8. Compléments sur les fonctions

8.1. Fonctions récursives

8.1.1. Principe

Revenons rapidement sur la façon d'utiliser des fonctions en Python vue au premier semestre :

— d'abord, on *définit* la fonction à l'aide du mot-clé `def` suivi de l'*identifiant* de la fonction, ses éventuels *paramètres* entre parenthèses, puis son *corps*,

— ensuite, on peut autant de fois que l'on veut *appeler* cette fonction en passant des valeurs concrètes en paramètre.

On rappelle que dans le corps d'une fonction `f`, il est tout à fait possible de faire appel à une autre fonction `g`. Il suffit qu'au moment où `f` sera appelée, la fonction `g` ait bien été définie.

À tester. Lors de l'appel à `contains_prime` à la ligne 15 du programme ci-dessous, quelles opérations sont effectuées ?

```
1  def is_prime(n):
2      # Check if n is prime
3      for d in range(2, n):
4          if n % d == 0:
5              return False
6      return True
7
8  def contains_prime(L):
9      # Check if L contains at least one prime number
10     for x in L:
11         if is_prime(x):
12             return True
13     return False
14
15 print(contains_prime([9, 10, 4, 3, 6, 7]))
16 print(contains_prime([8, 12, 10, 6, 10, 15]))
```

Mieux encore, dans le corps d'une fonction `f` il est possible de faire appel à la fonction `f` elle-même comme dans l'exemple suivant.

À tester. On considère la définition suivante.

```
1  def foo(i):
2      print("Début de l'apppel à foo avec i qui vaut", i)
3      if i == 0:
4          print("Terminé")
5      else:
6          foo(i - 1)
7      print("Fin de l'appel à foo avec i qui vaut", i)
```

Que se passera-t-il si on exécute l'instruction `foo(4)` ? Essayez d'anticiper le résultat vous-même, puis vérifiez avec Python.

Dans l'exemple ci-dessus, le « calcul » de `foo(4)` demande de calculer `foo(3)`, qui demande lui-même de calculer `foo(2)`, et ainsi de suite … On appelle **fonction récursive** une fonction qui, lors de son appel, peut être amenée à devoir se réinvoquer elle-même. Ainsi, parmi toutes les fonctions des exemples précédents, les fonctions `is_prime` et `contains_prime` ne sont pas récursives, tandis que la fonction `foo` est récursive.

Remarque. Il arrive qu'un même algorithme puisse être codé aussi bien de façon récursive que de façon non-récursive. Par exemple, on aurait aussi pu définir la fonction `contains_prime` de la façon suivante.

```
def contains_prime(L):
    # Check if L contains at least one prime number
    if len(L) == 0:
        return False

    return is_prime(L[0]) or contains_prime(L[1:])
```

Dans ce cas, la fonction `contains_prime` réalise exactement la même tâche que précédemment, mais c'est maintenant une fonction récursive.

Rappels premier semestre : Que signifie la syntaxe `L[a:b]` *? Et en particulier* `L[1:]` *?*

Les fonctions récursives apparaissent naturellement dans des problèmes dont la résolution pour un objet donné peut s'exprimer via le même problème appliqué à des objets plus petits. Par exemple :

— une liste contient au moins un nombre premier si son premier terme est premier ou si le reste de la liste contient un nombre premier,

— la factorielle d'un entier n est égal au produit entre n et la factorielle de $n-1$.

Lors de la définition d'une fonction récursive, il faut impérativement penser à traiter à part les « cas limite » auxquels on se ramène (par exemple la liste vide dans le premier exemple, et $0!$ ou $1!$ dans le second exemple). Sinon, la fonction ne cessera jamais de s'appeler elle-même et tombera dans un problème similaire à celui d'une boucle infinie.

À tester. Analysez le code suivant.

```
def fact(n):
    if n == 0:
        return 1

    return n * fact(n - 1)

print(fact(5))
```

Que se passe-t-il si on essaie maintenant d'appeler `fact(-1)` ?

Exercice 8.1. Écrire une fonction récursive `recursive_sum(L)` qui prend en argument une liste de nombres `L` et qui renvoie la somme des éléments de `L`. L'aspect récursif se basera sur le principe suivant :

— **Cas limite.** Par convention, la somme des éléments d'une liste vide vaut zéro.

