Basking and remaining 10% of Python Optimiza...

http://www.sweetapp.com/optimize

June 2004

Brian Quinlan

brian@sweetapp.com

Who are you?

- A few core Python contributions:
  - SimpleXMLRPCServer & DocXMLRPCServer
  - A bit of Unicode stuff
  - Bug fixes
- Other Python projects:
  - Pyana (Xalan-C bindings)
  - PyMobile (control GSM phones)
  - SilverCity (used to color these slides!)
- Won math competition in grade 11

Never optimize

Reduces clarity, maintainability and flexibility, increases effort

Response time

.1 second	limit for perception of reacting instantaneously
1 second	limit for the uninterrupted flow of thought
10 seconds	limit for keeping the user's attention

sanitize.py (1/2)

```
def adjust_char(c):
    if c.isdigit(): return '#'
    elif c == '$': return ''
    elif c == '!': return '.'
    else: return c

def adjust_str(s):
    r = ''
    for c in s:
        r += adjust_char(c)
    return r

def adjust_file(fin, fout):
    fout.write(adjust_str(fin.read()))
```

sanitize.py (2/2)

```
def main():
    import sys
    adjust_file(
        file(sys.argv[1], 'r'),
        file(sys.argv[2], 'w+'))

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

Add profiling switch

```
def profile_main():
    import hotshot
    prof = hotshot.Profile('adjust.prof')
    prof.runcall(main)
    prof.close()

if __name__ == '__main__':
    import sys
    if '-profile' in sys.argv:
        sys.argv.remove('-profile')
        profile_main()
    else:
        main()
```

Using hotshot.stats

```
C:\> python sanitize.py –profile
            policy100.txt policy-eur.txt
C:\> python
Python 2.3.3 (#51, Dec 18 2003, ...
>>> from hotshot import stats
>>> sstats = \
... stats.load('sanitize.prof')
>>> sstats.sort_stats('time', 'calls')
<pstats.Stats instance at 0x00909F58>
>>> sstats.print_stats()
```

hotshot.stats results

```
160912 function calls in 250.777 CPU seconds

   Ordered by: internal time, call count

ncalls  tottime  percall  cumtime  percall function
     1  237.674  237.674  250.713  250.713 adj_str
160909   13.039    0.000   13.039    0.000 adj_char
     1    0.033    0.033  250.746  250.746 adj_file
     1    0.032    0.032  250.777  250.777 main
     0    0.000            0.000          profile
```

Measure with timeit

```
>>> import timeit
>>> f = file('policy100.txt', 'r')
>>> text = f.read()
>>> t = timeit.Timer(
... 'adj_str(%r)' % text,
... 'from sanitize import adj_str')
>>> t.timeit(5)/5
60.557139245351024
```

Deviance is normal

```
>>> from pprint import pprint
>>> pprint(t.repeat(10, 1))
[52.466439729071681,
  62.095288647020766,
  58.866749544984032,
  58.704351530711278,
  58.403590959186204,
  58.50212024153916,
  58.860085010804369,
  58.746528120193943,
  66.207049778672854,
  58.952417416179969]
```

A small change

```
from StringIO import StringIO
def adj_str(s):
    r = StringIO()
    for c in s:
        r.write(adj_char(c))
    return r.getvalue()
```

Measure again

```
>>> import timeit
>>> f = file('policy100.txt', 'r')
>>> text = f.read()
>>> t = timeit.Timer(
... 'adj_str(%r)' % text,
... 'from sanitize import adj_str')
>>> t.timeit(5)/5
2.1199982120632739
```

Stat results (again)

```
321825 function calls in 14.018 CPU seconds

  Ordered by: internal time, call count

  ncalls  tottime  percall  cumtime  percall function
  160909    6.012    0.000    6.012    0.000 write
       1    3.941    3.941   13.163   13.163 adj_str
  160909    2.680    0.000    2.680    0.000 adj_char
       1    0.818    0.818   14.018   14.018 main
       1    0.528    0.528    0.528    0.528 gettvalue
       1    0.036    0.036   13.199   13.199 adj_file
       1    0.001    0.001    0.002    0.002 ?
       1    0.000    0.000    0.000    0.000 StringIO
       1    0.000    0.000    0.000    0.000 __init__
       0    0.000             0.000          profiler
```

Hotspot limitations

- C functions are invisible
- Clock accuracy
- Profiler overhead
- Line numbers profiling isn't useful

http://www.python.org/doc/lib/profile_limits.html

timeit limitations

- Using it requires ugly syntax
- Dynamically builds a function containing your setup and test code (ugly error messages):

