Pascal ORTIZ

Chronométrer, profiler

Version du 11 octobre 2020

Table des matières

Mesurer le temps d'exécution d'un code Python

Lorsque qu'on veut mesurer de manière commode et efficace la vitesse d'exécution d'un code Python, on fait appel, *dans* le fichier Python contenant le code à chronométrer, à la fonction `perf_counter` du module standard `time`. Voici un exemple d'utilisation :

```
1  from time import perf_counter
2
3  begin_perf = perf_counter()
4
5  #  ---- Début de code à exécuter ----
6
7  n=400000
8  x=10**n
9  N=len(str(x))
10
11 #  ---- Fin de code à exécuter ----
12
13 delta = perf_counter() - begin_perf
14
15 print(f"Temps d'exécution : {delta:.2f}s")
16
17 print(N==n+1)
```

qui affiche

```
Temps d'exécution : 1.91s
True
```

— Ligne 1 : la fonction de chronométrage est `perf_counter` et elle doit être importée du module standard `time`

— Lignes 5-11 : le code dont on veut mesurer la durée d'exécution

— Ligne 3 : avant l'exécution du code, on prend le temps fourni par une horloge du processeur avec un appel à `perf_counter`

— Ligne 13 : après l'exécution du code, on prend, toujours avec `perf_counter`, le nouveau temps fourni par l'horloge et on calcule l'écart ce qui correspond au temps d'exécution du code.

— Ligne 15 : on affiche un message donnant le temps d'exécution. On a utilisé une f-chaîne et la durée est arrondie deux chiffres après la virgule avec le spécifieur `.2f`

— Ligne 17 : optionnellement, on vérifie la justesse du code.

Le temps mesuré n'est pas strictement le temps réel entre le début et la fin de l'exécution (dit *temps réel* ou encore *wall time* pour temps de l'horloge murale). C'est le temps mesuré par une horloge du processeur et qui correspond au temps consacré par le processeur à l'exécution du code. En effet, pendant que le processeur traite le code, il gère aussi bien d'autres actions qui prennent un certain temps ce qui doit provoquer une correction de mesure. En pratique, assez souvent, le temps mesuré par `perf_counter` correspond au temps d'exécution réel mais il peut y avoir des exceptions, en particulier en cas d'exécutions parallèles sur un processeur multicœur.

La fonction `perf_counter` est la fonction à utiliser par excellence pour faire des comparaisons de performance. Comme dit l'aide de la fonction :

```
perf_counter(...)
    perf_counter() -> float

    Performance counter for benchmarking.
```

La fonction `perf_counter` est le timer par défaut utilisé dans Python (fonction `default_timer` du module `time`). Par voie de conséquence, c'est le timer utilisé par exemple par la fonction `timeit`.

Temps d'exécution avec `timeit`

Soit à mesurer le temps d'exécution d'une ou plusieurs instructions Python, le plus souvent juste une ligne de code, le but étant de faire une expérimentation rapide et simple à réaliser. Par exemple soit le code Python suivant qui évalue 10 à la puissance un million :

```
x=10**10**6
```

On cherche à déterminer le temps moyen d'exécution d'un tel code. Pour cela, on peut lancer Python dans un terminal-système comme ci-après :

```
$ python -m timeit "x=10**1000000"
10 loops, best of 3: 172 msec per loop
```

Commentaires :

- Le programme est lancé dans un terminal-système (cf. le symbole $) et pas dans une console Python
- La commande `python -m` timeit est obligatoire. Elle appelle Python, en chargeant le module (cf. l'option `-m`) nommé `timeit` qui est un module standard.
- le code dont on veut chronométrer l'exécution, est placé à la fin, c'est une chaîne de caractères, entre guillemets
- Le programme lance Python sur le code proposé, le code est exécuté un certain nombre de fois, ici 10, et un temps moyen d'exécution est affiché.

A moins que votre chaîne ne contienne des appels à la fonction `print`, la commande n'affiche rien d'autre qu'un message comme ci-dessus. Le nombre de répétitions est déterminé en interne par Python.

Cette utilisation n'est adaptée que pour chronométrer des petits morceaux de code, qui tiennent sur une ligne et pas davantage, bien que ce soit possible (cf. plus bas).

`timeit` : utilisation typique

En pratique, la commande `python -m timeit` est utilisée pour comparer les performances de deux codes effectuant la même tâche :

```
$ python3 -m timeit -s "N=10**3" -s "s = ''" "for i in range(N):" "    s += 'X'"
10000 loops, best of 3: 59.1 usec per loop
$ python3 -m timeit -s "N=10**3" -s "s = ''" "for i in range(N):" "    s = 'X' + s"
[1000 loops, best of 3: 24.9 msec per loop
$
```

Ce n'est pas toujours pratique, car ça peut être très long à exécuter. Une unité de temps en *usec* représente des microsecondes et en *msec* des millisecondes.

Remarques

— Je ne donnerai pas tous les détails de l'utilisation de `timeit`. Sa documentation officielle est ici.

— `timeit` est codé en Python dans le fichier timeit.py dont la lecture est utile pour comprendre certains détails peu clairement expliqués dans la documentation.

— On notera aussi que timeit utilise une instruction exec.

`timeit` : mode de calcul et options utiles

Avant de parler des options qu'on peut donner à `timeit`, expliquons comment `timeit` calcule le temps d'exécution d'un code comme :

```
$ python -m timeit my_snippet
```

`timeit` dispose au départ de deux valeurs entières positives `n` et `r`. Le calcul de la durée d'exécution qui va être affichée est effectué de la manière suivante :

— réalisation de `n` chronométrages de l'exécution de l'instruction Python `my_snippet`

— calcul de la moyenne de ces `n` chronométrages

— répétition `r` fois du calcul de moyenne précédent

— détermination de la meilleure moyenne (temps le plus court)

— affichage du résultat.

Par exemple, si `n=50` et `r=20` et si la meilleure moyenne est de 5 ms, le code affichera le message :

```
50 loops, best of 20: 5 msec per loop
```

Cela veut dire que `timeit` a exécuté le morceau de code au total 1000 fois.

Par défaut, `r=5` et `n` est estimé au fur et à mesure à une valeur suffisamment grande pour que le temps obtenu soit significatif. Cela peut conduire à des valeurs telles que `n=100000` ou beaucoup plus petites comme `n=20`.

Les options

On peut donner en option à `timeit` des valeurs choisies pour n et r. Voici la syntaxe à utiliser par exemple, pour `n=50` et `r=4` :

```
$ python3 -m timeit -n 50 -r 4 "x=10**1000000"
50 loops, best of 4: 172 msec per loop
```

Les options sont introduites par `-n` et `-r` sans espace après le tiret. Noter qu'on peut par exemple écrire `-n50` au lieu de `-n 50`.

Noter que la ligne de commande ci-dessus met un temps non négligeable à s'exécuter environ $0.172\text{s} \times 50 \times 4 \simeq 35\text{s}$. Voilà aussi pourquoi il est utile de disposer des options n et r.

Chronométrer avec `timeit` dans des feuilles Jupyter

L'environnement IPython (par exemple sous Jupyter Notebook) dispose nativement d'une commande, dite « magique », `timeit` qui permet de chronométrer l'exécution :

— soit d'une expression (avec `%timeit` en début de ligne)
— soit d'une cellule de code (avec `%%timeit` en début de cellule)

Par exemple, la cellule

```
1 def calc(n):
2     a=n*10**5
3     x=10**a
4     s=str(x)
5     N=len(s)
6     return N==a+1
7
8 %timeit calc(4)
```

de par sa ligne 8, affiche à l'exécution :

```
1.95 s ± 24.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

On pourrait aussi chronométrer toute une cellule :

```
%%timeit

n=4
a=n*10**5
x=10**a
s=str(x)
N=len(s)
print (N==a+1)
```

qui affiche

```
True
True
True
True
True
True
True
True
1.93 s ± 11.9 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Par défaut, `timeit` répète l'exécution du bloc ou de l'expression 7 fois (en plus de l'exécution préalable de la cellule). Avec l'option `-r`, on peut modifier ce nombre, par exemple ci-dessous à 3 :

```
def calc(n):
    a=n*10**5
    x=10**a
    s=str(x)
    N=len(s)
    return N==a+1
```

```
%timeit -r3 calc(4)
```

On peut placer plusieurs commandes `%timeit` dans une même cellule et à n'importe quel endroit :

```
%timeit 10**100000
print(42)
%timeit 10**200000
print(42)
```

qui affiche

```
4.34 ms ± 35.3 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
42
13.2 ms ± 76.3 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
42
```

Chronométrer avec `timeit` dans du code python

Le module standard `timeit` permet de mesurer de petits morceaux de code dans du code Python (et pas simplement en ligne de commande). Il s'utilisera plus naturellement en mode interactif plutôt qu'en mode fichier. Voici un exemple :

```
>>> from timeit import timeit
>>> timeit("x=10**10**6", number=4)
0.6991793749984936
>>>
```

On donne une instruction Python entourée de guillemets, ici `"x=10**10**6"`, on indique avec l'argument nommé `number` combien de fois on veut qu'elle soit exécutée et le temps d'exécution est alors affiché.

