Profiling/Optimization

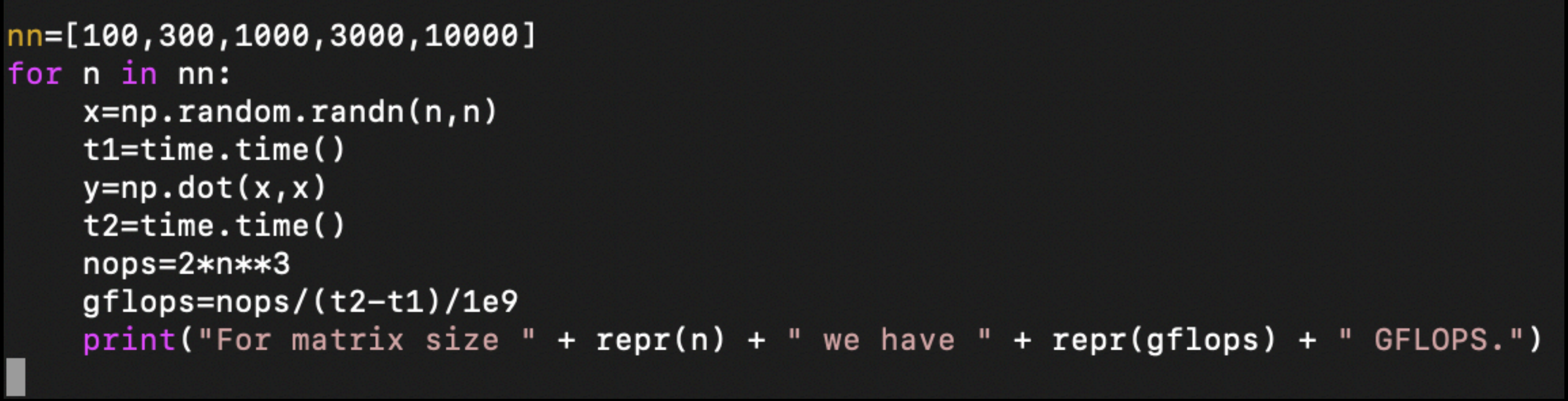
Do You Want Your Code to Run Fast?

- hopefully yes!
- Algorithms can make a huge difference - FFT vs. (slow) DFT.
- Even with fast algorithm, you still win if you code it well.

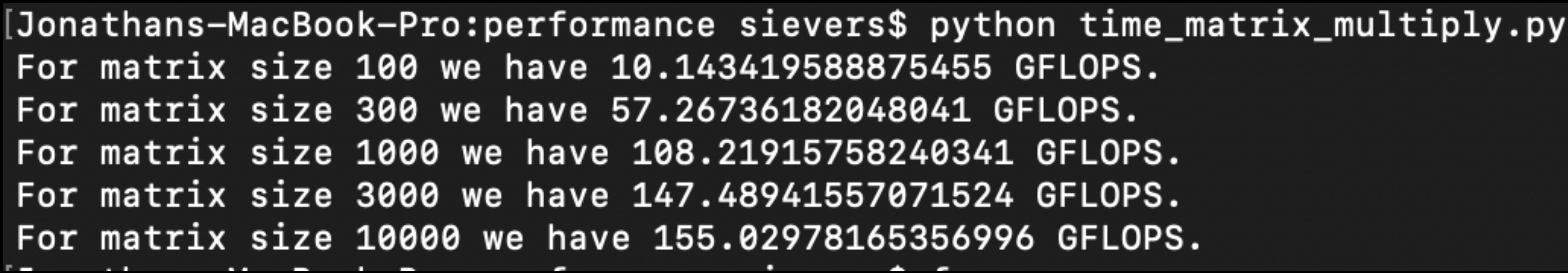
For a nice example of a detailed case study, see:
https://ppc.cs.aalto.fi/ch2/

What Takes Time?