```
def inner(_it, _timer):
    # setup code
    _t0 = _timer()
    for _i in _it:
        # test code
    _t1 = _timer()
    return _t1 - _t0
```

Optimization steps

1. Identify that a problem exists
2. Isolate it
3. Fix it
    1. Algorithm
    2. Bad design
    3. Micro-optimizations
    4. Use something other than Python

sum(1..n)

```
def sum_to1(n):
    '''sum_to(5) => 15

    Returns 1 + 2 + 3 ... + n'''
    return sum(range(1, n+1))

def sum_to2(n):
    '''sum_to(5) => 15

    Returns 1 + 2 + 3 ... + n'''
    return n*(n + 1)/2
```

sum(1..n) - timings

```
import timeit
for function in ['sum_to1', 'sum_to2']
    t = timeit.Timer(
  'sum_to(1000000)',
  'from __main__ import %s as sum_to'
        % function)
    print function, '=>', t.timeit(10)
```

```
sum_to1 => 10.1560606166
sum_to2 => 0.0708553740769
```

Mutability reminder

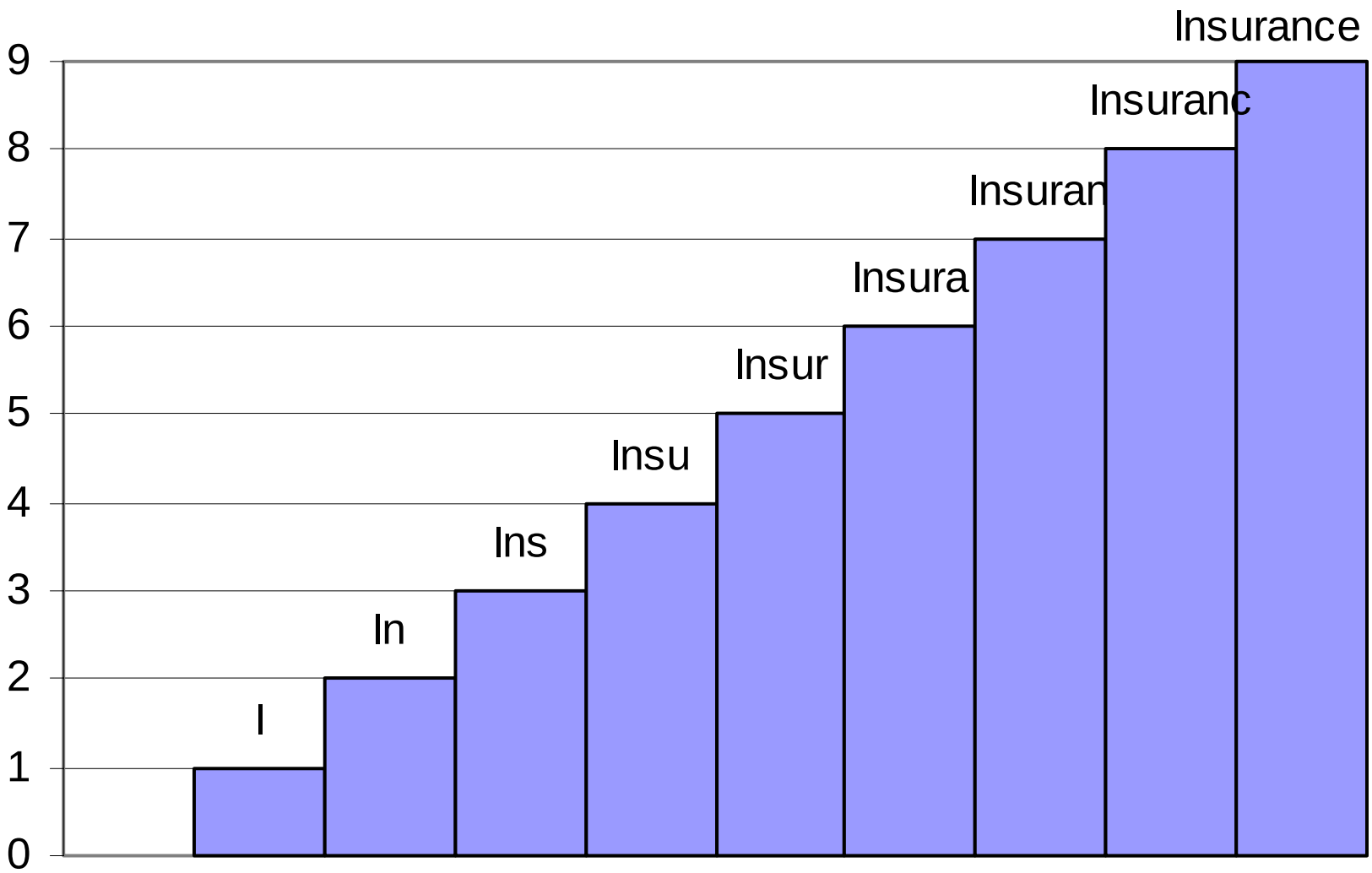
```
>>> a = ['Hello']
>>> b = a
>>> b
['Hello']
>>> a += ['World']
>>> a
['Hello', 'World']
>>> b
['Hello', 'World']
```

