WRITING FASTER CODE

WRITING FASTER CODE AND NOT HATING YOUR JOB AS A SOFTWARE DEVELOPER

WRITING FASTER PYTHON

@SebaWitowski

SEBASTIAN WITOWSKI

CERN

```r
# https://en.wikipedia.org/wiki/R_(programming_language
install.packages("caTools")
library(caTools)
jet.colors <- colorRampPalette(c("green", "blue", "red",
                                 "cyan", "#7FFF7F",
                                 "yellow", "#FF7F00",
                                 "red", "#7F0000"))
m <- 1000
C <- complex( real=rep(seq(-2.10,0.8, length.out=m), ea
              imag=rep(seq(-1.2,1.2, length.out=m), m )
C <- matrix(C,m,m)
Z <- 0
X <- array(0, c(m,m,20))
for (k in 1:20) {
  Z <- Z^2+C
  X[,,k] <- exp(-abs(Z))
}
write.gif(X, "Mandelbrot.gif", col=jet.colors, delay=90
```

2.1

2.2

SMALL SCRIPTS...

```
# Fibonacci sequence (from rosettacode)

recfibo <- function(n) {
  if ( n < 2 ) n
  else Recall(n-1) + Recall(n-2)
}

# Print the first 21 elements
print.table(lapply(0:20, recfibo))
```

...WILL GROW

2.4

Shell - Konsole <2>

```
In [6]: s='IPython uses TAB for name completion. Hit TAB after the dot:'

In [7]: s.
s.capitalize  s.expandtabs  s.islower     s.lower       s.rstrip      s.title
s.center      s.find        s.isspace     s.lstrip      s.split       s.translate
s.count       s.index       s.istitle     s.replace     s.splitlines  s.upper
s.decode      s.isalnum     s.isupper     s.rfind       s.startswith
s.encode      s.isalpha     s.join        s.rindex      s.strip
s.endswith    s.isdigit     s.ljust       s.rjust       s.swapcase

In [7]: # Functions are automatically parenthesized, saving typing:

In [8]: s.replace 'TAB', 'tab'
------> s.replace ('TAB', 'tab')
Out[8]: 'IPython uses tab for name completion. Hit tab aft

In [9]: # Using ',' as the first character the quotes are

In [10]: ,s.replace tab TAB
-------> s.replace ("tab", "TAB")
Out[10]: 'IPython uses TAB for name completion. Hit TAB af

In [11]: # It offers functions to check your current names

In [12]: whos
Variable   Type    Data/Length
----------------------------
a          list    ['Python', 3000]
s          str     IPython uses TAB for<...>t TAB after t
x          int     45
z          float   5.6

In [13]: # and powerful object introspection:

In [14]: list2dict?
Type:           function
Base Class:     <type 'function'>
String Form:    <function list2dict at 0x8072e5c>
Namespace:      User-defined configuration
File:           /usr/local/home/fperez/local/python/IPytho
Definition:     list2dict(lst)
Docstring:
    Takes a list of (key,value) pairs and turns it into a

In [15]:
```

spectrogram - Iceweasel

localhost:8889/notebooks/spectrogram.ipynb

IP[y]: Notebook spectrogram Last Checkpoint: a few seconds ago (autosaved)

IPython (Python 3)

File Edit View Insert Cell Kernel Help

Code Cell Toolbar: None

Simple spectral analysis

An illustration of the Discrete Fourier Transform using windowing, to reveal the frequency content of a sound signal.

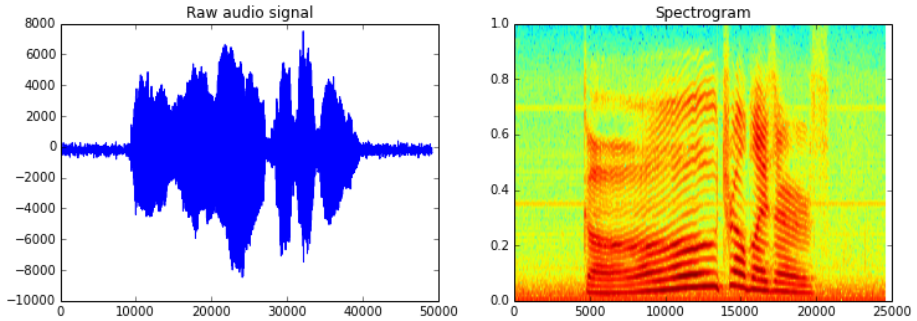
$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} kn} \qquad k = 0, \ldots, N-1$$

We begin by loading a datafile using SciPy's audio file support:

In [1]:
```
from scipy.io import wavfile
rate, x = wavfile.read('test_mono.wav')
```

And we can easily view its spectral structure using matplotlib's builtin `specgram` routine:

In [2]:
```
%matplotlib inline
from matplotlib import pyplot as plt
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))
ax1.plot(x); ax1.set_title('Raw audio signal')
ax2.specgram(x); ax2.set_title('Spectrogram');
```



PYTHON WAS NOT MADE TO BE FAST...

