Programarea calculatoarelor si limbaje de programare II

Benchmarking in Python

Universitatea Politehnica din Bucureşti

Sumar

- ***Masurarea duratei iteratiilor***
- Cu *timeit*
- Alte metode de benchmarking
- Detalii de implementare a functiilor

timer0

- In principiu, colectiile iterative (*list*) sunt mai rapide decat instructiunea *for,* iar *map()* este mai rapid decat ambele, in special daca sunt prezente apeluri de functii predefinite

```
# Fisierul timer0.py:
import time
def timer(func, *args): # Versiune simplista
    start = time.perf_counter() #time.clock()
    for i in range(10000):
        func(*args)
    return time.perf_counter() - start # Durata, in secunde
    ###return time.clock() - start
```

```
>>> from timer0 import timer
>>> timer(pow, 2, 1000) # Durata apelului lui pow(2, 1000) de 10000 de ori
0.0012753999999997262
>>> timer(str.upper, 'spam') # Durata lui 'spam'.upper() de 10000 de ori
8.869999999916445e-05
```

timer0...

Erori in proiectarea lui timer0:

- Nu suporta argumente cu cuvinte cheie
- Numarul de repetitii este fix
- Generarea cu *range()* afecteaza durata executiei
- *time.clock()* nu mai este implementat, *time.perf_counter()* se va folosi in loc.
- Nu demonstreaza ca apelul lui func a mers
- Produce doar durata totala, care variaza cu incarcarea calculatorului (multiuser, multitasking)

Scriptul timer

```
# Fisierul timer.py:
"""
Durata apelurilor de functii:
Durata totala, cea mai rapida, si cea mai rapida
    din mai multe incercari
"""
import time, sys
timer = time.perf_counter ###time.clock if
    sys.platform[:3] == 'win' else time.time


def total(reps, func, *pargs, **kargs):
    """
Durata apelului de reps ori a func().
Returneaza tuplul (timp total, ultimul rezultat
    al apelului lui func)
    """
    repslist = list(range(reps)) # Nemasurat,
    merge in Python v2.x si  v3.x
    start = timer()
    for i in repslist:
        ret = func(*pargs, **kargs)
    elapsed = timer() - start
    return (elapsed, ret)
```

timer...

```
def bestof(reps, func, *pargs, **kargs):
    """
Apelul cel mai rapid de func()
Returneaza tuplul (timpul cel mai scurt, ultimul
rezultat)
    """
    best = 2 ** 32 # 136 ani...
    for i in range(reps): # range nu este masurat
        start = timer()
        ret = func(*pargs, **kargs)
        elapsed = timer() - start # Sau apel de
    total() cu reps=1
        if elapsed < best: best = elapsed
    return (best, ret)
```

```
def bestoftotal(reps1, reps2, func, *pargs,
    **kargs):
    """
Cea mai rapida executie:
(reps1 apeluri de (reps2 apeluri de func))
    """
    return bestof(reps1, total, reps2, func,
    *pargs, **kargs)
```

timer...

Avantajele lui timer:

- Alege ~~*time.clock()* pentru Windows~~ (precizie in microsecunde) ~~sau~~ *time.time()* ~~pentru Linux~~; acum *time.perf_counter()*.
- Apelul lui *range()* nu este masurat, iar cu list(range()) sunt suportate atat Python v3.x cat si v2.x
- Argumentul reps, din prima pozitie, determina numarul de repetitii
- Sunt suportate oricate argumente pozitionale (cu *pargs) si cu cuvinte cheie (cu **kargs)
- Functia total() returneaza un tuplu (chiar si fara paranteze) cuprinzand durata totala si ultimul rezultat al apelului
- Functia bestof() returneaza cea mai rapida executie si ultimul rezultat, tot intr-un tuplu, putandu-se inlatura impactul celorlalte procese din sistem
- Functia bestoftotal() apeleaza bestof() care repeta de reps1 ori apelul lui total() cu argumentele reps2, func si celelalte argumente ramase; reps1 << reps2

timer...

```
>>> # Teste cu modulul timer:
>>> from timer import total,bestof,bestoftotal
>>> total(1000, pow, 2, 1000)[0] # Rezultat tuplu, indexare pozitia 0 e durata
0.0012618000000657048
>>> total(1000, str.upper, 'spam') # Rezultat tuplul (durata, ultimul rezultat al apelului de func)
(8.310000021083397e-05, 'SPAM')
>>> bestof(1000, str.upper, 'spam') # Ar trebui sa fie 1/1000 din totalul de mai sus...
(9.000000318337698e-07, 'SPAM')

>>> bestof(1000, pow, 2, 1000000)[0]
0.003583899999739515
>>> bestof(50, total, 1000, str.upper, 'spam')
(9.210000007442432e-05,
    (7.580000010420918e-05, 'SPAM'))
>>> bestoftotal(50, 1000, str.upper, 'spam')
(9.18999999157677e-05,
    (7.550000009359792e-05, 'SPAM'))
>>> min(total(1000, str.upper, 'spam') for i in range(50)) # Cu generator si functia predefinita min()
(7.520000008298666e-05, 'SPAM')
>>> ((((2 ** 32) / 60) / 60) / 24) / 365 # 2**32 secunde = 136 ani
136.19251953323186
>>> ((((2 ** 32) // 60) // 60) // 24) // 365 # Cu floor: //
136
```