```
>>> a = 'Hello '
>>> b = a
>>> b
'Hello '
>>> a += 'World'
>>> a
'Hello World'
>>> b
'Hello '
```

So…?

```
def adjust_str(s):
    r = ''
    for c in s: # 160909 times
        r += adjust_char(c)
    return r
```

So…!?



Ah so!

```
sum(1…n) = n*(n+1)/2
sum(1..9) = 9*(9+1)/2 = 9*10/2 = 90/2 = 45

>>> n = 160909
>>> n*(n+1)/2
12945933595L
```

160909 strings containing over 12 billion characters were created!

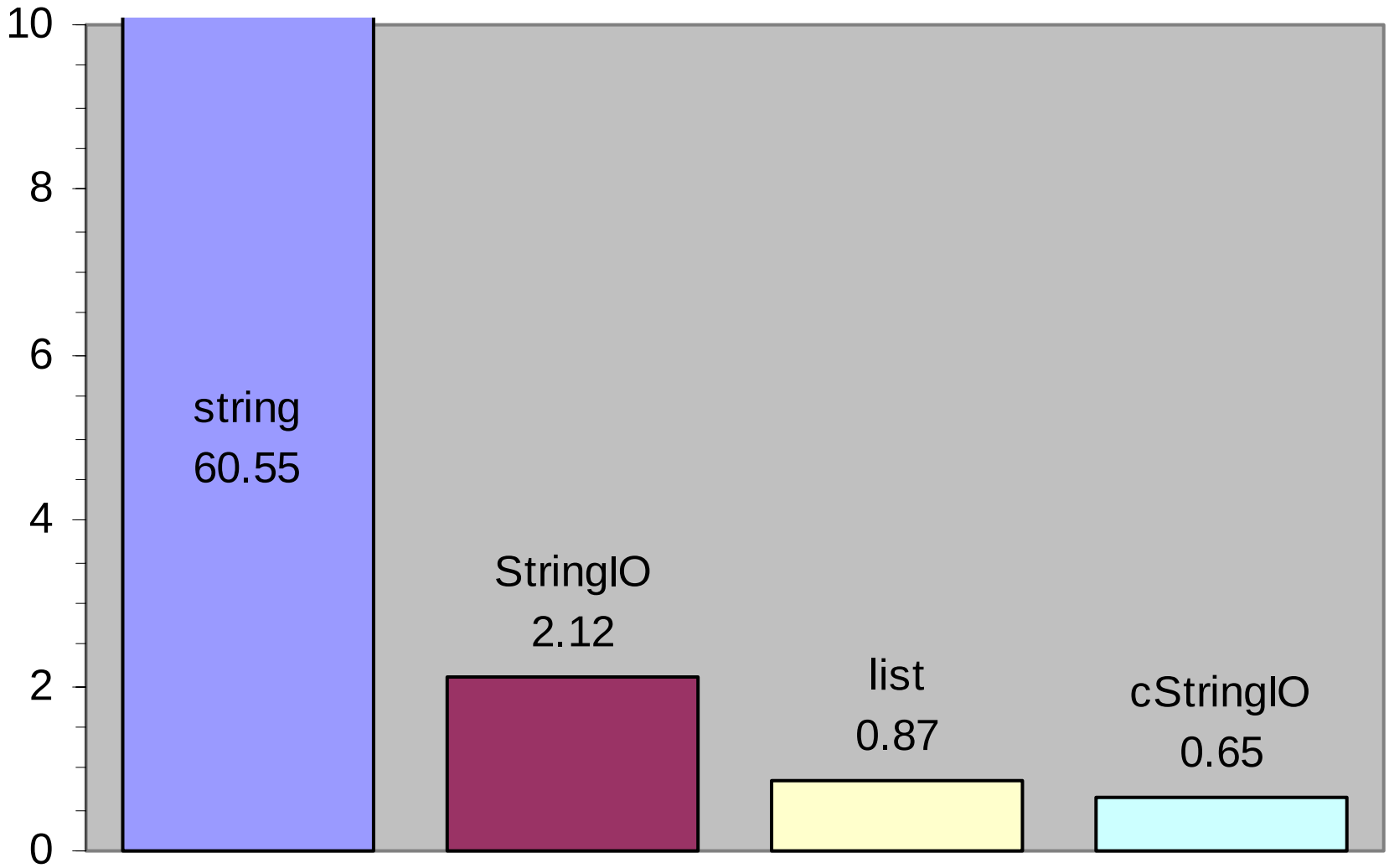
Use a list