Il existe aussi une classe `timeit` qui permet de chronométrer du code. D'une manière générale, l'usage de `timeit` est assez rare ailleurs que dans un terminal système.

`timeit` pour un code-source plus volumineux

On peut utiliser `timeit` pour mesurer le temps d'exécution d'un morceau de code plus volumineux qu'un simple instruction, par exemple le code

```
N=10**6
s = ''
for i in range(N):
    s += 'X'
```

Ce n'est pas très pratique mais on peut le chronométrer dans un terminal-système de la manière suivante :

```
$ python -m timeit -s "N=10**6" -s "s = ''" "for i in range(N):" "    s += 'X'"
10 loops, best of 3: 92.2 msec per loop
```

— La commande commence toujours par `$ python -m timeit`

— La ligne de commande peut être suivie par plusieurs `-s` chacun suivi d'une commande : il s'agit de commandes d'initialisation, exécutées avant chaque mesure de la ligne de code à chronométrer. Dans l'exemple ci-dessus, deux instructions sont pré-exécutées.
— La fin de la ligne de commande est constituée du code Python dont la durée d'exécution va être mesurée en en calculant la moyenne du temps d'exécution en la répétant un certain nombre de fois. Si le code Python tient sur plusieurs lignes, il faut placer chaque ligne entre guillemets, en respectant l'éventuelle indentation de la ligne.

La fonction `process_time`

Il existe une autre fonction que la fonction `perf_counter` pour mesurer un temps d'exécution. Il s'agit de la fonction `process_time` du module standard `time`.

Son utilisation est très semblable à celle de `perf_counter` :

```
from time import process_time

begin_process = process_time()

#  ---- Début de code à exécuter ----

n=400000
x=10**n
N=len(str(x))

#  ---- Fin de code à exécuter ----

delta = process_time() - begin_process

print(f"Temps d'exécution : {delta:.2f}s")

print(N==n+1)
```

qui affiche

```
Temps d'exécution : 1.92s
True
```

En pratique, pour un programme mono-processeur et lié au processeur (*CPU-bounded* en anglais), les temps fournis par les fonctions `process_time` et `perf_counter` sont proches. Toutefois, `perf_counter` compte les temps donnés à la fonction `sleep` alors que ce n'est pas le cas de la fonction `process_time` :

```
from time import perf_counter, process_time, sleep

begin = perf_counter()

sleep(2)

delta = perf_counter() - begin

```

```
print(f"perf_counter: {delta:.2f}s")


begin = process_time()

sleep(2)

delta = process_time() - begin

print(f"process_time : {delta:.2f}s")
```

qui affiche

```
perf_counter: 2.00s
process_time : 0.00s
```

Comportements différents de `process_time` et `perf_counter`

Les deux fonctions `process_time` et `perf_counter` ne comptent pas les mêmes actions. Selon la documentation, `perf_counter` est *system-wide* tandis que `process_time` est *process-wide*. Pour une définition de ces deux notions on pourra se reporter à Understanding time.perfcounter and time.processtime.

Soit par exemple le code suivant qui écrit successivement 10 fichiers de grosse taille (plus de 130 Mo) :

```
from time import perf_counter, process_time

N=10**6

cpt=0
for i in range(10):
    begin = perf_counter()

    with open(f"test_perf{cpt}.txt", "w") as f:
        f.write('\n'.join("X"*140 for _ in range(N)))

    delta = perf_counter() - begin
    cpt+=1
    print(f"perf_counter: {delta:.2f}s")


    begin = process_time()

    with open(f"test_proc{i}.txt", "w") as f:
        f.write('\n'.join("X"*140 for _ in range(N)))

    delta = process_time() - begin
    cpt+=1

```

```
25      print(f"process_time : {delta:.2f}s")
26
```

— Lignes 9-10 et 19-20 : on écrit dans un fichier dont le nom est de la forme `test_perfXX` un texte de taille importante.

A l'exécution (sous Linux), il affiche :

```
perf_counter: 0.34s
process_time : 0.33s
------------------------------
perf_counter: 0.33s
process_time : 0.33s
------------------------------
perf_counter: 0.33s
process_time : 0.34s
------------------------------
perf_counter: 0.65s
process_time : 0.34s
------------------------------
perf_counter: 0.92s
process_time : 0.34s
------------------------------
perf_counter: 1.34s
process_time : 0.33s
------------------------------
perf_counter: 1.42s
process_time : 0.33s
------------------------------
perf_counter: 1.44s
process_time : 0.34s
------------------------------
perf_counter: 1.31s
process_time : 0.33s
------------------------------
perf_counter: 1.36s
process_time : 0.33s
------------------------------
```

On observe que :

— les temps des deux fonctions sont **très** différents (sauf au début)

— le temps donné par `process_time` est très stable

— le temps enregistré par `perf_counter` semble augmenter avec les écritures de fichiers ce qui correspond, peut-être, au fait que l'accumlulation des fichiers nécessite plus de ressources système.

Pour savoir quel est le temps CPU correct, on crée un fichier Python qui exécute la tâche, sans rien d'autre :

write_file.py

```
1 from random import sample
2 name=''.join(sample("abcdefgh",3))
3
4 N=10**6
5
6 with open(f"test_perf_{name}.txt", "w") as f:
7     f.write('\n'.join("X"*140 for _ in range(N)))
```

Le nom du fichier créé par la tâche est de la forme `test_perf_ebf` (la fin du nom est aléatoire, cf. lignes 2 et 6). On évalue alors (sous Linux) avec la commande système `time` le temps d'exécution :

```
$ time python3 write_file.py

real    0m0,362s
user    0m0,164s
sys     0m0,199s
```

Le temps CPU d'exécution est la somme des temps CPU user et sys (système) soit environ 0.36s (sommes des deux dernières lignes). On peut même répéter l'action par le biais d'un script python (version 3.7 au moins) :

```
import subprocess

for _ in range(10):
    p=subprocess.run(['time' ,'python3', 'write_file.py'], capture_output=True)

    user, system=p.stderr.split()[:2]
    print(f"CPU time: {float(user[:4])+float(system[:4]):.2f}s")
    print("---------------")
```

qui affiche

```
CPU time: 0.34s
---------------
CPU time: 0.34s
---------------
CPU time: 0.34s
---------------
CPU time: 0.34s
---------------
CPU time: 0.35s
---------------
CPU time: 0.35s
---------------
CPU time: 0.35s
---------------
CPU time: 0.36s
---------------
CPU time: 0.35s
---------------
```

```
CPU time: 0.37s
---------------
```

Ainsi, dans le cas présent, c'est la fonction `process_time` qui donne la valeur la plus réaliste.

D'après l'aide, la fonction `process_time` est ainsi décrite :

```
Process time for profiling: sum of the kernel and user-space CPU time.
```

Cela correspond à ce que l'expérimentation ci-dessus donne.

Pour des informations supplémentaires, on pourra consulter cette étude.

Chronométrage du multiprocessing

Comparons les mesures de durées quand un programme utilise plusieurs processus via le module multiprocessing. La machine utilisée dispose de 4 cœurs. Le programme qui servira d'exemple est le produit de deux matrices.

Examinons d'abord l'exécution réelle en mono-processus et en mesurant le temps d'exécution via la commande bash `time`. Le programme est :

single.py

```
from random import randrange

def split(A):
    n=len(A)
    lines=[[A[i][j] for j in range(n)] for i in range(n)]
    cols=[[A[i][j] for i in range(n)] for j in range(n)]
    return lines, cols

def linXcol(L, C):
    return sum(a*b for (a,b) in zip(L,C))

def prod(lines, cols,n):
    P=[[None]*n for _ in range(n)]
    for i in range(n):
        for j in range(n):
            P[i][j]=linXcol(lines[i], cols[j])
    return P

def product(A):
    n=len(A)
    lines, cols=split(A)
    P=prod(lines, cols, n)


def test(n):

    A=[[randrange(2) for _ in range(n)] for _ in range(n)]

    product(A)
```

```
test(400)
```

qui affiche

```
$ time python3 single.py

real    0m4,269s
user    0m4,247s
sys     0m0,008s
```

On voit donc que le temps d'exécution du carré d'une matrice de taille 400×400 est de l'ordre de 4.3s. On observe que le temps CPU user est équivalent au temps d'exécution.

