Profiling and Optimization

Karl Kosack
CEA Paris-Saclay

ASTERICS-OBELICS International School
Annecy, June 2017

H2020-Astronomy ESFRI and Research Infrastructure Cluster
(Grant Agreement number: 653477).

“

Premature optimization is the root of all evil…

- probably Don Knuth

Why optimize?

Why optimize?

However… once code is working, you do want it to be efficient!

- want a balance between usability/cleanness and speed/memory efficiency
- These are not always both achievable, so err on the side of *usability*

Why optimize?

However… once code is working, you do want it to be efficient!

- want a balance between usability/cleanness and speed/ memory efficiency
- These are not always both achievable, so err on the side of *usability*

Some things:

- Python is interpreted (though some compilation happens), and can therefore be *slow*
- For-loops in particular are 100 - 1000x slower than C loops…
- There are some nice ways to speed up code, however, and get close to low-level language speed

Steps to optimization

1) Make sure code *works correctly* first

- DO NOT optimize code you are writing or debugging!

2) Identify use cases for optimization:

- how often is the code called? Is it useful to optimize it?
- If it is not called often and finishes with reasonable time/memory, stop!

3) *Profile* the code to identify bottlenecks in a more scientific way

- Profile time spent in each function, line, etc
- Profile memory use

4) try to re-write as little as possible to achieve improvement

5) refactor if it is still problematic…

Speed profiling 1: the notebook

Simplest method: *timeit*

- *no need to calculate start and stop times, python's standard lib has a nice module to help with that…*
- *easiest way is to use interactive* ***%timeit*** *magic ipython function*

DEMO NOTEBOOK

- *Usage:*

```
%timeit <python statement>
```

Why not just roll your own?

```
start = time.now()
[code]
stop = time.now()
print(stop-start)
```

this measures only wall-clock time! You want CPU time… then you want many trials, etc…

note you can also import the `timeit` module and use it similar to the magic %timeit function

Speed profiling 2: profiler!

A profiler is better than a simple %timeit, in that it checks the time in *all* functions and sub-functions at once and generates a report.

Python provides several profilers, but the most common is *cProfile* (note: gprof for c++)

Profile an entire script:

- Run your script with the additional options:

```
python -m cProfile -o output.pstats  <script>
```

- this generates a binary data file (*output.pstats*) that contains the info… you need a way to view it
- There is a built-in **pstats** module that displays it, for example

An example from CTA low-level data analysis…

```python
"""
The most basic pipeline, using no special features of the framework
other than a for-loop. This is useful for debugging and profiling of
speed.
"""

from ctapipe.io.hessio import hessio_event_source
from ctapipe.calib import (HessioR1Calibrator, CameraDL1Calibrator,
                           CameraDL0Reducer)
import sys

if __name__ == '__main__':

    filename = sys.argv[1]

    source = hessio_event_source(filename)

    cal_r0 = HessioR1Calibrator(None,None)
    cal_dl0 = CameraDL0Reducer(None,None)
    cal_dl1 = CameraDL1Calibrator(None,None)

    for event in source:

        print("EVENT", data.r0.event_id)
        cal_r0.calibrate(event)
        cal_dl0.reduce(event)
        cal_dl1.calibrate(event)
```

Generate Profile

```
% python -m cProfile -o output.pstats simple_pipeline.py ~/Data/CTA/Prod3/gamma.simtel.gz
```

```
I/O block extended by 256776 to 1256776 bytes
Trying to read event data before run header.
Skipping this data block.
I/O block extended by 370044 to 1626820 bytes
I/O block extended by 1385148 to 3011968 bytes
WARNING: ErfaWarning: ERFA function "taiutc" yielded
1 of "dubious year (Note 4)" [astropy._erfa.core]
EVENT 6911
EVENT 20505
EVENT 20514
EVENT 32700
EVENT 32704
EVENT 32708
EVENT 32710
EVENT 32711
I/O block extended by 368640 to 3380608 bytes
EVENT 32718
…
```

