

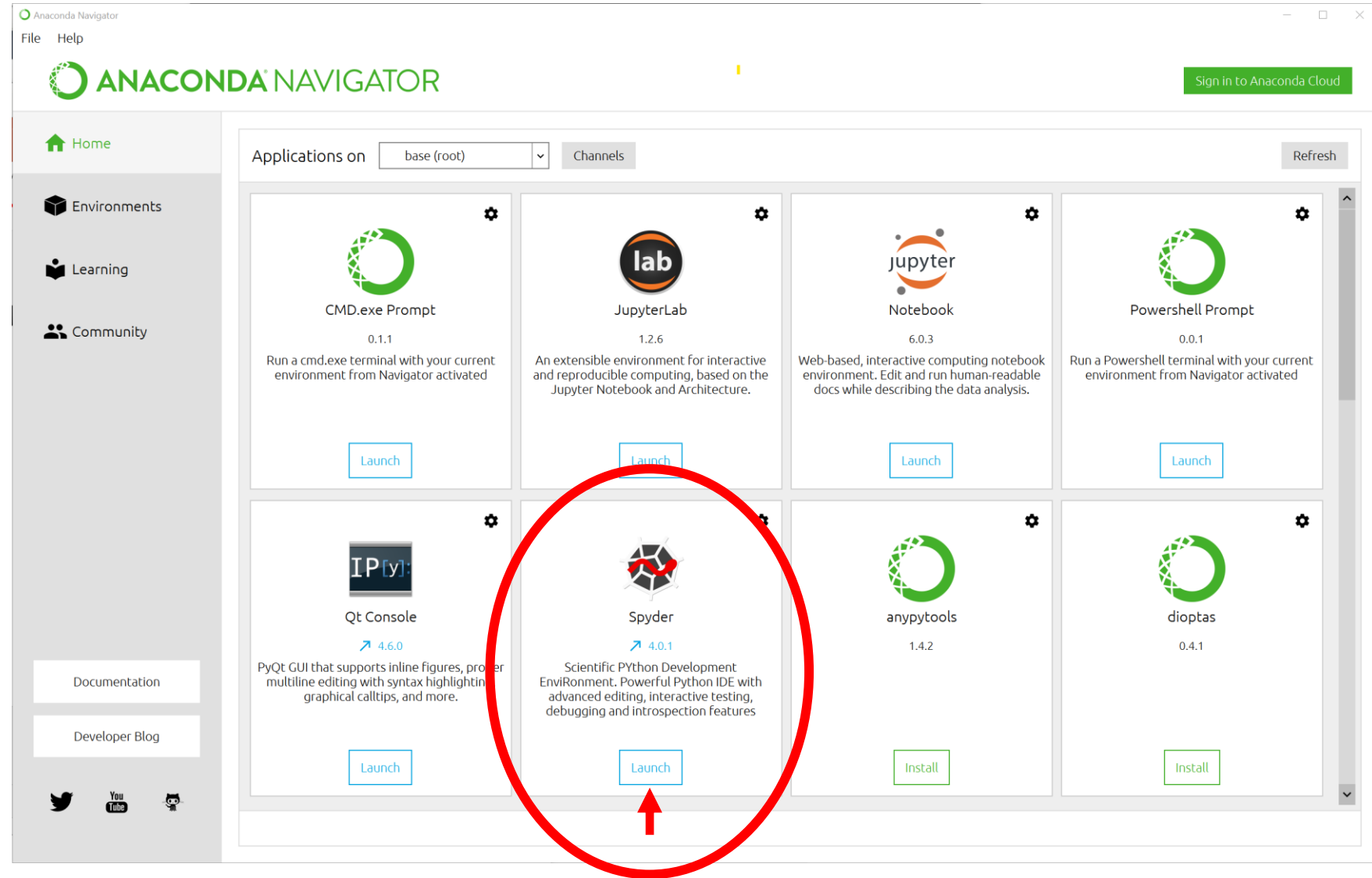
Python Optimization

v0.7

Research Computing Services
IS & T

Run Spyder

- Start the Anaconda Navigator
- Click on Spyder's Launch button
- Be patient...it takes a while to start.



Outline

- Introduction
- Profiling
- Data Structures
- Generators
- Accelerators
- Syntax

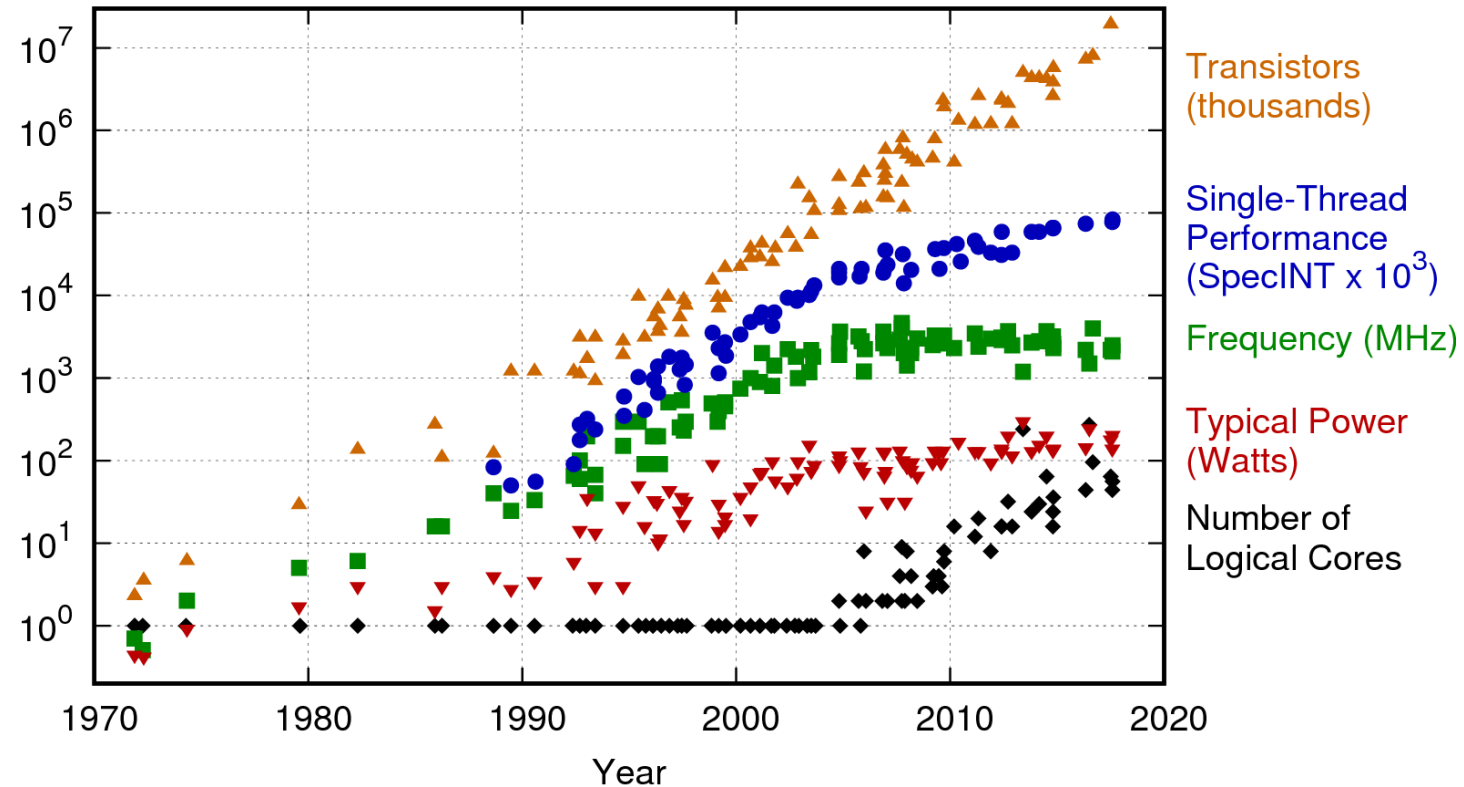
Optimization

- What are you optimizing?
 - Run time
 - Memory usage
 - I/O (storage read/write)
 - Code structure
 - Algorithm selection
- How do you decide when optimization is necessary?
- What should be changed in a program during optimization?
- Is Python fast?

