

Qualité algorithmique

2 — Optimisation des performances

Anthony Labarre

Université Gustave Eiffel



Plan d'aujourd'hui

- 1 La compilation et les types
- 2 Mémorisation des résultats
- 3 Les décorateurs
 - Fonctions imbriquées
 - L'opérateur *
 - Fonctions à nombre variable d'arguments
- 4 La sérialisation
- 5 Le parallélisme

La compilation et les types

La lenteur de Python

Python est réputé “lent” pour plusieurs raisons :

- langage interprété : à chaque exécution, chaque instruction doit être analysée, interprétée, puis seulement exécutée ;

La lenteur de Python

Python est réputé “lent” pour plusieurs raisons :

- langage interprété : à chaque exécution, chaque instruction doit être analysée, interprétée, puis seulement exécutée ;
- machine virtuelle : le programme ne “parle” pas directement à la machine ;

La lenteur de Python

Python est réputé “lent” pour plusieurs raisons :

- langage interprété : à chaque exécution, chaque instruction doit être analysée, interprétée, puis seulement exécutée ;
- machine virtuelle : le programme ne “parle” pas directement à la machine ;
- langage typé dynamiquement :

La lenteur de Python

Python est réputé “lent” pour plusieurs raisons :

- langage interprété : à chaque exécution, chaque instruction doit être analysée, interprétée, puis seulement exécutée ;
- machine virtuelle : le programme ne “parle” pas directement à la machine ;
- langage typé dynamiquement :
 - toutes les vérifications de type doivent se faire à l’exécution ;

La lenteur de Python

Python est réputé “lent” pour plusieurs raisons :

- langage interprété : à chaque exécution, chaque instruction doit être analysée, interprétée, puis seulement exécutée ;
- machine virtuelle : le programme ne “parle” pas directement à la machine ;
- langage typé dynamiquement :
 - toutes les vérifications de type doivent se faire à l’exécution ;
 - le code ne peut donc pas être optimisé à l’avance ;

Solutions pénibles

Diverses solutions existent :

... qu'on va courageusement éviter d'utiliser car elles nous compliqueraient grandement la vie.

(mais allez voir <https://compilers.pydata.org/> si vous êtes curieux)

Solutions pénibles

Diverses solutions existent :

- compiler le code : Cython, Numba, ...

... qu'on va courageusement éviter d'utiliser car elles nous compliqueraient grandement la vie.

(mais allez voir <https://compilers.pydata.org/> si vous êtes curieux)

Solutions pénibles

Diverses solutions existent :

- compiler le code : Cython, Numba, ...
- ajouter des annotations de type (via `cdef` en Cython) ;

... qu'on va courageusement éviter d'utiliser car elles nous compliqueraient grandement la vie.

(mais allez voir <https://compilers.pydata.org/> si vous êtes curieux)

Solutions pénibles

Diverses solutions existent :

- compiler le code : Cython, Numba, ...
- ajouter des annotations de type (via `cdef` en Cython) ;
- importer du code provenant d'autres langages (C, Fortran, ...) : `ctypes`, `cffi`, ...;

... qu'on va courageusement éviter d'utiliser car elles nous compliqueraient grandement la vie.

(mais allez voir <https://compilers.pydata.org/> si vous êtes curieux)

Solutions pénibles

Diverses solutions existent :

- compiler le code : Cython, Numba, ...
- ajouter des annotations de type (via `cdef` en Cython) ;
- importer du code provenant d'autres langages (C, Fortran, ...): `ctypes`, `cffi`, ...;
- écrire directement le code en CPython (\neq Cython) ;

... qu'on va courageusement éviter d'utiliser car elles nous compliqueraient grandement la vie.

(mais allez voir <https://compilers.pydata.org/> si vous êtes curieux)



- Une des manières les plus rentables d'accélérer son code est d'utiliser l'interpréteur PyPy ;
- Si tout se passe bien, on n'a rien à modifier dans notre code : il suffit d'écrire `pypy3` au lieu de `python3` à l'exécution ;



Examinons un exemple.

```
cayley.py --operation prefix-reversal 9
```

PyPy est-il magique ?

- Deux approches de compilation permettent d'améliorer les performances :
 - ① AOT (pour ahead of time) : comme en C ;
 - ② JIT (pour just in time) : au fur et à mesure des besoins ;
- PyPy effectue de la compilation JIT ;
- Les gains en performance sont corrélés avec la proportion de code pouvant en bénéficier ;

Inconvénients de PyPy

- PyPy a toujours un train de retard sur la dernière version de Python ;
- Il peut donc y avoir des incompatibilités, pas toujours rattrapables ;
- Dans certains cas (rares), PyPy peut être moins performant que Python ;
- ... mais vu l'investissement, ça vaut le coup d'essayer ;

(plus d'infos sur les performances :

<https://www.pypy.org/performance.html>)

Et dans d'autres langages ?

- Certains autres langages interprétés proposent peut-être des interpréteurs améliorés équivalents ;
- Cherchez “JIT”, “compilation”, ...
- Si vous utilisez un langage compilé, examinez les options de votre compilateur ;
 - par exemple pour `gcc` : l'option `-O` permet de définir le niveau d'optimisation ;

Mémorisation des résultats

- Rappels sur les compromis performances / mémoire : on peut accélérer les calculs en stockant des résultats intermédiaires ;

- Rappels sur les compromis performances / mémoire : on peut accélérer les calculs en stockant des résultats intermédiaires ;
- On a vu que les dictionnaires pouvaient nous éviter facilement ces calculs ;

- Rappels sur les compromis performances / mémoire : on peut accélérer les calculs en stockant des résultats intermédiaires ;
- On a vu que les dictionnaires pouvaient nous éviter facilement ces calculs ;
- En Python, c'est extrêmement simple à implémenter : il suffit d'utiliser le **décorateur** `lru_cache` du module standard `functools` ;

Utilisation de `lru_cache`

- `lru_cache` construit un dictionnaire `{paramètres : résultat}`, pour la fonction décorée, qui est automatiquement consulté à chaque appel ;

Utilisation de `lru_cache`

- `lru_cache` construit un dictionnaire `{paramètres : résultat}`, pour la fonction décorée, qui est automatiquement consulté à chaque appel ;
- De notre point de vue, on peut considérer ce dictionnaire comme global (même si c'est faux) ;

Utilisation de `lru_cache`

- `lru_cache` construit un dictionnaire `{paramètres : résultat}`, pour la fonction décorée, qui est automatiquement consulté à chaque appel ;
- De notre point de vue, on peut considérer ce dictionnaire comme global (même si c'est faux) ;
- On limite (ou non) sa taille avec le paramètre `maxsize` ;