View stats with builtin stats viewer

```
% python -m pstats output.pstats
```

```
Welcome to the profile statistics browser.
output.pstats% sort cumtime
output.pstats% stats 10

Wed Apr 19 14:48:12 2017    output.pstats

         3975674 function calls (3926391 primitive calls) in 18.386 seconds

   Ordered by: cumulative time
   List reduced from 6335 to 10 due to restriction <10>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   1347/1    0.047    0.000   18.388   18.388 {built-in method builtins.exec}
        1    0.002    0.002   18.387   18.387 simple_pipeline.py:4(<module>)
      100    0.010    0.000    9.626    0.096 /Users/kosack/Projects/CTA/Working/ctapipe/ctapipe/calib/camera/dl1.py:221(calibrate)
      307    0.006    0.000    9.183    0.030 /Users/kosack/Projects/CTA/Working/ctapipe/ctapipe/calib/camera/charge_extractors.py:271(extract_charge)
      307    0.004    0.000    8.456    0.028 /Users/kosack/Projects/CTA/Working/ctapipe/ctapipe/calib/camera/charge_extractors.py:309(get_peak_pos)
      307    7.299    0.024    8.452    0.028 /Users/kosack/Projects/CTA/Working/ctapipe/ctapipe/calib/camera/charge_extractors.py:464(_obtain_peak_position)
      101    0.030    0.000    6.508    0.064 /Users/kosack/Projects/CTA/Working/ctapipe/ctapipe/io/hessio.py:70(hessio_event_source)
      221    5.638    0.026    5.640    0.026 /Users/kosack/anaconda/lib/python3.6/site-packages/pyhessio/__init__.py:273(move_to_next_event)
   1310/6    0.006    0.000    1.949    0.325 <frozen importlib._bootstrap>:958(_find_and_load)
   1310/6    0.005    0.000    1.949    0.325 <frozen importlib._bootstrap>:931(_find_and_load_unlocked)
```

most time is spent
in extract_charge

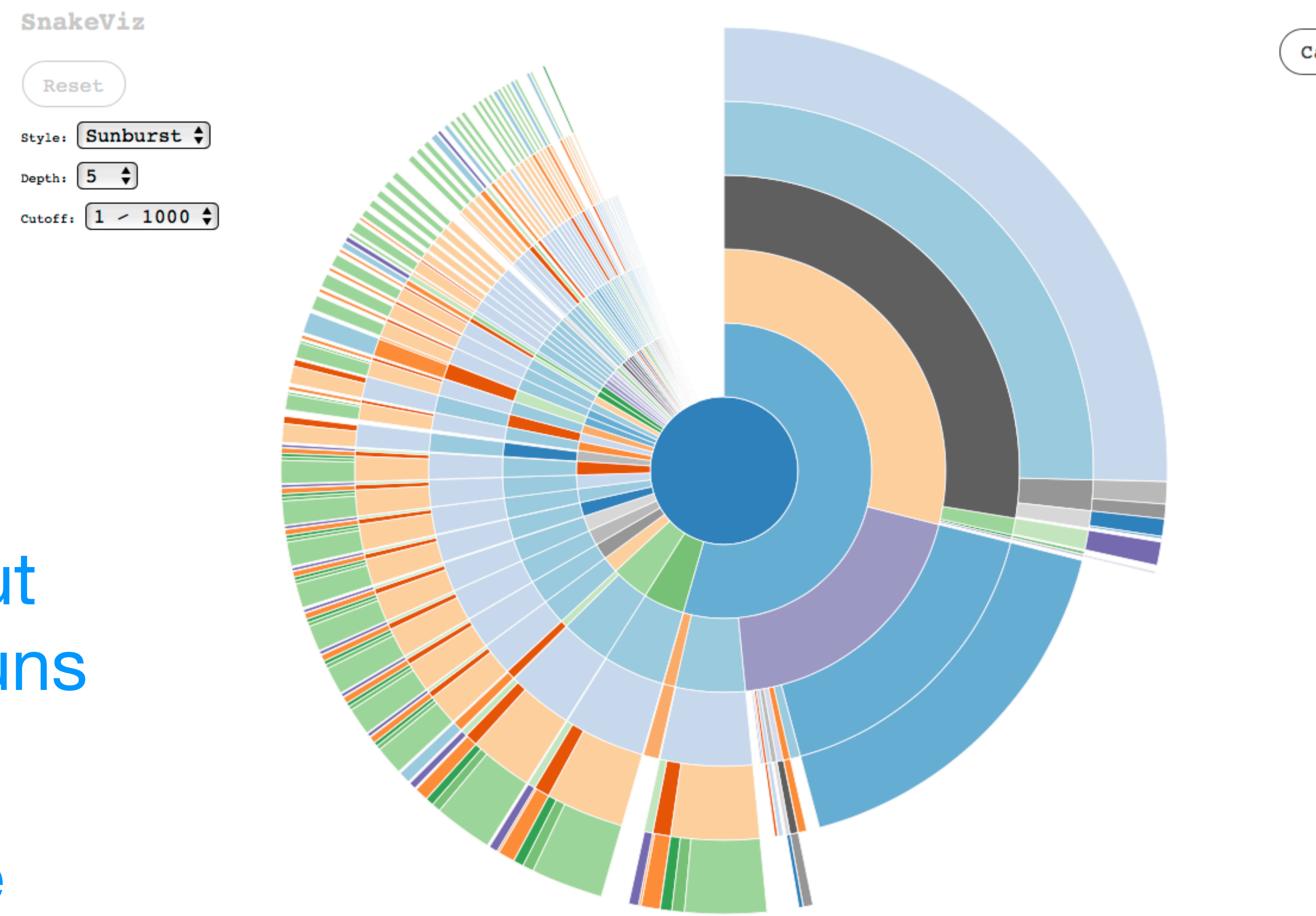
Note that the data are really hierarchical so we'd like to select only stats for functions called ***within*** *extract_charge to see where the slowness is… you can do this with the command-line, but…*