Why Bother to Optimize?

- Computers aren't getting much faster.
- Easier access to data means there's more computation possible than in the past.
- Better code means you can get more done!

42 Years of Microprocessor Trend Data



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

<https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/>

Some words of wisdom, lightly paraphrased

The Elements of Programming Style, by Brian W. Kernighan and P. J. Plauger, 1974.

- Before you make your code faster:
 - Make it right
 - Make it clear
- Keep it right when you make it faster.
- Fundamental improvements in performance are most often made by algorithm changes, not by tuning.

Outline

- Introduction
- Profiling
- Data Structures
- Generators
- Accelerators
- Syntax

Profiling

- Before making code changes you must profile the code.
- We are **really** bad at guessing how different parts of programs perform!
 - This is true independent of your degree of programming experience.
- Ways to profile:
 - Insert timing statements into your code
 - Laborious but useful.
 - Use profiling tools that can measure:
 - Function call times
 - Line-by-line execution times
 - Memory consumption
 - CPU hardware utilization

Profiling Drawbacks

- Your program can take much longer to run when it is being profiled.
- It may consume more memory.
- A small test problem might be too small to reveal performance problems.

Python Profiling Tools

- The Python Standard Library includes two profilers:
 - cProfile (default)
 - profile
- [Documentation is online.](#)
- The IPython interpreter in Spyder and Jupyter will also provide timing information with the special commands *%time* and *%timeit*.

Additional Profiling Tools

- We'll use two other profilers today:
- *line_profiler*
 - Line-by-line timing statistics for selected functions.
- *memory_profiler*
 - Line-by-line memory usage statistics for selected functions.
 - For all python3 modules.

We'll pause here to make sure everyone has this installed in their Anaconda setups. If you're using the SCC and a python3 module they're already installed.

A simple sample code

- Open *row_vs_col_orig.py*
- This does a simple calculation where a Numpy matrix (i.e. a 2D table) is multiplied by a constant value.
- 3 implementations:
 - Calculate the new matrix values by multiplying the constant against whole rows at a time
 - ...by multiplying against whole columns at a time
 - ...using Numpy's built-in element-by-element multiplication syntax.

Profiling: manual timing

- Use the Python *time* library:

<https://docs.python.org/3.8/library/time.html>

- Next, open *row_vs_col_timing.py*

- This does manual timing of the function calls.
- Which version is the fastest?
- Change the size of the matrix – does this change your result?

- Two ways:

- `time.perf_counter()`:
 - Returns a floating point value representing a time.
- `time.time()`:
 - Floating point value of seconds since Jan. 1, 1970, 00:00:00

```
import time

st = time.perf_counter()
# do something...
et = time.perf_counter()
print(f'Elapsed (sec): {et-st:.3f}')
```

Function Decorators

- These are wrappers around functions.
 - Written as Python functions.
 - You can intercept a function call and do whatever you like before calling the wrapped function.
 - After the function you can again do whatever you like before returning values.

decorator

do something...

my_func(args)

do more...

```
# Add a decorator to a
# function
@decorator
def my_func(x, y, z):
    ...
    ...

# call the function:
my_func(1, 2, 3)
```

Better Manual Profiling

- Open *row_vs_col_decorator.py*
- This implements a function decorator to automatically time function calls.
- How it works:
 - Intercept a call to a function and start a timer.
 - Call the function.
 - Intercept the function return, stop the timer.
 - Print out the elapsed time.
 - Return the function's return value.