— **Récursivité.** recursive_sum$([x_0, x_1, \ldots, x_{n-1}])$ est égal à $x_0 +$ recursive_sum$([x_1, \ldots, x_{n-1}])$.

Exercice 8.2. Écrire une fonction récursive `is_sorted(L)` qui prend en argument une liste de nombres `L` et qui renvoie un booléen indiquant si la liste `L` est triée par ordre croissant ou non. L'aspect récursif se basera sur le principe suivant :

— **Cas limite.** Une liste de 0 ou 1 éléments est, par convention, toujours triée par ordre croissant.

— **Récursivité.** Une liste $L = [x_0, x_1, x_2, \ldots, x_{n-1}]$ est triée par ordre croissant si $x_0 \leq x_1$ et si la sous-liste $L' = [x_1, x_2, \ldots, x_{n-1}]$ est triée par ordre croissant elle-aussi.

Remarque. Pour la culture, sachez que la récursivité peut aussi se cacher dans des scénarios plus subtils. Dans l'exemple ci-dessous, la fonction `a` ne s'appelle pas directement, mais elle appelle `b` qui rappelle `a` à son tour, et ainsi de suite … Les fonctions `a` et `b` rentrent alors elles-aussi dans la catégorie des fonctions récursives.

```
def a(n):
    if n < 0:
        return 1
    else:
        return n + b(n-1)
```

```
def b(n):
    if n < 0:
        return 0
    else:
        return n * a(n-1)
```

8.1.2. Intérêt et limites

Ci-dessous on donne deux versions, l'une récursive et l'autre non-récursive :

— d'une fonction `fact` qui renvoie la factorielle $n!$ de l'entier n passé en argument,

— d'une fonction `fibo` qui renvoie, pour l'entier n passé en argument, le terme u_n de la *suite de* FIBONACCI définie par $u_0 = 1$, $u_1 = 1$ et pour tout $n \geq 2$,

$$u_{n+2} = u_{n+1} + u_n.$$

On voit que ce sont deux problèmes qui se prêtent bien à la récursivité, cette dernière fournissant des algorithmes un peu plus simples et lisibles que dans la version itérative.

Version récursive

```
def fact(n):
  if n <= 1:
    return 1

  return n * fact(n-1)
```

Version itérative

```
def fact(n):
  P = 1
  for i in range(2, n+1):
    P = P * i
  return P
```

```
def fibo(n):
  if n <= 1:
    return 1

  return fibo(n-1) + fibo(n-2)
```

```
def fibo(n):
  a, b = 1, 1
  for i in range(2, n+1):
    a, b = b, a + b
  return b
```

Toutefois, la version récursive proposée par la fonction `fibo` est loin d'être aussi efficace que sa version itérative. La figure 8.1 illustre les appels récursifs qui sont faits lors du calcul de `fact(5)` et `fibo(5)` respectivement. On constate que pour `n` égal à 5, le calcul de `fact(n)` cache 4 appels récursifs alors que celui