As usual there is a better way…

GUI stats viewing

```
% conda install snakeviz
% snakeviz output.pstats
```

- interactive call statistics viewer
- this is not the only one, but it's nice and simple and runs in your browser.
- Click and zoom to see the results



ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
307	7.299	0.02378	8.452	0.02753	charge_extractors.py:464(_obtain_peak_position)
221	5.638	0.02551	5.64	0.02552	__init__.py:273(move_to_next_event)
5749	0.4125	7.175e-05	0.4125	7.175e-05	~:0(<method 'reduce' of 'numpy.ufunc' objects>)
630887	0.2995	4.747e-07	0.4726	7.492e-07	traitlets.py:543(__get__)
100	0.2832	0.002832	0.2911	0.002911	r1.py:99(calibrate)
307	0.2478	0.0008072	0.2701	0.0008799	numeric.py:1936(indices)
118/88	0.1963	0.00223	0.2487	0.002826	~:0(<built-in method _imp.create_dynamic>)
2	0.1866	0.0933	0.1866	0.0933	__init__.py:368(close_file)

Profiling in a Notebook

You can also run the profiler directly on a statement in a notebook.

- use the magic %prun function

```
%prun <python statement>
```

- Pops up a sub-window with the results (the same as if you ran cProfile and then pstats (though you don't get an interactive viewer)

```
In [27]: %prun create_array_loop(1000,1000)
```

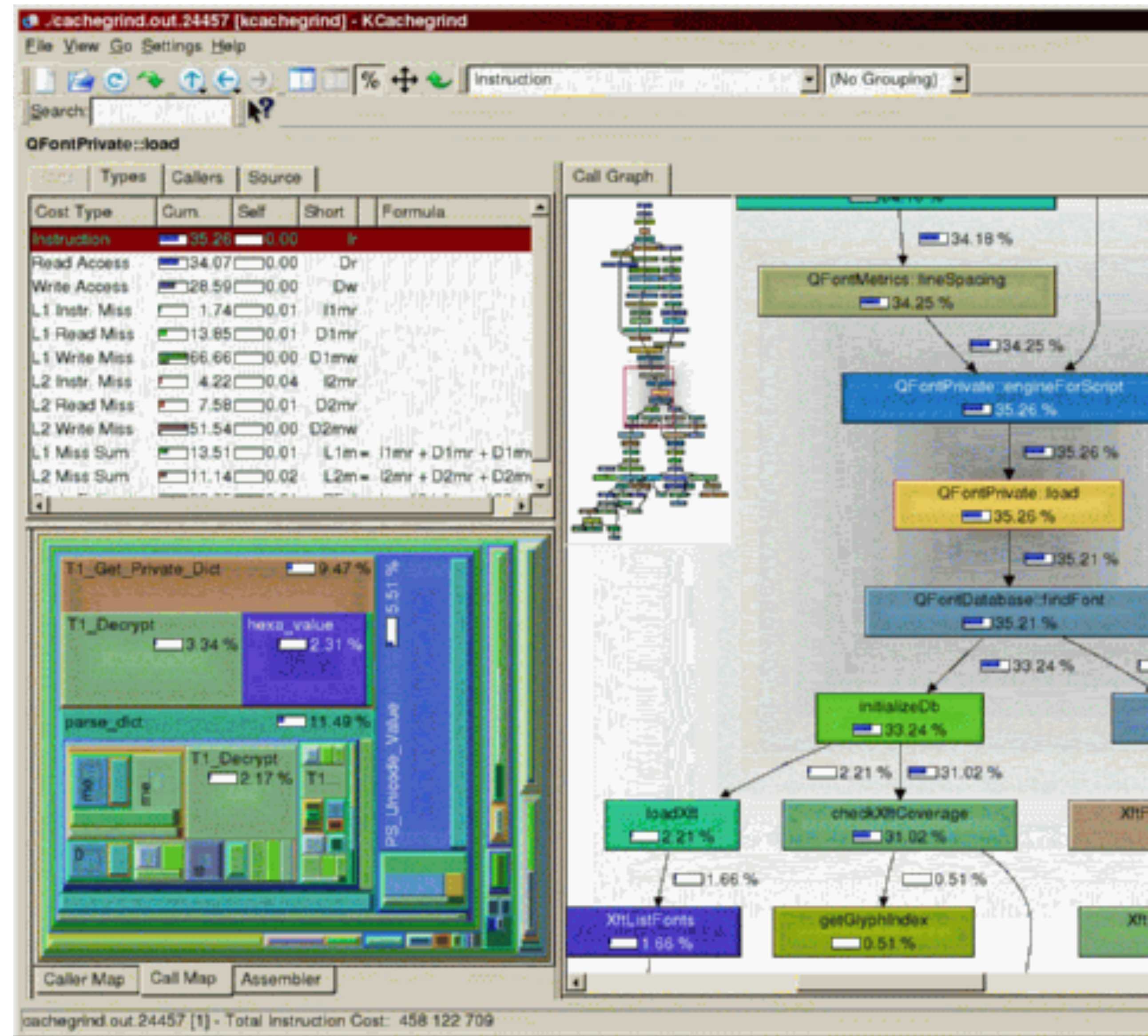
```
         3001004 function calls in 0.845 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.477    0.477    0.835    0.835 <ipython-input-12-6d84b414c957>:1(create_array_loop)
  1000000    0.136    0.000    0.136    0.000 {built-in method math.cos}
  1000000    0.133    0.000    0.133    0.000 {built-in method math.sin}
  1001000    0.089    0.000    0.089    0.000 {method 'append' of 'list' objects}
        1    0.010    0.010    0.845    0.845 <string>:1(<module>)
        1    0.000    0.000    0.845    0.845 {built-in method builtins.exec}
```

