Qualité algorithmique

1 — Profilage

Anthony Labarre

Université Gustave Eiffel

Objectifs du cours

Détecter et résoudre les problèmes de performances dans le code.

I. Diagnostics

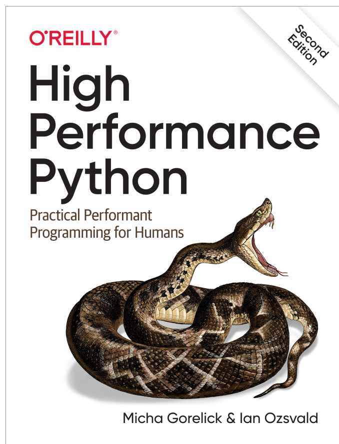
- maîtriser les outils d'évaluation du code : analyse du temps de calcul, consommation de la mémoire, ...
- interpréter les résultats pour repérer les problèmes de performances ;

II. Solutions

- techniques : notions avancées du langage (Python) ;
- généralistes : principes transposables à d'autres langages ;

Pour aller plus loin

Le cours se base notamment sur l'ouvrage suivant :



(attention aux erreurs : voir l'errata)

Prélude

Pour vous simplifier la vie, installez (outre bien sûr Python 3) :

- IPython, un interpréteur amélioré ;
- un environnement de développement (PyCharm, ...) ;
- `pip`, un gestionnaire de paquets pour Python (pour les installations sans droits admin) ;

On suppose un environnement GNU ; les raccourcis suivants signifient :

`paquet` = `sudo apt-get install paquet`

`paquet` = `pip install paquet`

Environnements virtuels

Il est possible que l’OS ne vous laisse pas exécuter `pip` :

```
$ pip install memray
error: externally-managed-environment

× This environment is externally managed
    To install Python packages system-wide, try apt install
    python3-xyz, where xyz is the package you are trying to
    install.

    If you wish to install a non-Debian-packaged Python package,
    create a virtual environment using python3 -m venv path/to/venv.
    Then use path/to/venv/bin/python and path/to/venv/bin/pip. Make
    sure you have python3-full installed.

    If you wish to install a non-Debian packaged Python application,
    it may be easiest to use pipx install xyz, which will manage a
    virtual environment for you. Make sure you have pipx installed.

    See /usr/share/doc/python3.11/README.venv for more information.

note: If you believe this is a mistake, please contact your Python installation or
↪  OS distribution provider. You can override this, at the risk of breaking your
↪  Python installation or OS, by passing --break-system-packages.
hint: See PEP 668 for the detailed specification.
```

On peut contourner cette interdiction en utilisant un **environnement virtuel.**

Environnements virtuels

Un **environnement virtuel** est une copie modifiable de l'environnement standard Python. Les changements dans cet environnement virtuel n'affectent que celui-ci.

Exemple (utilisation basique d'un environnement virtuel)

```
$ python3 -m venv .venv  # création de l'environnement virtuel
$ source .venv/bin/activate  # activation de l'environnement virtuel
(.venv) $ pip install line_profiler
Collecting line_profiler
  Downloading
  ↪  line_profiler-4.0.3-cp311-cp311-manylinux_2_17_x86_64.manylinux2014_x86_64.whl
  ↪  (698 kB)
Installing collected packages: line_profiler
Successfully installed line_profiler-4.0.3
(.venv) $ deactivate  # désactivation
$ # retour à l'environnement réel: line_profiler n'est plus accessible
```

Ils ont bien d'autres utilités qui sortent du cadre de ce cours (packaging, ...). Voir la documentation du module standard `venv`.

Précautions sur les environnements virtuels

Les seuls modules disponibles par défaut dans les environnements virtuels sont les modules standards !

- **`import`** échouera pour les modules tiers installés globalement (par exemple `python3-memory-profiler`) ;
- Il faudra les réinstaller dans l'environnement virtuel avec `pip` ;

`pip install -r requirements.txt` installe automatiquement les dépendances spécifiées.