```
def adj_str(s):
    r = []
    for c in s:
        r.append(adj_char(c))
    return ''.join(r)

>>> t.timeit(5)/5
0.8657419867608870 9
```

Use cStringIO

```
from cStringIO import StringIO
def adj_str(s):
    r = StringIO()
    for c in s:
        r.write(adj_char(c))
    return r.getvalue()

>>> t.timeit(5)/5
0.65127612587633343
```

String building graph



Code reuse is good

- Standard library:
  - StringIO/cStringIO
  - array
  - struct
- 3rd party extensions:
  - Numeric
  - PIL
  - SciPy

Building identity list (code)

```
def identity(lst):
    l = []
    for i in lst:
        l.append(i)
    return l

def identity2(lst):
    return [i for i in lst]
```

Building identity list (timings)

```
for function in ['identity', 'identity2']:
    t = timeit.Timer(
        'identity(lst)',
        'from __main__ import %s as identity;'
        'lst = range(5000)' % function)
    print function, '=>', t.timeit(1000)
```

```
identity => 4.78896256368
identity2 => 3.39620203126
```

Building useful list (for)

```
def filter_files(files):
    filtered_files = []
    for f in files:
        if f.endswith('.dll') or \
           f.endswith('.exe'):
        filtered_files.append(f)
    return filtered_files
```

Building useful list (list comp)

```
def filter_files2(files):
    return [f for f in files
                if f.endswith('.dll') or
                   f.endswith('.exe')]
```

Building useful list (timings)

```
import timeit
for func in ['filter_files', 'filter_files2']:
    t = timeit.Timer(
      'filter_files(files)',
      'from __main__ import %s as filter_files,'
      '                      get_files;'
      'files = get_files("c:\Windows")' % func)
    print func, '=>', t.timeit(100)
```

```
filter_files => 2.52770533685
filter_files2 => 2.46247190635
```

Scoping vs. Speed

```
def somefunction(...):
    ...
    print file
    ...
```

1. Check for “file” in local variables
2. Check for “file” in global variables
3. Check for “file” in built-ins

Global pi

```
from math import pi
def quadrant(a):
    if 0 <= a < pi/2: return 1
    elif pi/2 <= a < pi: return 2
    elif pi <= a < 3*pi/2: return 3
    elif 3*pi/2 <= a < 2*pi: return 4
    else:
        while a >= 2*pi:
            a -= 2*pi
        return quadrant(a)
```

Local pi