Faisons la même chose pour un calcul distribué sur les 4 cœurs. Le programme est :

pool.py

```
from random import randrange
from multiprocessing import Pool

def split(A):
    n=len(A)
    lines=[[A[i][j] for j in range(n)] for i in range(n)]
    cols=[[A[i][j] for i in range(n)] for j in range(n)]
    return lines, cols

def linXcol_pool(t):
    (i, j), lines, cols=t
    L=lines[i]
    C=cols[j]
    return sum(a*b for (a,b) in zip(L,C))

def product_pool(A, nproc):

    n=len(A)
    lines, cols=split(A)
    L=[((i,j), lines, cols) for i in range(n) for j in range(n)]

    pool=Pool(nproc)
    P=pool.map(linXcol_pool, L)
    return [P[k*n:(k+1)*n] for k in range(n)]

def test(n):
    A=[[randrange(2) for _ in range(n)] for _ in range(n)]
    product_pool(A, 4)

test(400)
```

qui affiche

```
$ time python3 pool.py

```

```
3 real    0m1,445s
4 user    0m4,545s
5 sys     0m0,090s
```

On constate que le temps réel (ligne 3) est environ le tiers du temps d'exécution CPU (somme des temps user et sys) qui est de l'ordre de grandeur du temps de calcul sur un seul processeur.

Maintenant, effectuons des mesures avec les fonctions `perf_counter` et `process_time` :

```python
from multiprocessing import Pool
from random import randrange
from time import  perf_counter, process_time


def split(A):
    n=len(A)
    lines=[[A[i][j] for j in range(n)] for i in range(n)]
    cols=[[A[i][j] for i in range(n)] for j in range(n)]
    return lines, cols

def linXcol(L, C):
    return sum(a*b for (a,b) in zip(L,C))

def prod(lines, cols,n):
    P=[[None]*n for _ in range(n)]
    for i in range(n):
        for j in range(n):
            P[i][j]=linXcol(lines[i], cols[j])
    return P

def product(A):
    n=len(A)
    lines, cols=split(A)
    P=prod(lines, cols, n)

    return P

def linXcol_pool(t):
    (i, j), lines, cols=t
    L=lines[i]
    C=cols[j]
    return sum(a*b for (a,b) in zip(L,C))

def product_pool(A, nproc):
    n=len(A)
    lines, cols=split(A)
    L=[((i,j), lines, cols) for i in range(n) for j in range(n)]

    pool=Pool(nproc)
    P=pool.map(linXcol_pool, L)
```

```
    return [P[k*n:(k+1)*n] for k in range(n)]


def test(n):

    begin_perf=perf_counter()

    A=[[randrange(2) for _ in range(n)] for _ in range(n)]
    print("Matrix generation: %.2fs\n" %(perf_counter()-begin_perf))


    begin_perf=perf_counter()

    P1=product(A)

    print("Product serial: %.2fs\n" %(perf_counter()-begin_perf))


    begin_perf=perf_counter()

    P2=product_pool(A, 4)

    print("Product pool perf_counter: %.2fs\n" %(perf_counter()-begin_perf))


    begin_process=process_time()

    P3=product_pool(A, 4)

    print("Product pool process_time: %.2fs\n" %(process_time()-begin_process))

    # check
    print(P1 == P2 == P3)


test(400)
```

qui affiche

```
Matrix generation: 0.10s

Product serial: 3.98s

Product pool perf_counter: 1.22s

Product pool process_time: 0.25s

True
```

Le temps fourni par la fonction `process_time` ne semble ne mesurer que le temps d'exécution

du processus principal, autrement dit il ignore l'essentiel de ce qui est fait à cette ligne :

```
P=pool.map(linXcol_pool, L)
```

qui distribue le calcul sur les différents processus. C'est sans doute ce qu'il faut comprendre quand la documentation dit que `process time is process-wide`.

Par contre, `perf_counter` mesure l'exécution des processus mais sans additionner leurs durées. Le rapport entre le temps mono-processus et multiprocessus est dans un ratio cohérent d'un peu plus de 3. Donc, `perf_counter` semble plus adapté à la mesure de benchmarks.

Curiosité : chronométrer avec un décorateur

Il est classique d'utiliser un décorateur pour afficher le chronométrage de l'exécution d'une fonction. Le principe est simple si on a compris ce que fait un décorateur. Voici un exemple :

```
from time import perf_counter

def print_elapsed(f):
    def wrapped(*args):
        begin=perf_counter()
        r=f(*args)
        delta=perf_counter() - begin
        print(f"Temps d'exécution : {delta:.2f}s")
        return r
    return wrapped

@print_elapsed
def calc(n):
    return 10**(n*10**5)

calc(12)
```

qui affiche

```
Temps d'exécution : 0.24s
```

On pourrait enrichir le message affiché en plaçant l'argument de la fonction ou son nom, etc.

Toutefois ce code présente plusieurs inconvénients comme la perte du nom de la fonction, ou un affichage inadapté si la fonction est récursive même si on peut contourner ces difficultés. Il est également possible d'envisager un décorateur à arguments.

Bien que signalé dans les manuels, cette technique semble assez peu utilisé dans des projets d'accès public. Le seul exemple que j'ai vu est le module sympy de calcul formel qui utilise un décorateur timethis.

Curiosité : chronométrer avec un gestionnaire de contexte

Un gestionnaire de contexte permet d'envelopper le chronométrage d'un morceau de code. Voici un exemple :

```python
from time import perf_counter

class Timer:
    def __enter__(self):
        self.begin = perf_counter()
        return self

    def __exit__(self, *args):
        self.delta = perf_counter() - self.begin
        print(f"Temps d'exécution : {self.delta:.2f}s")

def calc(n):
    return 10**(n*10**5)

with Timer() as t:
    calc(12)

print("La durée était :", t.delta)
```

qui affiche :

```
Temps d'exécution : 0.24s
La durée était : 0.24153239900624612
```

L'idée est déjà assez ancienne (2011) : Timing Your Code Using Python's with Statement.

Les profileurs

Pour comprendre ce qu'est un profileur, le plus simple est d'en utiliser un, cf. les paragraphes traitant du plus connu d'entre eux, cProfile, même si la prise en main n'est pas complètement immédiate.

Un profileur d'un code-source Python est un programme qui monitore les appels et les retours des fonctions, le nombre d'appels et leurs durées d'exécution. Le but est de savoir comment se répartit le flux d'exécution du code Python entre les différentes fonctions qui le constituent.

Les profileurs s'utilisent

— ou bien en interne à un fichier source Python que l'on exécute

— ou bien en ligne de commande

Ils permettent de récupérer des statistiques de distribution des temps d'exécution, parfois visualisables via des graphiques. De par leur nature, un profileur travaille en dynamique, en même temps que s'exécute le programme du code-source et ses analyses peuvent ralentir donc plus ou moins l'exécution du programme. Selon la nature du profileur (déterministe ou statistique), la pénalisation sera plus ou moins importante.

Le profileur Python le plus connu est cProfile, il est inclus dans la bibliothèque standard de Python. Il sera décrit ici en détail. Bien que ce soit un profileur déterministe, il est considéré comme étant relativement efficace, provoquant un ralentissement modéré de l'exécution du code analysé. Noter que c'est un programme dont l'utilisation est très peu visible dans les codes-sources, par exemple à l'examen de 40000 fichiers de bibliothèques Python connues et disponibles sur Github, sa mention n'apparaît que dans une vingtaine de fichiers.

Un autre profileur est py-spy qui est un **programme externe** pouvant profiler un code Python. C'est un profileur statistique, provoquant un ralentissement très modéré de l'exécution du code analysé et qui, à ce titre, peut être utilisé en production (et pas seulement en conception, avant son utilisation par des « clients ») et même pendant que le programme-source s'exécute. Il offre des possibilités de consultation graphique du diagramme d'exécution. Ce programme sera rapidement examiné dans la présente documentation.

Le profileur `line_profiler` est lui capable de mesurer l'exécution d'un code ligne par ligne.

cProfile : l'essentiel

Soit le code suivant :

`test_cprofile.py`

```
1  from cProfile import run
2
3  def A(n):
4      str(10**(n))
5      B(n)
6      B(n)
7
8  def B(n):
9      str(10**(n))
10
11 run("A(288500)")
```

— Lignes 3 et 8 : deux fonctions « artificielles » `A` et `B`,

— Lignes 5-6 : la fonction `A` appele deux fois la fonction `B`

— Lignes 4 et 9 : écriture en chiffres décimaux d'un nombre assez grand, opération qui peut être assez coûteuse. La valeur de 288500 a été choisie pour que, sur ma machine, la durée d'exécution de l'opération soit très proche de 1s (à des fins pédagogiques).