Aspects pratiques

Organisation prévisionnelle :

- 3 séances de CM ;
- 3 séances de TP ;

Évaluation :

- un mini-projet ;

Ressources :

- `http://igm.univ-mlv.fr/~alabarre/teaching.php` ;
- et une page sur l'intranet ;

Plan d'aujourd'hui

1. Profilage : généralités
2. Profiler les performances
   - `time` et `timeit`
   - `cProfile`
   - `line_profiler`
3. Profiler la mémoire
   - `memray`
   - `memory_profiler`

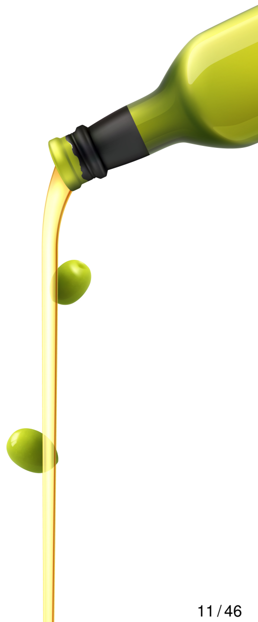
Profilage : généralités

Profilage

Le **profilage** consiste en l'évaluation en pratique du code. En particulier :

- mesurer la consommation en CPU ou en mémoire d'un programme (ou de ses morceaux) ;
- déterminer les bottlenecks, c'est-à-dire les parties les plus responsables des problèmes de performance ;

L'objectif est de rentabiliser son travail, en se concentrant sur les parties du code où l'optimisation aura le plus d'impacts. (image © macrovector / Freepik)



Mises en garde

N'optimisez jamais “au feeling” : profilez votre code pour savoir ce qui mérite votre attention !

Sans cela, vous risquez de perdre du temps à :

- réaliser des micro-optimisations ;
- optimiser des parties du code peu utilisées ;
- dégrader les performances ;

Une optimisation à la fois $\longrightarrow$ tests $\longrightarrow$ on recommence.

Mises en garde sur le profilage

Le profilage est une science expérimentale !

Il faut donc faire attention :

- aux variations : effectuez plusieurs fois le même test, faites des moyennes, variez les paramètres, ...
- aux interférences : fermez tous vos autres programmes, désactivez les options “douteuses” de l’OS ou du BIOS, ...

Si vous ne maîtrisez pas l’environnement, donnez des informations sur les conditions de test (avec les outils standards : `ps`, `free`, ...).

Exemple réel d'interférences

Lançons le même test sous des conditions différentes :

Conditions normales

```
f_1:   8.03
f_2:   6.76
f_3:   5.80
```

Pendant une réunion discord

Fonctions	Essai 1	Essai 2	Essai 3
f_1	32.83	13.71	9.41
f_2	10.85	26.17	16.33
f_3	12.16	16.60	20.57

$\Rightarrow$ Ce n'est donc pas un problème de durée : les benchmarks ne sont plus du tout fiables car les résultats sont trop variables !

Profiler les performances

`time` **et** **`timeit`**

Chronométrage

- La manière la plus basique d'évaluer les performances du code est de le chronométrer ;
- On préférera le module `timeit` au module `time` :
  - il calcule des temps moyens d'exécution (échantillon spécifiable) ;
  - il désactive le garbage collector, ce qui réduit les interférences sur le temps d'exécution ;
  - il sélectionne automatiquement le chronomètre le plus précis selon l'OS ;
  - utilisable dans un programme, dans l'interpréteur, ou en ligne de commande ;

Chronométrage : applications

- On a déjà vu comment s'y prendre avec le module `time` ;
- Les cas typiques d'utilisations seront les suivants :
  - comparer plusieurs fonctions pour la même tâche (cf. TP) ;
  - savoir rapidement quelle est la meilleure option quand on hésite entre plusieurs manières de procéder ;

`time` et `timeit` sont très faciles à utiliser grâce aux "commandes magiques" de `ipython`.