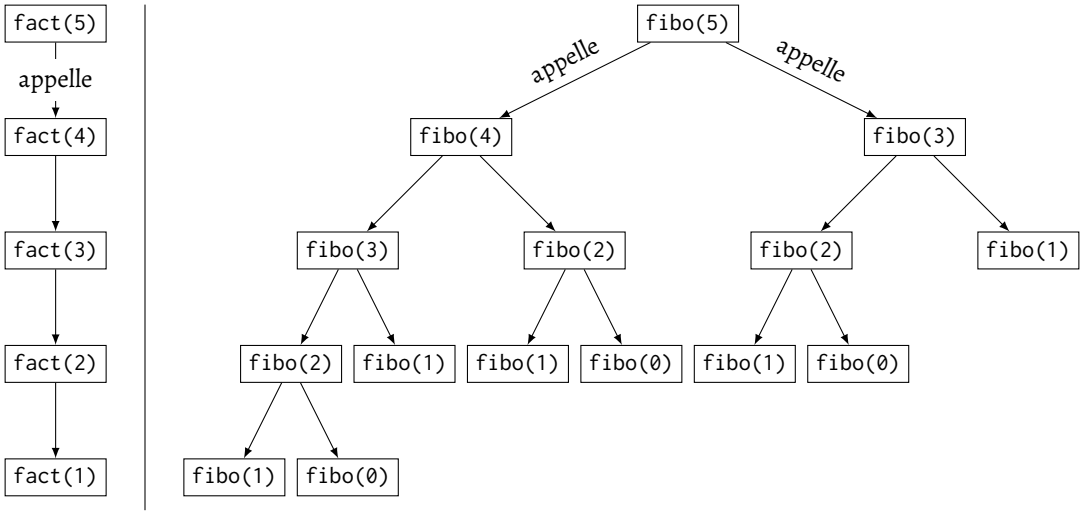


FIGURE 8.1. – Appels récursifs effectués pour les implémentations proposées de la factorielle de la suite de FIBONACCI

de `fibo(n)` cache déjà 14 appels récursifs. Cette différence s'accentue d'autant plus que la valeur de `n` augmente, et vous pouvez déjà constater un écart de temps d'exécution non-négligeable entre ces deux calculs pour `n` valant 40. La version itérative de la fonction `fibo`, en revanche, demande $n-1$ tours de boucle pour le calcul de u_n : en particulier le calcul de `fibo(40)` est alors quasi-immédiat.

Remarque. Dans le cas de la fonction `fibo`, un problème évident est déjà que l'on recalcule plusieurs fois des termes identiques. Par exemple, le calcul de `fact(5)` effectue à trois moments différents le calcul de `fibo(2)`. Une première amélioration de cette fonction pourrait déjà être de stocker quelque part les résultats intermédiaires obtenus pour les réutiliser ensuite. Toutefois nous ne rentrerons pas dans ces détails : retenez simplement que la récursivité est à utiliser avec parcimonie. En première année, essayez tant que possible d'utiliser des algorithmes itératifs, et réservez la récursivité aux problèmes qu'il serait trop complexe de traiter autrement.

Exercice 8.3.

1. Écrire une fonction récursive `another_sum(L)` qui prend en argument une liste de nombres `L`, et qui renvoie à nouveau la somme des éléments de `L` mais en se basant cette fois sur le procédé suivant :
   - **Cas limite.** La somme des éléments d'une liste vide vaut zéro, et la somme des éléments d'une liste qui ne contient qu'un seul nombre est égale à ce nombre.
   - **Récursivité.** Pour $k = n/2$ (arrondi si besoin), another_sum$([x_0, x_1, \ldots, x_{n-1}])$ est égal à
   $$\text{another\_sum}([x_0, x_1, \ldots, x_{k-1}]) + \text{another\_sum}([x_k, x_1, \ldots, x_{n-1}]).$$
2. On pose `L = [5, 2, 8, 1, 2, 3, 6]`. Illustrer les calculs de `another_sum(L)` et `recursive_sum(L)` (exercice 8.1) sur des arbres similaires à ceux de la figure 8.1. Comparez ces deux arbres en termes de nombre d'appels effectués, et en termes de « profondeur ».
3. Pourquoi était-il important d'inclure dans le cas limite les listes de taille 1 ?

Enfin, notons une dernière limite importante de la récursivité. Au moment de calculer `fibo(1)` dans la figure 8.1, l'interpréteur doit se rappeler qu'il s'agit d'un calcul intervenu dans la ligne 5 de l'appel à `fibo(2)`, afin de pouvoir y revenir une fois que `fibo(1)` aura été calculé. Mais de même, l'interpréteur gardait déjà en mémoire que `fibo(2)` était un calcul intervenu à la ligne 5 lors de l'appel à `fibo(3)`, qui était lui-même

un calcul intervenu à la ligne 5 de `fibo(4)`, et ainsi de suite. Cette nécessité de garder une trace des appels de fonctions effectués n'est pas propre à la récursivité, elle s'applique plus généralement à toute situation où des appels de fonctions sont imbriqués les uns dans les autres.

