M?moisation_en_Python_et_OCaml

October 5, 2017

1 Table of Contents

2 Mémoïsation, en Python et en OCaml

Ce document montre deux exemples d'implémentations d'un procédé générique (mais basique) de mémoïsation en Python et en OCaml

2.1 En Python

2.1.1 Exemples de fonctions à mémoïser

On commence avec des fonctions inutilement lentes :

```
In [1]: from time import sleep
```

```
In [2]: def f1(n):
            sleep(3)
            return n + 3

        def f2(n):
            sleep(4)
            return n * n
```

```
In [3]: %timeit f1(10)
```

```
3 s ś 1.05 ms per loop (mean ś std. dev. of 7 runs, 1 loop each)
```

```
In [4]: %timeit f2(10)
```

```
4 s ś 1.25 ms per loop (mean ś std. dev. of 7 runs, 1 loop each)
```

2.1.2 Mémoïsation générique, non typée

C'est étrangement court !

```
In [5]: def memo(f):
            memoire = {}  # dictionnaire vide, {} ou dict()
            def memo_f(n):              # nouvelle fonction
                if n not in memoire:    # verification
                    memoire[n] = f(n)   # stockage
                return memoire[n]       # lecture
            return memo_f               # ==> f memoisée !
```

2.1.3 Essais

```
In [6]: memo_f1 = memo(f1)

        print("3 secondes...")
        print(memo_f1(10))  # 13, 3 secondes après
        print("0 secondes !")
        print(memo_f1(10))  # instantanné !

        # différent de ces deux lignes !

        print("3 secondes...")
        print(memo(f1)(10))
        print("3 secondes...")
        print(memo(f1)(10))  # 3 secondes aussi !
```

```
3 secondes...
13
0 secondes !
13
3 secondes...
13
3 secondes...
13
```

```
In [7]: %timeit memo_f1(10)  # instantanné !
```

```
122 ns ś 5.67 ns per loop (mean ś std. dev. of 7 runs, 10000000 loops each)
```

Et :

```
In [8]: memo_f2 = memo(f2)

        print("4 secondes...")
        print(memo_f2(10))  # 100, 4 secondes après
        print("0 secondes !")
        print(memo_f2(10))  # instantanné !
```

```
4 secondes...
100
0 secondes !
100
```

```
In [9]: %timeit memo_f2(10)  # instantanné !
```

```
120 ns ś 3.43 ns per loop (mean ś std. dev. of 7 runs, 10000000 loops each)
```

2.1.4 Mémoïsation générique et typée

Ce n'est pas tellement plus compliquée de typer la mémoïsation.

```
In [10]: def memo_avec_type(f):
             memoire = {}  # dictionnaire vide, {} ou dict()
             def memo_f_avec_type(n):
                 if (type(n), n) not in memoire:
                      memoire[(type(n), n)] = f(n)
                 return memoire[(type(n), n)]
             return memo_f_avec_type
```

Avantage, on obtiens un résultat plus cohérent "au niveau de la reproducibilité des résultats", par exemple :

```
In [11]: def fonction_sur_entiers_ou_flottants(n):
             if isinstance(n, int):
                 return 'Int'
             elif isinstance(n, float):
                 return 'Float'
             else:
                 return '?'
```

```
In [12]: test0 = fonction_sur_entiers_ou_flottants
         print(test0(1))
         print(test0(1.0))  # résultat correct !
         print(test0("1"))
```

```
Int
Float
?
```

```
In [14]: test1 = memo(fonction_sur_entiers_ou_flottants)
         print(test1(1))
         print(test1(1.0))  # résultat incorrect !
         print(test1("1"))
```

```
Int
Int
?
```

```
In [15]: test2 = memo_avec_type(fonction_sur_entiers_ou_flottants)
         print(test2(1))
         print(test2(1.0))  # résultat correct !
         print(test2("1"))
```

```
Int
Float
?
```

2.1.5 Bonus : on peut utiliser la syntaxe d'un décorateur en Python

```
In [16]: def fibo(n):
             if n <= 1: return 1
             else: return fibo(n-1) + fibo(n-2)

         print("Test de fibo() non mémoisée :")
         for n in range(10):
             print("F_{} = {}".format(n, fibo(n)))
```

```
Test de fibo() non mémoisée :
F_0 = 1
F_1 = 1
F_2 = 2
F_3 = 3
F_4 = 5
F_5 = 8
F_6 = 13
F_7 = 21
F_8 = 34
F_9 = 55
```