Another stats viewer

You can also view pstats output with KDE's kcachegrind GUI, just like you would with C++ profiling output:

```
% pip install pyprof2calltree
% pyprof2calltree -i output.pstats -k
```

Then, open the resulting file with KCacheGrind



disclaimer: *I have not tried this, but have used KCacheGrind for C++ projects, and it's nice!*

Line Profiling

Sometimes you need more detail than function-level stats…
What about time spent in each line of code?

The line_profiler module can help:

```
% conda install line_profiler
```

- mark code with @profile:

```
from line_profiler import profile

@profile
def slow_function(a, b, c):
    ...
```

- Then run:
  - % **kernprof** -l script_to_profile.py

```
File: pystone.py
Function: Proc2 at line 149
Total time: 0.606656 s

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
   149                                           @profile
   150                                           def Proc2(IntParIO):
   151     50000        82003      1.6     13.5      IntLoc = IntParIO + 10
   152     50000        63162      1.3     10.4      while 1:
   153     50000        69065      1.4     11.4          if Char1Glob == 'A':
   154     50000        66354      1.3     10.9              IntLoc = IntLoc - 1
   155     50000        67263      1.3     11.1              IntParIO = IntLoc -
   156     50000        65494      1.3     10.8              EnumLoc = Ident1
   157     50000        68001      1.4     11.2          if EnumLoc == Ident1:
   158     50000        63739      1.3     10.5              break
   159     50000        61575      1.2     10.1      return IntParIO
```

- which generates a .lprof file that can be viewed with:
  - % **python -m line_profiler** script_to_profile.py.lprof

Line-profiling in a Notebook

Like with cProfile and timeit, you can do line profiling in a notebook:

- unlike %timeit, need to load an extension first:

```
%load_ext line_profiler
```