...BUT TO MAKE DEVELOPERS FAST.

> *It was nice to learn Python;*
> *a nice afternoon*

Donald Knuth

Would you like your FIRST program EVER to be like:

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
}
```

or

```
print("Hello, world!")
```

Source: https://www.shoop.io/en/blog/25-of-the-most-popular-python-and-django-websites

OPTIMIZATION

Google

optimization rules

All | Images | Videos | News | Shopping | More | Search tools

About 119.000.000 results (0,29 seconds)

Rules Of Optimization
c2.com/cgi/wiki?**Rules**Of**Optimization**
The "**rules**" of optimising are a rhetorical device intended to dissuade novice programmers from cluttering up their programs with vain attempts at writing optimal ...

The Rules of Code Optimization | The Audio Fool - MSDN Blogs
https://blogs.msdn.microsoft.com/.../06/.../the-**rules**-of-code-**optimization**...
Jun 14, 2007 - Steve also makes a point about premature **optimization**, and how it affects readability. This reminded me of a list of the **Rules** of **Optimization** ...

Program optimization - Wikipedia, the free encyclopedia
https://en.wikipedia.org/wiki/Program_**optimization**
In computer science, program **optimization** or software **optimization** is the process of modifying ..... The Second **Rule** of Program **Optimization** (for experts only!) ...

People also ask

What is an optimization problem?

What do you mean by code optimization?

What is optimization in software?

Rules Of Optimization

The "rules" of optimising are a rhetorical device intended to dissuade novice programmers from cluttering up their programs with vain attempts at writing optimal code. They are:

1. **FirstRuleOfOptimization - Don't.**
2. **SecondRuleOfOptimization - Don't... *yet.***
3. **ProfileBeforeOptimizing**

It is uncertain at present, whether cute devices such as this have, or ever will, change any attitudes.

It changed mine.

Mine, too.

Source:

MichaelJackson used to say (when asked about optimization):

1. Don't.
2. Don't Yet (for experts only).

This is republished in JonBentley's ProgrammingPearls.

And lets not forget these famous quotes:

> "The best is the enemy of the good."
>
> -- MrVoltaire
>
> "More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason - including blind stupidity."
>
> -- W.A. Wulf
>
> "We should forget about small efficiencies, say about 97% of the time: PrematureOptimization is the root of all evil."
>
> -- DonKnuth (who attributed the observation to CarHoare)

See: OptimizeLater, LazyOptimization, OptimizationUnitTest, OptimizationStories, http://c2.com/cgi/wiki?search=optimiz, UniformlySlowCode, CodeDepreciation, RulesOfOptimizationClub

---

CategoryOptimization

View edit of May 6, 2009 or FindPage with title or text search

RULES OF OPTIMIZATION:

1. DON'T
2. DON'T... YET
3. PROFILE

OPTIMIZATION IS ALL ABOUT THE SPEED

... AND MEMORY

... AND DISK SPACE

... DISK I/O

... NETWORK I/O

... POWER CONSUMPTION

... AND MORE.

> *Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live*

John Woods

CPROFILE

