9. Rendimiento en Python

May 5, 2014

Dicen por ahí que Python es lento. Y lo es, a veces (en unas situaciones particulares): básicamente en bucles / ciclos

Un ejemplo:

```
In [4]: total = 0
        for i in xrange(100):
            total += i

        print total


          File "<ipython-input-4-f736e62d08ed>", line 5
        i = "hola
                ^
    SyntaxError: EOL while scanning string literal
```

```
In [9]: total = 0
        total2 = ""
        for i in xrange(10):
            total += i
            print i

            i = "hola"
            total2 += i
            print i

        print total
        print total2
```

```
0
hola
1
hola
2
hola
3
hola
4
hola
5
hola
6
```

```
hola
7
hola
8
hola
9
hola
45
holaholaholaholaholaholaholaholaholahola
```

La flexibilidad misma lleva a la lentitud aquí, por ¡la necesidad de checar el tipo de los objetos todo el tiempo!

0.1 Diagnóstico por perfilamiento

Queremos acelerar un código de cómputo científico con muchos bucles. ¿Cuál es el procedimiento? Por ejemplo, "eliminar" los bucles. Antes que esto: veamos cuánto tarda en cada parte del programa. ¡Pero no a mano, sino con un *perfilador*!

La manera más fácil en IPython es con `%time` o `%timeit` (son diferentes): `%time` corre una vez un pedazo de código, mientras `%timeit` hace estadística. Y hay versiones con comandos mágicos de celda: `%%time` y `%%timeit` (corren todo el código de la celda completa):

```
In [10]: def sumar(n):
             total = 0
             for i in xrange(n):
                 total += i

             return total

In [40]: %time sumar(10000000)

CPU times: user 624 ms, sys: 8.84 ms, total: 633 ms
Wall time: 626 ms

Out[40]: 49999995000000

In [41]: %timeit sumar(10**8)

1 loops, best of 3: 6.3 s per loop

In [20]: %%timeit
         total = 0
         for i in xrange(1000):
             total += i

10000 loops, best of 3: 57.4 µs per loop
```

Sin usar las bondades de IPython:

```
In [33]: import profile

In [34]: profile.run?

In [38]: profile.run("sumar(100000)")
```

```
4 function calls in 0.012 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.000    0.000 :0(setprofile)
        1    0.012    0.012    0.012    0.012 <ipython-input-10-6939f211451f>:1(sumar)
        1    0.000    0.000    0.012    0.012 <string>:1(<module>)
        0    0.000             0.000          profile:0(profiler)
        1    0.000    0.000    0.012    0.012 profile:0(sumar(100000))
```

0.2 Vectorizar

Reemplazar bucles explícitos de Python con operaciones vectoriales sobre `array`s de `numpy`: `numpy` está diseñado tal que sus operaciones son rápidas –todos los bucles (implícitos) están escritos en C

```
In [42]: import numpy as np
```

```
In [49]: %%timeit
         #a = np.arange(10**6)
         s = sum(xrange(10**6))
         print s
```

```
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
```

499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000

499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000

499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000

499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000

499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000

499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000

499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000
499999500000

```
499999500000
100 loops, best of 3: 9.98 ms per loop

In [48]: %timeit sumar(10**6)

10 loops, best of 3: 62.1 ms per loop

In [53]: a = np.linspace(1, 10**6, 10**6)
         b = np.arange(0.1, 1, 10**6)

In [55]: %timeit c = a + b

1000 loops, best of 3: 1.24 ms per loop

In [59]: %%timeit
         c = np.zeros(10**6)

         for i in range(10**6):
             c[i] = a[i] + b[i]


        ---------------------------------------------------------------------------
    IndexError                                Traceback (most recent call last)

        <ipython-input-59-6fa598d3b2ad> in <module>()
    ----> 1 get_ipython().run_cell_magic(u'timeit', u'', u'\nc = np.zeros(10**6)\n\nfor i in range(10**6)


        /Users/dsanders/development/ipython/IPython/core/interactiveshell.pyc in run_cell_magic(self, ma
       2136             magic_arg_s = self.var_expand(line, stack_depth)
       2137             with self.builtin_trap:
    -> 2138                 result = fn(magic_arg_s, cell)
       2139             return result
       2140


        /Users/dsanders/development/ipython/IPython/core/magics/execution.pyc in timeit(self, line, cell


        /Users/dsanders/development/ipython/IPython/core/magic.pyc in <lambda>(f, *a, **k)
        189     # but it's overkill for just that one bit of state.
        190     def magic_deco(arg):
    --> 191         call = lambda f, *a, **k: f(*a, **k)
        192
        193         if callable(arg):


        /Users/dsanders/development/ipython/IPython/core/magics/execution.pyc in timeit(self, line, cell
        984             number = 1
        985             for i in range(1, 10):
    --> 986                 if timer.timeit(number) >= 0.2:
        987                     break
        988                 number *= 10


        /usr/local/Cellar/python/2.7.5/Frameworks/Python.framework/Versions/2.7/lib/python2.7/timeit.pyc
```

```
    193         gc.disable()
    194         try:
--> 195             timing = self.inner(it, self.timer)
    196         finally:
    197             if gcold:


<magic-timeit> in inner(_it, _timer)


IndexError: index 1 is out of bounds for axis 0 with size 1
```

```
In [57]: len(a)
```

```
Out[57]: 1000000
```

```
In [63]: a = 3
         s = "print(a)"
```

```
In [64]: s
```

```
Out[64]: 'print(a)'
```

```
In [66]: exec(s)
```

```
3
```

```
In [67]: expr = "3*a"
         eval(expr)
```

```
Out[67]: 9
```

0.3 List comprehensions

Supongamos que queramos aplicar una función a cada elemento de una lista, por ejemplo duplicar cada elemento:

```
In [68]: a = np.arange(10)
         b = 2*a
```

```
In [69]: a
```

```
Out[69]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In [70]: b
```

```
Out[70]: array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
```

```
In [71]: def f(x):
             return x*x + 1
```

```
In [72]: map(f, a)
```

```
Out[72]: [1, 2, 5, 10, 17, 26, 37, 50, 65, 82]
```

```
In [73]: f(a)
```

```
Out[73]: array([ 1,  2,  5, 10, 17, 26, 37, 50, 65, 82])
```

List comprehension:
Matemáticamente, escribiríamos:

$$\{x^2 + 1 : x \in A\}$$

```
In [74]: [x*x+1 for x in range(10)]
```

```
Out[74]: [1, 2, 5, 10, 17, 26, 37, 50, 65, 82]
```

```
In [92]: %time l = [x*x+1 for x in xrange(1000000) if x%2==1]
```

```
CPU times: user 223 ms, sys: 11.7 ms, total: 235 ms
Wall time: 229 ms
```

```
In [95]: %%timeit
         l = []
         for x in xrange(1000000):
             if x%2==1:
                 l.append(x*x+1)
```

```
10 loops, best of 3: 165 ms per loop
```

0.4

```
In [78]: np.sin(a)
```

```
Out[78]: array([ 0.        ,  0.84147098,  0.90929743,  0.14112001, -0.7568025 ,
                -0.95892427, -0.2794155 ,  0.6569866 ,  0.98935825,  0.41211849])
```

```
In []:
```

0.5 Siguiente paso: Cython y numba

Cython y numba compilan pedazos de Python a C

```
In []:
```