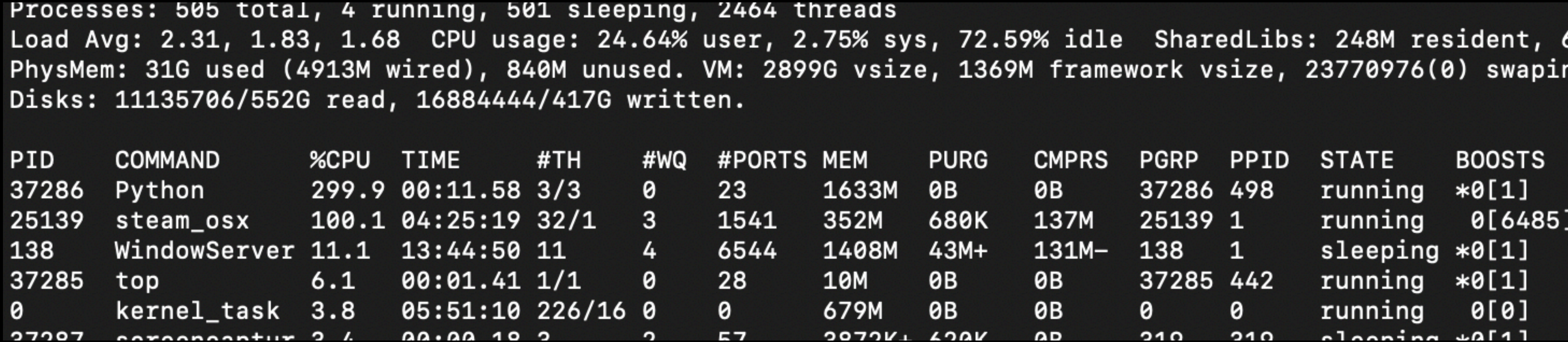
- It’s very useful to have an idea of how long things take.
- Floating point - AVX512 e.g. lets CPU crunch on 512 bits at a time. For 64 bit numbers, this is 8 operations per cycle. Often FMAD (fused multiply-add) does 1 multiply and 1 add in one cycle. 4 cores*8 FMADS/cycle*2 ops/FMAD*3 GHz = 200 GFLOPS.
- This is a hard limit - we’ll never go faster than this.



```
nn=[100,300,1000,3000,10000]
for n in nn:
    x=np.random.randn(n,n)
    t1=time.time()
    y=np.dot(x,x)
    t2=time.time()
    nops=2*n**3
    gflops=nops/(t2-t1)/1e9
    print("For matrix size " + repr(n) + " we have " + repr(gflops) + " GFLOPS.")
```



```
[Jonathans-MacBook-Pro:performance sievers$ python time_matrix_multiply.py
 For matrix size 100 we have 10.143419588875455 GFLOPS.
 For matrix size 300 we have 57.26736182048041 GFLOPS.
 For matrix size 1000 we have 108.21915758240341 GFLOPS.
 For matrix size 3000 we have 147.48941557071524 GFLOPS.
 For matrix size 10000 we have 155.02978165356996 GFLOPS.
[Jonathans-MacBook-Pro:performance sievers$ top
```



```
Processes: 505 total, 4 running, 501 sleeping, 2464 threads
Load Avg: 2.31, 1.83, 1.68  CPU usage: 24.64% user, 2.75% sys, 72.59% idle  SharedLibs: 248M resident, 6
PhysMem: 31G used (4913M wired), 840M unused. VM: 2899G vsize, 1369M framework vsize, 23770976(0) swapin
Disks: 11135706/552G read, 16884444/417G written.

PID    COMMAND      %CPU  TIME     #TH    #WQ  #PORTS MEM    PURG   CMPRS  PGRP  PPID  STATE    BOOSTS
37286  Python       299.9 00:11.58 3/3    0    23     1633M  0B     0B     37286 498   running  *0[1]
25139  steam_osx    100.1 04:25:19 32/1   3    1541   352M   680K   137M   25139 1     running   0[6485]
138    WindowServer 11.1  13:44:50 11     4    6544   1408M  43M+   131M-  138   1     sleeping *0[1]
37285  top          6.1   00:01.41 1/1    0    28     10M    0B     0B     37285 442   running  *0[1]
0      kernel_task  3.8   05:51:10 226/16 0    0      679M   0B     0B     0     0     running   0[0]
37287  screencaptur 3.4   00:00.18 3      2    57     3872K+ 620K   0B     319   319   sleeping *0[1]
```

Only using 3 cores, so expect ~150 GFLOPS.
Matrix multiplies can be very efficient!

Memory Bandwith

- Another bottleneck - data has to get to and from memory to the CPU.
- A few 10s of GBs is (often) optimistic here. (see memcpy_numba.py)
- How many numbers per second can I get to my CPU(s)?
- How many operations do I need to do while they're there to be floating-point limited?
- Say 50 GB/s, 16 bytes (2 numbers), 3 billion #'s/s. At 192 GFLOPS, need to do 64 floating point operations.