```
In [1]: import re

In [2]: import cProfile

In [3]: cProfile.run('re.compile("foo|bar")')
         185 function calls (180 primitive calls) in 0.001 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.000    0.000 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 re.py:222(compile)
        1    0.000    0.000    0.000    0.000 re.py:278(_compile)
        1    0.000    0.000    0.000    0.000 sre_compile.py:221(_compile_charset)
        1    0.000    0.000    0.000    0.000 sre_compile.py:248(_optimize_charset)
        1    0.000    0.000    0.000    0.000 sre_compile.py:412(_compile_info)
        2    0.000    0.000    0.000    0.000 sre_compile.py:513(isstring)
        1    0.000    0.000    0.000    0.000 sre_compile.py:516(_code)
        1    0.000    0.000    0.000    0.000 sre_compile.py:531(compile)
      3/1    0.000    0.000    0.000    0.000 sre_compile.py:64(_compile)
        3    0.000    0.000    0.000    0.000 sre_parse.py:105(__init__)
        5    0.000    0.000    0.000    0.000 sre_parse.py:153(__len__)
       12    0.000    0.000    0.000    0.000 sre_parse.py:157(__getitem__)
        7    0.000    0.000    0.000    0.000 sre_parse.py:165(append)
      3/1    0.000    0.000    0.000    0.000 sre_parse.py:167(getwidth)
        1    0.000    0.000    0.000    0.000 sre_parse.py:217(__init__)
        8    0.000    0.000    0.000    0.000 sre_parse.py:226(__next)
        2    0.000    0.000    0.000    0.000 sre_parse.py:242(match)
        6    0.000    0.000    0.000    0.000 sre_parse.py:247(get)
        1    0.000    0.000    0.000    0.000 sre_parse.py:276(tell)
        1    0.000    0.000    0.000    0.000 sre_parse.py:429(_parse_sub)
        2    0.000    0.000    0.000    0.000 sre_parse.py:491(_parse)
        1    0.000    0.000    0.000    0.000 sre_parse.py:70(__init__)
        2    0.000    0.000    0.000    0.000 sre_parse.py:75(groups)
        1    0.000    0.000    0.000    0.000 sre_parse.py:797(fix_flags)
        1    0.000    0.000    0.000    0.000 sre_parse.py:819(parse)
        1    0.000    0.000    0.000    0.000 {built-in method _sre.compile}
        1    0.000    0.000    0.001    0.001 {built-in method builtins.exec}
       17    0.000    0.000    0.000    0.000 {built-in method builtins.isinstance}
    25/24    0.000    0.000    0.000    0.000 {built-in method builtins.len}
        2    0.000    0.000    0.000    0.000 {built-in method builtins.max}
        9    0.000    0.000    0.000    0.000 {built-in method builtins.min}
        6    0.000    0.000    0.000    0.000 {built-in method builtins.ord}
       48    0.000    0.000    0.000    0.000 {method 'append' of 'list' objects}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
        5    0.000    0.000    0.000    0.000 {method 'find' of 'bytearray' objects}
        1    0.000    0.000    0.000    0.000 {method 'items' of 'dict' objects}
```

PSTATS

```
In [1]: import pstats

In [2]: p = pstats.Stats('stats.txt')

In [3]: p.sort_stats('cumulative').print_stats(10)
Thu Sep  8 11:37:40 2016    stats.txt

         1817 function calls (1795 primitive calls) in 0.006 seconds

   Ordered by: cumulative time
   List reduced from 118 to 10 due to restriction <10>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      3/1    0.000    0.000    0.006    0.006 {built-in method builtins.exec}
        1    0.000    0.000    0.006    0.006 myscript.py:1(<module>)
      5/1    0.000    0.000    0.006    0.006 <frozen importlib._bootstrap>:966(_find_and_load)
      5/1    0.000    0.000    0.006    0.006 <frozen importlib._bootstrap>:939(_find_and_load_unlocked)
      5/1    0.000    0.000    0.006    0.006 <frozen importlib._bootstrap>:659(_load_unlocked)
      2/1    0.000    0.000    0.006    0.006 <frozen importlib._bootstrap_external>:656(exec_module)
      8/1    0.000    0.000    0.005    0.005 <frozen importlib._bootstrap>:214(_call_with_frames_removed)
        1    0.000    0.000    0.005    0.005 /home/switowsk/.pyenv/versions/3.5.1/lib/python3.5/random.py:37(<module>)
        5    0.000    0.000    0.003    0.001 <frozen importlib._bootstrap>:570(module_from_spec)
        1    0.000    0.000    0.002    0.002 /home/switowsk/.pyenv/versions/3.5.1/lib/python3.5/hashlib.py:53(<module>)


Out[3]: <pstats.Stats at 0x7f6b3da98780>

In [4]: p.sort_stats('ncalls').print_stats(10)
Thu Sep  8 11:37:40 2016    stats.txt