L'exécution du fichier `test_cprofile.py` va provoquer le profilage du code exécuté lors d'un appel à la fonction `A`, ligne 11, entre les guillemets. C'est la fonction `run` du module cProfile qui lance le profilage. Lorsque le fichier est exécuté, on lit en console :

```
1          6 function calls in 3.009 seconds
2
3    Ordered by: standard name
4
5    ncalls  tottime  percall  cumtime  percall filename:lineno(function)
6         1    0.000    0.000    3.009    3.009 <string>:1(<module>)
7         1    0.998    0.998    3.009    3.009 test_cprofile.py:3(A)
8         2    2.011    1.005    2.011    1.005 test_cprofile.py:8(B)
9         1    0.000    0.000    3.009    3.009 {built-in method builtins.exec}
10        1    0.000    0.000    0.000    0.000 {method 'disable' of
11                                                '_lsprof.Profiler' objects}
```

Pour une meilleure lisibilité, j'ai changé la longueur d'une ligne de la sortie affichée

Les lignes 6, 9 et 10-11 n'étant pas très intéressantes, je les supprime manuellement de la sortie :

```
1          6 function calls in 3.009 seconds
2
3    Ordered by: standard name
4
5    ncalls  tottime  percall  cumtime  percall filename:lineno(function)
6         1    0.998    0.998    3.009    3.009 test_cprofile.py:3(A)
7         2    2.011    1.005    2.011    1.005 test_cprofile.py:8(B)
```

— Ligne 6 : la fonction `A` a été appelée 1 fois (colonne *ncalls*), la totalité de l'appel a duré 3.009s (colonne *cumtime*). Mais la fonction `A` appelle deux fois la fonction `B` donc si on décompte ces appels, l'exécution de `A` a seulement coûté une durée de 0.998s (colonne *tottime*). Cette durée correspond donc à la durée d'exécution de la ligne 4 du code-source, environ 1s.

— Ligne 7 : les deux appels de la fonction `B`, cf. la première et la dernière colonne qui montre aussi le numéro de ligne dans l'appel dans le code-source. Comme la fonction `B` n'appelle aucune autre fonction, les durées des colonnes *tottime* et *cumtime* sont identiques. La colonne *percall* est toujours *cumtime/ncalls*. Les résultats sont cohérents car la tâche effectuée lignes 4 ou 10 du code source dure environ 1s.

— Dans le code-source (ligne 11), on notera que l'appel que l'on va profiler est écrit dans une chaîne de caractères.

Donnons un exemple beaucoup plus simple et qui montre qu'on peut profiler une simple expression :

```
from cProfile import run
run("10**10**6")
```

qui affiche

```
         3 function calls in 0.173 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.173    0.173    0.173    0.173 <string>:1(<module>)
        1    0.000    0.000    0.173    0.173 {built-in method builtins.exec}
        1    0.000    0.000    0.000    0.000 {method 'disable' of
                                                '_lsprof.Profiler' objects}
```

A des fins pédagogiques, je prendrai parfois des exemples utilisant la fonction `sleep(t)` qui suspend le processeur une durée de `t` secondes. Ainsi :

```
1  from cProfile import run
2  from time import sleep
3
4  def A(n):
5      sleep(4)
6      for i in range(n):
7          B()
8
9  def B():
10     sleep(1)
```

```
11
12 run("A(3)")
```

qui affiche

```
1          11 function calls in 7.007 seconds
2
3    Ordered by: standard name
4
5    ncalls  tottime  percall  cumtime  percall filename:lineno(function)
6         1    0.000    0.000    7.007    7.007 <string>:1(<module>)
7         1    0.000    0.000    7.007    7.007 c.py:4(A)
8         3    0.000    0.000    3.003    1.001 c.py:9(B)
9         1    0.000    0.000    7.007    7.007 {built-in method builtins.exec}
10        4    7.007    1.752    7.007    1.752 {built-in method time.sleep}
11        1    0.000    0.000    0.000    0.000 {method 'disable' of
12                                                '_lsprof.Profiler' objects}
```

La fonction `B` met 1 seconde à s'exécuter et la fonction `A` a une instruction d'attente de 4 secondes. Donc la durée d'exécution de `A` est $4 + n$ secondes. Par exemple, cf. code-source ligne 12, si $n = 3$, le code s'exécute en environ 7 secondes, ce qu'on lit bien dans le compte-rendu statistique, ligne 1 et aussi ligne 7, colonne *cumtime*.

Placer les résultats dans un fichier

Plutôt que d'afficher les résultats d'un profilage sur la sortie standard, on peut par exemple les placer dans un fichier-texte. Pour cela, il faut utiliser le module standard Stats. L'opération n'est, selon moi, pas très intuitive. Voici un exemple :

```
1  from cProfile import run
2  from time import sleep
3
4  def A(n):
5      sleep(4)
6      for i in range(n):
7          B()
8
9  def B():
10     sleep(1)
11
12
13 filename="my_results"
14 run("A(3)", filename)
15
16 from pstats import Stats
17 p = Stats(filename, stream=open("results.txt", "w"))
18
19 p.print_stats()
```

qui écrit le fichier `"results.txt"` de contenu

```
Tue Jul  7 19:45:28 2020    my_results

         11 function calls in 7.007 seconds

   Random listing order was used

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        4    7.007    1.752    7.007    1.752 {built-in method time.sleep}
        1    0.000    0.000    7.007    7.007 {built-in method builtins.exec}
        1    0.000    0.000    0.000    0.000 {method 'disable' of
                                                '_lsprof.Profiler' objects}
        1    0.000    0.000    7.007    7.007 <string>:1(<module>)
        1    0.000    0.000    7.007    7.007 a.py:4(A)
        3    0.000    0.000    3.003    1.001 a.py:9(B)
```

Il ne semble pas que l'on puisse avec cProfile seul placer les résultats dans un fichier (autre que la sortie standard, ce qui est possible avec la méthode `Profile.print_stats`).

Si on ne souhaite pas travailler avec un fichier, on peut aussi simuler un fichier par une chaîne en utilisant le module standard `io.StringIO`.

Trier les résultats

Un des intérêts d'utiliser le module Stats est qu'il peut trier les résultats fournis par cProfile. Voici un exemple :

```
1  from cProfile import run
2  from time import sleep
3
4  def A(n):
5      sleep(4)
6      for i in range(n):
7          B()
8
9  def B():
10     sleep(1)
11
12
13 filename="my_results"
14 run("A(3)", filename)
15
16 from pstats import Stats
17 p = Stats(filename, stream=open("results.txt", "w"))
18 p.sort_stats('cumulative')
19 p.print_stats()
```

qui affiche :

```
Tue Jul  7 21:11:15 2020    my_results
```

```
         11 function calls in 7.007 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    7.007    7.007 {built-in method builtins.exec}
        1    0.000    0.000    7.007    7.007 <string>:1(<module>)
        1    0.000    0.000    7.007    7.007 a.py:5(A)
        4    7.007    1.752    7.007    1.752 {built-in method time.sleep}
        3    0.000    0.000    3.003    1.001 a.py:10(B)
        1    0.000    0.000    0.000    0.000 {method 'disable'
                                               of '_lsprof.Profiler' objects}
```

Comme on peut le voir ligne 18 du fichier-source, les résultats sont affichés suivant la colonne `cumtime`. Regarder dans la doc les mots-clés à utiliser pour le critère de tri.

Filtrer les noms de fonctions appelées

L'affichage fournit par cProfile peut-être très long (des centaines de lignes) et comporter de nombreuses fonctions extérieures au fichier, en particulier des appels pour faire des importations, ce qui peut rendre les résultats assez illisibles. Heureusement, il est possible, en plus de trier les résultats, de les filtrer, par rang par exemple mais aussi suivant une expression régulière. Voici un exemple pour le cas d'une expression régulière :

```
1  from cProfile import run
2  from time import sleep
3
4  def A(n):
5      sleep(4)
6      for i in range(n):
7          B()
8
9  def B():
10     sleep(1)
11     B1()
12     B22()
13
14 def B1():
15     sleep(1)
16
17 def B2():
18     sleep(1)
19
20 def B22():
21     sleep(1)
22
23
24 filename="my_results"
```

```
25 run("A(3)", filename)
26
27 from pstats import Stats
28 p = Stats(filename, stream=open("results.txt", "w"))
29 p.sort_stats('cumulative')
30 p.print_stats("B\d")
```

qui affiche

```
Tue Jul  7 21:38:50 2020    my_results

         23 function calls in 13.014 seconds

   Ordered by: cumulative time
   List reduced from 8 to 2 due to restriction <'B\\d'>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        3    0.000    0.000    3.003    1.001 a.py:14(B1)
        3    0.000    0.000    3.003    1.001 a.py:20(B22)
```

Les résultats ont été triés suivant `cumtime`. Mais surtout, ne sont affichés que les lignes qui contiennent un nom comme B1 ou B2, cf. ligne 30 du code source où l'expression régulière `"B\d"` représente un `B` majuscule suivi d'un seul chiffre. Donc dans la sortie, ne sont gardées que les lignes qui matchent l'expression régulière, ce qui a pour effet de sélectionner les fonctions du code source `B1`, `B2` tout en ignorant `B22`, `B` ou `A`. On voit que l'affichage en est très allégé.