À tester. Vérifiez que le code à gauche produit le message d'erreur à droite ci-dessous, et expliquez ce message. Pour information, *traceback* se traduit par *trace d'appels*.

```
1  def foo(x):
2      x = x + 1
3      y = bar(x)
4      return x + y
5
6  def bar(x):
7      x = 1 + baz(x)
8      return x
9
10 def baz(x):
11     y = x / 0
12     return y
13
14 foo(2)
15
16
```

```
Traceback (most recent call last):
  (...)

  File "...", line 14, in <module>
    foo(2)

  File "...", line 3, in foo
    y = bar(x)

  File "...", line 7, in bar
    x = 1 + baz(x)

  File "...", line 11, in baz
    y = x / 0

ZeroDivisionError: division by zero
```

Toutefois, la récursivité en particulier peut nécessiter de tracer un très grand nombre d'appels imbriqués les uns dans les autres. Or, il y a en pratique une limite du nombre d'appels que peut tracer l'interpréteur, et si celle-ci est dépassée, alors le programme s'arrête et déclenche une erreur. Il faut donc toujours privilégier soit des fonctions récursives pour lesquelles le nombre d'appels imbriqués n'augmentera pas trop vite en fonction de la taille de l'entrée, soit revenir sur des versions itératives de ces mêmes fonctions.

À tester.

1. Comparez le résultat de `fact(3000)` suivant si on a choisi l'implémentation récursive ou l'implémentation itérative de la fonction `fact`.
2. Reprenez les fonctions `recursive_sum` et `another_sum` des exercices précédents et essayez de les appliquer à une liste contenant 3000 nombres.

8.2. La fonction en tant qu'objet

En Python, les fonctions sont des « objets » au même titre que les entiers, les flottants, les listes, etc. On peut donc les manipuler exactement de la même manière, ce qui inclut par exemple les quelques situations présentées ci-après.

On peut stocker les fonctions dans une collection. Comme tout objet, il est possible de stocker des fonctions dans une liste, un dictionnaire, un ensemble, …

À tester. Analysez et testez le code ci-dessous.

```
def add_one(x):
    return x + 1

def substract_one(x):
    return x - 1

operations = {"+": add_one, "-": substract_one}

x = 0
for c in "++-+--+":
    x = operations[c](x)
    print(x)
```

Exercice 8.4.

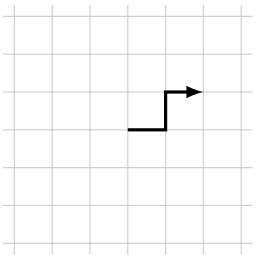
1. Importez le module `turtle` et créez un dictionnaire `moves` contenant les entrées suivantes :

Clé	Valeur
`'F'`	Une fonction qui fait avancer la tortue de 10 pixels.
`'B'`	Une fonction qui fait reculer la tortue de 10 pixels.
`'L'`	Une fonction qui fait tourner la tortue de 90 degrés sur sa gauche.
`'R'`	Une fonction qui fait tourner la tortue de 90 degrés sur sa droite.

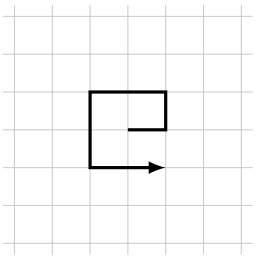
2. Créez une fonction `draw_string(S)` qui prend en argument une chaîne de caractères `S` formée uniquement des caractères `F`, `B`, `L` et `R`, et qui la traduit en une figure dessinée avec `turtle`. La chaîne `S` sera lue de la gauche vers la droite, et chaque caractère sera traduit à l'aide du dictionnaire `moves`.

   Exemples.

`draw_string("FLFRF")`



`draw_string("FLFLFFLFFLFF")`



On peut passer des fonctions en argument d'un fonction.

À tester. Analysez et tester le code ci-dessous.

```
def value_at_one(f):
    return f(1)

def f1(x):
    return 2 * x + 3

def f2(x):
    return x ** 2 + 1

print(value_at_one(f1))
print(value_at_one(f2))
```

Remarque. On rappelle que lorsqu'on définit une fonction, les arguments que l'on pose n'ont pas de type associé *a priori*. Tout ce qui importe est que, lorsqu'on appellera la fonction plus tard dans le programme, on s'assure bien de passer en arguments des objets dont le type est cohérent avec l'utilisation qui en est faite dans le corps de la fonction. Considérons par exemple les deux fonctions ci-dessous :

```
def foo(x):
    return x + x
```

```
def bar(L):
    return L[0]
```

La fonction `foo` peut aussi bien prendre un argument un entier qu'un flottant, une liste ou encore une chaîne de caractères. En revanche le corps de la fonction `bar` suppose implicitement que cette dernière s'applique seulement à un objet *subscriptable* admettant 0 comme indice (par exemple une liste contenant au moins un élément, ou encore un dictionnaire ayant 0 pour clé ...). Mais en pratique rien n'empêche de l'appeler avec un autre type d'objet, simplement on obtiendra un message d'erreur au moment de l'exécution.