- Then, if you have a function defined, you must "mark" it to be profiled by adding "-f <func>"

```
%lprun -f <function name> <python statement that uses function>
```

for example:

```
%lprun -f myfunc myfunc(100,100)
```

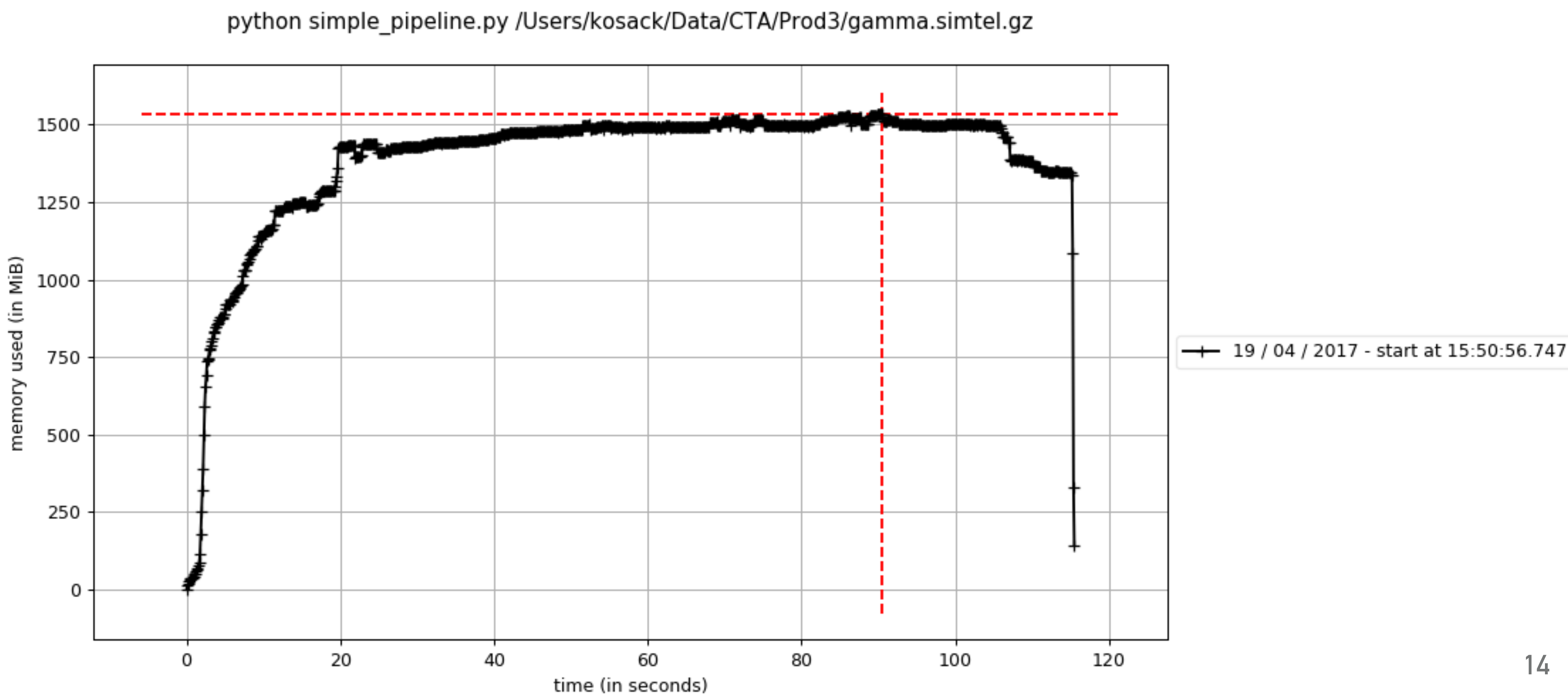
Note you can mark more than one func

```
In [51]: %lprun -f create_array_loop create_array_loop(1000,1000)

Timer unit: 1e-06 s

Total time: 1.31799 s
File: <ipython-input-12-6d84b414c957>
Function: create_array_loop at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def create_array_loop(N,M):
     2         1            2      2.0      0.0      arr = []
     3      1001          477      0.5      0.0      for y in range(M):
     4      1000         5244      5.2      0.4          row = []
     5   1001000       463343      0.5     35.2          for x in range(N):
     6   1000000       848316      0.8     64.4              row.append(sin(x)*cos(0.1*y))
     7      1000          606      0.6      0.0          arr.append(row)
     8         1            1      1.0      0.0      return arr
```

Memory Profiling

Use of CPU is not the only thing to worry about… what about RAM? Let's first check for memory leaks…

```
% conda install memory_profiler
% mprof run python <script>
% mprof plot
```



Memory Profiling in detail

Cumulative is nice, but we want to see the memory for a particular function or class…

- decorate the function you want to profile (line-wise) with memory_profiler.profile

> % **python -m memory_profiler** <script>

```python
from memory_profiler import profile

@profile
def main():

    filename = sys.argv[1]

    source = hessio_event_source(filename)

    cal_r0 = HessioR1Calibrator()
    cal_dl0 = CameraDL0Reducer()
    cal_dl1 = CameraDL1Calibrator()

    for data in source:

        print("EVENT", data.r0.event_id)
        cal_r0.calibrate(data)
        cal_dl0.reduce(data)
        cal_dl1.calibrate(data)


if __name__ == '__main__':
    main()
```