Utilisation de `lru_cache`

- `lru_cache` construit un dictionnaire {paramètres : résultat}, pour la fonction décorée, qui est automatiquement consulté à chaque appel ;
- De notre point de vue, on peut considérer ce dictionnaire comme global (même si c'est faux) ;
- On limite (ou non) sa taille avec le paramètre `maxsize` ;
- Utilisation :

```
# limite de taille par défaut  
@lru_cache  
def ma_fonction(...):  
    ...
```

```
# taille illimitée  
@lru_cache(maxsize=None)  
def ma_fonction(...):  
    ...
```

lru_cache en action

Mesurons le temps mis pour calculer F_{35} récursivement (visualisez les problèmes ici : <https://recursion.vercel.app/>) :

```
In [1]: def fib(n):  
...:     return n if n < 2 else fib(n-1) + fib(n-2)  
...:  
In [2]: %time fib(35)  
CPU times: user 1.47 s, sys: 202 µs, total: 1.47 s  
Wall time: 1.47 s  
Out[2]: 9227465
```

lru_cache en action

Mesurons le temps mis pour calculer F_{35} récursivement (visualisez les problèmes ici : <https://recursion.vercel.app/>) :

```
In [1]: def fib(n):
...:     return n if n < 2 else fib(n-1) + fib(n-2)
...:
In [2]: %time fib(35)
CPU times: user 1.47 s, sys: 202 µs, total: 1.47 s
Wall time: 1.47 s
Out[2]: 9227465
```

... et après décoration :

```
In [3]: from functools import lru_cache
In [4]: @lru_cache(maxsize=None)
...: def fib(n):
...:     return n if n < 2 else fib(n-1) + fib(n-2)
...:
In [5]: %time fib(35)
CPU times: user 59 µs, sys: 1 µs, total: 60 µs
Wall time: 64.6 µs
Out[5]: 9227465
```

On peut facilement vérifier la différence en termes de nombre d'appels avec `cProfile` :

```
$ python3 -m cProfile -s ncalls fibo.py # sans lru_cache
29860706 function calls (4 primitive calls) in 3.478 seconds

Ordered by: call count

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
29860703/1  3.478    0.000    3.478    3.478 fibo.py:1(fib)
...

$ python3 -m cProfile -s ncalls fibo.py # avec lru_cache
57 function calls (22 primitive calls) in 0.000 seconds

Ordered by: call count

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
36/1    0.000    0.000    0.000    0.000 fibo.py:3(fib)
...
```

Contourner les limitations de la récursivité

Python limite le nombre d'appels récursifs que l'on peut effectuer.

Contourner les limitations de la récursivité

Python limite le nombre d'appels récursifs que l'on peut effectuer.



Utilisez des appels “poubelle” pour remplir le dictionnaire de `lru_cache` !

Contourner les limitations de la récursivité

Python limite le nombre d'appels récursifs que l'on peut effectuer.



Utilisez des appels “poubelle” pour remplir le dictionnaire de `lru_cache` !

Exemple

Si calculer F_{1500} provoque une erreur, on peut effectuer un appel plus raisonnable juste pour réduire le nombre d'appels récursifs :

```
>>> fib(1500) # décorée par lru_cache
...
RecursionError: maximum recursion depth exceeded
>>> fib(1400) # appel "poubelle"
[le résultat s'affiche]
>>> fib(1500) # assez de valeurs stockées
[le résultat s'affiche]
```

Les décorateurs

- Les **décorateurs** sont des fonctions modifiant d'autres fonctions ;

Les décorateurs

- Les **décorateurs** sont des fonctions modifiant d'autres fonctions ;
- Lorsqu'on "décore" une fonction `f` à l'aide d'une fonction `deco`, la fonction `deco` exécute `f` ET d'autres lignes de code rajoutées ;

Les décorateurs

- Les **décorateurs** sont des fonctions modifiant d'autres fonctions ;
- Lorsqu'on "décore" une fonction `f` à l'aide d'une fonction `deco`, la fonction `deco` exécute `f` ET d'autres lignes de code rajoutées ;
- Intérêts :

- Les **décorateurs** sont des fonctions modifiant d'autres fonctions ;
- Lorsqu'on "décore" une fonction `f` à l'aide d'une fonction `deco`, la fonction `deco` exécute `f` ET d'autres lignes de code rajoutées ;
- Intérêts :
 - 1 modifier le comportement du code à moindre coût ;

Les décorateurs

- Les **décorateurs** sont des fonctions modifiant d'autres fonctions ;
- Lorsqu'on "décore" une fonction `f` à l'aide d'une fonction `deco`, la fonction `deco` exécute `f` ET d'autres lignes de code rajoutées ;
- Intérêts :
 - 1 modifier le comportement du code à moindre coût ;
 - 2 tester rapidement des changements avant de les appliquer ;

- Les **décorateurs** sont des fonctions modifiant d'autres fonctions ;
- Lorsqu'on "décore" une fonction `f` à l'aide d'une fonction `deco`, la fonction `deco` exécute `f` ET d'autres lignes de code rajoutées ;
- Intérêts :
 - 1 modifier le comportement du code à moindre coût ;
 - 2 tester rapidement des changements avant de les appliquer ;
 - 3 modifier le comportement de fonctions auxquelles on n'a pas accès en écriture ;

- Les **décorateurs** sont très simples à utiliser :

- Les **décorateurs** sont très simples à utiliser :
 - `@decorateur` devant la définition de la fonction à décorer ;

- Les **décorateurs** sont très simples à utiliser :
 - `@decorateur` devant la définition de la fonction à décorer ;
 - ou `@decorateur(arguments)` le cas échéant ;

Utilisation de décorateurs

- Les **décorateurs** sont très simples à utiliser :
 - `@decorateur` devant la définition de la fonction à décorer ;
 - ou `@decorateur(arguments)` le cas échéant ;
- On peut les cumuler :

```
@deco_1
@deco_2
def fonction(...):
    ...
```

applique **d'abord** deco_2, **puis** deco_1 ;

Écriture de décorateurs

Si l'utilisation des décorateurs est simple, leur écriture ...

```
from functools import wraps

def decorateur(f):
    @wraps(f)
    def decoration(*args, **kwargs):
        return f(*args, **kwargs)
    return decoration
```

... mérite quelques mots d'explication.



Écriture de décorateurs

Si l'utilisation des décorateurs est simple, leur écriture ...

```
from functools import wraps

def decorateur(f):
    @wraps(f)
    def decoration(*args, **kwargs):
        return f(*args, **kwargs)
    return decoration
```

... mérite quelques mots d'explication.

Intuitivement, le `decorateur` définit une nouvelle fonction de `decoration` qui va remplacer la fonction `f` lorsqu'on va la décorer ($\equiv f = \text{decorateur}(f)$).



Écriture de décorateurs

Si l'utilisation des décorateurs est simple, leur écriture ...

```
from functools import wraps

def decorateur(f):
    @wraps(f)
    def decoration(*args, **kwargs):
        return f(*args, **kwargs)
    return decoration
```

... mérite quelques mots d'explication.