Modulul *time* din v3.x

- *time.perf_counter()* calculeaza durate cu cea mai mare precizie disponibila (inclusiv timp in sleep, dormant)
- *time.process_time()* returneaza durata in fractiuni de secunda, per proces, fara sleep

```
if sys.version_info[0] >= 3 and
    sys.version_info[1] >= 3:
    timer = time.perf_counter # Durata per proces
else:
    timer = time.time ###time.clock if sys.platform[:3] == 'win' else time.time
```

- **Posibile utilizari in modulul timer**

```
try: # Mai simplu cu try, detecteaza lipsa functiei intr-o versiune mai mica decat v3.3
    timer = time.perf_counter
except AttributeError:
    timer = time.time ###time.clock if sys.platform[:3] == 'win' else time.time
```

Durata diverselor iteratii, in timeseqs.py

```
"Testeaza viteza diferitelor iteratii"
import sys, timer # Importari
reps = 10000
repslist = list(range(reps)) # Repetitii, intr-un list
def forLoop():
    res = []
    for x in repslist:
        res.append(abs(x))
    return res
def listComp():
    return [abs(x) for x in repslist]
def mapCall():
    return list(map(abs, repslist)) # list in v3.x
    # return map(abs, repslist) in v2.x
def genExpr():
    return list(abs(x) for x in repslist) # list()
    colecteaza toate valorile, expresie generator
def genFunc():
    def gen():
        for x in repslist:
            yield abs(x)
    return list(gen()) # list() colecteaza valorile
    generate de functia generator gen()
print(sys.version)
for test in (forLoop, listComp, mapCall, genExpr,
    genFunc):
    (bstof, (tot, result)) = timer.bestoftotal(5,
    1000, test) #atribuire de tuple
    print ('%-9s: %.5f => [%s...%s]' %
(test.__name__, bstof, result[0], result[-1]))
```

Durata...

C:\code>**py -3 timeseqs.py** 3.7.4 (tags/v3.7.4:e09359112e, Jul 8 2019, 20:34:20) [MSC v.1916 64 bit (AMD64)] forLoop : 0.86757 => [0...9999]	listComp : 0.51202 => [0...9999] mapCall : 0.24802 => [0...9999] genExpr : 0.74423 => [0...9999] genFunc : 0.75659 => [0...9999]
C:\code>**py -2 timeseqs.py** 2.7.17 (v2.7.17:c2f86d86e6, Oct 19 2019, 21:01:17) [MSC v.1500 64 bit (AMD64)] forLoop : 0.89017 => [0...9999]	listComp : 0.51768 => [0...9999] mapCall : 0.39244 => [0...9999] genExpr : 0.67139 => [0...9999] genFunc : 0.68432 => [0...9999]
C:\code>**pypy3 timeseqs.py** 3.6.9 (1608da62bfc7, Dec 23 2019, 12:38:24) [PyPy 7.3.0 with MSC v.1911 32 bit] forLoop : 0.07310 => [0...9999]	listComp : 0.07288 => [0...9999] mapCall : 0.06801 => [0...9999] genExpr : 0.14842 => [0...9999] genFunc : 0.14828 => [0...9999]

Durata...

- Obiectele generator sunt mai lente decat colectiile/listele iterative – fiindca implica salvari si restaurari de stare
- Python v2.7 pare mai rapid decat v3.x (3.7)
- PyPy este un ordin de magnitudine mai rapid (10X)

timeseqs2.py, cu functii in-line

```
# Fisierul timeseqs2.py (doar modificarile)
def forLoop():
    res = []
    for x in repslist:
        res.append(x + 10)
    return res
def listComp():
    return [x + 10 for x in repslist]
def mapCall():
    return list(map((lambda x: x + 10), repslist))
    # list() doar in v3.x
def genExpr():
    return list(x + 10 for x in repslist) # list() in 2.X si 3.X
def genFunc():
    def gen():
        for x in repslist:
            yield x + 10
    return list(gen()) # list() in 2.X si 3.X
```

```
C:\code>py -3 timeseqs2.py
3.7.4 (tags/v3.7.4:e09359112e, Jul  8 2019,
    20:34:20) [MSC v.1916 64 bit (AMD64)]
forLoop  : 0.82765 => [10...10009]
listComp : 0.47813 => [10...10009]
mapCall  : 1.01315 => [10...10009]
genExpr  : 0.70091 => [10...10009]
genFunc  : 0.70149 => [10...10009]
```