```
from math import pi
def quadrant2(a, pi=pi):
    if 0 <= a < pi/2: return 1
    elif pi/2 <= a < pi: return 2
    elif pi <= a < 3*pi/2: return 3
    elif 3*pi/2 <= a < 2*pi: return 4
    else:
        while a >= 2*pi:
            a -= 2*pi
        return quadrant2(a)
```

pi bakeoff

```
import timeit
for function in ['quadrant', 'quadrant2']:
    t = timeit.Timer(
        'quadrant(0.2); quadrant(1.6);'
        'quadrant(3.2); quadrant(6.0);'
        'quadrant(92385.6)',
        'from __main__ import %s as quadrant'
            % function)
    print function, '=>', t.timeit(100)
```

```
quadrant => 2.020
quadrant2 => 1.894
```

Dots vs. speed

```
print foo.bar.baz
print getattr(getattr(foo,'bar'),'baz')
```

Yet more pi

```
import math
def quadrant3(a):
    if 0 <= a < math.pi/2: return 1
    elif math.pi/2 <= a < math.pi: return 2
    elif math.pi <= a < 3*math.pi/2: return 3
    elif 3*math.pi/2 <= a < 2*math.pi: return 4
    else:
        while a >= 2*math.pi:
            a -= 2*math.pi
        return quadrant3(a)

quadrant3 => 2.93081746423
quadrant => 2.15314432421 # from math import pi
```

Counting words

```
>>> words = file("eula.txt").read().split()
>>> pprint(top_words(words, 15))
[('the', 185),
 ('to', 110),
 ('of', 107),
 ('or', 107),
 ('and', 70),
 ('you', 63),
 ('product', 60),
 ('any', 51),
 ('microsoft', 49),
 ('for', 44)]
```

Counting words – first try

```
def top_words(words, n=20):
    d = {} # {'and': 70, 'to': 15,...}
    for word in words:
        try:
            d[word.lower()] += 1
        except KeyError:
            d[word.lower()] = 1
    i = d.items() #[('and', 70), ('to', 15),...]
    # sort by the number, from highest to lowest
    i.sort(lambda x,y: cmp(y[1], x[1]))
    return i[:n] # only return the higest n

t.timeit(100) => 2.64283225941
```

Look before you leap

```
def top_words(words, n=20):
    d = {}
    for word in words:
        if d.has_key(word.lower()):
            d[word.lower()] += 1
        else:
            d[word.lower()] = 1
    i = d.items()
    i.sort(lambda x,y: cmp(y[1], x[1]))
    return i[:n]

t.timeit(100) => 1.9806651404
```

(Modified) decorated sort

```
def top_words(words, n=20):
    ...
    i = d.items() #[('and', 70), ('to', 15),...]
    i = [(num, word) for (word, num)
            in d.iteritems()] # [(70, 'and'),...]
    i.sort() # [(1, 'profits'),...,('the', 185)]
    i = i[-n:] # [(44, 'for'),..., ('the', 185)]
    i.reverse() # [(185,'the'), ..., ('for',44)]
    i = [(word, num) for (num, word) in i]
    return i

t.timeit(100) => 1.44930961896
```

Techniques summary

- Improve algorithm
- Avoid string concatenation
- Minimize scope
- Avoid dots
- Look before you leap (avoid exceptions)
- Decorate, sort, undecorate
- Cache results
- Avoid small functions
- Disable garbage collection
- Reduce thread polling interval
- Avoid object creation
- Avoid **exec** in functions (slows local variable lookup)

Still too slow? Recode in C!

When Python isn't fast enough, people traditionally turn to C

THIS IS TOTALLY CRAZY!

Writing C extensions is time consuming, error prone and requires expert knowledge

Static language compilers

```
// C, C++ or Java code
static float do_something(float x, float y)
{
    // do_something(18.4, 15.2) => 64.0
    return x + y * 3;
}

// Disassembly
D9450C          fld     dword ptr [ebp+0Ch]
D80D2C414200    fmul    dword ptr [0042412c]
D84508          fadd    dword ptr [ebp+8]
```

Same with Python, right?