Intuitivement, le `decorateur` définit une nouvelle fonction de `decoration` qui va remplacer la fonction `f` lorsqu'on va la décorer ($\equiv f = \text{decorateur}(f)$).

Examinons maintenant les détails techniques.



Fonctions imbriquées

```
from functools import wraps

def decorateur(f):
    @wraps(f)
    def decoration(*args, **kwargs):
        return f(*args, **kwargs)
    return decoration
```

- On peut définir des fonctions **dans** d'autres fonctions ;
- En général, on l'évite car les fonctions imbriquées ne sont pas accessibles ;
- Dans ce contexte, c'est justifié : la `decoration` n'a pas à être visible hors du `decorateur` ;

Fonctions imbriquées et récursivité

Cela permet aussi d'écrire des fonctions récursives plus efficaces :

Cela permet aussi d'écrire des fonctions récursives plus efficaces :

Version classique

```
def rech_rec(L, x, i):  
    if i < 0:  
        return  
    if L[i] == x:  
        return i  
    return rech_rec(L, x, i-1)  
  
def recherche(L, x):  
    return rech_rec(L, x, len(L)-1)
```

Fonctions imbriquées et récursivité

Cela permet aussi d'écrire des fonctions récursives plus efficaces :

Version classique

```
def rech_rec(L, x, i):  
    if i < 0:  
        return  
    if L[i] == x:  
        return i  
    return rech_rec(L, x, i-1)  
  
def recherche(L, x):  
    return rech_rec(L, x, len(L)-1)
```

Version améliorée

```
def recherche(L, x):  
    def rech_rec(i):  
        if i < 0:  
            return  
        if L[i] == x:  
            return i  
        return rech_rec(i-1)  
  
    return rech_rec(len(L)-1)
```

Fonctions imbriquées et récursivité

Cela permet aussi d'écrire des fonctions récursives plus efficaces :

Version classique

```
def rech_rec(L, x, i):  
    if i < 0:  
        return  
    if L[i] == x:  
        return i  
    return rech_rec(L, x, i-1)  
  
def recherche(L, x):  
    return rech_rec(L, x, len(L)-1)
```

Version améliorée

```
def recherche(L, x):  
    def rech_rec(i):  
        if i < 0:  
            return  
        if L[i] == x:  
            return i  
        return rech_rec(i-1)  
  
    return rech_rec(len(L)-1)
```

Dans la seconde version, les contextes empilés lors des appels récursifs ne comporteront plus que le paramètre `i`.

L'opérateur unaire de **déballage** * permet d'extraire le contenu d'un itérable en plusieurs morceaux.

```
from functools import wraps

def decorateur(f):
    @wraps(f)
    def decoration(*args, **kwargs):
        return f(*args, **kwargs)
    return decoration
```

L'opérateur unaire de déballage * permet d'extraire le contenu d'un itérable en plusieurs morceaux.

```
from functools import wraps

def decorateur(f):
    @wraps(f)
    def decoration(*args, **kwargs):
        return f(*args, **kwargs)
    return decoration
```

Exemple (utilisation de * *)

Comment extraire les éléments d'une liste dans des variables séparées ?

L'opérateur unaire de **déballage** * permet d'extraire le contenu d'un itérable en plusieurs morceaux.

```
from functools import wraps

def decorateur(f):
    @wraps(f)
    def decoration(*args, **kwargs):
        return f(*args, **kwargs)
    return decoration
```

Exemple (utilisation de * *)

Comment extraire les éléments d'une liste dans des variables séparées ?

```
✗ >>> premier, reste, dernier = list(range(10))
```

L'opérateur unaire de **déballage** * permet d'extraire le contenu d'un itérable en plusieurs morceaux.

```
from functools import wraps

def decorateur(f):
    @wraps(f)
    def decoration(*args, **kwargs):
        return f(*args, **kwargs)
    return decoration
```

Exemple (utilisation de * *)

Comment extraire les éléments d'une liste dans des variables séparées ?

✗ `>>> premier, reste, dernier = list(range(10))`

✓ `>>> premier, *reste, dernier = list(range(10))`

L'opérateur unaire de **déballage** * permet d'extraire le contenu d'un itérable en plusieurs morceaux.

```
from functools import wraps

def decorateur(f):
    @wraps(f)
    def decoration(*args, **kwargs):
        return f(*args, **kwargs)
    return decoration
```

Exemple (utilisation de *)

Comment extraire les éléments d'une liste dans des variables séparées ?

✗ `>>> premier, reste, dernier = list(range(10))`

✓ `>>> premier, *reste, dernier = list(range(10))`

```
>>> print(premier, dernier)
0 9
>>> print(reste)
[1, 2, 3, 4, 5, 6, 7, 8]
```

`reste` est une liste, indépendamment du type de la valeur à droite du `=`.

* et appels de fonctions

- * permet aussi de convertir un itérable en une liste de paramètres (donc des variables séparées par des virgules, et non une liste Python);
- Ceci permet des appels très concis;

Exemple (* et appels de fonctions)

`set.union` calcule l'union d'un nombre arbitraire d'ensembles.

* et appels de fonctions

- * permet aussi de convertir un itérable en une liste de paramètres (donc des variables séparées par des virgules, et non une liste Python) ;
- Ceci permet des appels très concis ;

Exemple (* et appels de fonctions)

`set.union` calcule l'union d'un nombre arbitraire d'ensembles.

```
>>> liste = [{i} for i in range(10)]
>>> set.union(liste[0], liste[1], liste[2], ...)
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

* et appels de fonctions

- * permet aussi de convertir un itérable en une liste de paramètres (donc des variables séparées par des virgules, et non une liste Python) ;
- Ceci permet des appels très concis ;

Exemple (* et appels de fonctions)

`set.union` calcule l'union d'un nombre arbitraire d'ensembles.

```
>>> liste = [{i} for i in range(10)]
>>> set.union(liste[0], liste[1], liste[2], ...)
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

```
>>> set.union(*liste)
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Fonctions à nombre variable d'arguments

```
from functools import wraps

def decorateur(f):
    @wraps(f)
    def decoration(*args, **kwargs):
        return f(*args, **kwargs)
    return decoration
```

- La syntaxe `*args` dans une définition de fonction spécifie un nombre variable (éventuellement nul) d'arguments ;
- `args` est un `tuple` de valeurs ;

Fonctions à nombre variable d'arguments

```
from functools import wraps

def decorateur(f):
    @wraps(f)
    def decoration(*args, **kwargs):
        return f(*args, **kwargs)
    return decoration
```

- La syntaxe `*args` dans une définition de fonction spécifie un nombre variable (éventuellement nul) d'arguments ;
- `args` est un `tuple` de valeurs ;

Exemple (nombre variable d'arguments)