Spyder Timing

```
In [3]: %time some_func(1,0.4,5.1)
Wall time: 2.18 s

In [4]: %time some_func(1,0.4,0.1)
Wall time: 515 ms

In [5]: %timeit some_func(2,0.1,0.5)
873 ms ± 6.85 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

- In the Python console use:
 - `%time ..python code...`
 - Prints time to run the code
 - `%timeit ..python code...`
 - Runs the code multiple times, reports timing statistics

Spyder Timing

- In source code you can label a cell with `#%%`
- Then put `%%time` or `%%timeit` at the top of the cell.
- These are NOT PYTHON commands – don't leave them in your code.

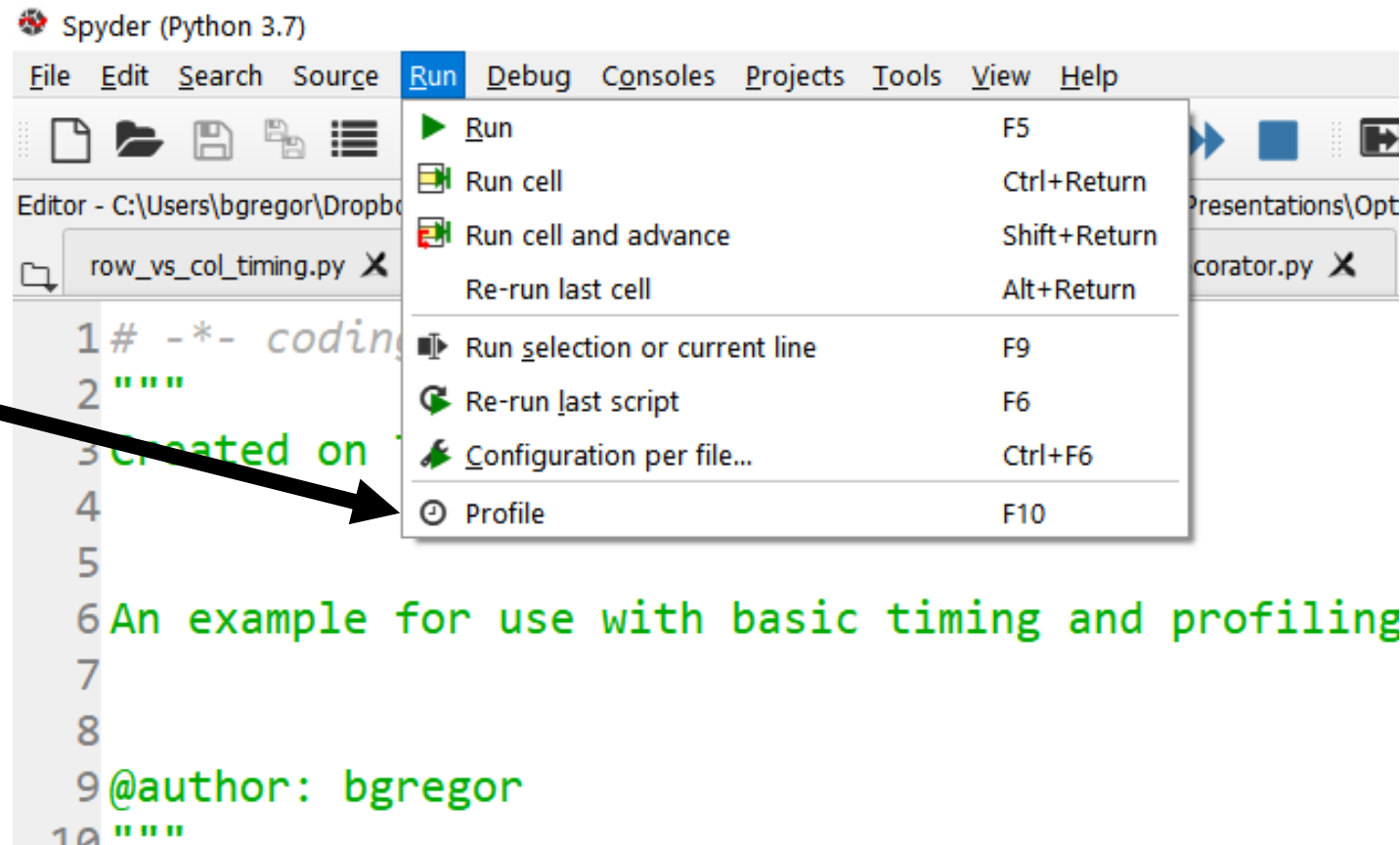
```
9 import time
10
11 def some_func(x,y,z):
12     time.sleep((x+y+z)/3)
13
14
15 # Define a cell. Note this starts with a # so it's just a comment
16 # to Python. iPython (used in Spyder) interprets this as a cell
17 #%%
18 %%time
19
20 some_func(1.5,0.1,2)
21
22 #%%
```

Run cell	Ctrl+Return
Run cell and advance	Shift+Return
Re-run last cell	Alt+Return
Run selection or current line	F9
Go to definition	Ctrl+G
Undo	Ctrl+Z
Redo	Ctrl+Shift+Z
Cut	Ctrl+X
Copy	Ctrl+C
Paste	Ctrl+V
Select All	Ctrl+A
Zoom in	Ctrl++
Zoom out	Ctrl+-
Zoom reset	Ctrl+0
Comment/Uncomment	Ctrl+1
Generate docstring	Ctrl+Alt+D
Format file or selection with Autopep8	Ctrl+Alt+I

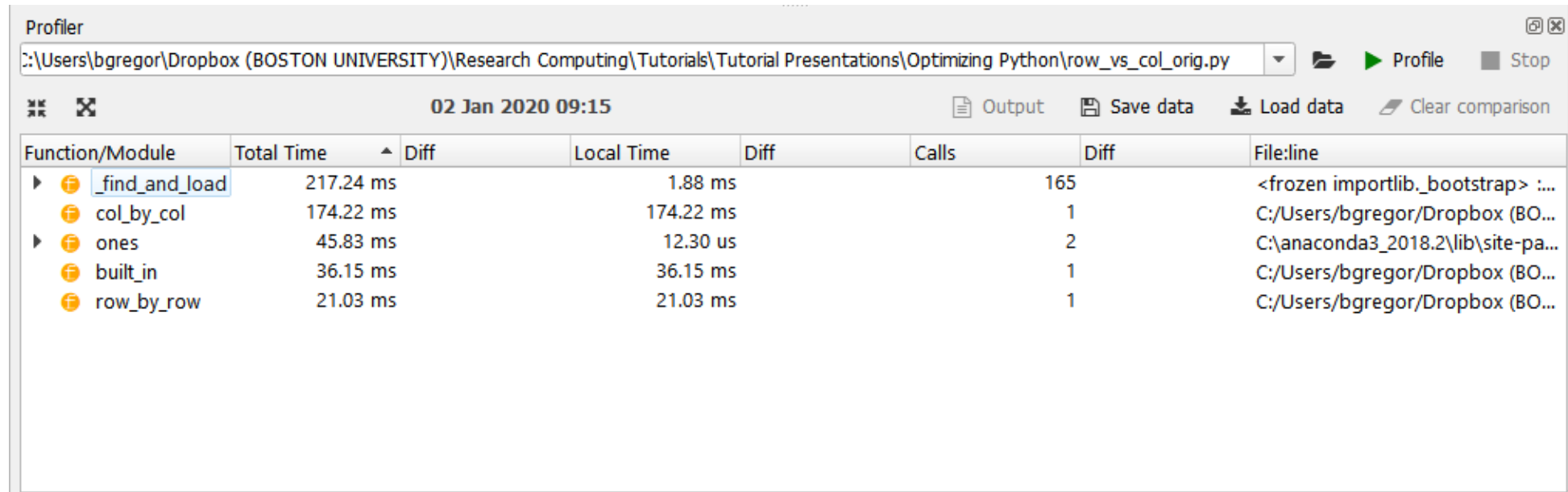
```
In [10]: runcell(1, 'C:/Users/bgregor/Dropbox (BOSTON UNIVERSITY)/Research Computing/Tutorials/Tutorial  
Presentations/Python Tutorials/Python Parallelization/v0.4/code/untitled4.py')  
Wall time: 1.21 s  
In [11]:
```

Profiling: Using the Python Profiler











- Return to *row_vs_col_orig.py*
- Spyder can run the Python profiler for you.
- Choose the menu option Run→Profile



Spyder Profiling Output



The screenshot shows the Spyder Profiler window. The title bar is 'Profiler'. The file path is 'C:\Users\bgregor\Dropbox (BOSTON UNIVERSITY)\Research Computing\Tutorials\Tutorial Presentations\Optimizing Python\row_vs_col_orig.py'. The date and time are '02 Jan 2020 09:15'. There are buttons for 'Output', 'Save data', 'Load data', and 'Clear comparison'. The main table has columns: 'Function/Module', 'Total Time', 'Diff', 'Local Time', 'Diff', 'Calls', 'Diff', and 'File:line'. The table contains the following data:

Function/Module	Total Time	Diff	Local Time	Diff	Calls	Diff	File:line
▶   _find_and_load	217.24 ms		1.88 ms		165		<frozen importlib._bootstrap> :...
▶   col_by_col	174.22 ms		174.22 ms		1		C:/Users/bgregor/Dropbox (BO...
▶   ones	45.83 ms		12.30 us		2		C:\anaconda3_2018.2\lib\site-pa...
▶   built_in	36.15 ms		36.15 ms		1		C:/Users/bgregor/Dropbox (BO...
▶   row_by_row	21.03 ms		21.03 ms		1		C:/Users/bgregor/Dropbox (BO...

- The Profiler tab shows total time spent in each function.
- If functions call functions those calls can be shown as well – click the triangles to expand the results.

Timing and Profiling in a Jupyter Notebook

- Simple timing can be done with the same commands.
 - `%time`, `%timeit` – apply to a single line of code
 - `%%time`, `%%timeit` – apply to a cell. Place these at the **top** of the cell.
- `%prun` runs the Python profiler for a function call.
- To see help add a `?: %time?`

```
In [6]: import numpy as np

mat_size = 500
# Let's just multiply by 2.
scaling_value = 2.0

def row_by_row(A,mat):
    ''' compute mat = A*mat row-by-row '''
    rows = mat.shape[0]
    for i in range(rows):
        mat[i,:] = A * mat[i,:]
    return mat
```

```
In [11]: mat = np.ones([mat_size,mat_size])
```

```
In [8]: %time mat = row_by_row(scaling_value, mat)
```

Wall time: 5.98 ms

```
In [12]: %timeit row_by_row(scaling_value, mat)
```

2.29 ms ± 349 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

```
In [13]: %prun row_by_row(scaling_value, mat)
```

4 function calls in 0.003 seconds

Ordered by: internal time

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.003	0.003	0.003	0.003	<ipython-input-6-652953e76d87>:7(row_by_row)
1	0.000	0.000	0.003	0.003	{built-in method builtins.exec}
1	0.000	0.000	0.003	0.003	<string>:1(<module>)
1	0.000	0.000	0.000	0.000	{method 'disable' of '_lsprof.Profiler' objects}

Command Line Python Profiling

- Command line profiling results are printed to the screen or can be saved to a file.
- This can be done inside of a batch job on the SCC...

- Syntax:

```
python -m cProfile run.py
```

- Sort by statistics:

```
python -m cProfile -s time run.py
```

- Best use – save to a file and use a utility to study the output:

```
python -m cProfile -o prof.out run.py
```

```
python -m cProfile -o prof.out run.py
```

Python Profiling Tools

- The SCC has 2 profiling visualization tools for Python

- [kcachegrind](#)

- Run using the Centos7 environment:
 - `scc-centos7 kcachegrind`

- Convert prof.out to the required file format open kcachegrind:

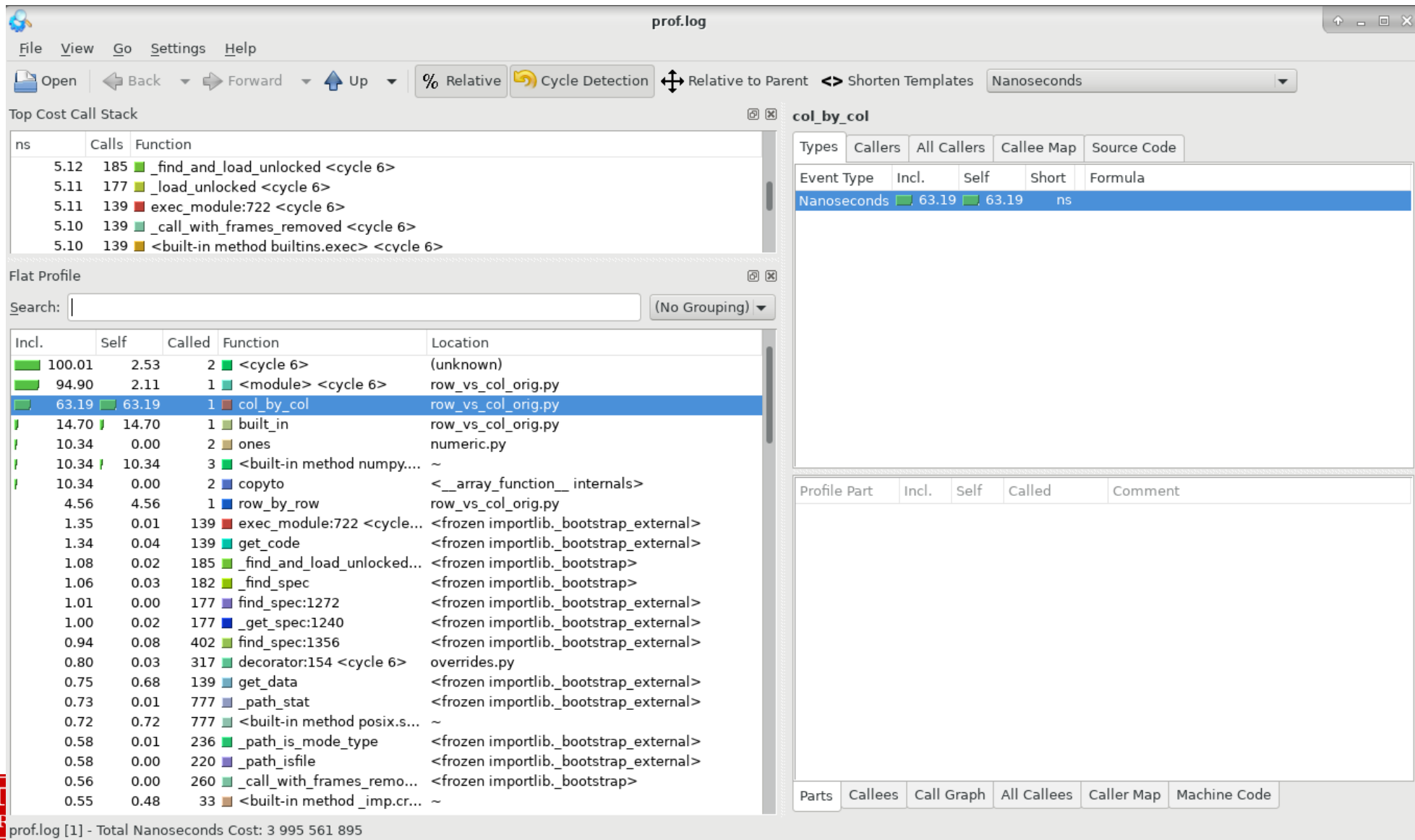
```
# pyprof2calltree is part of the  
# python3/3.10.12 module.  
pyprof2calltree -i prof.out -o prof.log  
scc-centos7 kcachegrind prof.log
```

- [snakeviz](#)

- Runs in a browser
- To use:

```
# 1st load your python3 module  
# one-time install  
pip install --user snakeviz  
~/.local/bin/snakeviz prof.out
```

kcachegrind



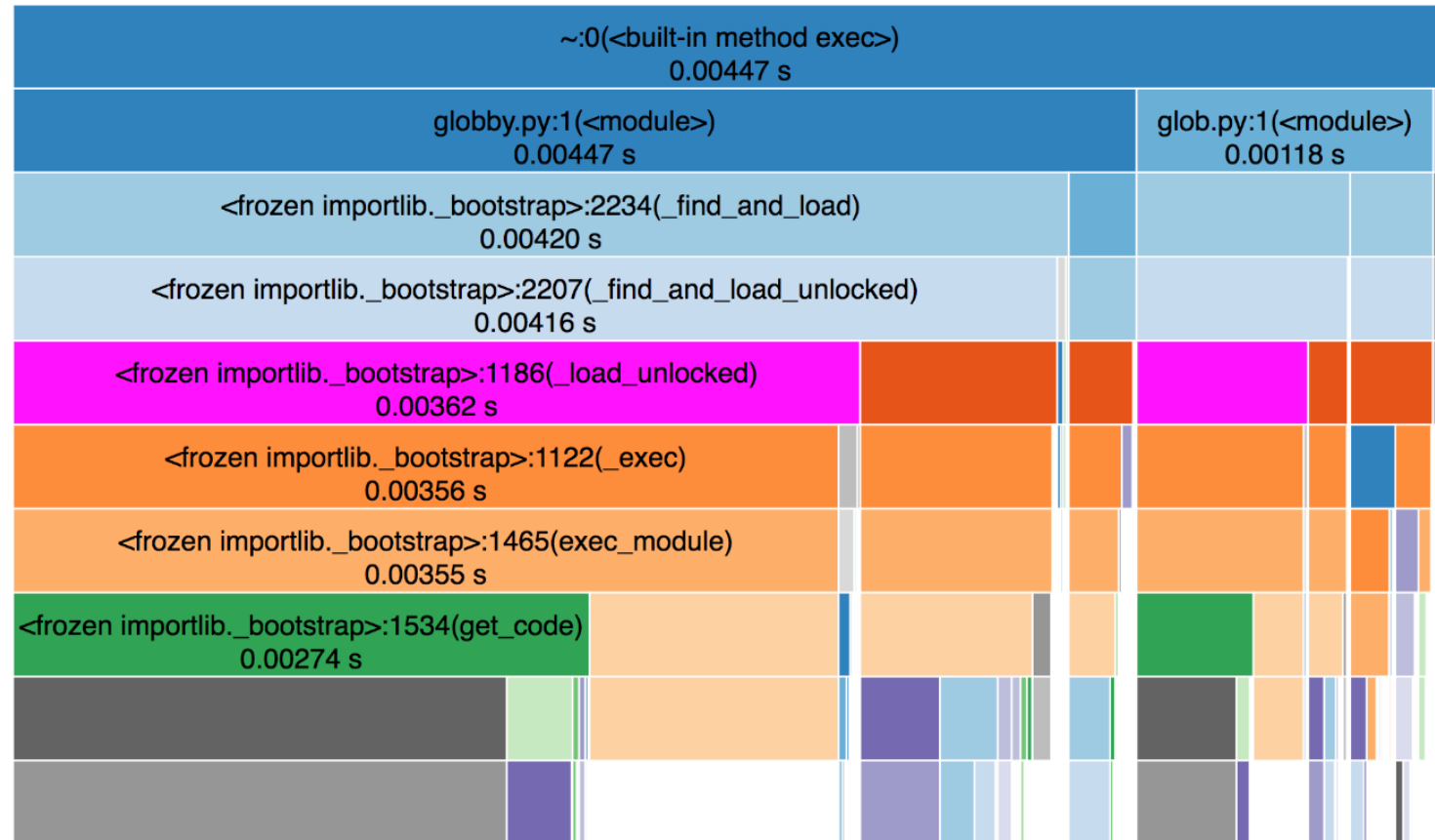
snakeviz

- Can be embedded into a Jupyter notebook:

In a Jupyter notebook:

```
%load_ext snakeviz
```

```
%snakeviz python_code_to_time...
```



Line-by-Line Profiling

- We've installed the *line_profiler* library.
- To use with *kernprof*, a command line tool:
 - Decorate functions with *@profile*
 - Do this for each function in *row_vs_col_orig.py*

```
from line_profiler import profile

@profile
def row_by_row(A,mat):
    ''' compute mat = A*mat row-by-row '''
    ...
```

Run kernprof

```
kernprof -l -o line.lprof row_vs_col_orig.py  
python -m line_profiler line.lprof
```

Timer unit: 1e-06 s

Total time: 0.204861 s

File: row_vs_col_orig.py

Function: row_by_row at line 32

Line #	Hits	Time	Per Hit	% Time	Line Contents
32					@profile
33					def row_by_row(A,x):
34					''' compute x = A*x row-by-row '''
35	1	5.0	5.0	0.0	rows = x.shape[0]
36	10001	5605.0	0.6	2.7	for i in range(rows):
37	10000	199251.0	19.9	97.3	x[i,:] = x[i,:] * A
38	1	0.0	0.0	0.0	return x

line_profiler from within Spyder

- Manually add the profiling to your script, run as usual.
- Older versions of Spyder (version 4) had a plug-in that loaded line_profiler results into the Spyder GUI. This does not exist for Spyder v5.