- **map() e mai lent cu functii definite de utilizator!**

timer2.py – cu cuvinte cheie

```
# Fisierul timer2.py (2.X si 3.X):
"""
total(spam, 1, 2, a=3, b=4, _reps=1000)
      apeleaza de _reps ori spam(1, 2, a=3, b=4)
      si returneaza durata totala si ultimul
      rezultat.
bestof(spam, 1, 2, a=3, b=4, _reps=5) testeaza
      de _reps ori, spre a elimina incarcarea
      variabila a sistemului, si returneaza timpul
      cel mai scurt
bestoftotal(spam, 1, 2, a=3, b=4, _reps1=5,
      _reps=1000) testeaza de _reps1 ori  apelul
      lui total cu _reps repetari, returnand timpul
      cel mai scurt
"""
import time, sys
timer = time.perf_counter #time.clock if
      sys.platform[:3] == 'win' else time.time
```

```
def total(func, *pargs, **kargs):
    _reps = kargs.pop('_reps', 1000) # Transmis
    sau cu valoarea 1000 implicita; argumentele
    ramase dupa pop() sunt pentru func
    repslist = list(range(_reps)) # Necontorizat
    start = timer()
    for i in repslist:
        ret = func(*pargs, **kargs)
    elapsed = timer() - start
    return (elapsed, ret)
```

timer2.py...

```
def bestof(func, *pargs, **kargs):
    _reps = kargs.pop('_reps', 5)
    best = 2 ** 32 # Secunde ~136 ani
    for i in range(_reps):
        start = timer()
        ret = func(*pargs, **kargs)
        elapsed = timer() - start
        if elapsed < best: best = elapsed
    return (best, ret)
def bestoftotal(func, *pargs, **kargs):
    _reps1 = kargs.pop('_reps1', 5)
    return min(total(func, *pargs, **kargs) for i
in range(_reps1)) # Se foloseste min() cu
argument de tip expresie generator
```

```
>>> import sys, timer2
>>> from timeseqs import * # Pt. iteratii
....
>>> for test in (forLoop, listComp, mapCall,
    genExpr, genFunc):
        (tot, result) =
    timer2.bestoftotal(test, _reps1=5,
    _reps=1000)
        print ('%-9s: %.5f => [%s...%s]' %
    (test.__name__, tot, result[0], result[-1]))
forLoop  : 0.83452 => [0...9999]
listComp : 0.50851 => [0...9999]
mapCall  : 0.24853 => [0...9999]
genExpr  : 0.74109 => [0...9999]
genFunc  : 0.74006 => [0...9999]
```

- Rezultate similare, map mai rapid ca for, colectii, iar generatorii sunt cei mai lenti

timer2.py...

- Teste interactive:

```
>>> from timer2 import total, bestof, bestoftotal
>>> total(pow, 2, 1000)[0] # 2**1000, repetat de 1000 de ori, implicit
0.0012397000000028413
>>> total(pow, 2, 1000, _reps=1000)[0] # 2**1000, repetat de 1000 ori
0.001226400000000183
>>> total(pow, 2, 1000, _reps=1000000)[0] # 2**1000, repetat de 1M ori
1.1547132000000033
>>> bestof(pow, 2, 100000)[0] # 2**100K, repetata de 1000 ori, implicit
0.00030600000000191585
>>> bestof(pow, 2, 1000000, _reps=30)[0] # 2** 1M, repetat de 30 ori
0.0039655000000067275
>>> bestoftotal(str.upper, 'spam', _reps1=30, _reps=1000) # 30 incercari, fiecare de 1000 ori
(7.629999998925996e-05, 'SPAM')
>>> bestof(total, str.upper, 'spam', _reps=30) # Apel inclus!
(9.509999999579577e-05, (7.839999999248448e-05, 'SPAM'))
```

timer2.py...

- Functie apelata cu cuvinte cheie:

```
>>> def spam(a, b, c, d): return a + b + c + d
>>> total(spam, 1, 2, c=3, d=4, _reps=1000)
(0.0002384000000006381, 10)
>>> bestof(spam, 1, 2, c=3, d=4, _reps=1000)
(1.0999999631167157e-06, 10)
```

```
>>> bestoftotal(spam, 1, 2, c=3, d=4,
        _reps1=1000, _reps=1000)
(0.0002163999999933934, 10)
>>> bestoftotal(spam, *(1, 2), _reps1=1000,
        _reps=1000, **dict(c=3, d=4))
(0.0002167000000099506, 10)
```

timer3.py, doar in Python v3.x

- Cu argumente neaparat cu cuvinte cheie (dupa cele pozitionale):

```
# Fisierul timer3.py (numai in 3.X):
"""
Acelasi mod de utilizare ca timer2.py, dar
    foloseste argumente neaparat cu cuvinte
    cheie, in loc de pop(), cod mai simplu.
    range() este generator in v3.x. Nu merge in
    v2.x!
"""
import time, sys
timer = time.perf_counter ###time.clock if
    sys.platform[:3] == 'win' else time.time
def total(func, *pargs, _reps=1000, **kargs):
    start = timer()
    for i in range(_reps):
        ret = func(*pargs, **kargs)
    elapsed = timer() - start
    return (elapsed, ret)
def bestof(func, *pargs, _reps=5, **kargs):
    best = 2 ** 32
    for i in range(_reps):
        start = timer()
        ret = func(*pargs, **kargs)
        elapsed = timer() - start
        if elapsed < best: best = elapsed
    return (best, ret)
def bestoftotal(func, *pargs, _reps1=5,
    **kargs):
    return min(total(func, *pargs, **kargs) for i
    in range(_reps1))
```