Matrix Multiply

- If I wrote matrix multipy as:

-
  ```
  for i in range(n):
    for j in range(m):
      for ii in range(k):
        c[i,j]=c[i,j]+a[i,ii]*b[ii,j]
  ```

- Every element of a&b gets pulled through i times. For 10k by 10k, this is $2*10000^3*8$ bytes = 16TB. At 50 GB/s, this takes 16*(1000/50)=320s. 30x longer than it actually took.

Latency/Memory Order

- Memory latency is a performance issue as well. It takes a long time to get a value from RAM.

- We often use multiple memory values in a row. The CPU will usually load values after the one we request so it’s ready (see “cache lines”).

- Write code to take advantage of this. In python/numpy, you (usually) get the next entry by increasing the *last* index.

Moral of the Story (so far)

- When you write code, think about operations *and* bandwidth.
- Code you write will almost certainly be bandwidth-limited.
- Try to use your data in order (look at numba_2d_sum.py).

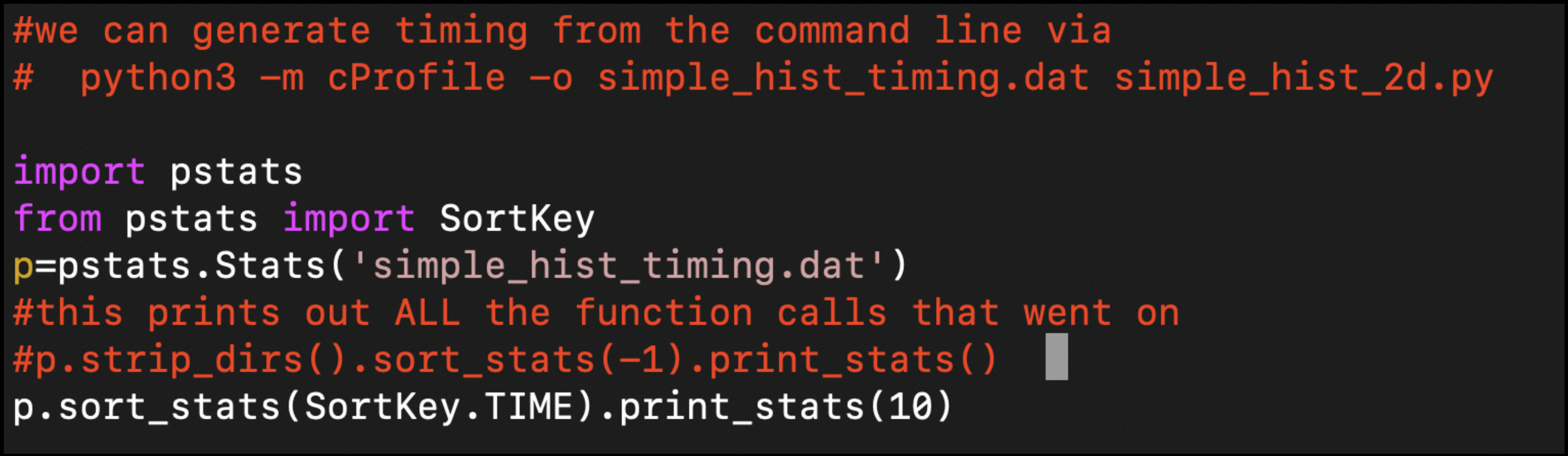
Profiling

- There are various tools (for pretty much all languages) to tell you how long your code is taking.

