Structurer les programmes – les fonctions

1 Utilité des fonctions

1.1 Introduction

Au fur et à mesure que les programmes que l'on écrit deviennent plus ambitieux, leur longueur et leur complexité augmentent. Il devient nécessaire de les structurer, de les découper en tâches élémentaires pour faciliter leur écriture et leur maintenance.

Pour illustrer ce découpage, prenons un exemple qui n'a rien à voir avec de l'informatique : faire du café avec une cafetière traditionnelle. C'est une tâche complexe que l'on peut réaliser aisément en la décomposant en tâches plus simples. Ainsi, il nous faudra :

- mettre en place l'eau ;
- mettre en place le filtre et le café ;
- programmer l'appareil pour faire le café.

L'étape de mise en place du café est elle-même décomposable en étapes élémentaires :

- mettre en place un filtre ;
- se procurer du café moulu ;
- placer la bonne quantité de café moulu dans le filtre.

En ce qui concerne le café moulu, il faudra peut-être prendre du café en grain et le moudre. Et ainsi de suite. Bref, si l'on prend en compte les pots que l'on ouvre, l'accès aux placards, etc., c'est à l'arrivée probablement plus d'une centaine d'actes élémentaires que l'on doit effectuer pour parvenir au résultat obtenu.

La décomposition en tâches de plus en plus élémentaires permet de structurer le processus, et de le rendre beaucoup plus facile à appréhender.

Par ailleurs, certaines tâches élémentaires pourront être réutilisées dans d'autres situations. Si vous n'êtes pas dans *votre* cuisine, quelques tâches élémentaires devront être adaptées, mais vous parviendrez quand même à faire du café.

Remarquons aussi que si vous avez besoin d'aller chercher un pot quelconque dans un placard, ce n'est pas une action que vous avez mise en place seulement lorsque vous avez appris à faire du café. Gageons que c'est une tâche qui vous sert dans beaucoup d'activités différentes ! C'est une tâche complexe (ben oui, ouvrir le placard, prendre le pot, fermer le placard…) qui sera réutilisable dans d'autres activités !

Évidemment, un être humain fait cela très naturellement, sans trop y penser. Mais cette démarche peut être utilisée dans bien d'autres cadres avec succès.

En sciences de l'ingénieur, cette méthode de décomposition de tâches en tâches élémentaires est appelée *analyse fonctionnelle descendante* et permet de simplifier grandement la réalisation d'appareils réalisant des tâches complexes, leur modification ou leur maintenance.

1.2 Les fonctions en informatique

En informatique, cette démarche de décomposition en tâche élémentaires est tout aussi indispensable. Plutôt qu'un seul programme de plusieurs centaines de milliers de lignes dans lequel on se perd, on écrira des milliers de petits morceaux de code de quelques dizaines de lignes au plus, qui seront bien plus faciles à écrire, à tester, et à faire évoluer ! C'est le mécanisme des procédures et des fonctions qui se dessine ici.

Certaines de ces actions peuvent parfois nécessiter des « paramètres » : on peut vouloir faire du café pour *5* personnes, aussi faudra-t-il mettre de l'eau pour *5* tasses, et du café pour autant. On n'écrira évidemment pas une fonction qui met de l'eau pour une tasse, une autre fonction pour deux tasses, et ainsi de suite, mais une fonction qui met de l'eau pour n tasse(s), et on précisera le n lorsque l'on effectuera la tâche.

En résumé, la décomposition d'un programme en tâches élémentaires grâce au mécanisme des fonctions répond à deux impératifs :

- *structurer* le programme pour le rendre plus facile à écrire et à maintenir ;
- *modulariser* l'ensemble, car certaines tâches élémentaires pourront servir à plus d'un usage.

Le but de ce chapitre consistera donc à présenter la structure de contrôle permettant de structurer et modulariser les programmes en Python, les *fonctions*. Fonctions dont nous nous somme d'ailleurs déjà servis à plus d'une reprise, en fait ! L'intérêt d'utiliser `sum` (qui se trouve être une fonction) plutôt que de réécrire à chaque besoin une boucle totalisant les éléments d'une liste ne vous a par exemple certainement pas échappé.

2 Premières fonctions

2.1 Définir et appeler une fonction

Une fonction consiste donc en une suite d'instructions, pouvant dépendre d'un ou plusieurs paramètres, que l'on exécutera plus tard autant de fois qu'on le souhaitera. La suite d'instructions, qui sera rangée quelque part dans la mémoire de l'ordinateur, devra être associée à un nom pour que l'on puisse y faire référence ultérieurement.