Alte tehnici de benchmarking

- Cu modulul *timeit*
- Cu biblioteca standard *profile*
- Masurarea vitezei de executie a colectiilor iterative de tip *set* si *dict* (adaptare a script-elor anterioare)

Sumar

- Masurarea duratei iteratiilor
- ***Cu timeit***
- Alte metode de benchmarking
- Detalii de implementare a functiilor

timeit, mod de utilizare

- Modulul *timeit* este standard, flexibil si se executa la fel indiferent de platforma (sistem de operare)
- Testele sunt specificate fie cu un obiect apelabil, fie cu instructiuni cuprinse in *str* – separate cu **; \n** spatiu si tab pentru indentare (e.g. \n\t)
- Se pot specifica si actiuni de initializare
- Se executa fie din linia de comanda, cu apeluri API (**A**pplication **P**rogramming **I**nterface), dintr-un script sau cu sesiuni interactive

Mod interactiv, cu API

- Functia *timeit.repeat()* returneaza un *list* care cuprinde durata unui test executat de *number* ori, repetat de *repeat* ori:

```
C:\code>py -3
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul  8 2019, 20:34:20) [MSC v.1916 64 bit (AMD64)] on win32
>>> import timeit
>>> min(timeit.repeat(stmt="[x ** 2 for x in range(1000)]", number=1000, repeat=5))
0.3578413000000005
C:\code>py -2
Python 2.7.17 (v2.7.17:c2f86d86e6, Oct 19 2019, 21:01:17) [MSC v.1500 64 bit (AMD64)] on win32
>>> import timeit
>>> min(timeit.repeat(stmt="[x ** 2 for x in range(1000)]", number=1000, repeat=5))
0.0562915
C:\code>pypy3
Python 3.6.9 (1608da62bfc7, Dec 23 2019, 12:38:24)
[PyPy 7.3.0 with MSC v.1911 32 bit] on win32
>>>> import timeit
>>>> min(timeit.repeat(stmt="[x ** 2 for x in range(1000)]", number=1000, repeat=5))
0.01041439999999999
```

- *min()* calculeaza timpul cel mai scurt

Din linia de comanda

- Fie ca script, fie ca modul cu parametrul **-m**:

C:\code>**python -m timeit -n 1000 "[x ** 2 for x in range(1000)]"**

1000 loops, best of 5: 359 usec per loop

C:\code>**py -3 -m timeit -n 1000 -r 5 "[x ** 2 for x in range(1000)]"**

1000 loops, best of 5: 358 usec per loop

C:\Program Files\Python37\Lib>**python "c:\Program Files\Python37\Lib\timeit.py" -n 1000 "[x ** 2 for x in range(1000)]"**

1000 loops, best of 5: 358 usec per loop

- ~~Cu **-c** spre a forta *time.clock* in toate versiunile de Python:~~

C:\code>**py -3 -m timeit -n 1000 -r 5 -c "[x ** 2 for x in range(1000)]"**

1000 loops, best of 5: 357 usec per loop

C:\code>**py -2 -m timeit -n 1000 -r 5 -c "[x ** 2 for x in range(1000)]"**

1000 loops, best of 5: 56.2 usec per loop

C:\code>**pypy3 -m timeit -n 1000 -r 5 -c "[x ** 2 for x in range(1000)]"**

1000 loops, average of 5: 63.3 +- 65.7 usec per loop (using standard deviation)

Din...

C:\code>**py -3 -m timeit -n 1000 -r 5 -c "[abs(x) for x in range(10000)]"**

1000 loops, best of 5: 727 usec per loop

C:\code>**py -2 -m timeit -n 1000 -r 5 -c "[abs(x) for x in range(10000)]"**

1000 loops, best of 5: 521 usec per loop

C:\code>**pypy3 -m timeit -n 1000 -r 5 -c "[abs(x) for x in range(10000)]"**

1000 loops, average of 5: 96.9 +- 36.3 usec per loop (using standard deviation)

- Aceleasi observatii: viteza PyPy > Cpython 2.7 > Cpython 3.7
- *range()* produce list in v2.x, dar este generator in v3.x, deci codul repetat este diferit

Cod multilinie

- Atentie la apostrofii inclusi, de protejat cu escape \

```
C:\code>py -3
>>> import timeit
>>> min(timeit.repeat(number=10000,
    repeat=3, stmt="L = [1, 2, 3, 4, 5]\nfor i in
    range(len(L)): L[i] += 1"))
0.007608000000004722
>>> min(timeit.repeat(number=10000,
    repeat=3, stmt="L = [1, 2, 3, 4,
    5]\ni=0\nwhile i < len(L):\n\tL[i] += 1\n\ti
    += 1")) # Indentare cu \n\t
0.009302099999999314
>>> min(timeit.repeat(number=10000,
    repeat=3, stmt="L = [1, 2, 3, 4, 5]\nM = [x +
    1 for x in L]"))
0.004526799999993614
```