```
>>> def fonction_1(*args):
...     print("J'ai reçu", len(args), "arguments")
...
>>> fonction_1()
J'ai reçu 0 arguments
>>> fonction_1(1, 2)
J'ai reçu 2 arguments
>>> fonction_1("bla", "blou", None)
J'ai reçu 3 arguments
```

Arguments nommés en nombre variable

```
from functools import wraps

def decorateur(f):
    @wraps(f)
    def decoration(*args, **kwargs):
        return f(*args, **kwargs)
    return decoration
```

- On peut nommer les paramètres, ce qui permet de les communiquer dans n'importe quel ordre ;

Arguments nommés en nombre variable

```
from functools import wraps

def decorateur(f):
    @wraps(f)
    def decoration(*args, **kwargs):
        return f(*args, **kwargs)
    return decoration
```

- On peut nommer les paramètres, ce qui permet de les communiquer dans n'importe quel ordre ;
- `**kwargs` dans la définition d'une fonction spécifie un dictionnaire arbitraire au format `{nom: valeur}`.

Arguments nommés en nombre variable

```
from functools import wraps

def decorateur(f):
    @wraps(f)
    def decoration(*args, **kwargs):
        return f(*args, **kwargs)
    return decoration
```

- On peut nommer les paramètres, ce qui permet de les communiquer dans n'importe quel ordre ;
- `**kwargs` dans la définition d'une fonction spécifie un dictionnaire arbitraire au format `{nom: valeur}`.

Exemple (arguments nommés en nombre variable)

```
>>> def fonction_2(**kwargs):
...     print("J'ai reçu", len(kwargs), "arguments")
...     for k, v in kwargs.items():
...         print("\tnom:", k, ", valeur:", v)
...
>>> fonction_2(taille=3, ens={1, 5}, temperature=37)
J'ai reçu 3 arguments
nom: taille , valeur: 3
nom: ens , valeur: {1, 5}
nom: temperature , valeur: 37
```

Arguments anonymes et nommés

- On peut mélanger les deux types d'arguments, en respectant le format suivant :
 - 1 d'abord tous les arguments "positionnels";
 - 2 ensuite tous les arguments nommés ;

Arguments anonymes et nommés

- On peut mélanger les deux types d'arguments, en respectant le format suivant :
 - 1 d'abord tous les arguments "positionnels";
 - 2 ensuite tous les arguments nommés ;
- Le format le plus général est donc :

```
def fonction(*args, **kwargs):  
    ...
```

qui accepte un nombre variable d'arguments anonymes suivi d'un nombre variable d'arguments nommés.

- C'est donc celui qu'on rencontrera le plus pour les décorateurs ;

Fonctionnement d'un décorateur

```
from functools import wraps

def decorateur(f):
    @wraps(f)
    def decoration(*args, **kwargs):
        return f(*args, **kwargs)
    return decoration
```

- Lorsqu'on décore une fonction, on la remplace par une nouvelle version décorée ;

Fonctionnement d'un décorateur

```
from functools import wraps

def decorateur(f):
    @wraps(f)
    def decoration(*args, **kwargs):
        return f(*args, **kwargs)
    return decoration
```

- Lorsqu'on décore une fonction, on la remplace par une nouvelle version décorée ;
- Autrement dit, écrire ceci :

```
@decorateur
def ma_fonction(...):
    ...
```

équivalent à écrire ceci :

```
ma_fonction = decorateur(ma_fonction)
```

Fonctionnement d'un décorateur

```
from functools import wraps

def decorateur(f):
    @wraps(f)
    def decoration(*args, **kwargs):
        return f(*args, **kwargs)
    return decoration
```

- Lorsqu'on décore une fonction, on la remplace par une nouvelle version décorée ;
- Autrement dit, écrire ceci :

```
@decorateur
def ma_fonction(...):
    ...
```

équivalent à écrire ceci :

```
ma_fonction = decorateur(ma_fonction)
```

- C'est pourquoi `decorateur` renvoie une fonction, et non le résultat d'un appel (sinon on ne pourrait pas appeler le résultat) ;

Le décorateur `wraps`

```
from functools import wraps

def decorateur(f):
    @wraps(f)
    def decoration(*args, **kwargs):
        return f(*args, **kwargs)
    return decoration
```

`@wraps` permet de rendre la décoration transparente :
l'utilisateur ne se rend plus compte que la fonction est décorée.

Le décorateur wraps

```
from functools import wraps

def decorateur(f):
    @wraps(f)
    def decoration(*args, **kwargs):
        return f(*args, **kwargs)
    return decoration
```

`@wraps` permet de rendre la décoration transparente :
l'utilisateur ne se rend plus compte que la fonction est décorée.

Exemple (avec et sans `@wraps`)

```
>>> @decorateur
... def ma_fonction():
...     """Je ne fais rien."""
...     pass
...
>>> ma_fonction.__name__, ma_fonction.__doc__ # sans @wraps
('decoration', None)
>>> ma_fonction.__name__, ma_fonction.__doc__ # avec @wraps
('ma_fonction', 'Je ne fais rien.')
```

Appliquer un décorateur à une fonction tierce

- Décorer une fonction que l'on définit soi-même est simple ;

Appliquer un décorateur à une fonction tierce

- Décorer une fonction que l'on définit soi-même est simple ;
- Mais comment décorer une fonction provenant d'un module tiers ?

Appliquer un décorateur à une fonction tierce

- Décorer une fonction que l'on définit soi-même est simple ;
- Mais comment décorer une fonction provenant d'un module tiers ?
 - `@deco` \n `def fonction(...)` n'est possible que si on peut écrire dans le fichier ;

Appliquer un décorateur à une fonction tierce

- Décorer une fonction que l'on définit soi-même est simple ;
- Mais comment décorer une fonction provenant d'un module tiers ?
 - `@deco` \n `def fonction(...)` n'est possible que si on peut écrire dans le fichier ;
 - `fonction = deco(fonction)` fonctionne ... sauf si `fonction` est récursive !

Appliquer un décorateur à une fonction tierce

- Décorer une fonction que l'on définit soi-même est simple ;
- Mais comment décorer une fonction provenant d'un module tiers ?
 - `@deco` \n `def fonction(...)` n'est possible que si on peut écrire dans le fichier ;
 - `fonction = deco(fonction)` fonctionne ... sauf si `fonction` est récursive !
- Il vaut mieux utiliser la fonction intégrée

```
setattr(objet, attribut, valeur)
```

qui permet de modifier un attribut d'un objet de manière dynamique.

Appliquer un décorateur à une fonction tierce

L'option la plus simple est d'importer le module, puis d'utiliser `setattr` pour appliquer le décorateur.

Exemple (décorer une fonction à la volée)

```
import fibo
from functools import lru_cache
setattr(fibo, "fib", lru_cache(fibo.fib))
...
```

Appliquer un décorateur à une fonction tierce

L'option la plus simple est d'importer le module, puis d'utiliser `setattr` pour appliquer le décorateur.

Exemple (décorer une fonction à la volée)