```
import line_profiler
profile = line_profiler.LineProfiler()

# function definitions here...

# Select the functions
profile.add_function(func_a)
profile.add_function(func_b)
profile.enable()

# run the rest of your program
# as usual...

# Turn off profiling, print the results.
profile.disable()

# Print the results
profile.print_stats()
```

line_profiler from within Jupyter with %lprun

- Option 1: Manually add to your script as in the previous slide.
- Option 2: Load the line-by-line profiler in your notebook and profile functions.
 - `%lprun` is the line-by-line profiler

```
def my_func(a,b,c):  
    ...python code...  
  
def looping(N,a,b,c):  
    for i in range(N):  
        my_func(a,b,c)  
  
%load_ext line_profiler  
  
N = 100  
a = b = c = 1.0  
  
# profile my_func as it gets  
# called by looping(). The  
# @profile decorator is not  
# needed.  
  
%lprun -f my_func looping(N,a,b,c)  
  
# line profiler output prints...
```

Memory Usage Profiling

- The *memory_profiler* library is used in a similar fashion.

```
@profile
def row_by_row(A,mat):
    ''' compute mat = A*mat row-by-row '''
    ...
```

- To use:
 - Decorate functions with *@profile*
- Run the script with the *memory_profiler* library.
- The output is printed to the screen.

```
python -m memory_profiler row_vs_col_orig.py
```

More ways to run...

- Import the library and decorate functions

```
import memory_profiler as mp

@mp.profile
def some_func(x,y,z):
    time.sleep((x+y+z)/3)

# Run as usual
```

- Jupyter
 - Load the memory profiler
 - Use `%memit` to get the peak memory used by a function call.

```
In [17]: %load_ext memory_profiler

%memit row_by_row(scaling_value,mat)
```

peak memory: 58.16 MiB, increment: 0.07 MiB

- Separate notebook files can be profiled with `%mprun`
- See [this web page](#) for details.

- Set *mat_size* = 10000. Output of: `python -m memory_profiler row_vs_col_orig.py`

Filename: row_vs_col_orig.py

Line #	Mem usage	Increment	Line Contents
40	808.535 MiB	808.535 MiB	@profile
41			def col_by_col(A,mat):
42			''' compute mat = A*mat col_by_col '''
43	808.535 MiB	0.000 MiB	cols = mat.shape[1]
44	808.535 MiB	0.000 MiB	for i in range(cols):
45	808.535 MiB	0.000 MiB	mat[:,i] = A * mat[:,i]
46	808.535 MiB	0.000 MiB	return x

Whoa!

Filename: row_vs_col_orig.py

Line #	Mem usage	Increment	Line Contents
48	808.535 MiB	808.535 MiB	@profile
49			def built_in(A,mat):
50			''' A*mat using built-in element-by-element'''
51	1571.477 MiB	762.941 MiB	return A * mat

2x memory usage...?

- This function is calculating the correct quantity.
- The syntax creates a new numpy array to hold the result which is returned.
- This is an in-place calculation:

```
def built_in(A,mat):  
    return A * mat  
  
mat = built_in(scaling_value, mat)
```

```
def row_by_row(A,mat):  
    rows = mat.shape[0]  
    for i in range(rows):  
        mat[i,:] = A * mat[i,:]  
    return x
```


Fix and re-profile the results.