```
>>> def do_something(x, y):
... return x + y * 3
...
>>> print do_something(18.4, 15.2)
64.0
```

Nope…

```
>>> print do_something('I say:',' Ni!')
I say: Ni! Ni! Ni!
>>> print do_something([1, 2, 2], [3])
[1, 2, 2, 3, 3, 3]
>>> from datetime import datetime,\
...                      timedelta
>>> print do_something(
...  datetime.now(), timedelta(days=1))
2004-05-26 11:25:15.501000
```

Consequences (code)

```
def count_to(start, incr, end):
    while start <= end:
        start = start + incr
    return start
```

```
static double count_to (
   double start, double incr, double end)
{
   while (start <= end)
      start = start + incr;
   return start;
}
```

Consequences (timings)

```
import timeit
t = timeit.Timer(
    'count_to(0.0, pi, 1000000000.0)',
    'from __main__ import count_to;'
    'from math import pi')
print t.timeit(1) # => 188.679708175
```

```
C time: 1.972000
```

Enter psyco

- Python specializing compiler
- Waits until the function is called at runtime and then generates machine code based on the actual types of the arguments
- If the function is called again with different argument types then new machine code is generated for the function

psyco pseudocode

```
def count_to(d={}, *args):
    types = tuple([type(arg) for arg in args])
    # types = (float, float, float)
    # types = (float, float, float)
    # types = (int, float, int)
    if not d.has_key(types):
        d[types] = generate_machine_code(types)
    d[types].call_machine_code(*args)

>>> count_to(0.0, pi, 1000000000.0)
>>> count_to(0.0, 5.3, 115.23)
>>> count_to(0, 5.5, 50000)
```

psyco (timings - I)

```
import psyco
psyco.full()
import timeit
t = timeit.Timer(
    'count_to(0.0, pi, 1000000000.0)',
    'from __main__ import count_to;'
    'from math import pi')
print t.timeit(1) # => 18.056491258
```

```
Python time: 188.679708175
C time: 1.972000
```

psyco (timings - II)

```
import timeit
    t = timeit.Timer(
    '''count_to(date(2004, 5, 23),
                timedelta(days=7),
                date(9999, 12, 25))''',
    'from __main__ import count_to;'
    'from datetime import *')
print t.timeit(100) # 39.71 vs 19.51
```

Psyco summary

- Strengths:
  - Possible massive performance improvements without any work
- Weaknesses:
  - X86 only
  - Can sometimes require a lot of memory
  - Some semantic changes e.g. locals() not available, KeyboardInterrupt not checked for, built-ins are assumed not to change
  - Inscrutable

Enter Pyrex

- A language for writing Python extensions
- Pyrex code is converted to C code and then compiled into a Python extension
- Pyrex can also directly use C libraries

count.pyx

```
def count_to(start, incr, end):
    while start <= end:
        start = start + incr
    return start
```

setup.py

```
from distutils.core import setup
from distutils.extension import \
                                Extension
from Pyrex.Distutils import build_ext

setup(
    name = 'Count',
    ext_modules=[
   Extension("count", ["count.pyx"]),],
    cmdclass = {'build_ext': build_ext}
)
```

count (timing I)

```
t = timeit.Timer(
    'count_to(0.0, pi, 1000000000.0)',
    'from count import count_to;'
    'from math import pi')
print t.timeit(1) # => 79.07236646

t = timeit.Timer(
     '''count_to(date(2004, 5, 23),
                timedelta(days=7),
                date(9999, 12, 25))''',
                'from count import count_to;'
                'from datetime import *')
print t.timeit(100) # => 39.511466401456047
```

count.pyx

```
def count_to( double start,
              double incr,
              double end):
    while start <= end:
        start = start + incr
    return start