- Din linia de comanda, spatiu alb pentru indentare, instructiuni separate (se concateneaza automat, cu \n intre stringuri):

```
C:\code>py -3 -m timeit -n 1000 -r 3 "L =
    [1,2,3,4,5]" "i=0" "while i < len(L):" " L[i] +=
    1" " i += 1"
1000 loops, best of 3: 881 nsec per loop
C:\code>py -3 -m timeit -n 1000 -r 3 "L =
    [1,2,3,4,5]" "M = [x + 1 for x in L]"
1000 loops, best of 3: 461 nsec per loop
```

Alte setari pentru *timeit*

- Specificarea codului de initializare, cu **-s** si **setup=**

```
C:\code>python -m timeit -n 1000 -r 3 "L =
    [1,2,3,4,5]" "M = [x + 1 for x in L]"
1000 loops, best of 3: 435 nsec per loop
C:\code>python -m timeit   -n 1000 -r 3 -s "L =
    [1,2,3,4,5]" "M = [x + 1 for x in L]" # Mai
```

rapid, iar variabilele initializate sunt vizibile in codul testat

1000 loops, best of 3: 376 nsec per loop

- Cu API, interactiv:

```
>>> from timeit import repeat
>>> min(repeat(number=1000, repeat=3,
    setup='from mins import min1, min2,
    min3\n'
          'vals=list(range(1000))',
    stmt= 'min3(*vals)'))
0.012894799999997986
```

```
>>> min(repeat(number=1000, repeat=3,
    setup='from mins import min1, min2,
    min3\n'
          'import
    random\nvals=[random.random() for i in
    range(1000)]',
    stmt= 'min3(*vals)'))
0.0912814999999938
```

Alte...

- Cu API *timeit.timeit()*, clasa *timeit.Timer*, cu functie in loc de *str*:

```
>>> import timeit
>>> timeit.timeit(stmt='[x ** 2 for x in range(1000)]', number=1000) # Durata totala
0.3609824000000117
>>> timeit.Timer(stmt='[x ** 2 for x in range(1000)]').timeit(1000) # API de class
0.36002239999993435
>>> timeit.repeat(stmt='[x ** 2 for x in range(1000)]', number=1000, repeat=3)
[0.35965279999993527, 0.3571412999999666, 0.3577203999999483]
>>> def testcase():
        y = [x ** 2 for x in range(1000)] # Functie de testat
...
>>> min(timeit.repeat(stmt=testcase, number=1000, repeat=3))
0.35724730000004
```

Scriptul pybench.py

- Poate testa versiunea curenta a Python sau dintr-o lista:

```
import sys, os, timeit
defnum, defrep= 1000, 5 # Variaza per stmt
def runner(stmts, pythons=None,
    tracecmd=False):
    """
Executa testele din lista, apelantul determina
    modul de utilizare.
stmts: [(number?, repeat?, stmt-string)], se
    inlocuieste $listif3 in stmt
pythons: None=versiunea curenta, sau [(ispy3?,
    python-executable-path)]
    """
    print(sys.version)
    for (number, repeat, stmt) in stmts:
        number = number or defnum
        repeat = repeat or defrep # 0=default
        if not pythons: # stmt se executa in
versiunea curenta de Python: cu apel API
            ispy3 = sys.version[0] == '3'
            stmt = stmt.replace('$listif3',
'list' if ispy3 else '')
            best =
min(timeit.repeat(stmt=stmt,
number=number, repeat=repeat))
            print('%.4f [%r]' % (best,
stmt[:70]))
        else: # stmt se executa cu toate
versiunile de Python indicate cu o cale
completa in lista
```

Scriptul...

```
        # Se acumuleaza argumentele pentru linia de comanda, executata cu os.popen()
        print('-' * 80)
        print('[%r]' % stmt)
        for (ispy3, python) in pythons:
            stmt1 = stmt.replace('$listif3', 'list' if ispy3 else '')
            stmt1 = stmt1.replace('\t', ' ' * 4)
            lines = stmt1.split('\n')
            args = ' '.join('"%s"' % line for line in lines)
            cmd = '"%s" -m timeit -n %s -r %s %s' % (python, number, repeat, args) # Atentie, python trebuie pus intre apostrofi dubli, pentru cazul de spatiu alb in cale!
            print(python)
            if tracecmd: print(cmd)
            print('\t' + os.popen(cmd).read().rstrip())
```

- Executia:

```
import pybench, sys # In fisierul pybench_cases.py
pythons = [
    (1, r'C:\Program Files\Python37\python'),
    (0, r'C:\Python27\python'),
    (1, r'C:\Users\Dan\Downloads\pypy3.6-v7.3.0-win32\pypy3')
]
```

Scriptul...

```
stmts = [ # (num,rpt,stmt)
(0, 0, "[x ** 2 for x in range(1000)]"), # Iteratii
(0, 0, "res=[]\nfor x in range(1000):
    res.append(x ** 2)"), # \n=multistmt
(0, 0, "$listif3(map(lambda x: x ** 2,
    range(1000)))"), # \n\t=indentare
(0, 0, "list(x ** 2 for x in range(1000))"), #
    $listif3=list sau ''
(0, 0, "s = 'spam' * 2500\nx = [s[i] for i in
    range(10000)]"), # Operatii pe str
(0, 0, "s = '?'\nfor i in range(10000): s += '?'"),
]
tracecmd = '-t' in sys.argv # -t: trasare daca -t in
    linia de comanda
pythons = pythons if '-a' in sys.argv else None #
    -a, toate versiunile de Python
pybench.runner(stmts, pythons, tracecmd)
```

Scriptul...