On utilisera (pas besoin d'**`import`**) :

- `%time` ou `%timeit` pour les instructions d'une ligne ;
- `%%time` ou `%%timeit` pour les instructions multilignes (commencez par un retour à la ligne) ;

Utiliser `timeit` dans un programme

La fonction `timeit` admet plusieurs **paramètres nommés** (le nom importe, mais pas l'ordre) :

Exemple (utilisation basique de `timeit`)

```
>>> from timeit import timeit
>>> timeit(stmt='"-".join(str(n) for n in range(100))')  # le code à exécuter
[résultat]
```

```
>>> from timeit import timeit
>>> timeit(
...     stmt='"-".join(str(n) for n in range(100))',  # le code à exécuter
...     number=100  # le nombre de répétitions
... )
[résultat]
```

```
>>> from timeit import timeit
>>> timeit(
...     stmt='"-".join(str(n) for n in range(taille))',  # le code à exécuter
...     number=100,  # le nombre de répétitions
...     setup="taille = 42;"  # l'initialisation
... )
[résultat]
```

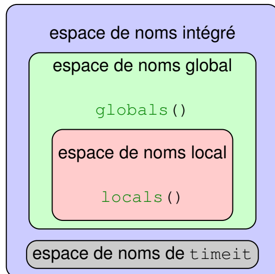
Attention, `number=1000000` par défaut, et ne s'adapte automatiquement qu'en ligne de commande ou en commande magique.

Interactions avec le programme actuel

Par défaut, `timeit` ne “voit” pas le reste de votre programme :

```
>>> from timeit import timeit
>>> def ma_fonction():
...     print("Bonjour!")
...
>>> timeit(stmt="ma_fonction()")
...
NameError: name 'ma_fonction' is not defined
```

- Nos variables et celles de `timeit` évoluent dans des **espaces de noms** différents ;
- On peut communiquer nos variables à `timeit` de deux façons :
  1. à l’aide du paramètre `setup` ;
  2. à l’aide du paramètre `globals` ;



Le paramètre `setup` de `timeit`

On peut communiquer des noms globaux (variables ou fonctions) à `timeit` via son paramètre `setup` :

```
>>> timeit(stmt="ma_fonction()", setup="from __main__ import ma_fonction")
[résultat]
```

Inconvénients :

1. peu ergonomique : on doit explicitement fournir tous les noms à importer ;
2. ne fonctionne pas pour les noms locaux ;

Le paramètre `globals` de `timeit` — `globals()`

- La fonction intégrée `globals()` renvoie le dictionnaire des noms globaux ;
- Il suffit de le passer au paramètre `globals` de `timeit` ;

Exemple

```
>>> from timeit import timeit
>>> def ma_fonction():
...     print("Bonjour!")
...
>>> timeit(stmt="ma_fonction()", globals=globals())
[résultat]
```

Le paramètre `globals` de `timeit` — `locals()`

Attention : **seuls** les noms **globaux** sont transmis par l'appel `globals()` !

Si l'on veut communiquer des noms locaux, on aura besoin de la fonction intégrée `locals()`, qui renvoie le dictionnaire des noms locaux.

Exemple

```
>>> from timeit import timeit
>>> def mon_super_algo(donnees):
...     # ... du code ...
...
>>> for instance in ...:
...     print(timeit(
...         stmt="mon_super_algo(instance)",
...         setup="from __main__ import mon_super_algo",
...         globals=locals()
...     ))
...
[résultat]
```

Le paramètre `globals` de `timeit`

Et s'il nous faut les deux ? On passe en paramètre le dictionnaire contenant l'union de `globals()` et de `locals()`.

Exemple

```
>>> timeit(
...     stmt="...",  # code à exécuter
...     globals=dict(globals(), **locals()),
...     # les autres paramètres
... )
[résultat]
```

La syntaxe étrange avec les `**` sera démystifiée au prochain cours.

Chronométrage : applications

- `time` et `timeit` conviennent bien pour mesurer des morceaux spécifiques de code ;
- Ils ne permettent pas de **détecter** facilement les endroits problématiques ;
- Pour cet usage, on utilisera le module `cProfile` ;

`cProfile`

Le module `cProfile`

- `cProfile` est un module standard mesurant le temps passé dans chaque fonction, ainsi que leur nombre d'appels ;
- Son utilisation en ligne de commande est très simple :
  `python3 -m cProfile mon_programme.py`
- On peut trier la table des résultats de manière décroissante selon divers critères ;

Examinons un exemple.

