Getting performance out of python

With notes from bzr

Robert Collins
robertc@robertcollins.net

SLUG August 2007

Python is slow. . .

Recursive fibonacci sequence

Python is slow. . .

Recursive fibonacci sequence

```
def fib(n):
  if n <= 1:
    return 1
  return 1 + fib(n-1) + fib(n-2)

print fib(35)
```

But really

Iterative fibonacci sequence

But really

Iterative fibonacci sequence

```
def fib(n):
  a, b = 1, 1
  for x in range(n):
    a, b = b, 1 + a + b
  return a

print fib(35)
```

It's how you use it

http://www.diveintopython.org/

It's how you use it

http://www.diveintopython.org/

Do

- Measure

Do

- Measure
- Write understandable code

Do

- Measure
- Write understandable code
- Use the standard library

Do

- Measure
- Write understandable code
- Use the standard library
- Use primitive types freely

Don't

- Optimise unless it is needed

Don't

- Optimise unless it is needed
- Use heavily recursive algorithms

Don't

- Optimise unless it is needed
- Use heavily recursive algorithms
- Hide reality

Don't

- Optimise unless it is needed
- Use heavily recursive algorithms
- Hide reality
- Overly generalise

Tackle the problem in a different way

Tackle the problem in a different way

log -v

push

Reality Bites

Most slow programs are slow due to interactions with the real world:

Reality Bites

Most slow programs are slow due to interactions with the real world:

disk

Reality Bites

Most slow programs are slow due to interactions with the real world:

disk

network

Silly numbers

- Making a tuple: 0.04 usec.

Silly numbers

- Making a tuple: 0.04 usec.
- Making a list: 0.21 usec.

Silly numbers

- Making a tuple: 0.04 usec.
- Making a list: 0.21 usec.
- Function calls: 0.22-0.29 usec.

Silly numbers

- Making a tuple: 0.04 usec.
- Making a list: 0.21 usec.
- Function calls: 0.22-0.29 usec.
- Making a dict statically: 0.53 usec.

Silly numbers

- Making a tuple: 0.04 usec.
- Making a list: 0.21 usec.
- Function calls: 0.22-0.29 usec.
- Making a dict statically: 0.53 usec.
- Making an object with slots: 1.12 usec.

Silly numbers

- Making a tuple: 0.04 usec.
- Making a list: 0.21 usec.
- Function calls: 0.22-0.29 usec.
- Making a dict statically: 0.53 usec.
- Making an object with slots: 1.12 usec.
- Making an object without slots: 1.39 usec.

Silly numbers

- Making a tuple: 0.04 usec.
- Making a list: 0.21 usec.
- Function calls: 0.22-0.29 usec.
- Making a dict statically: 0.53 usec.
- Making an object with slots: 1.12 usec.
- Making an object without slots: 1.39 usec.
- Making a dict statically: 2.12 usec.

Better examples

- EAFP v LBYL

Better examples

- EAFP v LBYL
- Matching types

Better examples

- EAFP v LBYL
- Matching types
- Cold cache IO

Better examples

- EAFP v LBYL
- Matching types
- Cold cache IO
- Death of 1000 cuts

Last Resort!

- pyrex

Last Resort!

- pyrex
- rctypes

Last Resort!

- pyrex
- rctypes
- ctypes

Last Resort!

- pyrex
- rctypes
- ctypes
- C api

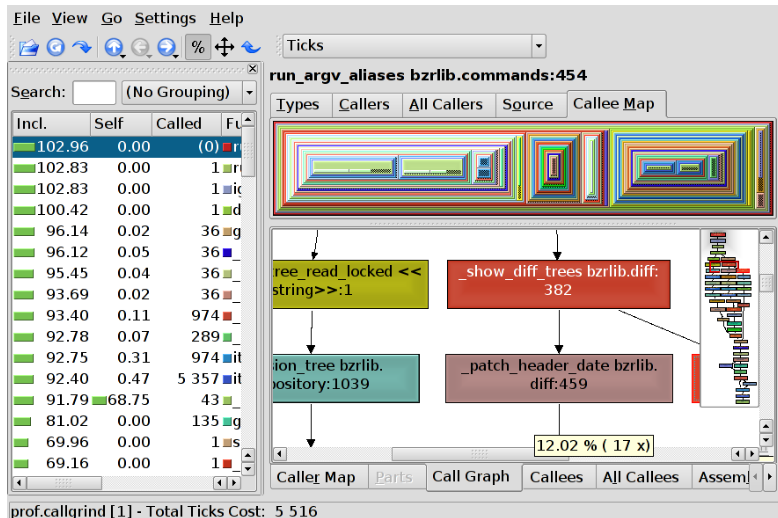
Where is my program spending its time?

Where is my program spending its time?

```
$ bzr diff -r 40..50
```

```
$ bzr --lsprof diff -r 40..50
```

```
$ bzr --lsprof-file foo.callgrind diff -r 40..50
```



File View Go Settings Help

Ticks

Search: (No Grouping)

Incl.	Self	Called	Fu
102.96	0.00	(0)	r
102.83	0.00	1	r
102.83	0.00	1	i
100.42	0.00	1	d
96.14	0.02	36	g
96.12	0.05	36	_
95.45	0.04	36	_
93.69	0.02	36	_
93.40	0.11	974	_
92.78	0.07	289	_
92.75	0.31	974	it
92.40	0.47	5 357	it
91.79	68.75	43	_
81.02	0.00	135	g
69.96	0.00	1	s
69.16	0.00	1	_

_buffer_all bzrlib.index:248

Types Callers All Callers Source Callee Map

_buffer_all bzrlib.index:248 91.79 %

<method 'split' of 'str' obje... <method 'appen... <len>

_buffer_all bzrlib.index: 248

<method 'split' of 'str' objects>

<method 'append' of 'list' objects>

Caller Map Parts Call Graph Callees All Callees Assem

prof.callgrind [1] - Total Ticks Cost: 5 516

adhoc python use

```
from bzrlib.lsprof import profile

_, stats = profile(list, t._iter_changes(t.basis_tree()))
stats.sort()
stats.pprint()
```

timeit

Whats the fastest way to write 1MB of data?

timeit

Whats the fastest way to write 1MB of data?

```
$ python -m timeit -s 'onek = "A"*1024'
  -s 'lines=[onek]*1024'
  "f = file('/dev/null', 'wb')"
  "for line in lines: f.write(line);"
  "f.close()"
```

1000 loops, best of 3: 1.03 msec per loop

1GB/second

no loop

```
$ python -m timeit -s 'onek = "A"*1024'
  -s 'lines=[onek]*1024'
  'onem = "".join(lines)'
  "f = file('/dev/null', 'wb')"
  "f.write(onem)"
  "f.close()"
```

1000 loops, best of 3: 628 usec per loop
1.6GB/second

fit the api to the data

```
$ python -m timeit -s 'onek = "A"*1024'
  -s 'lines=[onek]*1024'
  "f = file('/dev/null', 'wb')"
  "f.writelines(lines)"
  "f.close()"
```

1000 loops, best of 3: 510 usec per loop
2GB/second

but if the data fits better

```
$ python -m timeit -s 'onek = "A"*1024'
  -s 'lines=[onek]*1024'
  -s 'onem = "".join(lines)'
  "f = file('/dev/null', 'wb')"
  "f.write(onem)"
  "f.close()"
```

10000 loops, best of 3: 19.4 usec per loop
50GB/second

The End

Questions?

Questions?