         1817 function calls (1795 primitive calls) in 0.006 seconds

   Ordered by: call count
   List reduced from 118 to 10 due to restriction <10>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      244    0.000    0.000    0.000    0.000 {method 'rstrip' of 'str' objects}
      126    0.000    0.000    0.000    0.000 <frozen importlib._bootstrap_external>:363(_verbose_message)
      124    0.000    0.000    0.000    0.000 {method 'join' of 'str' objects}
      120    0.000    0.000    0.000    0.000 <frozen importlib._bootstrap_external>:50(_path_join)
      120    0.000    0.000    0.000    0.000 <frozen importlib._bootstrap_external>:52(<listcomp>)
      116    0.000    0.000    0.000    0.000 {method 'format' of 'str' objects}
       61    0.000    0.000    0.000    0.000 {method 'getrandbits' of '_random.Random' objects}
       44    0.000    0.000    0.000    0.000 {method 'rpartition' of 'str' objects}
       44    0.000    0.000    0.000    0.000 {built-in method builtins.hasattr}
       36    0.000    0.000    0.000    0.000 {built-in method builtins.getattr}
```

RUNSNAKERUN

tests/OpenGLContext.profile

File View

Percent File View

Name	Calls	RCalls	Loca
parame...	6	6	0.0
parents	14	14	0.0
parse	16	16	0.0
parse	2	2	0.0
parse	1	1	0.0
pathna...	1	1	0.0
permute	275	275	0.0
pickle	3	3	0.0
poll	87602	87602	0.3
poll	87602	87602	0.1
polygons	3571	3571	0.6
protoN...	72	72	0.0
prototype	3	3	0.0
pushOut	2	2	0.0
pyArgIn...	39	39	0.0
pygam...	1	1	0.0
pygam...	1	1	0.0
quatern...	2	2	0.0
quote	1	1	0.0
range	11	11	0.0
rankPac...	9	9	0.0
ravel	414	414	0.0
realpath	330	356	0.0
regDef...	771	771	0.0
register	187	187	0.0
register	50	50	0.0
register	11	11	0.0
register	1	1	0.0
register...	115	115	0.0
register...	11	11	0.0

Render@shape.py:30 [12.823s]

render@indexedpolygons.py:61 [5.097

callBound@indexedpo