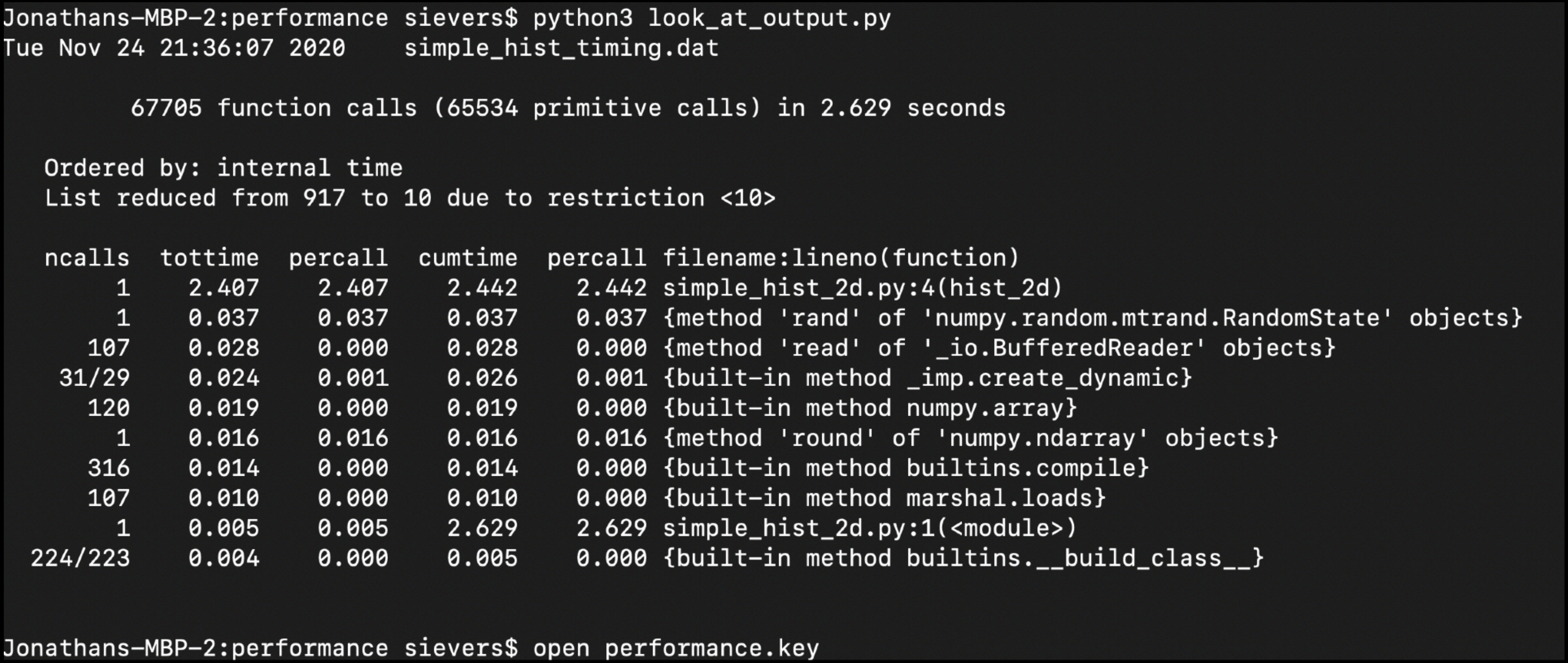
- They often track which functions are called, how many times, and how long each one takes.

- profile/cProfile can help with this.

- Always profile before you start optimizing - it can be non-obvious which parts are slow, and effort speeding up fast code is (mostly) wasted.

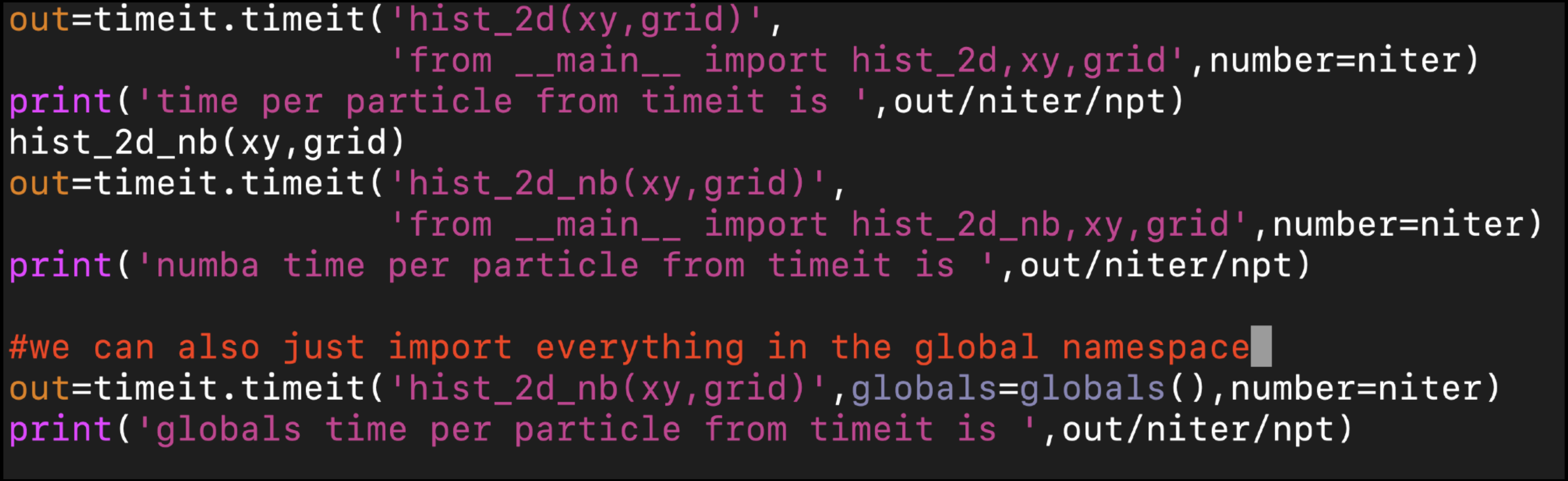


```
#we can generate timing from the command line via
#  python3 -m cProfile -o simple_hist_timing.dat simple_hist_2d.py

import pstats
from pstats import SortKey
p=pstats.Stats('simple_hist_timing.dat')
#this prints out ALL the function calls that went on
#p.strip_dirs().sort_stats(-1).print_stats()
p.sort_stats(SortKey.TIME).print_stats(10)
```