`cProfile` : tri des résultats (option `-s [mode]`)

Différents critères de tri nous intéresseront selon les cas :

- `cumtime` : le temps d'exécution “cumulé” ;
  → comment réduire le temps passé dans cette fonction ?
- `ncalls` : le nombre d'appels d'une fonction ;
  → comment réduire le nombre d'appels à cette fonction ?
- `percall` : le coût d'un appel (`cumtime / ncalls`) ;
  → la fonction est-elle vraiment si peu performante ?
  (pas possible en ligne de commande, on utilisera `snakeviz`)
- ...

`cProfile` : aller plus loin avec `pstats`

- On peut pousser les analyses plus loin en enregistrant les résultats dans un fichier :
  `python3 -m cProfile -o prog.stats prog.py`
- Puis en les chargeant à l'aide du module standard `pstats` :

```python
import pstats
p = pstats.Stats("prog.stats")
# p.sort_stats("cumtime")  # ou ncalls, etc.
```

- On peut ensuite afficher :
  - les stats habituelles (`p.print_stats()`);
  - qui appelle chaque fonction (`p.print_callees()`);
  - ce que chaque fonction appelle (`p.print_callers()`);
- Intérêt : tous les appels n'ont pas le même coût !

Options de `print_stats()`

`print_stats` permet de filtrer les résultats (cf. `grep`) et / ou de n'afficher qu'un certain pourcentage des lignes.

Exemple (filtrer les appels aux méthodes de `str`)

```
>>> p.print_stats("'str' objects")
...
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   459035    0.092    0.000    0.092    0.000 {method 'format' of 'str' objects}
      342    0.000    0.000    0.000    0.000 {method 'rstrip' of 'str' objects}
      195    0.000    0.000    0.000    0.000 {method 'join' of 'str' objects}
...
```

Exemple (n'afficher qu'un centième des lignes)

```
>>> p.print_stats(.01)
...
   List reduced from 425 to 4 due to restriction <0.01>
...
```

Les deux options sont compatibles, mais non commutatives.

Filtres multiples

Exemple (afficher un “top 10”)

```
>>> p.sort_stats("ncalls").print_stats(10)
...
   Ordered by: call count
   List reduced from 167 to 10 due to restriction <10>
...
```

On ne peut préciser qu’un seul filtre à la fois. Mais comme `print_stats` renvoie un `Stats`, on peut répéter les appels avec des filtres différents.

Exemple (filtrer les appels à `len` et `ord`)

```
>>> p.print_stats("len").print_stats("ord")
...
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
17132615/17132601    0.111    0.000    0.111    0.000 {built-in function len}
...
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        6    0.000    0.000    0.000    0.000 {built-in function ord}
...
```

Tri et affichage

- `Stats.sort_stats(cle_1, cle_2, ...)` admet un nombre arbitraire de critères de tri : on trie d'abord sur `cle_1`, puis sur `cle_2`, etc. ;
- `Stats.reverse_order()` renverse l'ordre (pratique pour voir rapidement les problèmes) ;

- Malheureusement, on ne peut pas “ignorer” des modules entiers ;

- `snakeviz` permet de visualiser les rapports générés par l'analyse de `cProfile` ;
- Utilisation : `python3 -m snakeviz rapport.stats`
- Ceci ouvre un navigateur permettant de visualiser les résultats et de les trier selon les statistiques habituelles ;

Examinons un exemple.