```
def built_in(A,mat):  
    mat[:] = A * mat  
    return mat
```

- Memory usage is down:

Line #	Mem usage	Increment	Line Contents
46	808.543 MiB	808.543 MiB	@profile
47			def built_in(A,mat):
48	808.543 MiB	0.000 MiB	mat[:] = A * mat
49	808.543 MiB	0.000 MiB	return mat

- Using the profiling in Spyder:

Function	Time (msec)
row_by_row	183.57
col_by_col	2870.05
built_in (original version)	576.27
built_in (in-place version)	764.87

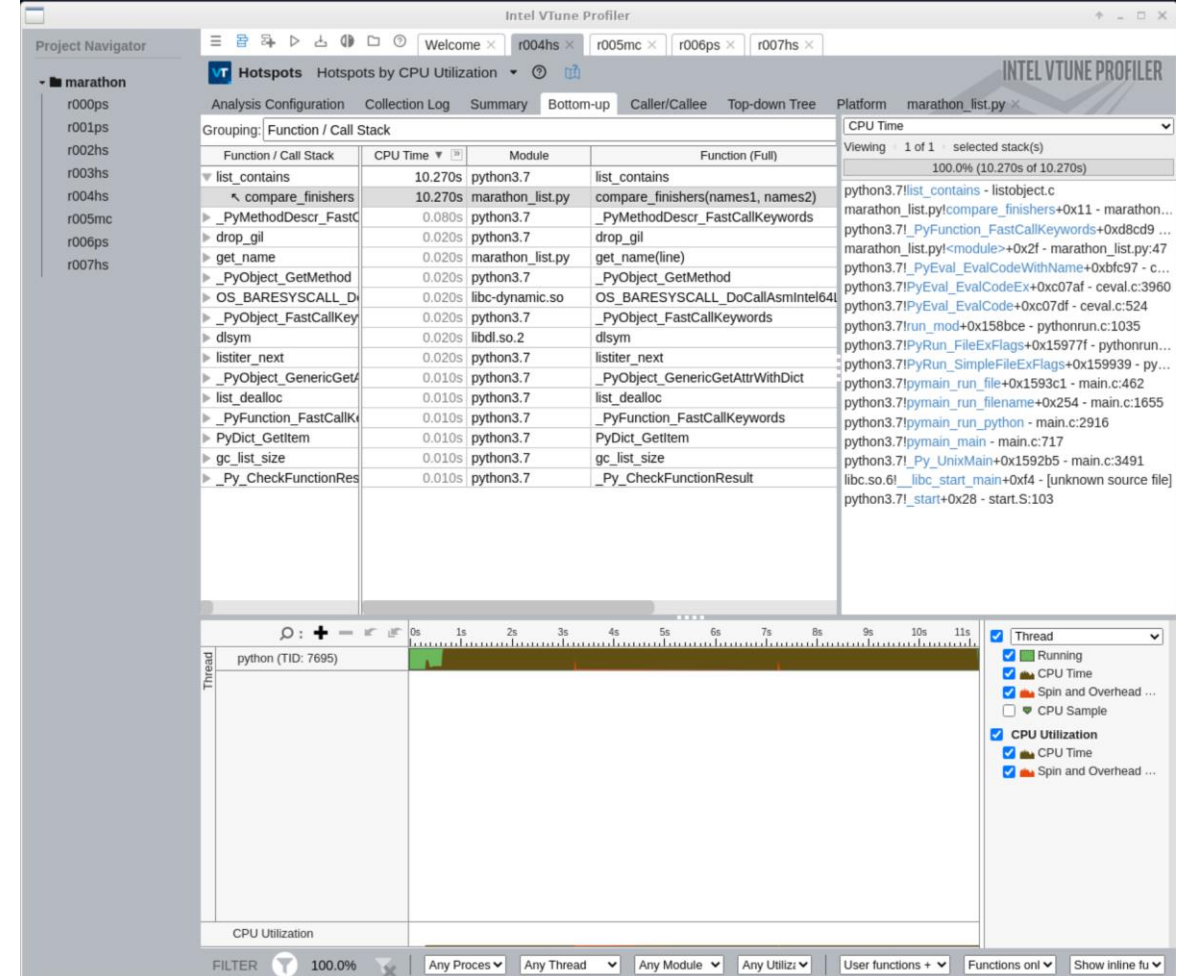
Interesting! More memory usage is faster...

Other Profiling Tools

- So far we've used:
 - Python's built-in profiler
 - line_profiler
 - memory_profiler
- Here are two more to consider:
 - Intel VTune Amplifier
 - Scalene

Intel Vtune Amplifier

- A comprehensive tool from Intel that can analyze Python scripts and the libraries they call for:
 - Function call times
 - “hotspots” – lines of code that consume excess time
 - Memory allocations
 - CPU and memory utilization
- Check out their [tutorials](#) and [documentation](#).
- Available in the intel/2024.0 module
module load intel/2024.0
vtune-gui &



Scalene



Select a profile (.json)

Time: Python | native | system Memory: Python | native Memory timeline: (max: 46.2MB, growth: 0.0%)

hover over bars to see breakdowns; click on COLUMN HEADERS to sort.

./test/testme.py: % of time = 99.9% out of 12.7s.

TIME	MEMORY average	MEMORY peak	MEMORY timeline	MEMORY activity	COPY (MB/s)	LINE PROFILE ./test/testme.py
						1 import numpy as np
						4 from numpy import linalg as LA
						12 x = [i*i for i in range(0,100000)][99999]
				10%		13 y1 = [i*i for i in range(0,200000)][199999]
				25%		14 z1 = [i for i in range(0,300000)][299999]
				63%		19 def doit2(x):
						24 while i < 100000:
						29 z = z * z
						30 z = x * x
						31 z = z * z
						32 z = z * z
						33 i += 1
				2%		46 y = np.random.randint(1, 100, size=500000)[4999999]
						47 x = 1.01

- A new profiling tool from [UMass Amherst](#).
- Easy to install (for Linux, Windows, and Mac):
 - pip install scalene
- Performs CPU, GPU, and memory profiling.

- The report is in HTML format and is displayed in a web browser.
- This can be called from within Jupyter notebooks as well as from a command line.

Profiling process

- Start with function-level profiling:
 - Spyder profiling
 - cProfile with kcache-grind or snakeviz
 - Identify problem functions.
- **LEARN THE TOOLS.**
 - Read the docs!
- Line profiler is slower than function timing so use where needed.
- Use memory profiling when it seems necessary.
 - Excess memory usage
 - Performance issues not easily solved with other methods.

Algorithm example

- Sometimes we have code that is written poorly.
- Profiling tells us where the problems are but we still need to find solutions.
- Let's look at an example and see if you can identify areas of poor performance:

bixi_slow.py

Outline

- Introduction
- Profiling
- Data Structures
- Generators
- Accelerators
- Syntax

Data Structures

- Algorithm implementation and performance is highly dependent on underlying data structures.
- Wikipedia has a [long list of established data structures](#).
- Find Python implementations at <https://pypi.org>
- Python data structures:
 - List
 - Dictionary
 - Aka “associative array”
 - Sets
 - Tuples
- These are sufficient to underpin a vast variety of algorithms.
- For manipulating numeric data use Numpy ndarrays or Pandas Dataframes.

Python data structures are fast when used for:

- Lists:
 - Appending
 - Element getting/setting
 - Removing from the end “pop()”
 - Get length
- Tuples (fixed lists):
 - Element getting
 - Get length
- Dictionaries:
 - Element getting/setting
 - Element membership
 - Element insertion
- Sets:
 - Element membership
 - Set operations (unions, intersections, etc)

Let's compare...

- Sample data: names of finishers of the 2015 and 2017 Boston Marathons
- Open files *marathon_list.py* and *marathon_set.py*
- Question: Who finished both the 2015 and 2017 marathons?
- Two implementations:
 - *marathon_list.py* loads the data into a pair of lists and then loops through them.
 - *marathon_set.py* loads the data into a pair of sets and intersects them.

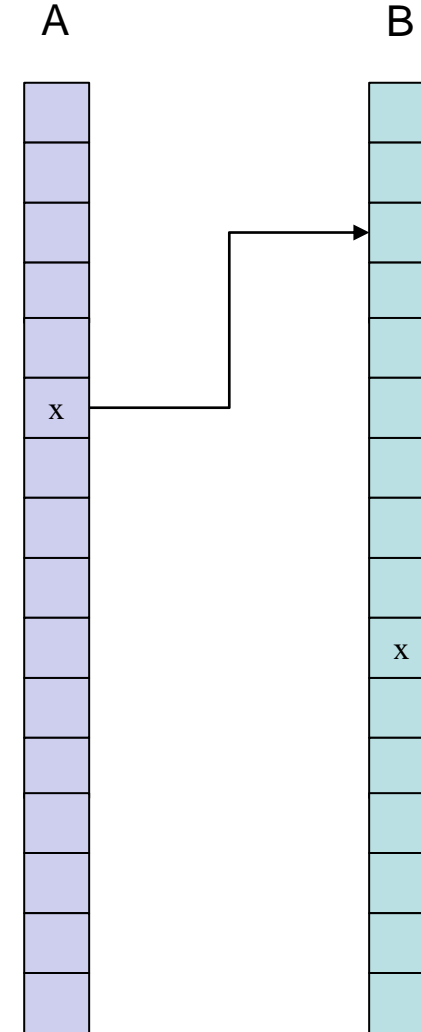


Performance

- Test each script using the Spyder profiler.
- Run it more than once – sometimes library or other code loading gives false timing.
- What did you find?
- Which one is faster?
- Are the results the same?

Lists

- The list lookup is $\sim 6300\times$ slower than the set intersection!
- Algorithm: For each element in list A check to see if it's in list B.
 - On average you need $\text{len}(B) / 2$ comparisons for every element in A.
 - That's approx. $\text{len}(A) * \text{len}(B) / 2$ operations. Each comparison is pretty fast.
 - For 26000 runners that's $\sim 350\text{M}$ string comparisons.



Sets

- Sets use a special data structure called a [hash table](#) to store elements.
 - Also used for dictionary keys.
 - The underlying hash function is **very** fast.
 - Lookup speed is nearly constant regardless of the size of the set.
- Algorithm: For each element in set A check to see if it's in set B.
 - You need $\text{len}(A)$ lookups into B. Each lookup in B takes a constant time τ .
 - That's $\text{len}(A)$ operations of time τ .
 - For 26000 runners there are 26000 hash comparisons.

- How do you choose?
- Test your code on different problem sizes.
- Profile your code if testing reveals problems.
- Read the documentation for available tools and libraries.
- Email RCS for help.

Understand your data

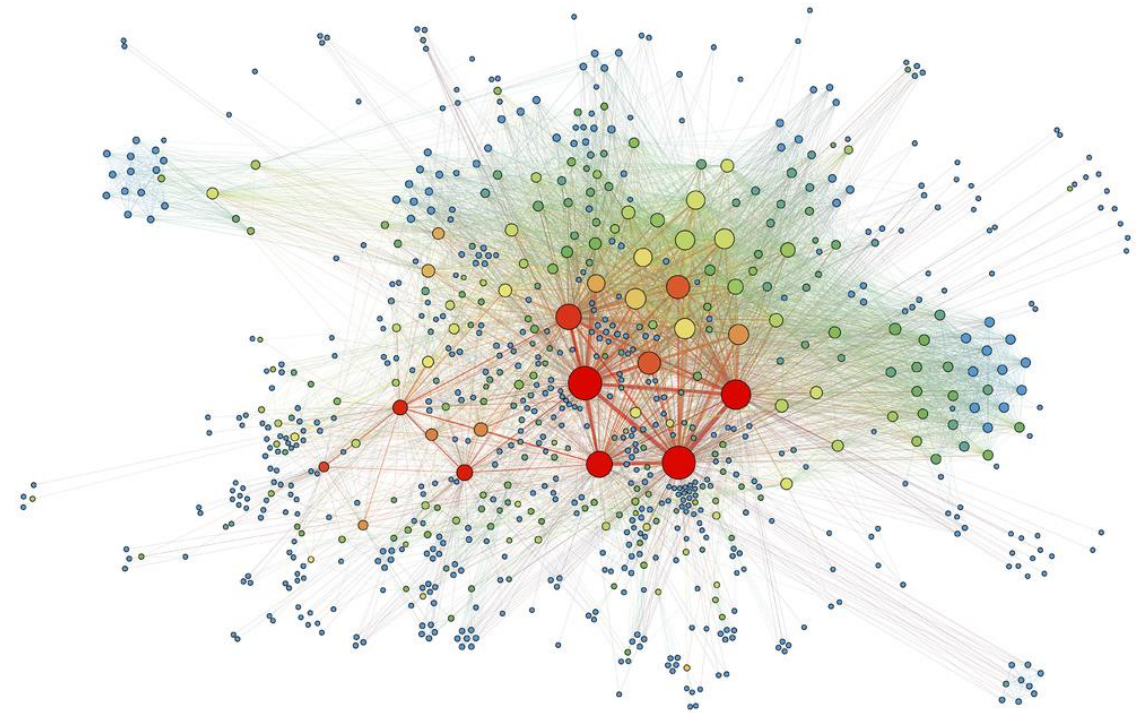
- Why were the results different?
- Sets only store unique values so some names got dropped.
- A better set solution would use a combination of factors to be more robust.
 - Example store this in the set as a tuple: (name, city, country, gender)



Graphs

- Some data is naturally understood as a graph.
 - A graph of people and their social connections to other people.
 - Contact tracing during a pandemic.
 - Journal articles and their authors.

- Python libraries: [networkx](#), [igraph](#), [graph-tool](#)



- Networkx is pure Python – it builds its graphs on a “dictionary of dictionaries of dictionaries”
- igraph is in C and C++.

Numpy and Pandas

- Numpy ndarrays are intended for high speed numeric calculations.
- Pandas dataframes are composed of ndarrays – similar pros & cons

Optimal usage:

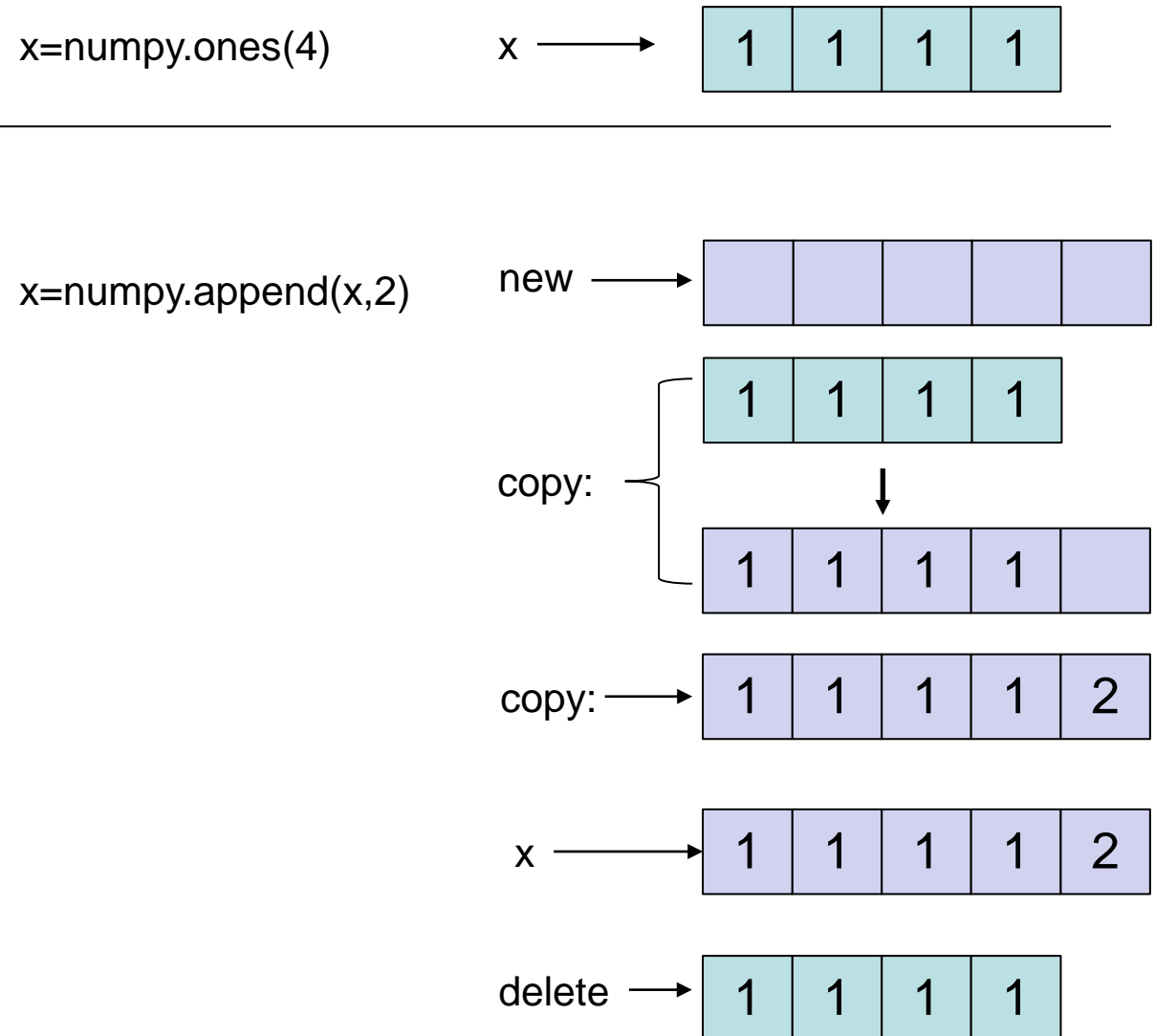
- Use built-in numpy functions wherever possible
 - If `x` is an ndarray...
 - `numpy.abs(x)` can operate on a whole ndarray.
 - `math.abs(x)` requires a Python loop
- Choose appropriate data types: float32, int, etc.
- Pre-allocate ndarrays to the correct size.
- Overwrite values with left-hand slice notation.

Pandas

- Read the [Pandas docs](#) that give performance tips.
- Also – docs on [scaling Pandas](#) to large data sets.
- Want to use multiple cores?
- Have really big data sets to process?
- Or both?
- Check out [Dask](#) and its DataFrame implementation.

Avoid numpy.append()!

- This **also** applies to `pandas.DataFrame.append()`
- `numpy.append()` for an ndarray with N elements:
 - Allocate a new ndarray of size N+1
 - Copy over the existing data
 - Copy in the new element.
 - Deallocate the old ndarray of N elements.



Some Numpy examples

- Open *numpy_solutions.py*
- Examples are provided for `append()`, pre-allocation, and proper use of library calls.

Outline

- Introduction
- Profiling
- Data Structures
- Generators
- Accelerators
- Syntax

Generators

- A Python generator is a function that behaves like an iterator.
- An iterator returns every element of a collection.
 - Example: a `for` loop iterates over the elements of a Python list.
- Generators can be used to create sequences of values one value at a time.

range()

- The range() function in Python is a generator.
- Try: `print(range(4))`
 - It won't print out any numbers – the output is not a list.
 - range() returns a generator that can be iterated over to produce a sequence of integers.