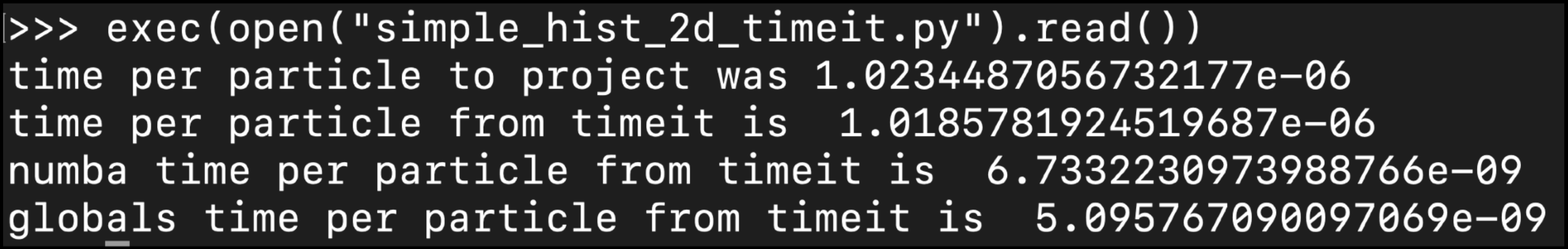


```
Jonathans-MBP-2:performance sievers$ python3 look_at_output.py
Tue Nov 24 21:36:07 2020    simple_hist_timing.dat

         67705 function calls (65534 primitive calls) in 2.629 seconds

   Ordered by: internal time
   List reduced from 917 to 10 due to restriction <10>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    2.407    2.407    2.442    2.442 simple_hist_2d.py:4(hist_2d)
        1    0.037    0.037    0.037    0.037 {method 'rand' of 'numpy.random.mtrand.RandomState' objects}
      107    0.028    0.000    0.028    0.000 {method 'read' of '_io.BufferedReader' objects}
    31/29    0.024    0.001    0.026    0.001 {built-in method _imp.create_dynamic}
      120    0.019    0.000    0.019    0.000 {built-in method numpy.array}
        1    0.016    0.016    0.016    0.016 {method 'round' of 'numpy.ndarray' objects}
      316    0.014    0.000    0.014    0.000 {built-in method builtins.compile}
      107    0.010    0.000    0.010    0.000 {built-in method marshal.loads}
        1    0.005    0.005    2.629    2.629 simple_hist_2d.py:1(<module>)
  224/223    0.004    0.000    0.005    0.000 {built-in method builtins.__build_class__}


Jonathans-MBP-2:performance sievers$ open performance.key
```

Timeit



```
out=timeit.timeit('hist_2d(xy,grid)',
                  'from __main__ import hist_2d,xy,grid',number=niter)
print('time per particle from timeit is ',out/niter/npt)
hist_2d_nb(xy,grid)
out=timeit.timeit('hist_2d_nb(xy,grid)',
                  'from __main__ import hist_2d_nb,xy,grid',number=niter)
print('numba time per particle from timeit is ',out/niter/npt)

#we can also just import everything in the global namespace
out=timeit.timeit('hist_2d_nb(xy,grid)',globals=globals(),number=niter)
print('globals time per particle from timeit is ',out/niter/npt)
```



```
>>> exec(open("simple_hist_2d_timeit.py").read())
time per particle to project was 1.0234487056732177e-06
time per particle from timeit is  1.0185781924519687e-06
numba time per particle from timeit is  6.7332230973988766e-09
globals time per particle from timeit is  5.095767090097069e-09
```

Timeit is another popular way to time individual functions. Will work for even very short (sub-microsecond) function calls. Several options, including command-line.

Performance

- Python is an interpreted language. This means things in loops are very slow!