Utiliser cProfile dans des feuilles Jupyter Notebook

L'environnement IPython (par exemple sous Jupyter Notebook) dispose nativement d'une commande, dite « magique » native, qui permet de profiler le code appelé par une expression ou un bloc de code. Son nom `prun`, semble signifier `profiler run`. Il y a deux façons d'utiliser le profileur selon qu'on place :

— `%prun` en début de ligne pour profiler une expression

— `%%prun` en début de cellule pour profiler tout le code de la cellule.

Voici un exemple du premier type :

```
1 def A(n):
2     str(10**(n))
3     B(n)
4     B(n)
5
6 def B(n):
7     str(10**(n))
8
9 %prun A(288500)
```

qui affiche :

```
6 function calls in 3.003 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        2    2.003    1.001    2.003    1.001 <ipython-input-2-626773c851c8>:6(B)
        1    1.001    1.001    3.003    3.003 <ipython-input-2-626773c851c8>:1(A)
        1    0.000    0.000    3.003    3.003 {built-in method builtins.exec}
        1    0.000    0.000    3.003    3.003 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 {method 'disable' of
                                                '_lsprof.Profiler' objects}
```

Les options

Les options recouvrent plus ou moins ce que peut faire cProfile hors des feuilles Jupyter Notebook, cf. la doc de la commande magique. Par exemple, on peut filtrer l'affichage et/ou le trier. Voici un exemple, en mode cellule :

```
%%prun -s cumulative -l B|A

from time import sleep

def A(n):
    sleep(2)
    for i in range(n):
        B()

def B():
    sleep(0.5)
    B1()
    B22()

def B1():
    sleep(0.5)

def B2():
    sleep(0.5)

def B22():
    sleep(0.5)

A(3)
```

qui affiche

```
25 function calls in 6.507 seconds

   Ordered by: cumulative time
   List reduced from 10 to 4 due to restriction <'B|A'>
```

```
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    6.507    6.507 <string>:3(A)
        3    0.000    0.000    4.505    1.502 <string>:8(B)
        3    0.000    0.000    1.502    0.501 <string>:13(B1)
        3    0.000    0.000    1.502    0.501 <string>:19(B22)
```

Utilisation de cProfile en tant que script

On peut invoquer cProfile en tant que script dans une ligne de commande. Par exemple soit le programme Python suivant :

```python
from time import sleep

def A(n):
    sleep(4)
    for i in range(n):
        B()

def B():
    sleep(1)

A(3)
```

On va le profiler directement depuis la ligne de commande. Pour cela, on lance :

```
python3 -m cProfile script_cprofile.py
```

ce qui affiche (avec une légère remise en forme) :

```
         11 function calls in 7.007 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    7.007    7.007 script_cprofile.py:1(<module>)
        1    0.000    0.000    7.007    7.007 script_cprofile.py:3(A)
        3    0.000    0.000    3.003    1.001 script_cprofile.py:8(B)
        1    0.000    0.000    0.000    0.000 {method 'disable' of
                                                '_lsprof.Profiler' objects}
        1    0.000    0.000    0.000    0.000 {range}
        4    7.007    1.752    7.007    1.752 {time.sleep}
```

Il est possible de passer des options, cf. la documentation. Par exemple, pour trier suivant la colonne *cumtime* :

```
python3 -m cProfile -s cumulative script_cprofile.py
```

qui affiche :

```
         13 function calls in 7.007 seconds
```

```
   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    7.007    7.007 {built-in method builtins.exec}
        1    0.000    0.000    7.007    7.007 script_cprofile.py:1(<module>)
        1    0.000    0.000    7.007    7.007 script_cprofile.py:3(A)
        4    7.007    1.752    7.007    1.752 {built-in method time.sleep}
        3    0.000    0.000    3.003    1.001 script_cprofile.py:8(B)
        1    0.000    0.000    0.000    0.000 {method 'disable' of
                                               '_lsprof.Profiler' objects}
        1    0.000    0.000    0.000    0.000 <frozen importlib._bootstrap>:
                                               997(_handle_fromlist)
        1    0.000    0.000    0.000    0.000 {built-in method builtins.hasattr}
```

Utilisation de cProfile plus élaborée

Voici un exemple d'utilisation plus élaborée de cProfile. On dispose d'une fonction `factor(n)` qui renvoie la liste triée des facteurs premiers d'un entier `n` (ligne 4 ci-dessous). Pour voir comment se distribue le code dans cette fonction, on la découpe en 5 petites fonctions (ce qui oblige donc à récrire du code et ce qui peut créer des erreurs) dont on veut déterminer la distribution d'exécution. Voici le code source :

```
1  from math import factorial
2  from cProfile import run
3
4  def factor(n):
5      L=[]
6      N=n
7      while n%2==0:
8          n//=2
9      if N!=n:
10         L.append(2)
11
12     k=3
13     while True:
14         if n%k !=0:
15             k+=2
16             if k*k>n:
17                 break
18         else:
19             L.append(k)
20             n//=k
21     if n!=1:
22         L.append(n)
23
24     return sorted(set(L))
25
26
```

```
27
28 def factor_splitted(n):
29     L=[]
30     n=power2(n, L)
31     n=loop(n, L)
32     if n!=1:
33         L.append(n)
34     S=sorted(set(L))
35     return S
36
37
38 def loop(n, L):
39     k=3
40     ok=True
41     while ok:
42         if n%k !=0:
43             k, ok=branch_true(n, k)
44         else:
45             n=branch_false(n, k, L)
46
47     return n
48
49 def branch_true(n, k):
50     k+=2
51     return k, k*k<=n
52
53 def branch_false(n, k, L):
54     L.append(k)
55     n//=k
56     return n
57
58 def power2(n, L):
59     N=n
60     while n%2==0:
61         n//=2
62     if N!=n:
63         L.append(2)
64     return n
65
66
67 def data():
68     return factorial(10000)
69
70 def main():
71     n=data()
72     P=factor_splitted(n)
73     print((P)==factor(n))
74
75
```

```
76 filename="my_results"
77 run("main()", filename)
78
79 from pstats import Stats
80 p = Stats(filename, stream=open("results.txt", "w"))
81 p.sort_stats("ncalls", 'cumulative', )
82 p.print_stats("factor|branch|power2|loop|data")
```

qui affiche (après léger reformatage) :

```
Fri Jul 10 17:34:08 2020    my_results

         70967 function calls in 4.354 seconds

Ordered by: call count, cumulative time
List reduced from 15 to 8 due to restriction <'factor|branch|power2|loop|data'>

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
21989    0.639    0.000    0.640    0.000 test_cprofile_plus.py:53(branch_false)
4983     0.001    0.000    0.001    0.000 test_cprofile_plus.py:49(branch_true)
1     0.000    0.000    2.179    2.179 test_cprofile_plus.py:28(factor_splitted)
1     2.172    2.172    2.173    2.173 test_cprofile_plus.py:4(factor)
1     0.664    0.664    1.304    1.304 test_cprofile_plus.py:38(loop)
1     0.874    0.874    0.874    0.874 test_cprofile_plus.py:58(power2)
1     0.000    0.000    0.002    0.002 test_cprofile_plus.py:67(data)
1     0.002    0.002    0.002    0.002 {built-in method math.factorial}
```

Les statistiques sont affichées dans le fichier texte `results.txt` (cf. ligne 80 du code source). Les résultats sont triés d'abord par nombre d'appels puis par durées cumulées (cf. lignes 81). Les fonctions dont les appels sont profilés sont filtrées grâce à l'expression régulière de la ligne 82 pour appeler uniquement les fonctions des découpages. On notera que le filtrage est perfectible, il a affiché la fonction `factorial` dernière ligne des statistiques alors que ce n'était pas souhaité.

En examinant les statistiques en vue d'améliorer les performances de la fonction `factor`, on voit que la fonction `branch_true`, bien qu'appelée près de 5000 fois n'a aucun coût. En revanche, la fonction `branch_false` a un coût assez important et est appelée très souvent. La fonction `power2` bien qu'appelée une seule fois, a elle aussi un coût important (qui ne dépend que d'elle et pas de sous-appels).

Profiler partiellement

Il est possible de ne profiler qu'une partie du code-source, en encadrant la portion de code avec les méthodes `enable` et `disable`. Voici un exemple :

```
1 from time import sleep
2 import cProfile
3
4 pr = cProfile.Profile()
5
6 def A(n):
```

```
7       sleep(4)
8       for i in range(n):
9           B()
10
11  pr.enable()
12
13  def B():
14      sleep(1)
15
16  A(2)
17  pr.disable()
18
19  B()
20
21  pr.print_stats()
```

qui affiche (après léger reformatage) :

```
    7 function calls in 6.006 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
2    0.000    0.000    2.002    1.001 test_cprofile.py:14(B)
1    0.000    0.000    6.006    6.006 test_cprofile.py:7(A)
3    6.006    2.002    6.006    2.002 {built-in method time.sleep}
1    0.000    0.000    0.000    0.000 {method 'disable' of
                                       '_lsprof.Profiler' objects}
```

Dans le code-source, seul l'appel ligne 16 est profilé, l'appel ligne 19 est ignoré, comme on le voit puisque l'appel `A(2)` engendre 2 appels de `B` alors que sur la totalité du code-source, `B` est exécutée 3 fois. La ligne `pr.enable()` aurait pu être placée ligne 5 (ou encore à l'endroit le plus logique, ligne 15) sans que cela change quoi que ce soit au profilage effectué.