- Rezultate:

```
C:\>py -3 pybench_cases.py
3.7.4 (tags/v3.7.4:e09359112e, Jul  8 2019,
      20:34:20) [MSC v.1916 64 bit (AMD64)]
0.3599  ['[x ** 2 for x in range(1000)]']
0.4021  ['res=[]\nfor x in range(1000):
      res.append(x ** 2)']
0.4182  ['list(map(lambda x: x ** 2,
      range(1000)))']
0.3905  ['list(x ** 2 for x in range(1000))']
0.4822  ["s = 'spam' * 2500\nx = [s[i] for i in
      range(10000)]"]
1.9559  ["s = '?'\nfor i in range(10000): s += '?'"]
```

```
C:\>py -2 pybench_cases.py
2.7.17 (v2.7.17:c2f86d86e6, Oct 19 2019,
      21:01:17) [MSC v.1500 64 bit (AMD64)]
0.0567  ['[x ** 2 for x in range(1000)]']
0.0982  ['res=[]\nfor x in range(1000):
      res.append(x ** 2)']
0.1238  ['(map(lambda x: x ** 2, range(1000)))']
0.0710  ['list(x ** 2 for x in range(1000))']
0.4512  ["s = 'spam' * 2500\nx = [s[i] for i in
      range(10000)]"]
1.7125  ["s = '?'\nfor i in range(10000): s += '?'"]
```

Scriptul...

C:\>**pypy3 pybench_cases.py**

3.6.9 (1608da62bfc7, Dec 23 2019, 12:38:24)

[PyPy 7.3.0 with MSC v.1911 32 bit]

0.0043 ['[x ** 2 for x in range(1000)]']

0.0042 ['res=[]\nfor x in range(1000):
res.append(x ** 2)']

0.0067 ['list(map(lambda x: x ** 2,
range(1000)))']

0.0109 ['list(x ** 2 for x in range(1000))']

0.3996 ["s = 'spam' * 2500\nx = [s[i] for i in
range(10000)]"]

2.1398 ["s = '?'\nfor i in range(10000): s += '?'"]

C:\>**python pybench_cases.py -a**

3.7.4 (tags/v3.7.4:e09359112e, Jul 8 2019,
20:34:20) [MSC v.1916 64 bit (AMD64)]

['[x ** 2 for x in range(1000)]']

C:\Program Files\Python37\python

1000 loops, best of 5: 358 usec per loop

C:\Python27\python

1000 loops, best of 5: 56.2 usec per loop

C:\Users\Dan\Downloads\pypy3.6-v7.3.0-
win32\pypy3

1000 loops, average of 5: 5.2 +- 1.54 usec per
loop (using standard deviation)

**['res=[]\nfor x in range(1000): res.append(x **
2)']**

C:\Program Files\Python37\python

1000 loops, best of 5: 405 usec per loop

Scriptul...

C:\Python27\python

1000 loops, best of 5: 96.7 usec per loop

C:\Users\Dan\Downloads\pypy3.6-v7.3.0-win32\pypy3

1000 loops, average of 5: 4.86 +- 1.67 usec per loop (using standard deviation)

['$listif3(map(lambda x: x ** 2, range(1000)))']

C:\Program Files\Python37\python

1000 loops, best of 5: 421 usec per loop

C:\Python27\python

1000 loops, best of 5: 126 usec per loop

C:\Users\Dan\Downloads\pypy3.6-v7.3.0-win32\pypy3

1000 loops, average of 5: 0.728 +- 0.723 usec per loop (using standard deviation)

['list(x ** 2 for x in range(1000))']

C:\Program Files\Python37\python

1000 loops, best of 5: 390 usec per loop

C:\Python27\python

1000 loops, best of 5: 69.8 usec per loop

C:\Users\Dan\Downloads\pypy3.6-v7.3.0-win32\pypy3

1000 loops, average of 5: 12.6 +- 2.44 usec per loop (using standard deviation)

Scriptul...

["s = 'spam' * 2500\nx = [s[i] for i in range(10000)]"]

C:\Program Files\Python37\python

1000 loops, best of 5: 482 usec per loop

C:\Python27\python

1000 loops, best of 5: 444 usec per loop

C:\Users\Dan\Downloads\pypy3.6-v7.3.0-win32\pypy3

1000 loops, average of 5: 456 +- 7.42 usec per loop (using standard deviation)

["s = '?'\nfor i in range(10000): s += '?'"]