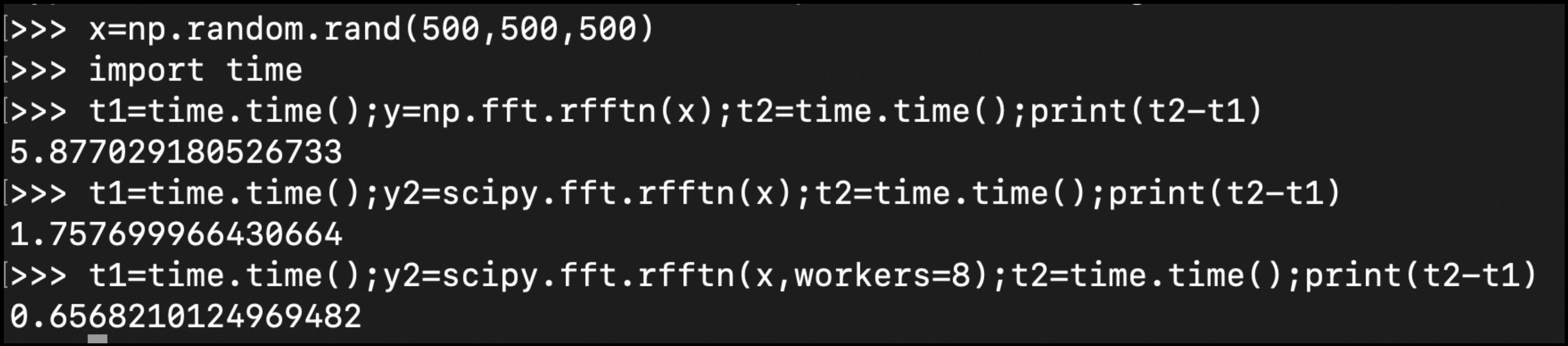
- Look at simple_hist_2d.py. This is a possibly relevant case where you take a list of points and grid them into a 2D array.

- How does this speed compare with anything you might expect?

- Numpy often has vectorized code: c=a+b is *much* faser than c[i]=a[i]+b[i] looped over i.

Beware...

- scipy≠numpy. e.g. scipy.fft.fft accepts optional argument “workers” which numpy.fft.fft doesn’t. You may wish to use scipy for this reason…



```
[>>> x=np.random.rand(500,500,500)
[>>> import time
[>>> t1=time.time();y=np.fft.rfftn(x);t2=time.time();print(t2-t1)
5.877029180526733
[>>> t1=time.time();y2=scipy.fft.rfftn(x);t2=time.time();print(t2-t1)
1.757699966430664
[>>> t1=time.time();y2=scipy.fft.rfftn(x,workers=8);t2=time.time();print(t2-t1)
0.6568210124969482
```

C

- Compiled code is usually much faster, at least where loops are involved.

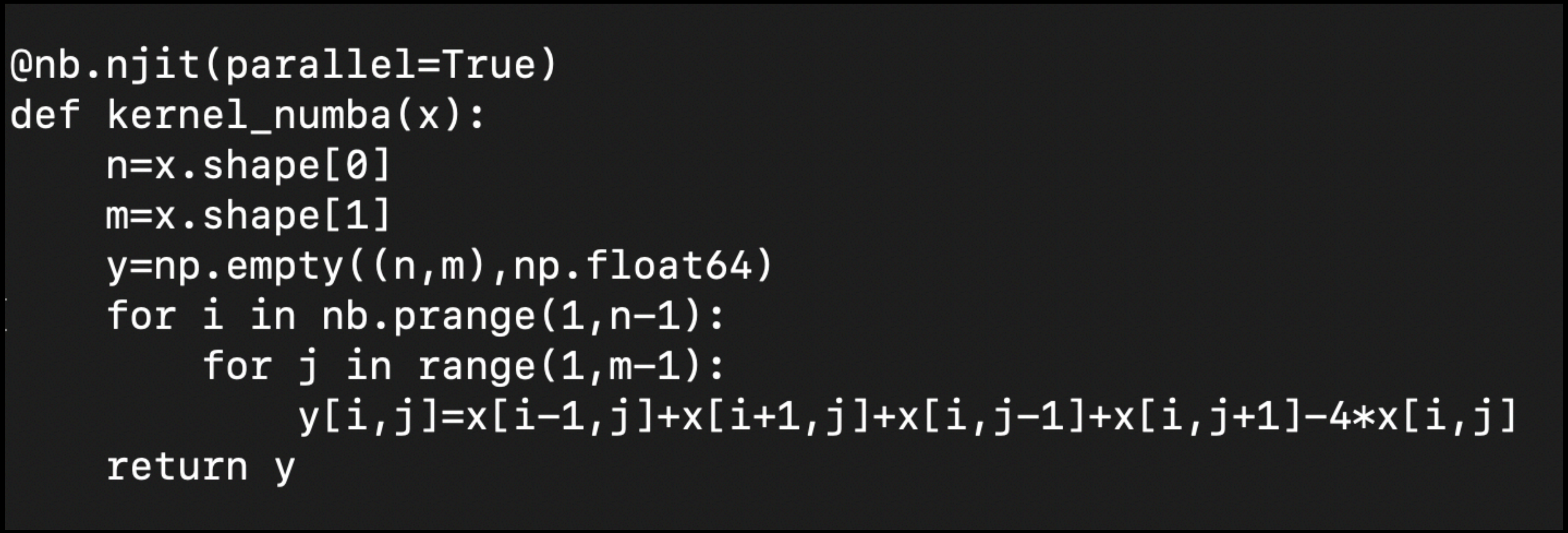
- We can link C to python, so python can call C libraries.