```
import fibo
from functools import lru_cache
setattr(fibo, "fib", lru_cache(fibo.fib))
...
```

Et on peut même décorer toutes les fonctions importées :

Exemple (décorer toutes les fonctions importées)

```
import mon_module
from functools import lru_cache

for nom in dir(mon_module):
    objet = getattr(mon_module, nom) # renvoie l'objet correspondant au nom donné
    if callable(objet):             # objet appellable -> fonction (ou classe)
        setattr(mon_module, nom, lru_cache(objet))
...
```

Appliquer un décorateur à une fonction tierce

- Si l'on ne veut pas importer tout le module, les choses se compliquent un peu ;

Appliquer un décorateur à une fonction tierce

- Si l'on ne veut pas importer tout le module, les choses se compliquent un peu ;
- `setattr(fibo, ...)` ne fonctionne pas puisqu'on n'a pas importé le module `fibo` ;

Appliquer un décorateur à une fonction tierce

- Si l'on ne veut pas importer tout le module, les choses se compliquent un peu ;
- `setattr(fibo, ...)` ne fonctionne pas puisqu'on n'a pas importé le module `fibo` ;
- On doit identifier **l'objet** correspondant au module qui contient `fib` ;

Appliquer un décorateur à une fonction tierce

- Si l'on ne veut pas importer tout le module, les choses se compliquent un peu ;
- `setattr(fibo, ...)` ne fonctionne pas puisqu'on n'a pas importé le module `fibo` ;
- On doit identifier **l'objet** correspondant au module qui contient `fib` ;
- Cela donne :

```
from fibo import fib
from functools import lru_cache
import sys
setattr(sys.modules[fib.__module__], "fib", lru_cache(fib))
...
```

Ajout de propriétés à une fonction



On peut rajouter des propriétés à une fonction.

- Une manière d'implémenter `lru_cache` est de rajouter un dictionnaire à la fonction décorée ;

Ajout de propriétés à une fonction



On peut rajouter des propriétés à une fonction.

- Une manière d'implémenter `lru_cache` est de rajouter un dictionnaire à la fonction décorée ;

Exemple

```
def lru_cache(f):  
    f.resultats = dict()  
    def decoration(*args, **kwargs):  
        # si args / kwargs appartient à f.resultats: renvoyer le résultat  
        # sinon, appeler f et mettre à jour f.resultats
```

Ajout de propriétés à une fonction



On peut rajouter des propriétés à une fonction.

- Une manière d'implémenter `lru_cache` est de rajouter un dictionnaire à la fonction décorée ;

Exemple

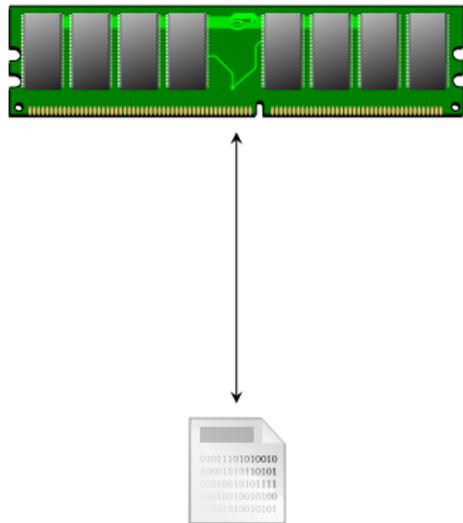
```
def lru_cache(f):  
    f.resultats = dict()  
    def decoration(*args, **kwargs):  
        # si args / kwargs appartient à f.resultats: renvoyer le résultat  
        # sinon, appeler f et mettre à jour f.resultats
```

- Et on peut faire cela avec n'importe quel ajout dont on aurait besoin lors d'appels différents ;

La sérialisation

Contexte

- Certains calculs coûteux doivent parfois être effectués juste pour une initialisation ;
- Même si on ne peut pas y échapper la première fois, on veut éviter de refaire le travail à chaque lancement du programme ;
- La **sérialisation** consiste à enregistrer un objet au format binaire sur le disque pour pouvoir le recharger en mémoire plus tard ;



Le module standard `pickle`

- Python propose la sérialisation à l'aide du module standard `pickle`;

Le module standard `pickle`

- Python propose la sérialisation à l'aide du module standard `pickle` ;
- Il est extrêmement simple à utiliser :

Le module standard `pickle`

- Python propose la sérialisation à l'aide du module standard `pickle` ;
- Il est extrêmement simple à utiliser :
 - 1 pour sauver un objet :

```
>>> import pickle
>>> objet = ...
>>> with open("mon_objet", "wb") as sortie:
...     pickle.dump(objet, sortie)
```

Le module standard `pickle`

- Python propose la sérialisation à l'aide du module standard `pickle` ;
- Il est extrêmement simple à utiliser :
 - 1 pour sauver un objet :

```
>>> import pickle
>>> objet = ...
>>> with open("mon_objet", "wb") as sortie:
...     pickle.dump(objet, sortie)
```

- 2 pour charger un objet :