C:\Program Files\Python37\python

1000 loops, best of 5: 1.94 msec per loop

C:\Python27\python

1000 loops, best of 5: 1.65 msec per loop

C:\Users\Dan\Downloads\pypy3.6-v7.3.0-win32\pypy3

1000 loops, average of 5: 2.24 +- 0.011 msec per loop (using standard deviation)

- Observatie: *timeit* fara cod, sau cu *pass*, calculeaza viteza de referinta pentru fiecare versiune de Python.

map() vs. PyPy

- Cu apeluri de functii, map poate fi mai rapid decat PyPy?:

```
# Fisierul pybench_cases2.py
pythons = [
    (1, r'C:\Program Files\Python37\python'),
    (0, r'C:\Python27\python'),
    (1, r'C:\Users\Dan\Downloads\pypy3.6-v7.3.0-win32\pypy3')
]
stmts += [
# Cu apeluri de functii, map e mai rapid; chiar si cu functii definite de utilizator -- ?!
    (0, 0, "[ord(x) for x in 'spam' * 2500]"),
    (0, 0, "res=[]\nfor x in 'spam' * 2500: res.append(ord(x))"),
    (0, 0, "$listif3(map(ord, 'spam' * 2500))"),
    (0, 0, "list(ord(x) for x in 'spam' * 2500)"),
# Set and dicts
    (0, 0, "{x ** 2 for x in range(1000)}"),
    (0, 0, "s=set()\nfor x in range(1000): s.add(x ** 2)"),
    (0, 0, "{x: x ** 2 for x in range(1000)}"),
    (0, 0, "d={}\nfor x in range(1000): d[x] = x ** 2"),
... # Vezi versiunea actualizata din code
]
```

map()...

- Executia – azi, PyPy e mai rapid!:

['[x ** 2 for x in range(1000)]']

C:\Program Files\Python37\python

1000 loops, best of 5: 361 usec per loop

C:\Users\Dan\Downloads\pypy3.6-v7.3.0-win32\pypy3

1000 loops, average of 5: 5.18 +- 1.52 usec per loop (using standard deviation)

['$listif3(map(lambda x: x ** 2, range(1000)))']

C:\Program Files\Python37\python

1000 loops, best of 5: 421 usec per loop

C:\Users\Dan\Downloads\pypy3.6-v7.3.0-win32\pypy3

1000 loops, average of 5: 5.96 +- 2.41 usec per loop (using standard deviation)

["[ord(x) for x in 'spam' * 2500]"]

C:\Program Files\Python37\python

1000 loops, best of 5: 602 usec per loop

C:\Users\Dan\Downloads\pypy3.6-v7.3.0-win32\pypy3

1000 loops, average of 5: 122 +- 5.43 usec per loop (using standard deviation)

["$listif3(map(ord, 'spam' * 2500))"]

C:\Program Files\Python37\python

1000 loops, best of 5: 314 usec per loop

C:\Users\Dan\Downloads\pypy3.6-v7.3.0-win32\pypy3

1000 loops, average of 5: 102 +- 2.63 usec per loop (using standard deviation)

Teste ad-hoc vs. *timeit*

- Rezultate similare:

C:\>**py -3 timeseqs3.py**

3.7.4 (tags/v3.7.4:e09359112e, Jul 8 2019, 20:34:20) [MSC v.1916 64 bit (AMD64)]

forLoop : **0.39429** => [0...998001]

listComp : 0.35199 => [0...998001]

mapCall : 0.41473 => [0...998001]

genExpr : 0.38181 => [0...998001]

genFunc : 0.38313 => [0...998001]

C:\>**py -3 pybench_cases.py**

3.7.4 (tags/v3.7.4:e09359112e, Jul 8 2019, 20:34:20) [MSC v.1916 64 bit (AMD64)]

0.3577 ['[x ** 2 for x in range(1000)]']

0.4027 ['res=[]\nfor x in range(1000): res.append(x ** 2)']

0.4190 ['list(map(lambda x: x ** 2, range(1000)))']

0.3895 ['list(x ** 2 for x in range(1000))']

0.4784 ["s = 'spam' * 2500\nx = [s[i] for i in range(10000)]"]

1.9321 ["s = '?'\nfor i in range(10000): s += '?'"]

Imbunatatiri cu *setup=*

```
# Fisier pybench2.py, diferentele
def runner(stmts, pythons=None,
    tracecmd=False):
    for (number, repeat, setup, stmt) in stmts:
        if not pythons:
            ...
            best = min(timeit.repeat(
setup=setup, stmt=stmt, number=number,
    repeat=repeat))
        else:
            setup = setup.replace('\t', ' ' *
4)
            setup = ' '.join('-s "%s"' % line
for line in setup.split('\n'))
            ...
            for (ispy3, python) in pythons:
                ...
                cmd = '%s -m timeit -n %s
-r %s %s %s' % (python, number, repeat,
setup, args)
```

```
# Fisier pybench2_cases.py, diferente
import pybench2, sys
stmts = [ # (num,rpt,setup,stmt)
(0, 0, "", "[x ** 2 for x in range(1000)]"),
(0, 0, "L = [1, 2, 3, 4, 5]", "for i in range(len(L)):
    L[i] += 1"),
(0, 0, "L = [1, 2, 3, 4, 5]", "i=0\nwhile i <
    len(L):\n\tL[i] += 1\n\ti += 1") ]
pybench2.runner(stmts, pythons, tracecmd)
```

Imbunatatiri...