- ctypes is one way to do this. There are others, but ctypes widely present and requires no setup.

- Look at simple_hist_2d_ctypes.py and hist2d_c.c as an example. We win by a factor of 100.

Numba

- We can sometimes have the best of both worlds. Numba allows real-time compilation of python into C(++?).
- @nb.(n)jit tells numba to try to compile code at runtime. If arg types change, it may recompile.
- Loops in Python are excruciatingly slow. Loops in Numba are fast. Often faster than vectorized code because needed memory bandwidth may decrease.
- Example - 2D stencil from Laplace solver. np.roll requires bringing an array through memory several times, but stencils will sit in cache in looped version.



```
@nb.njit(parallel=True)
def kernel_numba(x):
    n=x.shape[0]
    m=x.shape[1]
    y=np.empty((n,m),np.float64)
    for i in nb.prange(1,n-1):
        for j in range(1,m-1):
            y[i,j]=x[i-1,j]+x[i+1,j]+x[i,j-1]+x[i,j+1]-4*x[i,j]
    return y
```

- numba example. note code is perfectly standard python, except for @nb.njit

- note - np.empty requires tuple as size argument, not list.

- note - nb.prange will parallelize a loop for you! Make sure your loop is parallelizable first…

- look at time_kernel.py (and laplace_kernel.c) for smackdown for several different techniques.

Numba histd2d

- You might think that making an integer copy of coordinates, saving that, then using again would be slow. You would be right.

- simple_hist_2d_numba.py uses numba to do 2d hist both ways. hist_2d uses numpy to round/convert to ints, then calls numba. project_round pulls the round/convert into numba, then uses those values in-place.

- Speeds up by factor of ~2-3.

Simple Parallelization

- Numba may also parallelize loops for you.
- Loop iterations should be independent (“embarrassingly parallel”).
- Add parallel=True to @nb.njit, use nb.prange
- Why might making a histogram be dangerous?
- when two processes try to write to memory at the same time, behaviour is undetermined - this is a “race condition”.

MPI

- Another version of parallelization is Message Passing Interface (MPI). A bunch of workers run at the same time, usually on different physical machines.

- Each worker is responsible for a piece of the problem, they decide who does what, communicate with each other as needed (your job to figure out!)

- MPI wrapped in python via mpi4py.

mpi4py_example.py

- Look at mpi4py_example.py. We set up a communicator (comm) so workers can communicate. comm=MPI.COMM_WORLD has everybody in it.

- Every worker gets a unique ID (“rank”) running from 0 to n-1, where n is # of workers. rank=comm.Get_rank()

- Can also get total number of workers: nproc=comm.Get_size().