enableNorm

callBound@

void

bir

callBound@

get_vt

call

vol

polygons@indexedfaceset

polyg

__call__@in

vertexIn

polygons@indexedfaceset.py:310 [1.627s]

render@appeara

render@mater

getDa

__call

render@arraygeometry.py:126 [2.246s]

g

callBound@arraygeometr

draw@array

binc

unb

render@image

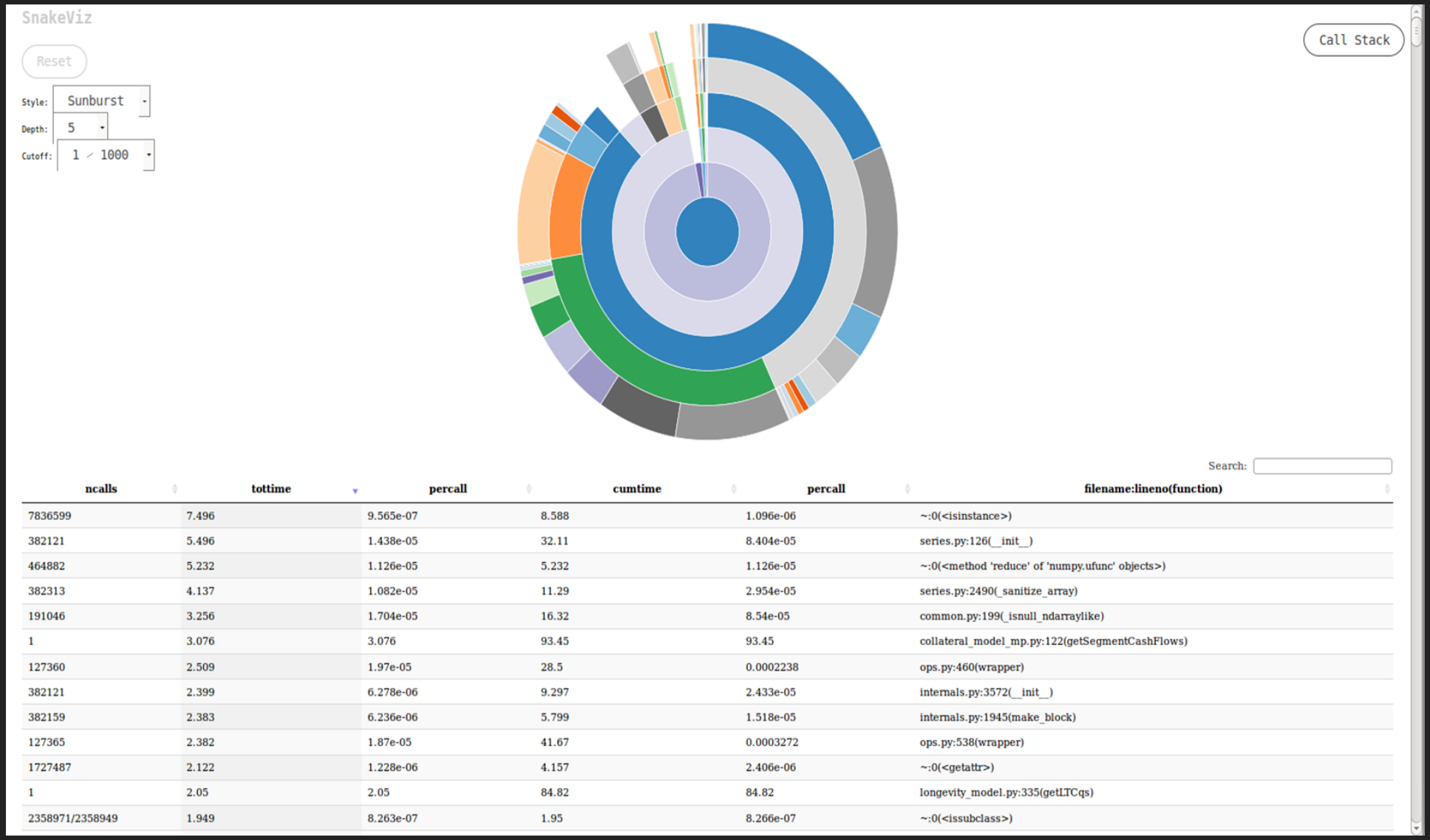
renderPo

Callees | All Callees | Callers | All Callers | Source Code

Name	Calls	RCalls	Local	/Call	Cum	/Call	File	Line	Directory
<len>	891503	914656	1.23234	0.00000	1.32067	0.00000	~	0	
__call__	37017	37017	0.07093	0.00000	0.07093	0.00000	convert...	155	/home/

polygons@indexedfaceset.py:310 [1.627s]

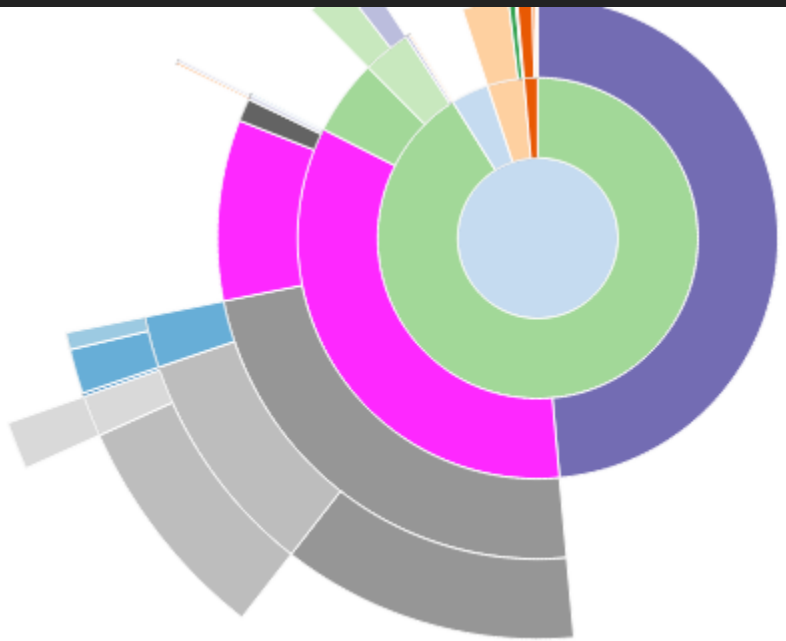
SNAKEVIZ



ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
7836599	7.496	9.565e-07	8.588	1.096e-06	~:0(<isinstance>)
382121	5.496	1.438e-05	32.11	8.404e-05	series.py:126(__init__)
464882	5.232	1.126e-05	5.232	1.126e-05	~:0(<method 'reduce' of 'numpy.ufunc' objects>)
382313	4.137	1.082e-05	11.29	2.954e-05	series.py:2490(_sanitize_array)
191046	3.256	1.704e-05	16.32	8.54e-05	common.py:199(_isnull_ndarraylike)
1	3.076	3.076	93.45	93.45	collateral_model_mp.py:122(getSegmentCashFlows)
127360	2.509	1.97e-05	28.5	0.0002238	ops.py:460(wrapper)
382121	2.399	6.278e-06	9.297	2.433e-05	internals.py:3572(__init__)
382159	2.383	6.236e-06	5.799	1.518e-05	internals.py:1945(make_block)
127365	2.382	1.87e-05	41.67	0.0003272	ops.py:538(wrapper)
1727487	2.122	1.228e-06	4.157	2.406e-06	~:0(<getattr>)
1	2.05	2.05	84.82	84.82	longevity_model.py:335(getLTCqs)
2358971/2358949	1.949	8.263e-07	1.95	8.266e-07	~:0(<issubclass>)

SNAKEVIZ

Name:
`filter`
Cumulative Time:
`0.000294 s (31.78 %)`
File:
`fnmatch.py`
Line:
`48`
Directory:
`/Users/jiffyclub/miniconda3/envs/snakevizdev/lib/python3.4/`



MEMORY_PROFILER

https://pypi.python.org/pypi/memory_profiler