# => 1.98855608744
```

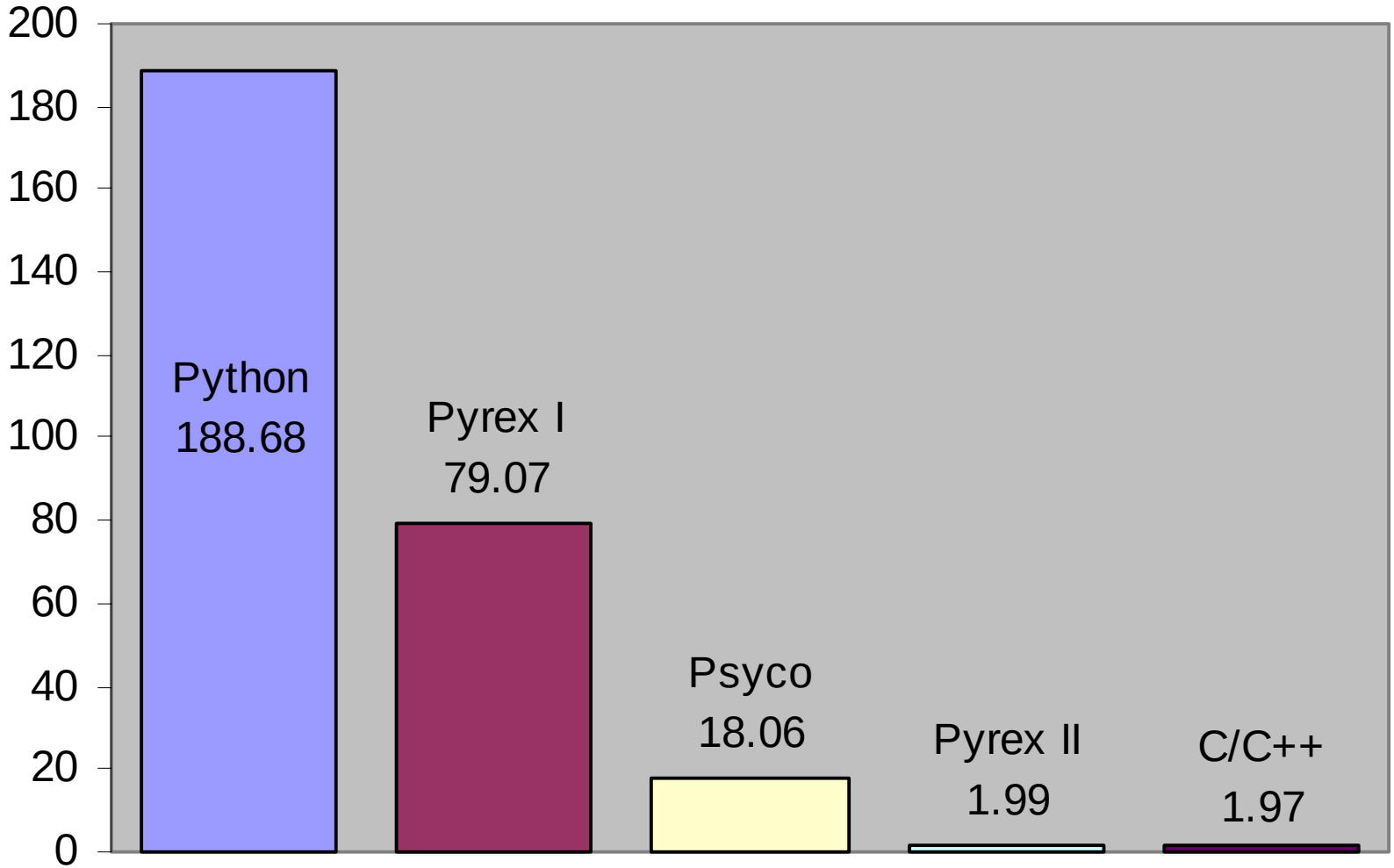
Try datetime

```
>>> from count import count_to
>>> from datetime import *
>>> count_to( date(2004, 5, 23),
...           timedelta(days=7),
...           date(9999,12,25))
Traceback (most recent call last):
File "<stdin>", line 1, in ?
TypeError: a float is required
```

Pyrex summary

- Strengths:
  - Possible massive performance improvements
  - Can work with both C and Python code
  - Cross platform
- Weaknesses:
  - Build system is a hassle i.e. need a C compiler, to write a setup script
  - Some Python constructs not supported
  - Modest performance improvements without reducing flexibility

Python vs Psyco vs Pyrex



Vancouver Python Workshop

- July 31st-August 2nd
- Right after OSCON (5 hour drive with other Pythonistas)
- Vancouver (you want to go there anyway)
- You've heard me speak before

Scionics is recruiting

- Python developer #1
  - Knowledge of image analysis or ability to read books and learn image analysis (good opportunity to get published)
- Python developer #2
  - Strong Python skills
  - Knowledge of CGI programming
  - Knowledge of database programming
  - Ability to tolerate me