Cette fonction récursive est terriblement lente !

```
In [18]: %timeit fibo(35)

3.15 s ś 74.8 ms per loop (mean ś std. dev. of 7 runs, 1 loop each)
```

```
In [33]: # version plus rapide !
         @memo
         def fibo2(n):
             if n <= 1: return 1
             else: return fibo2(n-1) + fibo2(n-2)

         print("Test de fibo() mémoisée (plus rapide) :")
         for n in range(10):
             print("F_{} = {}".format(n, fibo2(n)))

Test de fibo() mémoisée (plus rapide) :
F_0 = 1
F_1 = 1
F_2 = 2
F_3 = 3
F_4 = 5
F_5 = 8
F_6 = 13
F_7 = 21
F_8 = 34
F_9 = 55
```

```
In [21]: %timeit fibo2(35)

118 ns ś 2.59 ns per loop (mean ś std. dev. of 7 runs, 10000000 loops each)
```

Autre exemple, ou le gain de temps est moins significatif.

```
In [22]: def factorielle(n):
             if n <= 0: return 0
             elif n == 1: return 1
             else: return n * factorielle(n-1)

         print("Test de factorielle() non mémoisée :")
         for n in range(10):
             print("{}! = {}".format(n, factorielle(n)))

Test de factorielle() non mémoisée :
0! = 0
1! = 1
2! = 2
3! = 6
```

```
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
```

```
In [23]: %timeit factorielle(30)
```

```
4.53 µs ś 214 ns per loop (mean ś std. dev. of 7 runs, 100000 loops each)
```

```
In [24]: @memo
         def factorielle2(n):
             if n <= 0: return 0
             elif n == 1: return 1
             else: return n * factorielle2(n-1)

         print("Test de factorielle() mémoisée :")
         for n in range(10):
             print("{}! = {}".format(n, factorielle2(n)))
```

```
Test de factorielle() mémoisée :
0! = 0
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
```

```
In [25]: %timeit factorielle2(30)
```

```
124 ns ś 6.32 ns per loop (mean ś std. dev. of 7 runs, 10000000 loops each)
```

2.1.6 Conclusion

En Python, c'est facile, avec des dictionnaires génériques et une syntaxe facilitée avec un décorateur.

Bonus : ce décorateur est dans la bibliothèque standard dans le module functools !

```
In [32]: from functools import lru_cache  # lru = least recently updated
```

```
In [34]: @lru_cache(maxsize=None)
         def fibo3(n):
             if n <= 1: return 1
             else: return fibo3(n-1) + fibo3(n-2)

         print("Test de fibo() mémoisée avec functools.lru_cache (plus rapide) :")
         for n in range(10):
             print("F_{} = {}".format(n, fibo3(n)))
```

```
Test de fibo() mémoisée avec functools.lru_cache (plus rapide) :
F_0 = 1
F_1 = 1
F_2 = 2
F_3 = 3
F_4 = 5
F_5 = 8
F_6 = 13
F_7 = 21
F_8 = 34
F_9 = 55
```

```
In [35]: %timeit fibo2(35)
```

```
115 ns ś 2.83 ns per loop (mean ś std. dev. of 7 runs, 10000000 loops each)
```

```
In [36]: %timeit fibo3(35)
```

```
86.6 ns ś 1.62 ns per loop (mean ś std. dev. of 7 runs, 10000000 loops each)
```

```
In [37]: %timeit fibo2(70)
```

```
117 ns ś 1.44 ns per loop (mean ś std. dev. of 7 runs, 10000000 loops each)
```

```
In [38]: %timeit fibo3(70)
```

```
86.2 ns ś 1.12 ns per loop (mean ś std. dev. of 7 runs, 10000000 loops each)
```

(On obtient presque les mêmes performances que notre implémentation manuelle)

2.2 En OCaml

Je traite exactement les mêmes exemples.

> J'expérimente l'utilisation de deux kernels Jupyter différents pour afficher des exemples de codes écrits dans deux langages dans le même notebook... Ce n'est pas très propre mais *ça marche*.