```
$ pip install -U memory_profiler
```

```
@profile
def my_func():
    a = [1] * (10 ** 6)
    b = [2] * (2 * 10 ** 7)
    del b
    return a
```

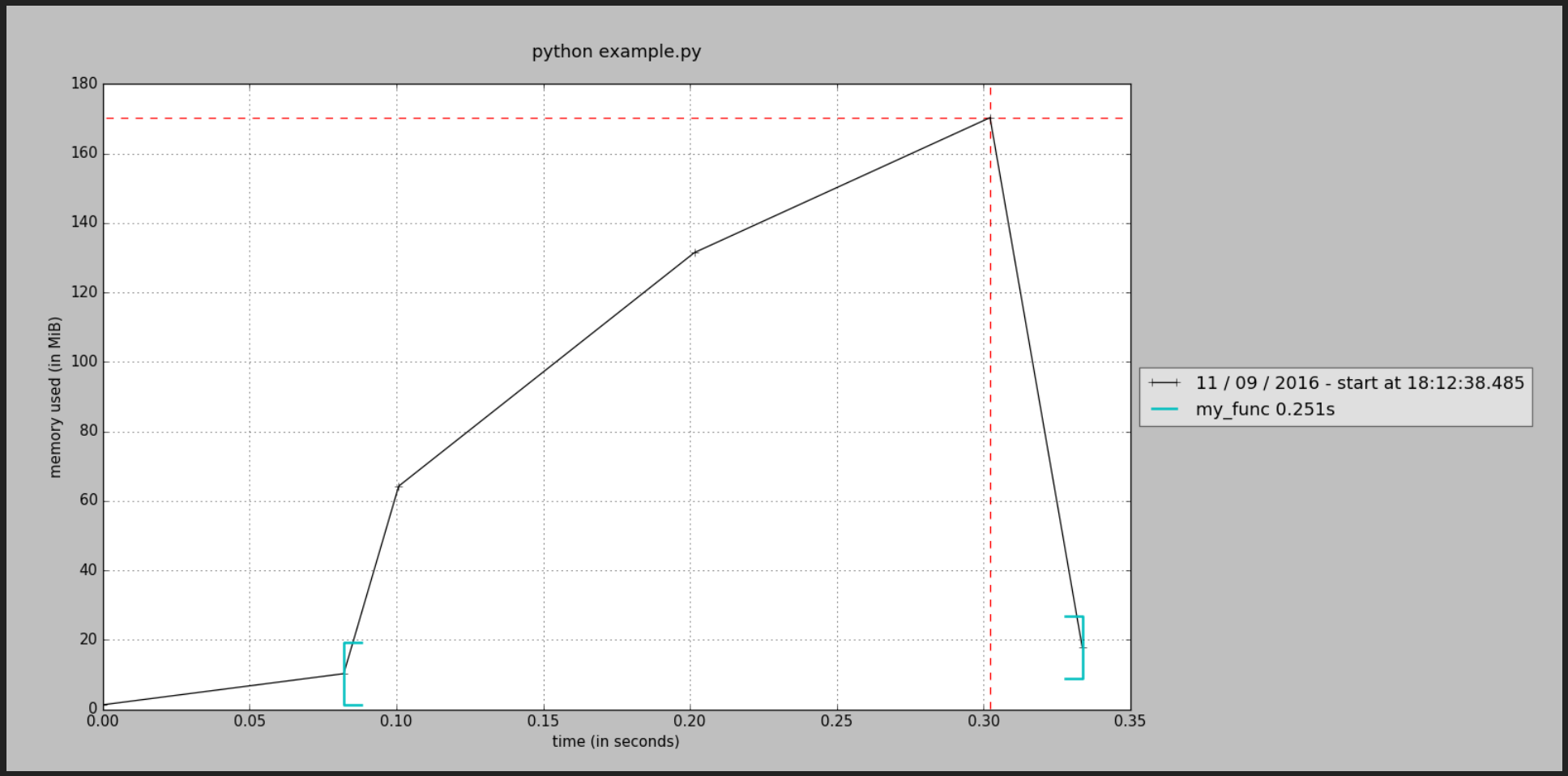
MEMORY_PROFILER