```
>>> import pickle
>>> with open("mon_objet", "rb") as entree:
...     objet = pickle.load(entree)
```

Limitations de pickle

- La plupart des objets “standards” de Python sont sérialisables ;
- Ce n’est pas garanti pour les objets des modules tiers ou ceux que vous définissez ;
- Attention à la compatibilité, notamment avec PyPy (voir [les formats dans la documentation de pickle](#)) ;



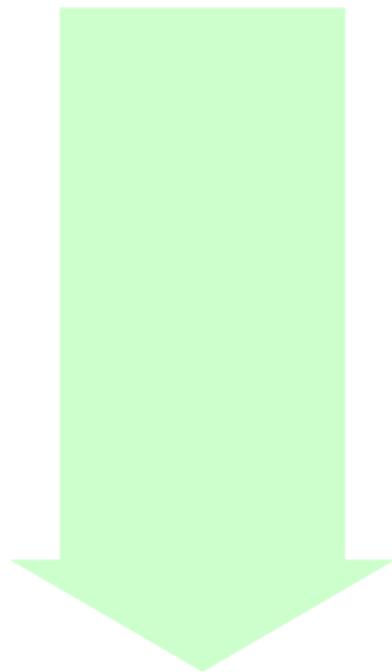
Consultez [la documentation de pickle](#) pour savoir ce qui n’est pas sérialisable et comment y remédier.

Le parallélisme

Threads et processus

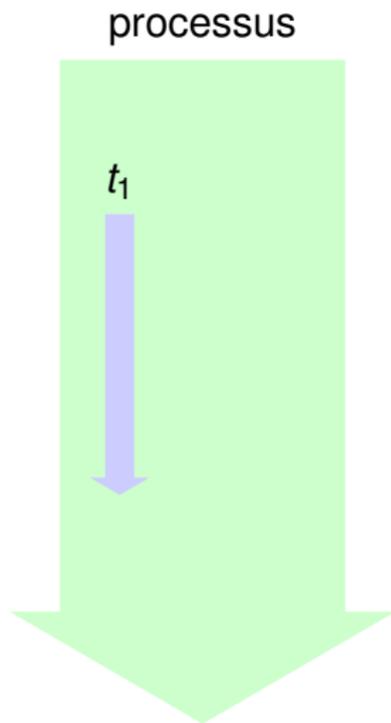
- Un **processus** est une instance d'un programme en cours d'exécution ;

processus



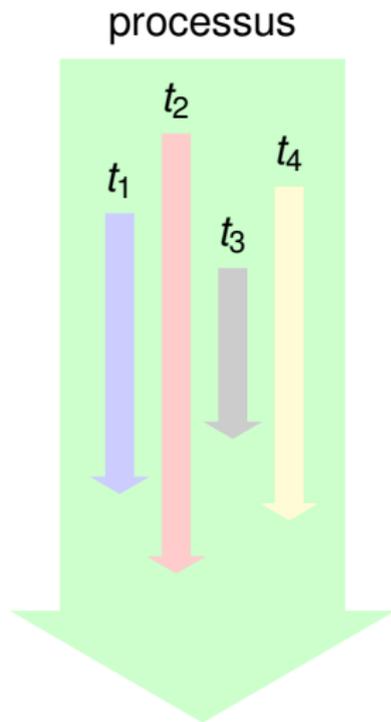
Threads et processus

- Un **processus** est une instance d'un programme en cours d'exécution ;
- Un **fil** (ou thread) peut être vu comme un sous-processus d'un processus donné ;



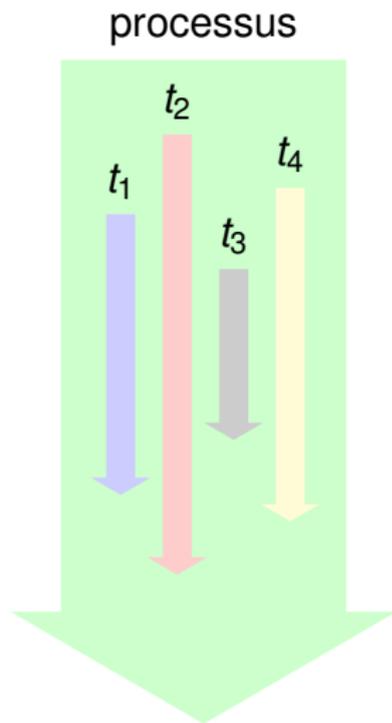
Threads et processus

- Un **processus** est une instance d'un programme en cours d'exécution ;
- Un **fil** (ou thread) peut être vu comme un sous-processus d'un processus donné ;
- Un processus peut comporter plusieurs threads, qui :



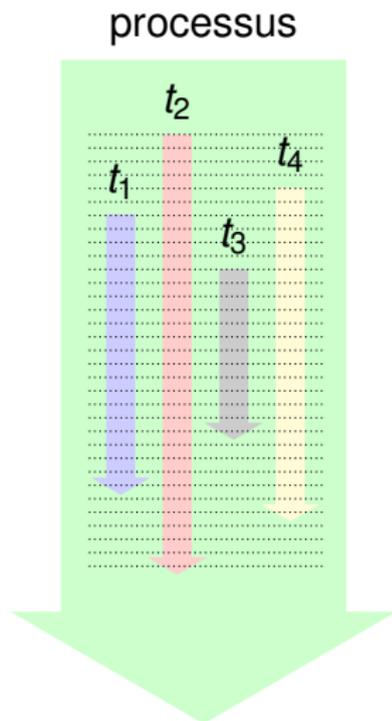
Threads et processus

- Un **processus** est une instance d'un programme en cours d'exécution ;
- Un **fil** (ou thread) peut être vu comme un sous-processus d'un processus donné ;
- Un processus peut comporter plusieurs threads, qui :
 - partagent le même espace mémoire et les mêmes ressources ;



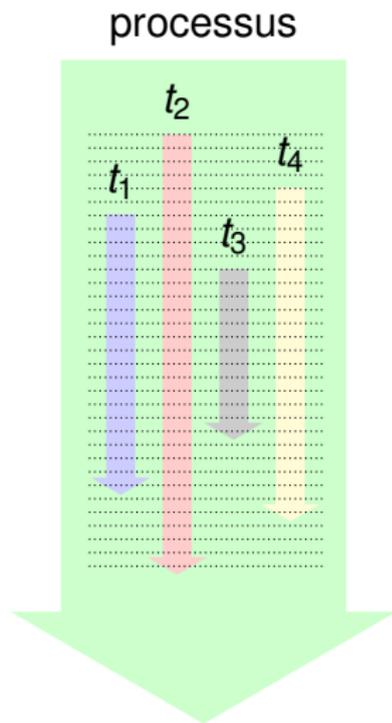
Threads et processus

- Un **processus** est une instance d'un programme en cours d'exécution ;
- Un **fil** (ou thread) peut être vu comme un sous-processus d'un processus donné ;
- Un processus peut comporter plusieurs threads, qui :
 - partagent le même espace mémoire et les mêmes ressources ;
 - **donnent l'illusion** de travailler en parallèle ;



Threads et processus

- Un **processus** est une instance d'un programme en cours d'exécution ;
- Un **fil** (ou thread) peut être vu comme un sous-processus d'un processus donné ;
- Un processus peut comporter plusieurs threads, qui :
 - partagent le même espace mémoire et les mêmes ressources ;
 - **donnent l'illusion** de travailler en parallèle ;
- En réalité, un processus s'exécute sur un seul cœur du processeur, que se partagent tour à tour ses threads ;



Threads

- Les threads ont leur utilité et sont disponibles via le module standard `threading`;

Threads

- Les threads ont leur utilité et sont disponibles via le module standard `threading`;
- Leur intérêt du point de vue des performances est limité :

Threads

- Les threads ont leur utilité et sont disponibles via le module standard `threading`;
- Leur intérêt du point de vue des performances est limité :
 - ① ils ne permettent pas d'exploiter plusieurs cœurs à la fois ;

- Les threads ont leur utilité et sont disponibles via le module standard `threading`;
- Leur intérêt du point de vue des performances est limité :
 - ① ils ne permettent pas d'exploiter plusieurs cœurs à la fois ;
 - ② en Python, le GIL (Global Interpreter Lock) ne permet pas aux threads d'agir en parallèle même si le processeur le permet ;

Threads

- Les threads ont leur utilité et sont disponibles via le module standard `threading`;
- Leur intérêt du point de vue des performances est limité :
 - ① ils ne permettent pas d'exploiter plusieurs cœurs à la fois ;
 - ② en Python, le GIL (Global Interpreter Lock) ne permet pas aux threads d'agir en parallèle même si le processeur le permet ;
- On va donc privilégier une approche multipliant les processus plutôt que les threads, puisque chaque processus pourra tourner en parallèle sur un cœur différent ;

Le module standard `multiprocessing`

- Le module standard `multiprocessing` nous donne accès aux processus multiples (et aux threads);
- Il démultiplie le processus actuel en plusieurs processus (rappelez-vous de `fork`);
- On obtient donc plusieurs interpréteurs Python pouvant chacun s'exécuter sur des cœurs différents;

- Le parallélisme n'est pas la réponse à tout ;
- Avec n cœurs, on va **au mieux** n fois plus vite ;
- Attention : le parallélisme a des coûts, proportionnels notamment à la communication entre les processus :
 - ① en temps de calcul ;
 - ② en mémoire ;
 - ③ ... et en difficulté d'implémentation ;

Exemple 1 (sans dépendances entre processus)



Calculons les n premières puissances de 2 en répartissant le travail sur tous les processeurs.