`line_profiler`

python3-line-profiler line_profiler

- line_profiler porte bien son nom : il permet d'évaluer chaque ligne de code d'un appel de fonction ;
- Cela permet de pousser plus loin l'analyse fournie par cProfile ;
- Il peut aussi identifier des problèmes que cProfile peut rater, puisque ce dernier ne s'intéresse qu'aux appels de fonctions et aux compréhensions ;
- Son utilisation est un peu plus complexe que cProfile et nécessite de modifier légèrement votre code ;

- Utilisation :
  1. rajoutez une ligne `@profile` précédant la définition de chaque fonction à analyser du module ;
  2. exécutez `kernprof -lv mon_module.py` ;
- Remarque évidente : vous devez avoir prévu des appels aux fonctions à analyser . . .

(On reviendra plus tard sur le **décorateur** `@profile`)

Examinons un exemple (scrabble).

Profiler la mémoire

Mesurer la consommation en mémoire

- On a déjà utilisé `sys.getsizeof`, avec les problèmes que l'on sait (itérables) ;
- Cela reste utile de manière ponctuelle ; en complément, on utilisera :
  1. `memray` comme "`cProfile` pour la mémoire" ;
  2. `memory_profiler` comme "`line_profiler` pour la mémoire" ;

`memray`

- `memray` est un équivalent de `cProfile` pour la mémoire ;
- Son utilisation en ligne de commande est très simple :
  `python3 -m memray run mon_programme.py` ou
  `python3 -m memray run --live mon_programme.py`
- Ceci génère un rapport au format `.bin`, qu'on peut ensuite visualiser dans le terminal ou en HTML ;
- Si l'on veut plus de précision, on peut utiliser `memory_profiler` ;

(voir la documentation pour plus d'infos)

Visualiser les résultats de `memray`

- Une fois le rapport généré (par exemple `resultats.bin`), on peut le visualiser de différentes manières ;
- On lance
  `python3 -m memray [mode] resultats.bin`
  où `mode` est :
  - `stats` : histogramme et top 5 des allocations les plus gourmandes et les plus nombreuses ;
  - `summary` : aperçu au moment où la consommation était la plus élevée ;
  - `tree` : vue hiérarchique des 10 plus grosses allocations ;
  - …
- Il y a plein d'autres modes, et plein d'options pour chaque mode ;

memory_profiler

python3-memory-profiler [python3-psutil]
memory_profiler [psutil]

- memory_profiler évalue chaque ligne de code d'un appel de fonction du point de vue de la mémoire ;
- Malheureusement, il ralentit énormément le code (et sans psutil c'est encore pire) ;
- Utilisation :
  1. rajoutez @profile devant la définition de chaque fonction à analyser du module ;
  2. exécutez python3 -m memory_profiler mon_module.py ;

Examinons un exemple.

Interpréter les résultats de `memory_profiler`

L'analyse produit le résultat suivant :

```
Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
     3   38.816 MiB   38.816 MiB           1   @profile
     4                                         def my_func():
     5   46.492 MiB    7.676 MiB           1       a = [1] * (10 ** 6)
     6  199.117 MiB  152.625 MiB           1       b = [2] * (2 * 10 ** 7)
     7   46.629 MiB -152.488 MiB           1       del b
     8   46.629 MiB    0.000 MiB           1       return a
```

1. `Line` : le numéro de la ligne analysée ;
2. `Mem usage` : la mémoire consommée **après** exécution de la ligne actuelle ;
3. `Increment` : la différence en consommation par rapport à la ligne précédente ;
4. `Line contents` : le code.

(source : documentation de `memory_profiler`)

memory_profiler dans IPython

- memory_profiler propose également une commande magique **%memit** pour IPython ;
- Comme memory_profiler n'est pas un module standard, il faudra explicitement dire à IPython de charger cette commande lorsque vous démarrez une session :

```
In [1]: %load_ext memory_profiler
In [2]: %memit
peak memory: 75.87 MiB, increment: 0.05 MiB
```

- On l'utilisera dans la partie "Solutions" ;

Un mot sur `del`

- En théorie, l'instruction `del x` efface la variable `x` ;
- En pratique, `del x` signale simplement à Python que `x` n'existe plus ;
- C'est le garbage collector qui décide quand libérer réellement la mémoire ;
- Ceci explique que les gains ne soient pas toujours visibles immédiatement avec `memory_profiler` ou `%memit` ;