```
$ python -m memory_profiler example.py
Filename: example.py

Line #    Mem usage    Increment   Line Contents
================================================
     1   10.289 MiB    0.000 MiB   @profile
     2                             def my_func():
     3   17.926 MiB    7.637 MiB       a = [1] * (10 ** 6)
     4  170.516 MiB  152.590 MiB       b = [2] * (2 * 10 ** 7)
     5   17.926 MiB -152.590 MiB       del b
     6   17.926 MiB    0.000 MiB       return a
```

MEMORY_PROFILER

```
mprof run example.py
mprof plot
```



https://www.huyng.com/posts/python-performance-analysis

LEVELS OF OPTIMIZATION

- Design
- Algorithms and data structures

```
sum = 0
for x in range(1, N + 1):
    sum += x
print sum
```

↓

```
print N * (1 + N) / 2
```

LEVELS OF OPTIMIZATION

- Design
- Algorithms and data structures
- Source code
- Build level
- Compile level
- Runtime level

WRITING FAST PYTHON

A.K.A SOURCE CODE OPTIMIZATION

SETUP

Python 3.5.1 (IPython 1.2.1)

```
def ultimate_answer_to_life():
    return 42
```

```
>>> %timeit ultimate_answer_to_life()
10000000 loops, best of 3: 87.1 ns per loop
```

2.72×10^{21} times faster than in The Hitchhiker's Guide to the Galaxy ;-)

#1 COUNT ELEMENTS IN A LIST

```
how_many = 0
for element in ONE_MILLION_ELEMENTS:
    how_many += 1
print how_many
```

26.5 ms

```
print len(ONE_MILLION_ELEMENTS)
```

96.7 ns

274 000 times faster

		Built-in Functions		
abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

And collections module.

#2 SUM ELEMENTS IN A LIST

```
sum(ONE_MILLION_ELEMENTS)
```

15.3 ms

```
# ONE_MILLION_ELEMENTS is a numpy.array
numpy.sum(ONE_MILLION_ELEMENTS)
```

630 µs

24 times faster

#3 FILTER A LIST

```
output = []
for element in MILLION_NUMBERS:
    if element % 2:
        output.append(element)
```

222 ms

```
list(filter(lambda x: x % 2, MILLION_NUMBERS))
```

234 ms

```
[item for item in MILLION_NUMBERS if item % 2]
```

127 ms

75% faster

#4 PERMISSIONS OR FORGIVENESS ?

```
class Foo(object):
    hello = 'world'
foo = Foo()
```

```
if hasattr(foo, 'hello'):
    foo.hello
```

149 ns

```
try:
    foo.hello
except AttributeError:
    pass
```

43.1 ns

3.5 times faster

#4 PERMISSIONS OR FORGIVENESS ?

```
if (hasattr(foo, 'foo') and hasattr(foo, 'bar')
    and hasattr(foo, 'baz')):
    foo.foo
    foo.bar
    foo.baz
```

401 ns

```
try:
    foo.foo
    foo.bar
    foo.baz
except AttributeError:
    pass
```

110 ns

3.64 times faster

#4 PERMISSIONS OR FORGIVENESS ?