```
import multiprocessing
```

```
def calculer(num):  
    return 2 ** num
```

```
def main():  
    n = int(1e5)  
    print(  
        "Calcul parallèle des", n, "premières puissances de 2 sur",  
        multiprocessing.cpu_count(), "cœurs"  
    )  
    nombres = range(n)  
    with multiprocessing.Pool() as p:  
        valeurs = p.map(func=calculer, iterable=nombres)
```

crée un ensemble de k processus (par défaut, k est le nombre de cœurs)

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

applique f à chaque élément de it (version parallèle de `map`)

Limitations de `Pool.map`

- `Pool.map(func=f, iterable=it)` suppose que chaque élément de `it` est un paramètre à passer à `f`;
- Comment faire pour utiliser des fonctions à plusieurs paramètres ?
- On utilise `Pool.starmap`;

Exemple 2 (sans dépendances entre processus)



Calculons $0^0, 1^1, 2^2, \dots, n^n$ en répartissant le travail sur tous les processeurs.

```
import multiprocessing
```

```
def calculer(mantisse, exposant):  
    return mantisse ** exposant
```

crée les couples (0, 0), (1, 1), (2, 2), ...

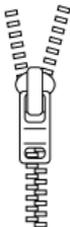
```
def main():  
    n = int(1e4) * 2  
    print(  
        "Calcul parallèle de 0 ** 0, 1 ** 1, 2 ** 2, ..., ", n, " ** ", n, " sur",  
        multiprocessing.cpu_count(), " cœurs"  
    )
```

```
    nombres = zip(range(n+1), range(n+1))  
    with multiprocessing.Pool() as p:  
        valeurs = p.starmap(func=calculer, iterable=nombres)
```

crée un ensemble de k processus (par défaut, k est le nombre de cœurs)

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

applique f à chaque itérable de it : si $it = [[a, b, c], \dots]$, on aura $f(a, b, c)$



Communication entre processus

- Dans les exemples précédents, les processus accèdent au même itérable, mais uniquement en lecture pour l'initialisation ;

Communication entre processus

- Dans les exemples précédents, les processus accèdent au même itérable, mais uniquement en lecture pour l'initialisation ;
- On ne sait pas dans quel ordre les tâches sont effectuées, mais le `Pool` attend que tous les processus se terminent avant de passer à la suite (à la fin du `with`) :

Communication entre processus

- Dans les exemples précédents, les processus accèdent au même itérable, mais uniquement en lecture pour l'initialisation ;
- On ne sait pas dans quel ordre les tâches sont effectuées, mais le `Pool` attend que tous les processus se terminent avant de passer à la suite (à la fin du `with`) :
- Les choses se compliquent quand il faut partager des données entre les processus ;

Communication entre processus

- Dans les exemples précédents, les processus accèdent au même itérable, mais uniquement en lecture pour l'initialisation ;
- On ne sait pas dans quel ordre les tâches sont effectuées, mais le `Pool` attend que tous les processus se terminent avant de passer à la suite (à la fin du `with`) :
- Les choses se compliquent quand il faut partager des données entre les processus ;
- `multiprocessing` le permet, mais avec des types particuliers ;

Communication entre processus

- Dans les exemples précédents, les processus accèdent au même itérable, mais uniquement en lecture pour l'initialisation ;
- On ne sait pas dans quel ordre les tâches sont effectuées, mais le `Pool` attend que tous les processus se terminent avant de passer à la suite (à la fin du `with`) :
- Les choses se compliquent quand il faut partager des données entre les processus ;
- `multiprocessing` le permet, mais avec des types particuliers ;
- Examinons un exemple simple : tester la primalité d'un naturel ;

Test parallèle de primalité : intuition

- L'algorithme naïf pour vérifier si un nombre n est premier teste tous les diviseurs impairs jusqu'à \sqrt{n} ;

Test parallèle de primalité : intuition

- L'algorithme naïf pour vérifier si un nombre n est premier teste tous les diviseurs impairs jusqu'à \sqrt{n} ;
- Pour le paralléliser, on demande à chaque processus de vérifier en parallèle un sous-intervalle ;

Test parallèle de primalité : intuition

- L'algorithme naïf pour vérifier si un nombre n est premier teste tous les diviseurs impairs jusqu'à \sqrt{n} ;
- Pour le paralléliser, on demande à chaque processus de vérifier en parallèle un sous-intervalle ;
- Si on trouve un diviseur, plus la peine d'en chercher un autre ...

Test parallèle de primalité : intuition

- L'algorithme naïf pour vérifier si un nombre n est premier teste tous les diviseurs impairs jusqu'à \sqrt{n} ;
- Pour le paralléliser, on demande à chaque processus de vérifier en parallèle un sous-intervalle ;
- Si on trouve un diviseur, plus la peine d'en chercher un autre ...
- ... et il faut donc le communiquer aux autres processus !

Test parallèle de primalité : implémentation

(explications sur les sources de “High Performance Python”)

Il nous faut définir :

- 1 un flag permettant de noter si un diviseur a été trouvé ;

Test parallèle de primalité : implémentation

(explications sur les sources de “High Performance Python”)

Il nous faut définir :

- 1 un flag permettant de noter si un diviseur a été trouvé ;
- 2 un manager pour partager cette variable entre les différents processus ;

Test parallèle de primalité : implémentation

(explications sur les sources de “High Performance Python”)

Il nous faut définir :

- 1 un flag permettant de noter si un diviseur a été trouvé ;
- 2 un manager pour partager cette variable entre les différents processus ;
- 3 un intervalle de vérification (vérifier si l'un des processus a trouvé un diviseur toutes les x secondes) ;

Test parallèle de primalité : implémentation

La fonction de vérification devient :