2.2.1 Préliminaires

Quelques fonctions nécessaires pour ces exemples :

```
In [7]: let print = Format.printf;;
        let sprintf = Format.sprintf;;
        let time = Unix.time;;
        let sleep n = Sys.command (sprintf "sleep %i" n);;

Out[7]: val print : ('a, Format.formatter, unit) format -> 'a = <fun>


Out[7]: val sprintf : ('a, unit, string) format -> 'a = <fun>


Out[7]: val time : unit -> float = <fun>


Out[7]: val sleep : int -> int = <fun>


In [11]: let timeit (repet : int) (f : 'a -> 'a) (x : 'a) () : float =
             let time0 = time () in
             for _ = 1 to repet do
                 ignore (f x);
             done;
             let time1 = time () in
             (time1 -. time0 ) /. (float_of_int repet)
         ;;

Out[11]: val timeit : int -> ('a -> 'a) -> 'a -> unit -> float = <fun>
```

2.2.2 Exemples de fonctions à mémoïser

```
In [12]: let f1 n =
             ignore (sleep 3);
             n + 2
         ;;

Out[12]: val f1 : int -> int = <fun>


In [13]: let _ = f1 10;; (* 13, après 3 secondes *)

Out[13]: - : int = 12


In [14]: timeit 3 f1 10 ();; (* 3 secondes *)
```

```
Out[14]: - : float = 3.
```

Et un autre exemple similaire :

```
In [15]: let f2 n =
             ignore (sleep 4);
             n * n
         ;;

Out[15]: val f2 : int -> int = <fun>


In [18]: let _ = f2 10;; (* 100, après 3 secondes *)

Out[18]: - : int = 100


In [19]: timeit 3 f2 10 ();; (* 4 secondes *)

Out[19]: - : float = 4.
```

2.2.3 Mémoïsation pour des fonctions d'un argument

On utilise le module Hashtbl de la bibliothèque standard.

```
In [20]: let memo f =
             let memoire = Hashtbl.create 128 in (* taille 128 par defaut *)
             let memo_f n =
                 if Hashtbl.mem memoire n then   (* lecture *)
                     Hashtbl.find memoire n
                 else begin
                     let res = f n in            (* calcul *)
                     Hashtbl.add memoire n res;  (* stockage *)
                     res
                 end
             in
             memo_f  (* nouvelle fonction *)
         ;;

Out[20]: val memo : ('a -> 'b) -> 'a -> 'b = <fun>
```

2.2.4 Essais

Deux exemples :

```
In [21]: let memo_f1 = memo f1 ;;
         let _ = memo_f1 10 ;; (* 3 secondes *)
         let _ = memo_f1 10 ;; (* instantanné *)
```

```
Out[21]: val memo_f1 : int -> int = <fun>


Out[21]: - : int = 12


Out[21]: - : int = 12


In [24]: timeit 100 memo_f1 20 ();; (* 0.03 secondes *)

Out[24]: - : float = 0.03


In [28]: let memo_f2 = memo f2 ;;
         let _ = memo_f2 10 ;; (* 4 secondes *)
         let _ = memo_f2 10 ;; (* instantanné *)

Out[28]: val memo_f2 : int -> int = <fun>


Out[28]: - : int = 100


Out[28]: - : int = 100


In [29]: timeit 100 memo_f2 20 ();; (* 0.04 secondes *)

Out[29]: - : float = 0.04
```

Ma fonction `timeit` fait un nombre paramétrique de répétitions sur des entrées non aléatoires, donc le temps *moyen* observé dépend du nombre de répétitions !

```
In [31]: timeit 10000 memo_f2 50 ();; (* 0.04 secondes *)

Out[31]: - : float = 0.0004
```

2.2.5 Exemple de la suite de Fibonacci

```
In [32]: let rec fibo = function
             | 0 | 1 -> 1
             | n -> (fibo (n - 1)) + (fibo (n - 2))
         ;;

Out[32]: val fibo : int -> int = <fun>


In [38]: fibo 40;;
```

```
Out[38]: - : int = 165580141


In [42]: timeit 10 fibo 40 ();; (* 4.2 secondes ! *)

Out[42]: - : float = 4.2
```

Et avec la mémoïsation automatique :

```
In [48]: let memo_fibo = memo fibo;;

Out[48]: val memo_fibo : int -> int = <fun>


In [49]: memo_fibo 40;;

Out[49]: - : int = 165580141


In [50]: timeit 10 memo_fibo 41 ();; (* 0.7 secondes ! *)

Out[50]: - : float = 0.7
```

2.2.6 Conclusion

En OCaml, ce n'était pas trop dur non plus en utilisant une table de hachage (dictionnaire), disponibles dans le module Hashtbl.

On est confronté à une limitation de Caml, à savoir que la la fonction `memo_f` doit être bien typée pour être renvoyée par `memo f` donc `memo` ne peut pas avoir un type générique : il faut écrire un décorateur de fonction pour chaque signature bien connue de la fonction qu'on veut mémoïser...