Il en est exactement de même dans la définition de `value_at_one` ci-dessus : l'interpréteur ne « sait » pas à l'avance que l'argument `f` désigne une fonction, il est uniquement de la responsabilité du programmeur d'appeler `value_at_one` en lui donnant en paramètre une fonction et non pas un entier, une liste, ...

Exercice 8.5. Écrire une fonction `apply(f, L)` qui prend comme arguments une fonction `f` et une liste `L`, et qui renvoie une nouvelle liste obtenue en appliquant la fonction `f` à chacun des éléments de `L`.

Par exemple, si on définit

```
def square(x):
    return x ** 2

def increment(x):
    return x + 1
```

alors :

- `apply(square, [2, 5, 1, 3])` devra renvoyer la liste `[4, 25, 1, 9]`,
- `apply(increment, [2, 5, 1, 3])` devra renvoyer la liste `[3, 6, 2, 4]`.

En général, on appelle *fonction de rappel* (*callback* en anglais) une fonction `f` passée en argument d'une autre fonction `g` dans le but d'être exécutée dans le corps de `g`. Nous ré-emploieront ce vocabulaire dans le chapitre sur les interfaces graphiques.

Une fonction peut renvoyer une fonction. Ce dernier point peut paraître perturbant au premier abord, mais un exemple tel que celui ci-dessous fonctionne parfaitement.

À tester. Vérifiez que le code suivant fonctionne et **ne crée pas** de variable `f` dans l'explorateur de variables.

```
def add(n):
    def f(x):
        return x + n
    return f
```

Testez ensuite la série d'instructions suivante, et observez les résultats obtenus dans la console et dans l'explorateur de variables.

```
add_one = add(1)
add_four = add(4)
print(add_one(3))
print(add_four(3))
```

Exercice 8.6.

1. Écrire une fonction `trinome(a, b, c)` qui prend en arguments trois nombres a, b, c, et qui renvoie la fonction $x \mapsto ax^2 + bx + c$.
2. Écrire une fonction `plot_function(f, a, b)` qui prend en argument une fonction f et deux nombres $a < b$, et qui utilise la fonction `plot` du module `matplotlib.pyplot` pour tracer le graphe de f sur l'intervalle $[a, b]$.
3. Utilisez ces deux fonctions pour tracer quelques trinômes du second degré.

Pour aller plus loin : le mot-clé `lambda`. La syntaxe `lambda args : expr` permet de créer des fonctions « à la volée ». Concrètement, l'instruction

```
foo = lambda args : expr
```

est équivalente à

```
def foo(args):
    return expr
```

L'intérêt du mot-clé `lambda` est de créer des fonctions simples en un ligne, sans même avoir besoin de leur donner un nom (on les appelle alors *fonctions anonymes* ou *fonctions lambda*). Il est particulièrement utile lorsqu'on veut justement passer des fonctions en argument ou en valeur de retour d'une autre fonction.

À tester. Vérifier que le code ci-dessous donne le même résultat que le code page 92 :

```
operations = {"+": (lambda x: x + 1), "-": (lambda x: x - 1)}

x = 0
for c in "++-+--+":
    x = operations[c](x)
    print(x)
```

À tester. Vérifier que le code ci-dessous donne le même résultat que le code page 93 :

```
def value_at_one(f):
    return f(1)

print(value_at_one(lambda x: 2 * x + 3))
print(value_at_one(lambda x: x ** 2 + 1))
```

À tester. Vérifier que le code ci-dessous donne le même résultat que le code page 94 :

```
def add(n):
    return (lambda x: x + n)

add_one = add(1)
add_four = add(4)
print(add_one(3))
print(add_four(3))
```

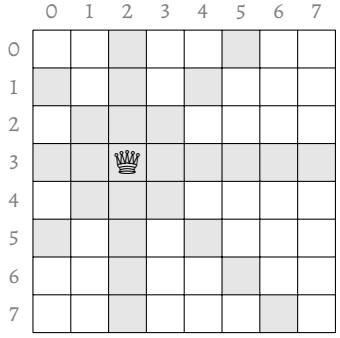
Exercice 8.7. Reprenez les exercices de cette section en utilisant le mot-clé `lambda` là où c'est possible et utile.