```
16 def check_prime_in_range(n_from_i_to_i):
17     (n, (from_i, to_i), value) = n_from_i_to_i
18     if n % 2 == 0:
19         return False
20     assert from_i % 2 != 0
21     check_every = CHECK_EVERY
22     # tester tous les diviseurs impairs dans l'intervalle donné
23     for i in range(from_i, int(to_i), 2):
24         # tous les CHECK_EVERY diviseurs, vérifier si l'un des
25         # processus a trouvé un diviseur
26         check_every -= 1
27         if not check_every:
28             if value.value == FLAG_SET: # oui -> on arrête tout
29                 return False
30             check_every = CHECK_EVERY
31
32     # si NOUS avons trouvé un diviseur, le signaler
33     if n % i == 0:
34         value.value = FLAG_SET
35         return False
36     return True
```

Test parallèle de primalité : implémentation

Une autre fonction (`check_prime`) s'occupe du découpage de l'intervalle de recherche et de la répartition :

```
48 from_i = to_i
49 to_i = int(math.sqrt(n)) + 1
50
51 ranges_to_check = create_range.create(from_i, to_i, nbr_processes)
52 ranges_to_check = list(zip(len(ranges_to_check) * [n], ranges_to_check,
↳ len(ranges_to_check) * [value]))
53 assert len(ranges_to_check) == nbr_processes
54 results = pool.map(check_prime_in_range, ranges_to_check)
55 if False in results:
56     return False
57 return True
```

Test parallèle de primalité : implémentation

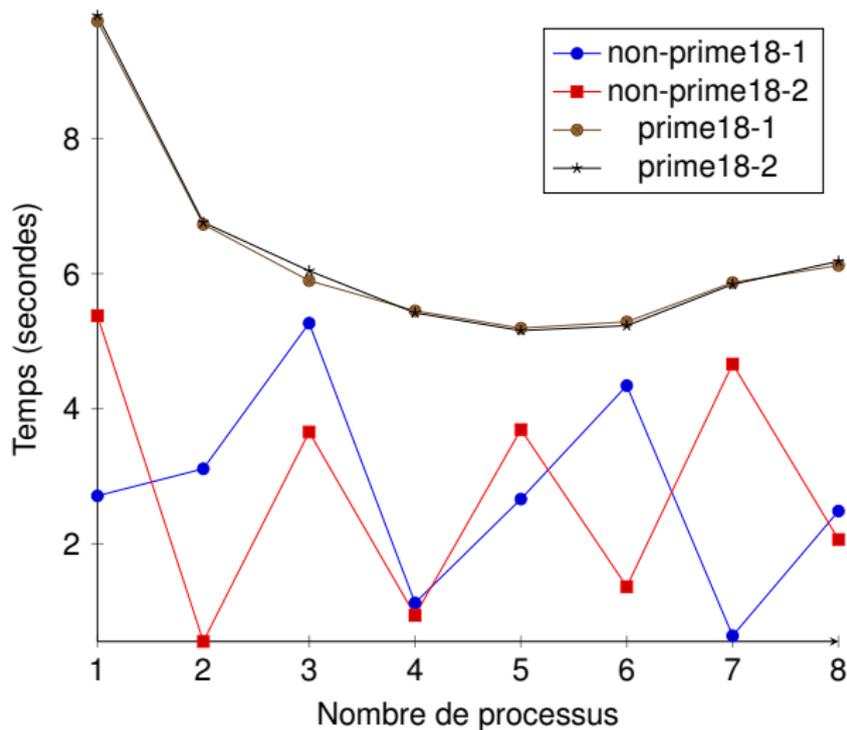
Une autre fonction (`check_prime`) s'occupe du découpage de l'intervalle de recherche et de la répartition :

```
48 from_i = to_i
49 to_i = int(math.sqrt(n)) + 1
50
51 ranges_to_check = create_range.create(from_i, to_i, nbr_processes)
52 ranges_to_check = list(zip(len(ranges_to_check) * [n], ranges_to_check,
53 ↪ len(ranges_to_check) * [value]))
54 assert len(ranges_to_check) == nbr_processes
55 results = pool.map(check_prime_in_range, ranges_to_check)
56 if False in results:
57     return False
58 return True
```

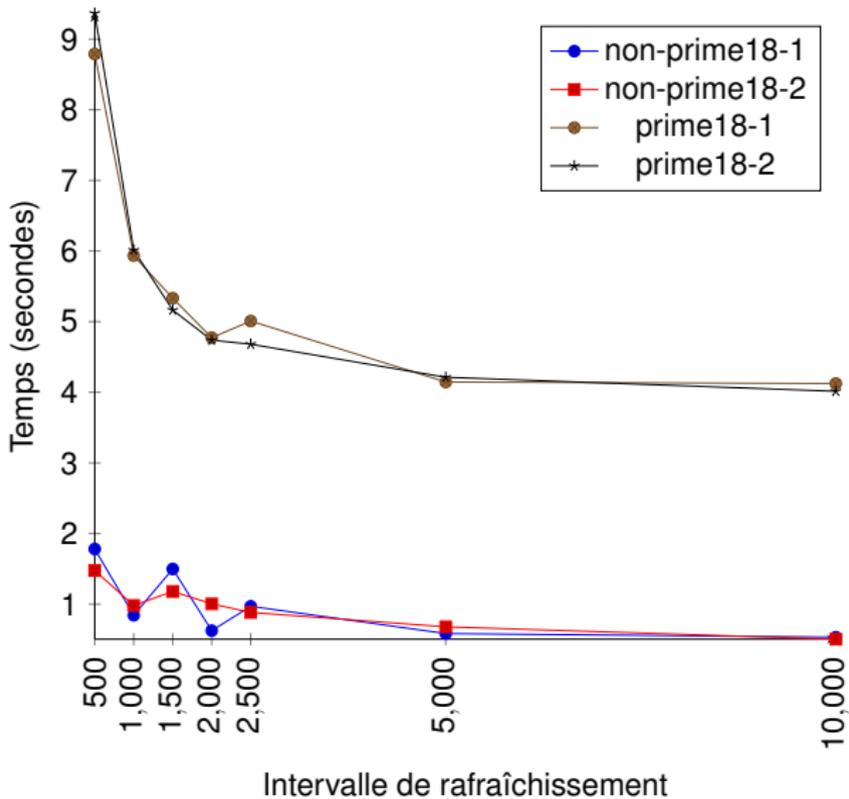
... que le code principal appelle après avoir initialisé le reste :

```
62 NBR_PROCESSES = 4 # int(sys.argv[1])
63 CHECK_EVERY = int(sys.argv[1])
64 manager = multiprocessing.Manager()
65 value = manager.Value(b'c', FLAG_CLEAR) # 1 byte character
66 pool = Pool(processes=NBR_PROCESSES)
```

Performances selon le nombre de processus



Performances selon le rafraîchissement (4 CPUs)



Profilage et parallélisme

 viztracer



`cProfile` ne donne que les statistiques liées au processus “père” !

Si l'on veut prendre en compte tous les processus lancés, il faut utiliser `viztracer` comme suit :

```
$ python3 -m viztracer --log_multiprocess programme.py
```