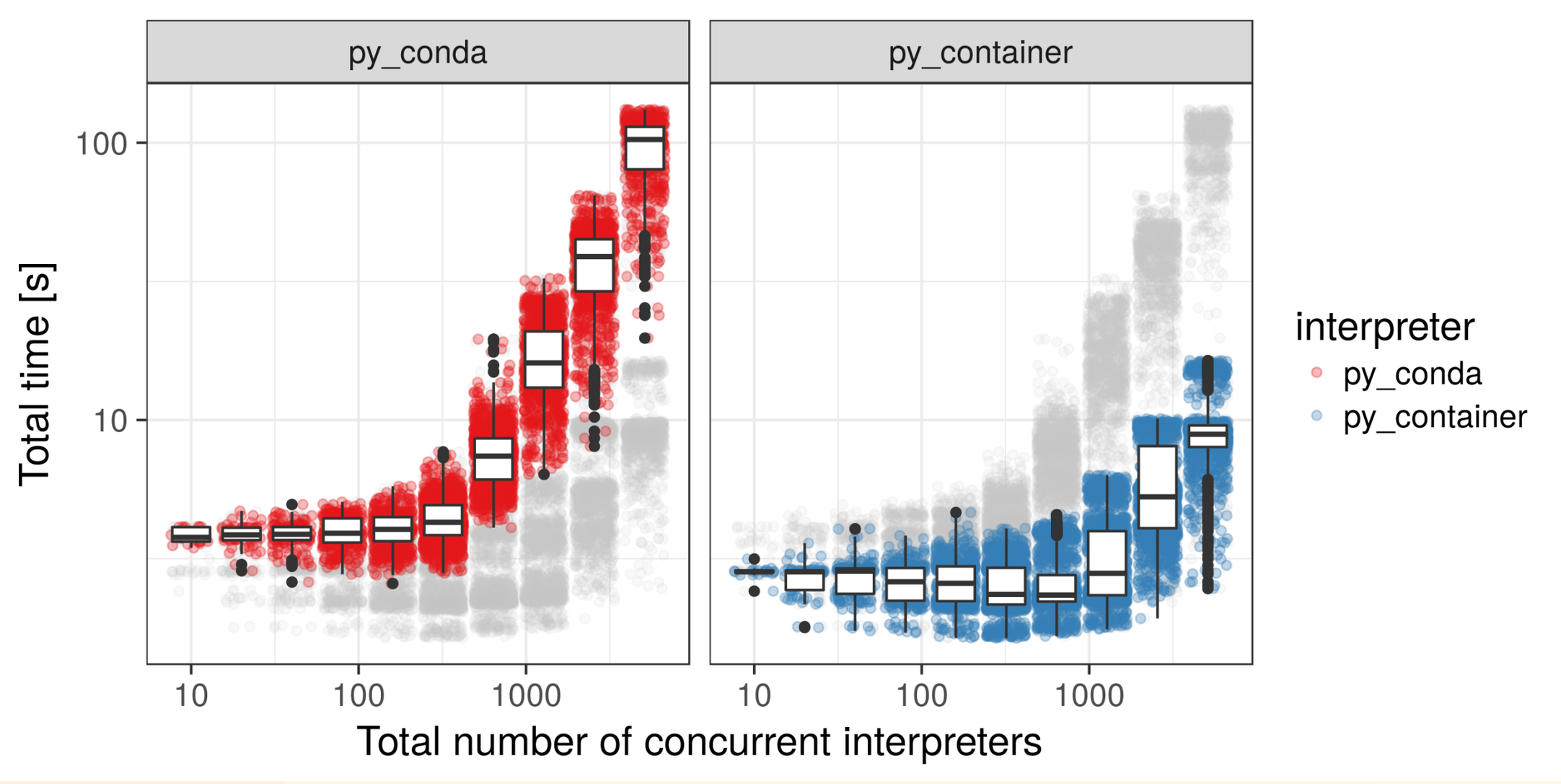
There are APIs in a number of languages: Python, R, Java, and Scala.

Home page

Python import problem

During startup python does a lot of small file operations as it locates all the files it needs. These metadata heavy operations can strain the file systems if many of them happen at the same time.

Python import problem



Python import problem

One solution is to containerize the python interpreter like in the example above.

Other solutions:

- For mpi processes: import in the root process and share files via MPI
- static python builds
- cache the shared objects / python packages

NERSC talk - NERSC paper - Python MPI bcast - Static python - Scalable python

Python on Biowulf

Main python modules

```
$ module load python/2.7
$ module load python/3.4
$ module load python/3.5
```

Packages in these environments are updated regularly. To see what packages are installed use

```
$ module load python/2.7
$ conda list
$ conda list numba
```

Conda environments: Create your own

```
$ module load Anaconda
$ conda create -n myenv python=3.5
```

Activate the environment and install packages with conda or pip

```
$ source activate myenv
$ which pip
~/.conda/envs/myenv/bin/pip
$ conda install numpy
$ pip install click
```

Conda environments: Default location

By default, new environments will be in `$HOME/.conda/envs`. This can be changed with

```
$ conda config --prepend envs_dirs /data/$USER/envs
```

Conda environments: A private Anaconda install

```
$ wget https://repo.continuum.io/miniconda/Miniconda3-latest-Linux-x86_64.sh
$ bash Miniconda3-latest-Linux-x86_64.sh -b -p /path/to/my/conda
$ export PATH=/path/to/my/conda/bin:$PATH
$ which python
/path/to/my/conda/bin/python
$ conda install numpy
...
```

Python science stack

Numeric: numpy, scipy, pandas, statsmodel, numba
Scientific: biopython, astropy, nipy.org
Visualization: matplotlib, seaborn, ggplot, bokeh
Machine learning: scikit learn, tensorflow, theano, keras
Tools: jupyter, ipython, conda, bioconda

More links

https://hpc.nih.gov/docs/python.html
https://hpc.nih.gov/apps/jupyter.html

Other talks:

HPC Python / TACC
HPC Python / Blue Waters
HPC Python / Exascale.org
HPC Python / PRACE
HPC Python tutorial

staff@hpc.nih.gov


Steve Bailey


Steven Fellini, Ph.D.


Susan Chacko, Ph.D.

Picture unavailable

Afif Elghraoui


Ainsley Gibson


David Hoover, Ph.D.


Patsy Jones

Picture unavailable

Charles Lehr


Jean Mao, Ph.D.


Tim Miller


Charlene Osborn


Mark Patkus

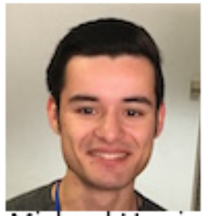

Dan Reisman


Wolfgang Resch, Ph.D.


Jerez Te, Ph.D.


Rick Troxel


Sylvia Wilkerson


Michael Harris (intern)