Analyzing Python Performance

Bob Hancock

hancock.robert@gmail.com

Section I

Measure

Measure

- If you didn't measure it, it did happen.
- Never trust your instincts.
- Is it reproducible?
- What do you measure?
- How do you measure?
- Why do you measure?

Section 2

Runtime

What is the runtime?

- How long does it take in ***real*** time?

```
time python myprogram.py
real    0m1.028s
user    0m0.001s
sys     0m0.003s
```

UNIX time

- real time – the actual (clock) elapsed time.

- user – the amount of time the program spent in user mode.

- sys – the amount of time the program spent in kernel mode.

Approximate cpu cycles

$$\text{cpu cycles} = \text{sys} + \text{user}$$

- Adding these two values gives you an approximation of cpu cycles.
- If cpu cycles < real time then it is an indicator then you want to investigate the IO waits.

Section 3

Profile

cProfile

- The profiler written in C for functions.
- ncalls – number of calls
- tottime – total time in a function, excluding sub-functions
- percall – tottime / ncalls
- cumtime – time in this function and sub-functions.
- percall – cumtime / primitive calls

cProfile

```
 126491937 function calls (41 primitive calls) in 38.119 seconds

  Ordered by: standard name

  ncalls  tottime  percall  cumtime  percall filename:lineno(function)
       1    0.000    0.000   38.119   38.119 fibonacci.py:11(main)
       1    0.000    0.000   38.119   38.119 fibonacci.py:2(<module>)
126491932/36   38.119    0.000   38.119    1.059 fibonacci.py:6(fib)
       1    0.000    0.000    0.000    0.000 memo.py:1(<module>)
       1    0.000    0.000    0.000    0.000 {len}
       1    0.000    0.000    0.000    0.000 {method 'disable' of
                                                 '_lsprof.Profiler' objects}
```

Section 4

Instrumentation

Instrumentation

- Code instructions that allow you to measure and monitor execution performance.
- Minimal intrusion versus logging.
- Start with time()

...profile/timer_example.py

Context Manager

- Context managers allow you to wrap sections of code.
- Minimal intrusion

…/modules/profile/timer.py

https://docs.python.org/2.7/reference/datamodel.html#context-managers

Section 5

Hotspots

Line Profiler

- Minimal intrusion and overhead
- Provides timing on a line by line basis.
- Allows you to identify hotspots.

Output

```
Timer unit: 1e-06 s

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 - IntGlob
   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
```

Line Profiler Columns

- Line #: The line number in the file.
- Hits: The number of times that line was executed.
- Time: The total amount of time spent executing the line in the timer's units. In the header information before the tables, you will see a line "Timer unit:" giving the conversion factor to seconds. It may be different on different systems.

Line Profiler Columns

- Per Hit: The average amount of time spent executing the line once in the timer's units.
- % Time: The percentage of time spent on that line relative to the total amount of recorded time spent in the function.
- Line Contents: The actual source code. Note that this is always read from disk when the formatted results are viewed, not when the code was executed. If you have edited the file in the meantime, the lines will not match up, and the formatter may not even be able to locate the function for display.

Line Profiler Example

- prof/sieve.py

Section 6

Memory Profiler

What does memory mean in Python?

- C/C++/Go execute native code and control memory through system calls.
- Python executes byte code via an interpreter written in C.
- The interpreter includes memory allocation optimizations. You don't control it directly.
- Automatic garbage collection.
- What is the GC?

memory_profiler

Section 7

Debuggers

pdb

- The Python line debugger.
- Import it at the top of your file.
- It is a line oriented debugger like GDB.
- Prefer pdb.set_trace() to setting dynamic breakpoints.
- This allows you to persist and go back and review your results.

  prof/sieve.py

Visual Debuggers

- If you are building any large project you will want a visual debugger.
- It will save you a lot of time.
- PyCharm
- Wing
- Let's compare pdb with Wing.

psutil

- psutil gives you safe access to low lever system calls.
- Used internally by memory_profiler.
- You can programmatically kill process.
- What happens when you attempt to kill a process?

Lab

- Select a Level 1 problem from LAB_PROBLEMS.
- Understand the problem.
- Craft a solution.
- Write a program to solve it.
- Debug it if need be.
- Ask questions.

Section 8

Optimizations

Avoid the Dot Operator

- For calculations that make heavy use of methods or modules lookups create a local variable.

  optimal/dot_operator.py

- Module lookup: 21.4790320396
- As var: 19.025177002
- 12% faster

list comprehensions

- list comprehensions are, on average, 10% - 15% faster than loops with append.

  optimal/list_comprehensions.py

concat versus join

- For strings of a significant size join is faster.
- Uniform versus non-uniform data.

optimal/concat_join_test.py

set operations

- frozensets are immutable and a little bit faster.
- All the arithmetic set operations are supported.
- Where would you use a set?
- Code it.

procedure versus class

- Classes are for re-use.
- class = noun
- method = verb
- What is the difference between a procedure/function and a method?
- Do not overuse classes.

slots

- An advanced operation! Be careful.
- It can save memory and speed up access to class variables.
- Instead of a dynamic library there is a static structure that is immutable.
- Can have side effects when interacting with third party libraries.
- Use when you have lots of instances of a class.

Section 9

What Not to Do

Expressions as Default Arguments

- What happens each time your run this?

```
def foo(bar=[]):
    bar.append(“one”)
    return bar
```

Using Class Variables Incorrectly

```
class A(object):
    x = 1
class B(A):
    pass
class C(A):
    pass
>>> print A.x, B.x, C.x
   1 1 1   <=== makes sense
```

Using Class Variables

```
>>> B.x = 2
>>> print A.x, B.x, C.x
1 2 1  <=== This makes sense

>>> A.x = 3
>>> print A.x, B.x, C.x
3 2 3  <=== What is going on?
```

MRO

- Class variables are handled as dictionaries.
- Since variable x is not found in C, it is looked up in the base class.
- C does not have its own variable x independent of A.
- C.x actually refernces A.x.
- MRO – Method Resolution Order

Catch both Exceptions

```
try:
...     l = ["a", "b"]
...     int(l[2])
... except ValueError, IndexError:
...     pass
...
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
IndexError: list index out of range
```

Multiple Exceptions - Use a tuple

- Exception ValueError, IndexError specifies IndexError as a parameter.
- Exception, e – binds the exception to the *optional* second parameter.
- e is now a variable pointing to the raised exception.

Exceptions - The Right Way

```
try:
    l = ["a", "b"]
    int(l[2])
 except (ValueError, IndexError) as e:
    pass
```

Local Enclosing Global Module

```
 x = 10
def foo():
    x += 1
    print x


>>> foo()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in foo
UnboundLocalError: local variable 'x' referenced before assignment
```

Don't Delete while Iterating

```
odd = lambda x : bool(x % 2)
numbers = [n for n in range(10)]
for i in range(len(numbers)):
    if odd(numbers[i]):
        del numbers[i]
```

```
Traceback (most recent call last):
    File "<stdin>", line 2, in <module>
IndexError: list index out of range
```