```
for x in range(1,4):  
    print(x)
```

```
# Output:
```

```
#     1
```

```
#     2
```

```
#     3
```

List comprehensions as generators

- List comprehensions are handy ways to create and manipulate lists.
- Intermediate lists or ones that are created and discarded still need to allocate memory.
- Generator syntax: use () instead of []
- No lists are created...little additional memory.

```
strs = ['call', 'me', 'ishmael']  
# uppercase all the strings  
caps = [L.upper() for L in strs]  
  
# Print them out  
for c in caps:  
    print(caps)  
  
gcaps = (L.upper() for L in strs)  
for g in gcaps:  
    print(g)
```

Generator Functions

- A generator function is written with the `yield` keyword.
- It will generate values until it reaches a `return` statement or throws a `StopIteration` exception.
- Every `yield` will return a value but the function keeps running until it returns.

```
import random

def triple_ran(N):
    ''' Return N tuples of 3
        random numbers.'''
    for i in range(N):
        vals=(random.random(),
              random.random(),
              random.random())
        yield vals
    return # optional

for triplet in triple_ran(4):
    print('%1.3f %1.3f %1.3f' % triplet)

# 0.070 0.363 0.821
# 0.668 0.705 0.235
# 0.384 0.817 0.071
# 0.633 0.303 0.591
```


Outline

- Introduction
- Profiling
- Data Structures
- Generators
- Accelerators
- Syntax

Accelerators

- These are some tools that can be used in conjunction with numpy to speed up numpy-based functions.

numba

- The [numba library](#) can translate portions of your Python code and compile it into machine code on demand.
- Achieves a significant speedup compared with regular Python.
- Compatible with numpy ndarrays.
- Can generate code to execute automatically on GPUs.

numba

- The @jit decorator is used to indicate which functions are compiled.
- Options:
 - GPU code generation
 - Parallelization
 - Caching of compiled code
- Can produce faster array code than pure NumPy statements.