Decorate what we want to measure

```
Filename: simple_pipeline.py

Line #    Mem usage    Increment   Line Contents
================================================
    19     87.8 MiB      0.0 MiB   @profile
    20                             def main():
    21
    22     87.8 MiB      0.0 MiB       filename = sys.argv[1]
    23
    24     87.8 MiB      0.0 MiB       source = hessio_event_source(filename, max_
    25     87.8 MiB      0.0 MiB                                    allowed_tels=r
    26
    27     87.8 MiB      0.0 MiB       cal_r0 = HessioR1Calibrator(None,None)
    28     87.8 MiB      0.0 MiB       cal_dl0 = CameraDL0Reducer(None,None)
    29     87.8 MiB      0.0 MiB       cal_dl1 = CameraDL1Calibrator(None,None)
    30
    31    929.2 MiB    841.4 MiB       for data in source:
    32
    33    929.2 MiB      0.0 MiB           print("EVENT", data.r0.event_id)
    34    929.2 MiB      0.0 MiB           cal_r0.calibrate(data)
    35    929.2 MiB      0.0 MiB           cal_dl0.reduce(data)
    36    935.6 MiB      6.4 MiB           cal_dl1.calibrate(data)
```

Not so exciting, of course all memory is in the data reader, but you get the idea…

Memory Profiling: jump to debugger

Automatic Debugger breakpoints:

- you can automatically start the debugging if the code tries to go above a memory limit, to see where the allocation is happening:

```
% python -m memory_profiler --pdb-mmem=100  <script>
```

will break and enter debugger after 100 MB is allocated, on the line where the last allocation occurred

Print out memory usage during program execution:

```
from memory_profiler import memory_usage
mem_usage = memory_usage(-1, interval=.2, timeout=1)
print(mem_usage)
    [7.296875, 7.296875, 7.296875, 7.296875, 7.296875]
```

- see the docs. you can also write it to a log periodically, etc.

Memory Profiling in a Notebook

Again, you can do memory profiling using magic commands in an iPython (Jupyter) notebook

- Enable the memory profiling notebook extension:

```
%load_ext memory_profiler
```

- Now you have access to several magic functions:

Like %timeit, but for memory usage:

```
%memit <python statement>
```

or a more full-featured report:

```
%mprun -f <function name> <statement>
```

```
In [40]: %memit range(100000)
peak memory: 89.61 MiB, increment: 0.00 MiB

In [41]: %memit np.arange(100000)
peak memory: 90.12 MiB, increment: 0.52 MiB
```

Caveats:

- the peak memory usage shown in the notebook may not relate to the function you are testing! It is the sum of all memory already allocated that has not yet been garbage collected. (so look at the "increment" instead).
- %mprun only works if your functions are **defined in a file** (not a notebook) and imported into the notebook

SO WE'VE IDENTIFIED SLOW CODE

NOW WHAT?

Speeding up python code: Numpy

Use NumPy vector operations as much as possible

- don't call a function on many small pieces of data when you can call it on an array all at once
- numpy is implemented in C *and* it uses fast numerical libraries, optimized for your CPU (e.g. Intel Math Kernel Library, BLAS, etc)
- usually just vectorizing your code to avoid some for-loops, will give you great performance.
  - bad:

```
for ii in range(100):
    x = ii*0.1
    y[ii] = f(x)
```

  - Good:

```
x = np.linspace(0,10,100)
y = f(x)
```

Speeding up 2: cython

***cython* is a special meta-language that lets you write *C code* with python syntax. It can be used to speed up core routines with minimal effort**

see documentation here:

[Cython: C-Extensions for Python]

You get access to all of C's functionality:

- compiled code (uses GCC or clang) with **fast loops**
- call C code directly
- explicit data types
- functions can be C-only for more speed, or have automatic python interfaces

And:

- numpy operations natively supported

To try it out in a notebook:

```
%load_ext cython
```

then any cell that starts with `%%cython` gets compiled automatically:

```
%%cython

def cython_func(x):
    cdef int ii
    cdef double y = 0
    for ii in range(100):
        y += ii
    return y
```

There is a LOT of functionality in cython, but the simplest thing that increases speed is to define your variable types with