Plutôt que d'encadrer le code à exécuter, il est possible, depuis Python 3.8, d'utiliser la syntaxe `with` en appelant le gestionnaire de contexte `Profile` :

```
1  from time import sleep
2  from cProfile import Profile
3
4  def A(n):
5      sleep(4)
6      for i in range(n):
7          B()
8
9  def B():
10     sleep(1)
11
12 with Profile() as pr:
13     A(2)
14
```

```
15 B()
16
17 pr.print_stats()
```

qui affiche (après léger reformatage), grâce à la méthode `print_stats` de la ligne 17 :

```
    8 function calls in 6.006 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
1    0.000    0.000    0.000    0.000 cProfile.py:133(__exit__)
1    0.000    0.000    6.006    6.006 test_cprofile_with.py:4(A)
2    0.000    0.000    2.002    1.001 test_cprofile_with.py:9(B)
3    6.006    2.002    6.006    2.002 {built-in method time.sleep}
1    0.000    0.000    0.000    0.000 {method 'disable' of
                                       '_lsprof.Profiler' objects}
```

profile vs cProfile

Les modules profiles et cProfile sont deux modules de la bibliothèque standard de Python. Ils ont la même interface et ce sont tous les deux des profileurs déterministes. La grosse différence est que profile est écrit en pur Python tandis que cProfile est une extension Python écrite en C. Il s'ensuit que la pénalisation d'exécution est beaucoup plus faible pour cProfile.

À cause des incertitudes sur les débuts et fin de chronométrages, le module profile est particulièrement sensible aux accumulations d'erreurs. C'est pour cela que profile dispose d'un mécanisme de calibration. La documentation montre sur un exemple comment on peut calibrer profile.

Outre les problèmes de performances, il semble donc que profile soit moins facile à utiliser que cProfile. La documentation indique qu'il faut plutôt utiliser profile si on veut étendre le profileur (sans plus de précision).

Par ailleurs, par défaut, profile utilise comme timer `process_time`. Donc, profile ne compte pas les durées d'appels à la fonction `sleep` :

```
1 from profile import run
2 from time import sleep
3
4 def A():
5     sleep(2)
6
7 run("A()")
```

qui affiche

```
         6 function calls in 0.002 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.000    0.000 :0(exec)
```

```
        1    0.001    0.001    0.001    0.001 :0(setprofile)
        1    0.000    0.000    0.000    0.000 :0(sleep)
        1    0.000    0.000    0.000    0.000 <string>:1(<module>)
        1    0.000    0.000    0.002    0.002 profile:0(A())
        0    0.000             0.000          profile:0(profiler)
        1    0.000    0.000    0.000    0.000 profile_sleep.py:4(A)
```

le temps d'exécution est presque de 0 seconde.

cProfile et multiprocessing

Profilons avec cProfile un code utilisant le module multiprocessing :

```python
from multiprocessing import Pool
from random import randrange
from cProfile import run

def split_matrix(A):
    n=len(A)
    lines=[[A[i][j] for j in range(n)] for i in range(n)]
    cols=[[A[i][j] for i in range(n)] for j in range(n)]
    return lines, cols

def linXcol(t):
    (i, j), lines, cols=t
    L=lines[i]
    C=cols[j]
    return sum(a*b for (a,b) in zip(L,C))

def product(A, nproc):
    for _ in range(5):
        n=len(A)
        lines, cols=split_matrix(A)
        L=[((i,j), lines, cols) for i in range(n) for j in range(n)]

    pool=Pool(nproc)
    P=pool.map(linXcol, L)
    return P

def go():
    n=400
    A=[[randrange(2) for _ in range(n)] for _ in range(n)]
    P=product(A, 4)

def profile_prod():

    filename="prod"
    run("go()", filename)

```

```
37      from pstats import Stats
38      p = Stats(filename, stream=open("prod_matrix.txt", "w"))
39      p.sort_stats("ncalls", 'cumulative', )
40      p.print_stats()
41
42  profile_prod()
```

Le multiprocessing est utilisé à la ligne 24 : la fonction `linXcol` est exécutée sur différents processus. C'est l'appel de la fonction go ligne 27 qui est profilée.

Le résultat est un fichier de 300 lignes, contenant de nombreux appels à des fonctions ou des méthodes de la bibliothèque standard (dont des fonctions, méthodes ou modules privés). Voici le début et la fin des statistiques affichées (formatage modifié) :

```
Wed Jul  8 15:02:25 2020    prod

         808997 function calls (808871 primitive calls) in 3.104 seconds

Ordered by: call count, cumulative time

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
320294    0.026    0.000    0.026    0.000 {method 'getrandbits' of '_random.
                                             Random' objects}
160032    0.069    0.000    0.102    0.000 /usr/lib/python3.6/random.py:223
                                             (_randbelow)
160032    0.007    0.000    0.007    0.000 {method 'bit_length' of 'int' objects}
160000    0.052    0.000    0.153    0.000 /usr/lib/python3.6/random.py:173
                                             (randrange)
    2000    0.039    0.000    0.039    0.000 prod_matrix_profile.py:10(<listcomp>)
    2000    0.034    0.000    0.034    0.000 prod_matrix_profile.py:9(<listcomp>)

    ...


1    0.000    0.000    0.000    0.000 /usr/lib/python3.6/tempfile.py:600
                                       (_TemporaryFileWrapper)
1    0.000    0.000    0.000    0.000 /usr/lib/python3.6/shutil.py:58
                                        (SameFileError)
1    0.000    0.000    0.000    0.000 /usr/lib/python3.6/shutil.py:61
                                       (SpecialFileError)
1    0.000    0.000    0.000    0.000 /usr/lib/python3.6/shutil.py:65(ExecError)
1    0.000    0.000    0.000    0.000 /usr/lib/python3.6/shutil.py:68(ReadError)
1    0.000    0.000    0.000    0.000 /usr/lib/python3.6/shutil.py:71(RegistryError)
1    0.000    0.000    0.000    0.000 /usr/lib/python3.6/multiprocessing/
                                       synchronize.py:123(Semaphore)
1    0.000    0.000    0.000    0.000 /usr/lib/python3.6/multiprocessing/pool.py:53
                                       (RemoteTraceback)
```

Le point le plus important à signaler est la fonction exécutée en parallèle, `linXcol`, **n'est pas appelée**. Autrement dit, il semble que ne soit profilé que ce qui est exécuté dans le processus principal. Cela semble être cohérent avec ce qu'on peut lire dans le code source de cProfile selon

lequel le timer par défaut utilisé est `process_time` et qui ne chronomètre que l'exécution du processus où il est appelé.

Le profileur py-spy

On va examiner le profileur py-spy qui présente certains avantages par rapport à cProfile. Il faut d'abord l'installer :

```
pip3 install py-spy
```

L'utilisation est différente de cProfile : en effet, il n'y a aucune importation de py-spy à faire dans le code Python. En fait py-spy est un programme « externe » qui va se charger lui-même d'exécuter le programme Python et d'en faire une analyse statistique. Faisons-le fonctionner sur le code Python suivant :

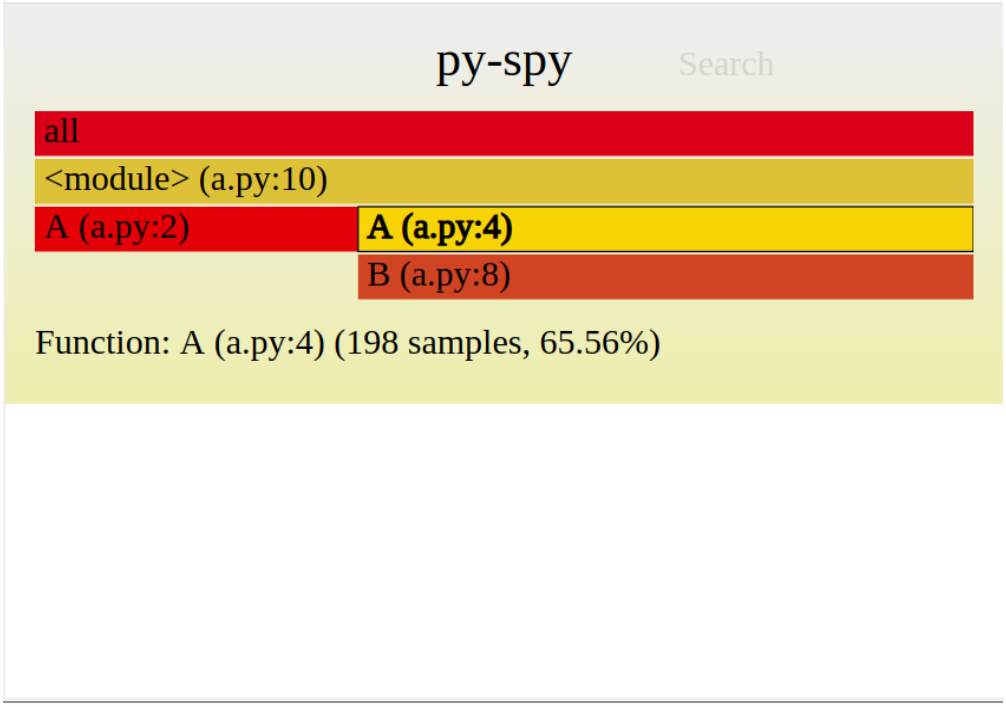
a.py

```
1  def A(n):
2      str(10**(n))
3      for i in range(2):
4          B(n)
5
6
7  def B(n):
8      str(10**(n))
9
10 A(288500)
```