```
from numba import njit
```

```
# This will get compiled when it's  
# first executed. The result will be  
# cached for re-use.
```

```
@njit(cache=True)
```

```
def average(x, y, z):  
    return (x + y + z) / 3.0
```

```
# With type information this one gets  
# compiled when the file is read.
```

```
@njit(float64(float64, float64, float64))
```

```
def average_eager(x, y, z):  
    return (x + y + z) / 3.0
```

numexpr

- Another acceleration library for Python.
 - This one seems to be waning in popularity
- Useful for speeding up specific ndarray expressions.
 - Typically 2-4x faster than plain NumPy
- Code needs to be edited to move ndarray expressions into the `numexpr.evaluate()` function:

```
import numpy as np
import numexpr as ne

a = np.arange(10)
b = np.arange(0, 20, 2)

# Plain NumPy
c = 2 * a + 3 * b

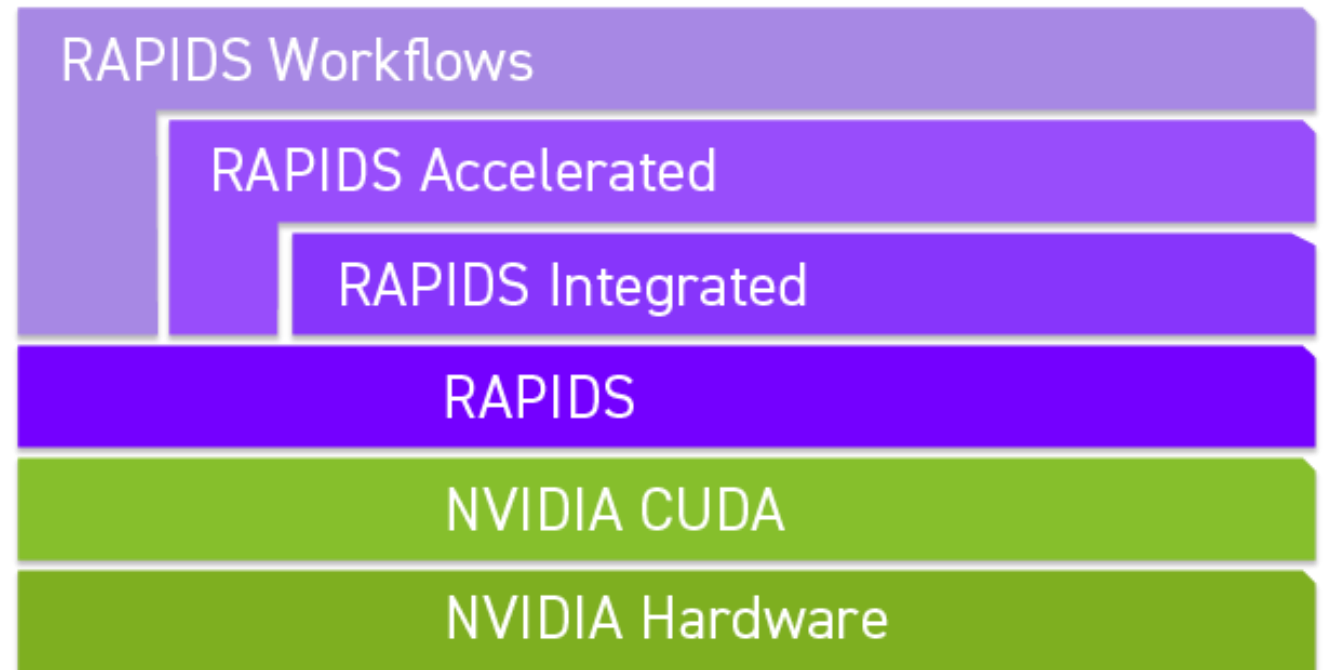
# Numexpr
d = ne.evaluate("2*a+3*b")
```

f2py

- Fortran code can be lightly modified and re-compiled into Python compatible functions.
- High performance routines are relatively easy to code in Fortran 95/2003.
- f2py is part of the numpy library.
- Compiled Fortran code can be >100x faster than equivalent Python code (even when based around numpy).

Rapids.ai

- “GPU Accelerated Data Science”
- Provides a number of libraries that execute on the GPU
- These are
 - Pandas → cudf
 - scipy, numpy → cupy
 - scikit-learn → cuml
 - Dask → dask-cuda
 - networkx → cudgraph
 - [And lots more](#)
- Easiest install on the SCC:
 - Use a [conda env](#)



Outline

- Introduction
- Profiling
- Data Structures
- Generators
- Accelerators
- Syntax

Python Syntax

- Here are some common ways where Python syntax can result in unintended consequences.

String concatenation

- Avoid excessive use of '+' as each '+' creates a temporary string.
- Very time and memory-intensive in loops.
- If strings are in a list (or similar thing) use the string join() function.
- Building a string in a loop? Append them to a list then join().

Best to avoid this

```
# a list of some strings
strs = ['a', 'b', 'c', ...]
s = ""
for i in strs:
    s += i
```

Less code, faster, and less memory.

```
strs = ['a', 'b', 'c', ...]
s = "".join(strs)
```

```
msg = []
for idx, elem in enumerate(some_data):
    # doing something...
    # record a message/result/etc
    msg.append('Step %s complete\n' % idx)
# Now concatenate
msg = ''.join(msg)
```

Slice Notation

- Lists:
 - RHS list slicing copies lists
 - LHS list slicing overwrites elements
- Numpy ndarrays:
 - RHS ndarray slicing creates a Numpy *view*
 - LHS ndarray slicing overwrites elements

```
x = [1,2,3,4]
# y is a new list
y = x[0:2]
# y --> [1,2]
x[0:2] = [-5,-6]
# 1st two elements
# of x are overwritten
# x --> [-5,-6,3,4]
```

```
x = numpy.array([1,2,3,4])
# y is a view into x
y = x[0:2]
# y --> x[0:2] --> [1,2]
x[0:2] = [-5,-6]
# 1st two elements
# of x are overwritten
# x --> [-5,-6,3,4]
# y --> x[0:2] --> [-5,-6]
```

The `del` command


- Temporary variables in loops – avoid the `del` command to clear out lists.
 - The `del` works by marking the elements of list `x` for deletion at some *later* time, *not* when the `del` is called.
 - The cleared elements of `x` aren't cleaned up until `x` goes out of scope.
 - This can result in a surprising amount of memory consumption!
- Instead re-declare `x` with each inner loop iteration.

No.

```
x=[]
sum = 0.0
for i in range(N):
    for j in range(M):
        # do something that
        # adds stuff to x
    sum += sum(x)
    # clear out x
    del x[:]
```

Yes.

```
sum = 0.0
for i in range(N):
    x = []
    for j in range(M):
        # do something that
        # adds stuff to x
    sum += sum(x)
```



Open files with `with`

- The Python `with` command when opening files will auto-handle the closing of the file.
- The operating system limits the number of files that can be opened...it's easy to forget a file `.close()` call.

```
import glob
import os
files = glob.glob(os.path.join(img_dir, '*.dat'))
# Do something with each data file
for datfile in files:
    dat=open(datfile,'r')
    some_func(dat.read())
    # If there are enough files and you
    # don't call this:
    # dat.close()
    # this loop WILL CRASH when you hit
    # your open file limit.

# Life is better with "with" :
for datfile in files:
    with open(datfile,'r') as dat:
        some_func(dat.read())
        # this guarantees the open file is
        # closed when this code block ends
        dat.close() ## this is now optional.
```

Python's itertools and functools libraries

- These two libraries are full of highly useful tools for manipulating Python functions and data structures.
- Well worth checking out!

- **itertools:**

- “The module standardizes a core set of fast, memory efficient tools that are useful by themselves or in combination. Together, they form an “iterator algebra” making it possible to construct specialized tools succinctly and efficiently in pure Python.”

- **functools:**

- “The functools module is for higher-order functions: functions that act on or return other functions. In general, any callable object can be treated as a function for the purposes of this module.”

End-of-course Evaluation Form

- Please visit this page and fill in the evaluation form for this course.
- Your feedback is highly valuable to the RCS team for the improvement and development of tutorials.
- If you visit this link later please make sure to select the correct tutorial – name, time, and location.

http://scv.bu.edu/survey/tutorial_evaluation.html