***cdef** type variable*

for numpy arrays, you can define their type as follows:

```
cimport numpy as cnp


cdef cnp.ndarray[double, mode="c", ndim=2] my_array
```

```
%%cython --compile-args=-O2

import numpy as np
cimport cython

def tailcuts_clean_cython_opt(image, neighbors, double picture_thresh, double boundary_thresh):
    cdef int ii
    cdef picture = np.zeros_like(image)
    cdef clean_mask = np.zeros_like(image)

    for ii in range(image.shape[0]):
        if image[ii] > picture_thresh:
            picture[ii] = 1
            clean_mask[ii] = 1

    for ii in range(image.shape[0]):
        if image[ii] > boundary_thresh:
            for neigh in neighbors[ii]:
                if neigh < 0:
                    break
                if picture[neigh]:
                    clean_mask[ii] = 1
                    break
    return clean_mask
```

Speeding up 3: Numba

Even newer technology:

- takes python code and *directly* uses **introspection to compile it under LLVM** (no python-to-c or cython translation)
- Pretty **automatic**, *but doesn't always help*! Still need code written in a way that can be optimized (for-loops are actually good here, it can't do much with numpy operations since they are already compiled code)
- Can generate **NumPy "ufuncs"** directly (function that works on scalars but is run on all elements of an array), which are too slow to write in python normally.
- the "pro" version can also generate **GPU code!** (@jit

Super simple to try though:

```
from numba import jit
from numpy import arange

# jit decorator tells Numba to compile this function.
# The argument types will be inferred by Numba when function is called.
@jit
def sum2d(arr):
    M, N = arr.shape
    result = 0.0
    for i in range(M):
        for j in range(N):
            result += arr[i,j]
    return result

a = arange(9).reshape(3,3)
print(sum2d(a))
```

just add this decorator, and it's magic

```python
from timeit import default_timer as timer
from matplotlib.pylab import imshow, jet, show, ion
import numpy as np

from numba import jit


@jit
def mandel(x, y, max_iters):
    """
    Given the real and imaginary parts of a complex number,
    determine if it is a candidate for membership in the Mandelbrot
    set given a fixed number of iterations.
    """
    i = 0
    c = complex(x,y)
    z = 0.0j
    for i in range(max_iters):
        z = z*z + c
        if (z.real*z.real + z.imag*z.imag) >= 4:
            return i

    return 255

@jit
def create_fractal(min_x, max_x, min_y, max_y, image, iters):
    height = image.shape[0]
    width = image.shape[1]

    pixel_size_x = (max_x - min_x) / width
    pixel_size_y = (max_y - min_y) / height
    for x in range(width):
        real = min_x + x * pixel_size_x
        for y in range(height):
            imag = min_y + y * pixel_size_y
            color = mandel(real, imag, iters)
            image[y, x] = color

    return image

image = np.zeros((500 * 2, 750 * 2), dtype=np.uint8)
s = timer()
create_fractal(-2.0, 1.0, -1.0, 1.0, image, 20)
e = timer()
print(e - s)
imshow(image)
```

example from the Numba docs

- note that you need to "jit" not only the parent function, but any function that it calls that needs to be sped up

Advanced Numba

Numba includes a lot of advanced features and options to *jit* that can help speed things up when automatic methods fail

- e.g. specify the input and output type mapping, rather than infer it

Ufunc generation with *vectorize* and *guvectorize (generalized)*

Options like target='GPU' for producing CUDA code or similar

```python
import numpy as np

from numba import guvectorize

@guvectorize(['void(float64[:], intp[:], float64[:])'], '(n),()->(n)')
def move_mean(a, window_arr, out):
    window_width = window_arr[0]
    asum = 0.0
    count = 0
    for i in range(window_width):
        asum += a[i]
        count += 1
        out[i] = asum / count
    for i in range(window_width, len(a)):
        asum += a[i] - a[i - window_width]
        out[i] = asum / count

arr = np.arange(20, dtype=np.float64).reshape(2, 10)
print(arr)
print(move_mean(arr, 3))
```