Pour lancer py-spy, on se place en ligne de commande :

```
$ py-spy record -o my_diagramm.svg -- python3 a.py
```

qui produit un diagramme au format svg (visualisable dans un navigateur), réactif au passage de la souris :



On peut obtenir l'équivalent mais en mode texte « animé » avec la ligne de commande suivante :

```
py-spy top -- python3 a.py
```

qui va être actif pendant l'exécution (comme le programme top sous Linux) puis se figer à la fin de l'exécution et qui affichera :

```
1  GIL: 100.00%, Active: 100.00%, Threads: 1
2
3    %Own   %Total  OwnTime  TotalTime  Function (filename:line)
4  100.00% 100.00%    1.90s     1.90s   B (a.py:8)
5    0.00%   0.00%    1.10s     1.10s   A (a.py:2)
6    0.00% 100.00%   0.000s     3.00s   <module> (a.py:10)
7    0.00% 100.00%   0.000s     1.90s   A (a.py:4)
8
9
10
11
12
13
14 Press Control-C to quit, or ? for help.
15
16 process 14090 ended
```

L'information n'est pas présentée comme pour les statistiques de cProfile. Pourtant, elles sont pertinentes :

— ligne 7 des statistiques : l'appel (ligne 4 de `a.py`) de la fonction `B` coûte 1.9 s, précisément à la ligne où est le coût (ligne 8 de `a.py`, ligne XX des statistiques).

— Ligne 6 des statistiques : l'appel de A (ligne 10 du code-source, ligne 6 des statistiques) coûte 3s qui correspondent à $1.1 + 1.9$.

— La partie coûteuse de la fonction `A` (hors sous-appels) est identifiée par **son numéro de ligne** dans le code source (ligne 2 de `a.py`, ligne 5 des statistiques)

Autre exemple

On dispose d'une fonction `factor(n)` qui renvoie la liste des facteurs premiers d'une entier `n` (ligne 2 ci-dessous) et on lui demande de factoriser un nombre factorisable très grand (factorielle de 1000). Voici le code :

`factor.py`

```
1  from math import factorial
2
3  def factor(n):
4      L=[]
5      N=n
6      while n%2==0:
7          n//=2
8      if N!=n:
9          L.append(2)
10
11     k=3
12     while True:
13         if n%k !=0:
14             k+=2
15             if k*k>n:
16                 break
17         else:
18             L.append(k)
19             n//=k
20     if n!=1:
21         L.append(n)
22
23     return sorted(set(L))
24
25
26 def data():
27     return factorial(10000)
28
29 n=data()
30 P=factor(n)
```

exécuté via :

```
$ py-spy top -- python3 a.py
```

et qui finit par afficher :

```
1 Collecting samples from 'python3 factor.py' (python v3.6.9)
2 Total Samples 200
```

```
GIL: 100.00%, Active: 100.00%, Threads: 1

  %Own   %Total  OwnTime  TotalTime  Function (filename:line)
 53.00%  53.00%   0.600s    0.600s   factor (factor.py:13)
  0.00%   0.00%   0.510s    0.510s   factor (factor.py:6)
 47.00%  47.00%   0.510s    0.510s   factor (factor.py:19)
  0.00%   0.00%   0.380s    0.380s   factor (factor.py:7)
  0.00% 100.00%   0.000s     2.00s   <module> (factor.py:30)
```

Dans un premier temps, c'est assez difficile à lire. Mais les résultats sont finalement assez cohérents. Le programme identifie bien des endroits qui s'avèreront être effectivement des points chauds (lire la colonne *TotalTime*) que sont les trois divisions effectuées

- division par 2
  - lignes 6, 7 de factor.py
  - lignes 7 et 9 des statistiques
- division par k
  - lignes 13, 19 de factor.py
  - lignes 6 et 8 des statistiques

Remarques

- py-spy est écrit en Rust
- py-spy est capable de profiler des processus, pas seulement des programmes Python
- py-spy est un profileur statistique et non déterministe comme l'est cProfile, et donc il n'entraîne qu'une pénalisation faible de performance, en particulier, il peut tourner en production avec le programme qu'il profile.

py-spy et multiprocessing

py-spy est capable de profiler des programmes qui s'exécute en parallèle sur plusieurs processus. Testons-le avec le programme de produit matriciel ci-dessous :

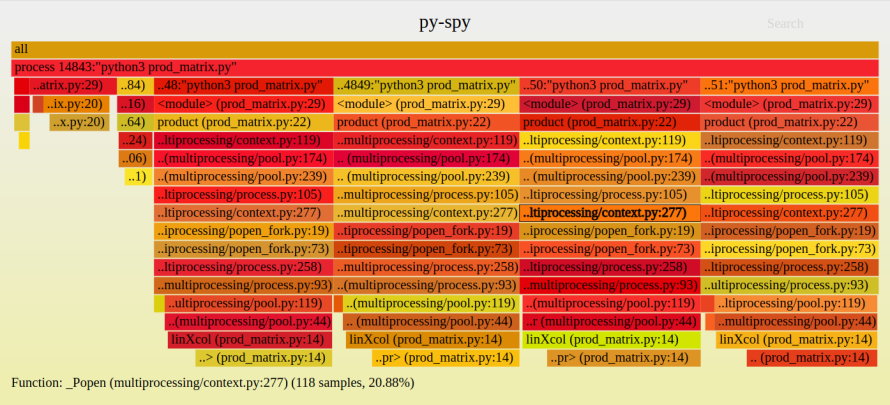
```python
from multiprocessing import Pool
from random import randrange

def split(A):
    n=len(A)
    lines=[[A[i][j] for j in range(n)] for i in range(n)]
    cols=[[A[i][j] for i in range(n)] for j in range(n)]
    return lines, cols

def linXcol(t):
    (i, j), lines, cols=t
    L=lines[i]
    C=cols[j]
    return sum(a*b for (a,b) in zip(L,C))

```

```python
def product(A, nproc):
    for _ in range(5):
        n=len(A)
        lines, cols=split(A)
        L=[((i,j), lines, cols) for i in range(n) for j in range(n)]

    pool=Pool(nproc)
    P=pool.map(linXcol, L)
    return P
    return [P[k*n:(k+1)*n] for k in range(n)]

n=400
A=[[randrange(2) for _ in range(n)] for _ in range(n)]
P=product(A, 4)
```

qu'on exécute via :

```
py-spy record -s -o matrix.svg -- python3 prod_matrix.py
```

(noter l'option `-s` pour *subprocess*) et qui affiche :



Sur le graphique, les 4 processus apparaissent clairement.

Profiler ligne par ligne

L'outil `line_profiler` permet de profiler un code Python ligne par ligne. Il faut d'abord l'installer :

```
pip3 install line_profiler
```

Son utilisation n'est pas très intuitive. Il faut d'abord savoir que l'installation du package `line_profiler` a créé un exécutable du nom de `kernprof`.

Maintenant, partons d'un fichier-source disons `exemple.py` :

exemple.py

```
def f(n):
    a=n*10**5
    x=10**a
    s=str(x)
    N=len(s)
    return N==a+1

print(f(4))
```

dont veut mesurer la durée d'exécution de chaque ligne du corps d'une de ses fonctions, ici `f`, suite à un appel, ici `f(4)`, présent dans le fichier. Pour cela, on **modifie** le source `exemple.py` en plaçant le décorateur `profile` au-dessus de la fonction, d'où le nouveau code

exemple_profile.py

```
@profile
def f(n):
    a=n*10**5
    x=10**a
    s=str(x)
    N=len(s)
    return N==a+1

print(f(4))
```

Une fois cette modification faite, dans un terminal système, on exécute la commande ci-dessous :

```
kernprof -l -v exemple_profile.py
```

L'option `-l` signifie **ligne par ligne** et l'option `-v` signifie que l'on pourra immédiatement **voir** dans la console les résultats du profilage. Après exécution, on pourrait lire :

```
True
Wrote profile results to exemple_profile.py.lprof
Timer unit: 1e-06 s

Total time: 1.92435 s
File: exemple_profile.py
Function: f at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           @profile
     2                                           def f(n):
     3         1          1.0      1.0      0.0      a=n*10**5
     4         1      38990.0  38990.0      2.0      x=10**a
     5         1    1885361.0 1885361.0     98.0       s=str(x)
     6         1          2.0      2.0      0.0      N=len(s)
     7         1          1.0      1.0      0.0      return N==a+1
```

L'exécution de la commande aura créé un fichier `example.py.lprof` à partir duquel les résultats

affichés sont extraits. Les résultats ci-dessus montrent les statistiques de l'exécution de chaque ligne, et de toute évidence, l'évaluation est cohérente.