```
class Bar(object):
    pass
bar = Bar()
```

```
if hasattr(bar, 'hello'):
    bar.hello
```

428 ns

```
try:
    bar.hello
except AttributeError:
    pass
```

536 ns

25% slower

#5 MEMBERSHIP TESTING

```
def check_number(number):
    for item in MILLION_NUMBERS:
        if item == number:
            return True
    return False
```

```
%timeit check_number(500000)
```

18 ms

```
500000 in MILLION_NUMBERS
```

8.45 ms

2 times faster

#5 MEMBERSHIP TESTING

```
100 in MILLION_NUMBERS
```

1.55 µs

```
999999 in MILLION_NUMBERS
```

15.7 ms

#5 MEMBERSHIP TESTING

```
MILLION_SET = set(MILLION_NUMBERS)
%timeit 100 in MILLION_SET
```

46.3 ns

33 times faster (vs list)

```
%timeit 999999 in MILLION_SET
```

63.3 ns

248 000 times faster (vs list)

```
%timeit set(MILLION_NUMBERS)
```

106 ms

#6 REMOVE DUPLICATES

```
unique = []
for element in MILLION_ELEMENTS:
    if element not in unique:
        unique.append(element)
```

8.29 s

```
set(MILLION_ELEMENTS)
```

19.3 ms

400 times faster

Trick with OrderedDict (if order matters)

#7 LIST SORTING

```
sorted(MILLION_RANDOM_NUMBERS)
```

467 ms

```
MILLION_RANDOM_NUMBERS.sort()
```

77.6 ms

6 times faster

#8 1000 OPERATIONS AND 1 FUNCTION

```
def square(number):
    return number**2
squares = [square(i) for i in range(1000)]
```

399 µs

```
def compute_squares():
    return [i**2 for i in range(1000)]
```

314 µs

27% faster

#9 CHECKING FOR TRUE

```
if variable == True:
```

35.8 ns

```
if variable is True:
```

28.7 ns

24% faster

```
if variable:
```

20.6 ns

73% faster

#9.1 CHECKING FOR FALSE

```
if variable == False:
```

35.1 ns

```
if variable is False:
```

26.9 ns

30% faster

```
if not variable:
```

19.8 ns

77% faster

#9.2 CHECKING FOR EMPTY LIST

```
if len(a_list) == 0:
```

91.7 ns

```
if a_list == []:
```

56.3 ns

60% faster

```
if not a_list:
```

32.4 ns

280% faster

#10 DEF VS LAMBDA

```
def greet(name):
    return 'Hello {}!'.format(name)
```

329 ns

```
greet = lambda name: 'Hello {}!'.format(name)
```

332 ns

#10 DEF VS LAMBDA

```
>>> dis.dis(greet)
 0 LOAD_CONST    1 ('Hello {}!')
 3 LOAD_ATTR     0 (format)
 6 LOAD_FAST     0 (name)
 9 CALL_FUNCTION 1 (1 positional, 0 keyword pair)
12 RETURN_VALUE
```

Stack Overflow question on when lambda might be necessary

#11 LIST() OR []

```
list()
```

104 ns

```
[]
```

22.5 ns

4.6 times faster

#11.1 DICT() OR {}

```
dict()
```

161 ns

```
{}
```

93 ns

1.7 times faster

DANGER ZONE

#12 VARIABLES ASSIGNMENT

```
q=1
w=2
e=3
r=4
t=5
y=6
u=7
i=8
o=9
p=0
```

71.8 ns

```
q,w,e,r,t,y,u,i,o,p = 1,2,3,4,5,6,7,8,9,0
```

56.4 ns

27% faster (but please don't)

#13 VARIABLES LOOKUP

```
def squares(MILLION_NUMBERS):
    output = []
    for element in MILLION_NUMBERS:
        output.append(element*element)
    return output
```

149 ms

```
def squares_faster(MILLION_NUMBERS):
    output = []
    append = output.append # <= !!!!!!!!
    for element in MILLION_NUMBERS:
        append(element*element)
    return output
```

110 ms

35% faster (and 42% more confusing)

SUMMARY

- There are different kinds of optimization
- There are different levels of optimization
- Source code optimizations adds up
- Source code optimizations is cheap
  - Idiomatic Python
  - Don't reinvent the wheel
- Profile your code and be curious!

THANK YOU!

HAPPY AND FAST CODING!

Check the slides at:
http://switowski.github.io/itweekend-2016