Exercices supplémentaires

Exercice 8.8. Le problème des huit dames (Difficile). Ce problème consiste à trouver une façon de placer 8 dames sur un échiquier de taille 8×8 sans qu'aucune d'entre elles n'en menace une autre. Aux échecs, la dame est une pièce qui peut se déplacer d'un nombre de cases quelconque horizontalement, verticalement ou en diagonale.

Dans cet exercice, on se propose de résoudre ce problème à l'aide d'un algorithme récursif. On modélisera une dame par un tuple (i, j) représentant sa position sur le plateau (avec $0 \leq i < 8$ et $0 \leq j < 8$), et commencera par implémenter quelques fonctions utiles à la résolution du problème.

1. Écrire une fonction `are_aligned(pos1, pos2)` qui prend en argument deux tuples `pos1` et `pos2` représentant des positions sur le plateau, et qui renvoie un booléen indiquant si deux dames en ces positions se menacent mutuellement (c'est-à-dire `True` si ces positions appartiennent à une même ligne, colonne ou diagonale et `False` sinon).
2. Écrire une fonction `locked_cells(pos)` qui prend en argument un tuple `pos` représentant une position sur le plateau, et qui renvoie un **ensemble** contenant toutes les positions menacées par une dame située en `pos`, y compris elle-même (c'est-à-dire les cases colorées sur la figures ci-dessous).



3. Écrire une fonction `all_cell()` qui renvoie un ensemble contenant toutes les positions possibles sur l'échiquier.
4. Écrire une fonction `solve(queens, free_cells)` qui prend comme arguments :
   — `queens` : un ensemble contenant des positions de reines déjà placées sur l'échiquier,
   — `free_cell` : un ensemble contenant les positions des cases sur lesquelles il est encore possible de placer une dame (c'est-à-dire des cases libres **et** menacées par aucune des reines déjà placées).

   Cette fonction devra alors renvoyer :
   — soit la valeur `None` s'il n'est plus possible de résoudre le problème des huits dames dans ces conditions,
   — soit un ensemble contenant 8 positions valides pour les dames sinon.

   Pour ce faire, on opèrera de manière récursive en suivant les étapes ci-dessous.
   a) On regarde d'abord la taille de l'ensemble `queens`. S'il contient 8 éléments, alors le problème est résolu et on renvoie l'ensemble `queens`.
   b) On regarde ensuite la taille de l'ensemble `free_cells`. S'il est vide, alors le problème n'est plus résoluble et on renvoie la valeur `None`.
   c) Sinon, pour chacune des positions `pos` possibles dans l'ensemble `free_cells` :
      i. on crée un nouvel ensemble `queens2` obtenu en ajoutant aux dames déjà placées une nouvelle dame en position `pos`,
      ii. on crée un nouvel ensemble `free_cells2` contenant seulement les positions qui restent libres une fois cette nouvelle dame placée,
      iii. on regarde le résultat de `solve(queens2, free_cells2)` : si c'est autre chose que `None`, alors on renvoie cette solution, sinon on poursuit l'algorithme.

   Pour résoudre le problème des huit dames, il n'y aura plus qu'à démarrer cet algorithme sur un plateau vide. On appellera pour cela la fonction `solve` avec `queens` l'ensemble vide et `free_cells` l'ensemble contenant toutes les cases du plateau.
5. Bonus : représentez visuellement la solution obtenue de la manière de votre choix (en console, avec `matplotlib`, avec `PIL`, …).

À retenir

— Savoir analyser ou implémenter une fonction *récursive*.
— Savoir distinguer un algorithme *récursif* d'un algorithme *itératif*.
— Savoir expliquer le message d'erreur « *RecursionError : maximum recursion depth exceeded* ».
— Savoir stocker des fonctions dans une collection quelconque (liste, dictionnaire, ensemble, …).
— Savoir écrire une fonction qui prend en argument et/ou renvoie d'autres fonctions.