Remarques

— `line_profiler` ne semble plus véritablement mis-à-jour depuis 2017, il y a toutefois une certaine maintenance et des adaptations pour assurer l'utilisation pour les nouvelles versions de Python, comme indiqué sur le repo Github.
— `line_profiler` est écrit en Cython.
— `line_profiler` est couramment utilisé dans IPython et les Jupyter Notebook
— une évolution intéressante de `line_profiler` semble être pprofile mais qui est écrit en pur Python.

Profiler par ligne dans une feuille Jupyter Notebook

Rappelons que les feuilles Jupyter Notebook disposent de commandes magiques. Grâce à une telle commande magique, on peut utiliser `line_profiler` dans Jupyter Notebook. Il faut que `line_profiler` soit installé. La commande magique s'appelle `%lprun` ce qui abrège *line profiler run*.

On commence par charger depuis une cellule le module `line_profiler` :

```
%load_ext line_profiler
```

Dans la cellule suivante, on écrit le code de la fonction à profiler par ligne, sans le décorateur `profile` (contrairement à la version Python hors Jupyter Notebook). On peut alors utiliser directement la commande magique `%lprun` (ligne 13 ci-dessous) :

```
1  def calc(n):
2      a=n*10**5
3      x=10**a
4      s=str(x)
5      N=len(s)
6      return N==a+1
7
8  def go(N):
9      from random import randint
10     n = randint(1,N)
11     return calc(n)
12
13 %lprun -f calc go(5)
```

ce qui affiche

```
Timer unit: 1e-06 s

Total time: 1.08239 s
File: <ipython-input-2-16ca1d6dc5c6>
Function: calc at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
```

```
============================================================
     1                                           def calc(n):
     2         1          1.0      1.0      0.0      a=n*10**5
     3         1      28273.0  28273.0      2.6      x=10**a
     4         1    1054111.0 1054111.0     97.4       s=str(x)
     5         1          2.0      2.0      0.0      N=len(s)
     6         1          1.0      1.0      0.0      return N==a+1
```

L'affichage montre le temps d'exécution ligne par ligne (colonne *Time*). L'option `f` de la commande doit être suivie du nom de la fonction dont on veut voir l'affichage des temps d'exécution de chaque ligne. La fin de la commande magique se termine par une expression à exécuter, ici `go(5)` mais on aurait pu imaginer `calc(4)`.

Exemple d'exploitation du profilage ligne par ligne

Soit la fonction `factor` suivante, qui donne la liste des facteurs premiers distincts d'un entier `n` :

```
def factor(n):
    L=[]
    N=n
    while n%2==0:
        n//=2
    if N!=n:
        L.append(2)

    k=3
    while True:
        if n%k !=0:
            k+=2
            if k*k>n:
                break
        else:
            L.append(k)
            n//=k
    if n!=1:
        L.append(n)

    return sorted(set(L))

print(factor(42*10*17*43))
```

qui affiche

```
[2, 3, 5, 7, 17, 43]
```

Pour ce qui suit, on utilise Jupyter Notebook et on aura besoin de la commande magique `%timeit` et aussi de `%lprun` de profilage par ligne. On veut tester `factor` sur de grandes valeurs de `n`, par exemple pour de grande factorielle comme $10000!$. Donc on fait un essai de mesure :

```
n=factorial(10000)
%timeit -r3 factor(n)
```

qui affiche

```
2.19 s ± 4.32 ms per loop (mean ± std. dev. of 3 runs, 1 loop each)
```

L'instruction est répétée seulement trois fois (cf. `%r3`) au lieu des 7 par défaut. Ensuite, on profile par ligne :

```
%load_ext line_profiler
```

puis

```
%lprun -f factor factor(n)
```

qui affiche :

```
Timer unit: 1e-06 s

Total time: 2.29126 s
File: <ipython-input-1-e9ee41573b8e>
Function: factor at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def factor(n):
     2         1          2.0      2.0      0.0      L=[]
     3         1          1.0      1.0      0.0      N=n
     4      9996     460990.0     46.1     20.1      while n%2==0:
     5      9995     460165.0     46.0     20.1          n//=2
     6         1          1.0      1.0      0.0      if N!=n:
     7         1          2.0      2.0      0.0          L.append(2)
     8
     9         1          1.0      1.0      0.0      k=3
    10         1          1.0      1.0      0.0      while True:
    11     26972     692133.0     25.7     30.2          if n%k !=0:
    12      4983       1442.0      0.3      0.1              k+=2
    13      4983       1667.0      0.3      0.1              if k*k>n:
    14         1          1.0      1.0      0.0                  break
    15                                               else:
    16     21989       7742.0      0.4      0.3              L.append(k)
    17     21989     666809.0     30.3     29.1              n//=k
    18         1          0.0      0.0      0.0      if n!=1:
    19         1          1.0      1.0      0.0          L.append(n)
    20
    21         1        299.0    299.0      0.0      return sorted(set(L))
```

Il est immédiat que ce qui est coûteux (cf. colonne *Time*), ce sont les opérations de divisions aux lignes 4, 5, 11 et 17 du code Python. Une optimisation évidente consiste à remarquer qu'il est inutile de calculer le quotient **ainsi que** le reste de `n` par `k`, car si on a seulement le quotient `q = n//k` alors on en déduit le reste `r = n - k * q` et qui est obtenu par des opérations moins

coûteuses. Donc on essaye cette optimisation pour les lignes 4 et 5, ce qui donne le nouveau code :

```
def factor1(n):
    L=[]
    N=n
    while True:
        q= n//2
        r=n-2*q
        if r==0:
            n=q
        else:
            break
    if N!=n:
        L.append(2)

    k=3
    while True:
        if n%k !=0:
            k+=2
            if k*k>n:
                break
        else:
            L.append(k)
            n//=k
    if n!=1:
        L.append(n)

    return sorted(set(L))
```

et on regarde si cela a un effet :

```
n=factorial(10000)
%timeit -r3 factor1(n)
```

ce qui affiche

```
1.84 s ± 3.09 ms per loop (mean ± std. dev. of 3 runs, 1 loop each)
```

et qui montre une amélioration non négligeable (15%). Donc on applique l'optimisation aux autres lignes :

```
def factor2(n):
    L=[]
    N=n
    while True:
        q= n//2
        r=n-2*q
        if r==0:
            n=q
        else:
            break
```

```
    if N!=n:
        L.append(2)

    k=3
    while True:
        q= n//k
        r=n-k*q
        if r !=0:
            k+=2
            if k*k>n:
                break
        else:
            L.append(k)
            n=q
    if n!=1:
        L.append(n)

    return sorted(set(L))
```

et on teste :

```
1 n=factorial(10000)
2 %timeit -r3 factor2(n)
```

ce qui affiche

```
1.31 s ± 7.69 ms per loop (mean ± std. dev. of 3 runs, 1 loop each)
```

et donc le temps d'exécution a été divisé par presque 2.

Une autre optimisation peut être envisagée. Comme le montre les lignes 4-5 du profilage, on effectue un grand nombre de division par 2. Or, pour faire une division par 2, on peut utiliser un opérateur bit-à-bit :

```
a=1000
d=3
print(a//(2**d))
print(a>>d)
```

qui souvent est optimisé, d'où le code :

```
def factor3(n):
    L=[]
    N=n
    while True:
        q= n>>1
        r=n-2*q
        if r==0:
            n=q
        else:
            break
    if N!=n:
```

```
        L.append(2)

    k=3
    while True:
        q= n//k
        r=n-k*q
        if r !=0:
            k+=2
            if k*k>n:
                break
        else:
            L.append(k)
            n=q
    if n!=1:
        L.append(n)

    return sorted(set(L))
```

ce qui donne en testant :

```
n=factorial(10000)
%timeit -r3 factor3(n)
```

```
3 850 ms ± 9.93 ms per loop (mean ± std. dev. of 3 runs, 1 loop each)
```

ce qui a divisé le temps initial par presque 2.6.

Il faudrait toutefois s'assurer que :

— la validité du code n'est pas entamée par les modifications successives

— le temps d'exécution n'est pas augmenté dans d'autres cas que celui qui a servi à créer les optimisations.