mpi4py_example.py 2

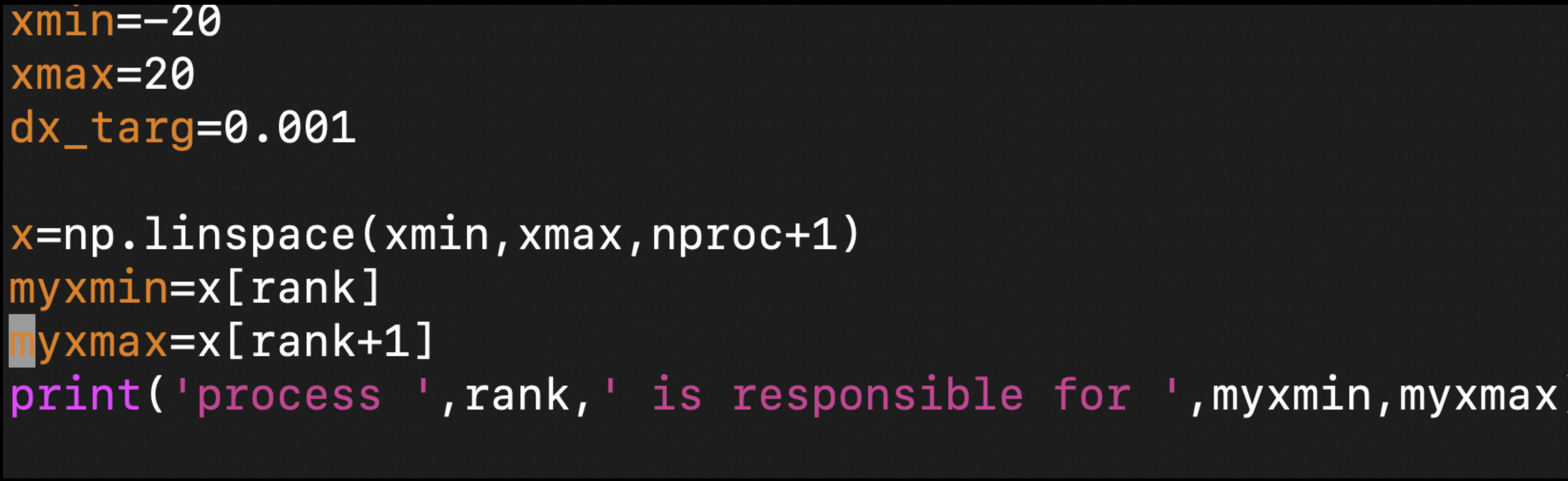
- I can add arrays across workers, send answer to all workers: comm.Allreduce(rr,rr_out), where rr,rr_out are (numpy) arrays. lower-case version (comm.allreduce) let’s you do same for scalars.

- Multiply works as well: comm.Allreduce(rr+1,rr_out,MPI.PROD)

- we can also send point-to-point messages:
  comm.send(rr,dest=my_send_buddy)
  mesg=comm.recv(source=my_receive_buddy)

MPI Integration

- We can split integration among multiple workers by giving each worker a piece of the domain, the combining answers.



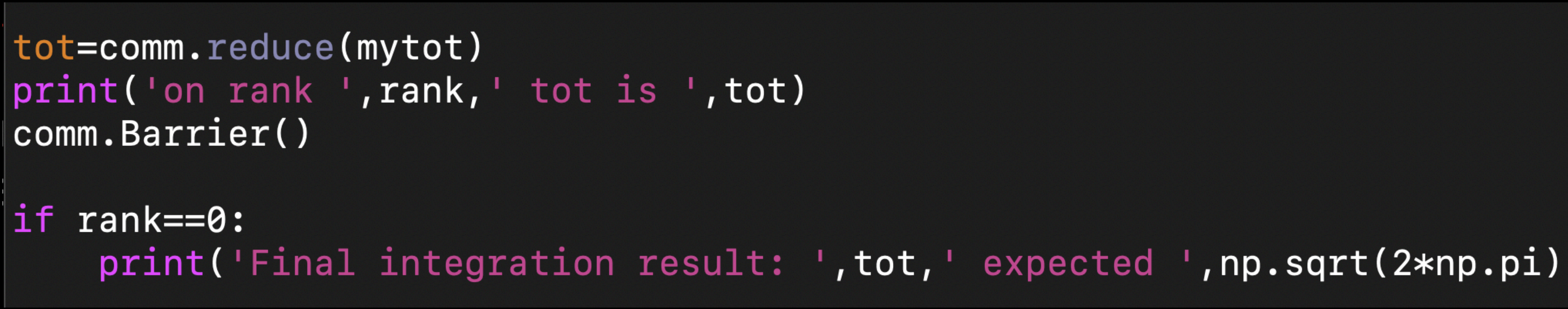
```
xmin=-20
xmax=20
dx_targ=0.001

x=np.linspace(xmin,xmax,nproc+1)
myxmin=x[rank]
myxmax=x[rank+1]
print('process ',rank,' is responsible for ',myxmin,myxmax
```

MPI Integration 2

- Once we split up the domain, each worker integrates from myxmin to myxmax.

- Once they’re done, every worker sends it partial result to get summed.

- In our case, we’ll use reduce, which takes everyone’s result and puts it on one process (by default the rank=0 process).



```
tot=comm.reduce(mytot)
print('on rank ',rank,' tot is ',tot)
comm.Barrier()

if rank==0:
    print('Final integration result: ',tot,' expected ',np.sqrt(2*np.pi)
```