- Observatie: codul API este mai clar, deoarece argumentele sunt transmise nemodificate, nefiind afectate de shell-ul liniei de comanda:

 # API:

```
(0, 0, "def f(x):\n\treturn x",
"res=[]\nfor x in 'spam' *
    2500:\n\tres.append(f(x))")
```

 # Sub shell, cu spatii pentru indentare:

```
python -m timeit -n 1000 -r 5 -s "def f(x):"
-s "    return x" "res=[]"
"for x in 'spam' * 2500: "
"    res.append(f(x))"
```

Sumar

- Masurarea duratei iteratiilor
- Cu *timeit*
- ***Alte metode de benchmarking***
- Detalii de implementare a functiilor

Benchmarking cu *pystones*

Alte metode de benchmarking in Python:

- Cu *pystone.py*, aflat in distributia de Python v2.7 in directorul Lib/test si in cea de PyPy (lib-python/3/test)
- La http://speed.python.org, unde se gasesc partajate diverse rezultate ale unor benchmark-uri
- La http://speed.pypy.org, asemanator, dar pentru PyPy

C:\Python27\Lib\test>**py -2 pystone.py**

Pystone(1.1) time for 50000 passes = 0.279277

This machine benchmarks at 179034 pystones/second

C:\Users\Dan\Downloads\pypy3.6-v7.3.0-win32\lib-python\3\test>**pypy3 pystone.py**

Pystone(1.2) time for 50000 passes = 0.0468709

This machine benchmarks at 1.06676e+06 pystones/second

Sumar

- Masurarea duratei iteratiilor
- Cu *timeit*
- Alte metode de benchmarking
- ***Detalii de implementare a functiilor***

Variabilele locale sunt detectate static

- Desi variabilele asignate sunt implicit locale functiilor, ele sunt gasite in faza de compilare a *def*-ului in mod static:

```
>>> X = 99
>>> def selector(): # X folosit, neasignat
        print(X) # X este cel global
>>> selector()
99
>>> def selector():
        print(X) # X este local, nu exista inca
        X = 88 # X este local, peste tot in functie, deci si mai sus!
# Se poate intampla si pentru import X sau def X...
>>> selector()
UnboundLocalError: local variable 'X' referenced before assignment
```

Variabilele...

- Rezolvare posibila, cu *global*:

```
>>> def selector():
        global X # X este global, peste tot in functie
        print(X)
        X = 88

>>> selector()
99
```

- Acces atat la variabila globala cat si la cea locala:

```
>>> X = 99
>>> def selector():
        import __main__ # __main__ este modulul global, in mod interactiv (Idle)
        print(__main__.X) # __main__.X este atribut al modulului, deci global
        X = 88 # X (necalificat) este local
        print(X) # Afiseaza X local

>>> selector()
99
88
```

Obiecte modificabile ca argumente implicite

- Argumentele cu valori implicite sunt evaluate si salvate o singura data, la executia *def*-ului, iara nu la apelurile functiei
- Ca urmare, exista un singur obiect per argument cu valoare implicita

```
>>> def saver(x=[]): # Valoare implicita, un list
        x.append(1) # Acelasi obiect este actualizat, mereu
        print(x)

>>> saver([2]) # Apel fara valoare implicita
[2, 1]

>>> saver() # Apel cu valoare implicita
[1]
>>> saver() # Lista creste la toate apelurile cu argument implicit (lipsa)
[1, 1]
>>> saver()
[1, 1, 1]
```

Obiecte...

- Rezolvare prin copierea valorii implicite la inceputul functiei:

```
>>> def saver(x=None):
        if x is None: # Fara argument?
            x = [] # Creeaza un list nou
        x.append(1) # Actualizare a lui x
        print(x)
```

```
>>> saver([2])
[2, 1]
>>> saver() # Nu mai creste
[1]
>>> saver()
[1]
```

- Rezolvare cu atribute de functii (unice):

```
>>> def saver():
        saver.x.append(1) # Desi numele saver este global, se modifica doar o componenta a sa, atributul x, permis!
        print(saver.x)
>>> saver.x = [] # Initializare, dupa executia def-ului
```

```
>>> saver()
[1]
>>> saver()
[1, 1]
>>> saver()
[1, 1, 1]
```

Functii fara *return*

- Instructiunile *return* si *yield* sunt optionale
- Fara return se executa corpul functiei si se returneaza *None* – proceduri:

```
>>> def proc(x):
        print(x) # Fara return => None
>>> x = proc('testing 123...')
testing 123...
>>> print(x)
None
```

```
>>> list = [1, 2, 3]
>>> list = list.append(4) # append este o
    procedura returneaza None
>>> print(list) # Lista s-a pierdut, atentie!
None
```

Alte erori

- Fabricile de functii care genereaza functii intr-un *for* permit memorarea **unei singure** variabile de stare dependenta de contor – ultima valoare din iteratie
  - Solutia: functiile generate vor folosi argumente cu valori implicite, calculate la definitia acestora, fiecare retinand o stare diferita – corespunzatoare iteratiei curente

Alte...

- Variabilele din *builtins* pot fi mascate prin asignari globale sau locale
  - Solutia: ori variabilele predefinite nu sunt necesare, ori verificati codul cu PyChecker:

```
C:\> pychecker [options] file1.py file2.py ...
```

--no-shadowbuiltin check if a variable shadows a builtin, off[implicit]