example from the Numba docs

example: tailcuts cleaning

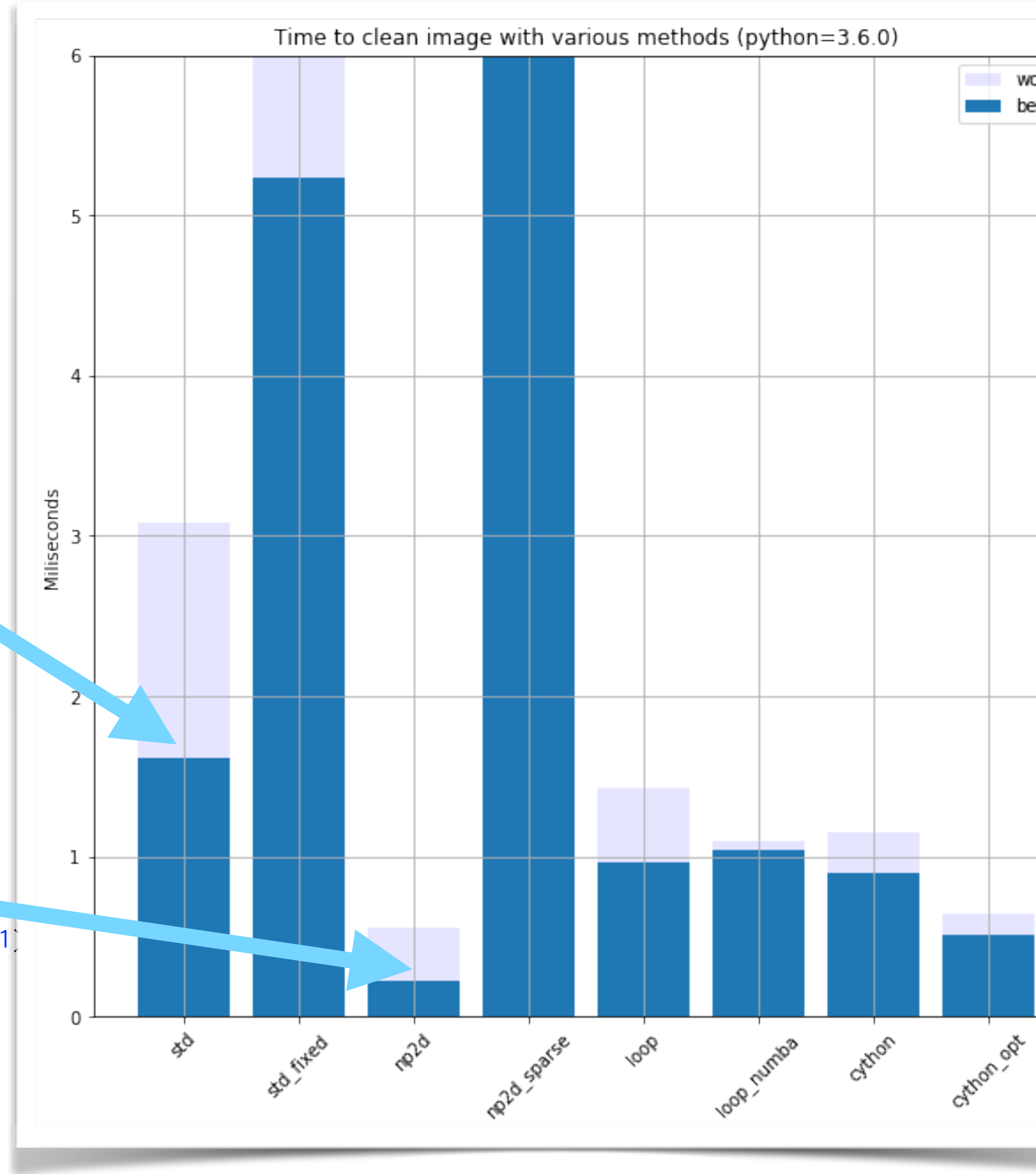
An example from CTA data processing:

- a simple 2-threshold nearest-neighbor image cleaning routine that works on non-cartesian pixel layouts

```
def tailcuts_clean(geom, image, picture_thresh, boundary_thresh):

    clean_mask = image >= picture_thresh
    boundary_mask = image >= boundary_thresh
    boundary_ids = [pix_id for pix_id in geom.pix_id[boundary_mask]
                    if clean_mask[geom.neighbors[pix_id]].any()]

    clean_mask[boundary_ids] = True
    return clean_mask
```

list-comprehension → numpy expression

```
def tailcuts_clean(geom, image, picture_thresh, boundary_thresh):

    pixels_in_picture = image >= picture_thresh
    pixels_above_boundary = image >= boundary_thresh
    pixels_with_picture_neighbors = (pixels_in_picture
                                     * geom.neighbor_matrix).any(axis=1)

    return (pixels_above_boundary
            & pixels_with_picture_neighbors) | pixels_in_picture
```



Future

Generally the CPython python "interpreter" speed increases with each release

There are a few projects to replace CPython with fully JIT-compiled python, in particular PyPy

- all PyPy code is JIT-compiled with LLVM
- support for most (but not all) of NumPy
- some support for C-extensions, but not all c-code can be run yet
- supports (so far) Python language up to version 3.5.3