Il est délicat d'utiliser l'opérateur d'affectation habituel pour écrire une fonction, car écrire

```
nom_de_fonction = suite_d'instructions
```

pourrait faire une ligne très très longue (et très très peu lisible) si les instructions sont nombreuses. En fait, cette manière de créer une fonction existe bien (avec une subtilité

pour préciser que l'on affecte les instructions, et non leur résultat, au nom situé à gauche du signe égal), mais nous n'en parlerons pas pour le moment.

Nous utiliserons la façon « normale » d'associer un nom à une suite d'instruction, qui utilise le mot-clé **def** :

```
def nom_de_fonction(arguments) :
    instruction_1
    instruction_2
    ...
    instruction_n
```

Nous allons clarifier les différents éléments qui interviennent dans les sections suivantes. Mais notons quand même dès à présent l'indentation (qui une fois de plus est provoquée par le signe :) qui permet de savoir *où se termine la séquence d'instructions constituant la fonction.*

2.2 Premier exemple

On suppose dans un premier temps que l'on souhaite dessiner une maison. Pour pouvoir la tracer plusieurs fois, nous allons créer une fonction que l'on appellera autant de fois que nécessaire.

Pour dessiner une maison, il faut dessiner un toit, puis des murs. Nous allons donc définir plusieurs fonctions. Une première pour tracer le toit [1] :

```
def TraceToit() :
    print("  ______  ")
    print(" _UUUUUU_ ")
    print("_UUUUUUUU_")
    print("UUUUUUUUUU")
```

Une seconde fonction pour tracer les murs :

```
def TraceMurs() :
    print("|        |")
    print("| OO  OO |")
    print("|________|")
```

Et une dernière pour tracer la maison en entier :

```
def TraceMaison() :
    TraceToit()
    TraceMurs()
```

1. On en appelle à la clémence du lecteur quant à l'apparence globale desdits tracés...

Vérifions que ce sont bien trois noms tout à fait classiques, qui désignent non des valeurs numériques mais des suites d'instructions :

```
In [1]: TraceMaison
Out[1]: <function __main__.TraceMaison>
```

On n'en apprends pas grand-chose [2] mais au moins, le nom désigne bien une *fonction*. Les fonctions sont des objets comme les autres, et qui donc, nous l'avons déjà vu, peuvent se voir librement associées à d'autres noms, placées dans des conteneurs comme des listes, etc.

Pour « exécuter » la série d'instructions contenus dans une fonction (on parle d'« appel » de la fonction), on utilise fait suivre la fonction (généralement son nom) de parenthèses. Lorsque l'on appelle notre fonction de tracé de maison depuis l'interpréteur, on obtient alors le résultat suivant :

```
In [2]: TraceMaison()
  ______
 _UUUUUU_
_UUUUUUUU_
UUUUUUUUUU
|        |
| OO  OO |
|________|
```

L'usage des parenthèses après le nom de la fonction est ici indispensable : c'est notre façon de dire à Python que l'on veut *exécuter* les instructions constituant la fonction que désigne le nom et non pas savoir simplement ce que le nom désigne.

Les fonctions ainsi définies peuvent être utilisées autant de fois qu'on le souhaite, soit directement, soit à l'intérieur d'autres fonctions, comme c'est ici le cas pour les deux premières dans la définition de la troisième.

Notons que Python vérifie grossièrement la syntaxe du contenu de la fonction lors de sa définition (il pourra vous signaler une erreur de parenthésage même si la fonction n'est pas utilisée) mais le contenu de la fonction n'est exécuté que lors d'un appel à celle-ci !

2.3 Arguments d'une fonction

Une fonction peut prendre un ou plusieurs *arguments* ou *paramètres* qui permettront d'avoir un comportement différent à chaque appel, ce qui en fait tout l'intérêt. Pour définir et utiliser des arguments, on indique dans un premier temps un certain nombre de noms entre les parenthèses dans la définition.

2. La fonction `getsourcelines` du module `inspect`, peut, dans certains cas, redonner les instructions constituant la fonction, mais dans certains cas, Python ne conserve pas les instructions sous une forme humainement lisible, et il n'est pas possible de savoir ce que la fonction contient exactement comme suite d'instructions.

Imaginons par exemple que l'on souhaite dessiner un immeuble comprenant *n* étages et un toit. On écrit alors la fonction suivante :

```
def TraceImmeuble(etages) :
    TraceToit()
    for i in range(etages) :
        TraceMurs()
```

On appelle la fonction toujours avec son nom, et en plaçant entre les parenthèses une valeur qui sera communiquée à la fonction, par exemple 2 comme ci-dessous :

```
In [3]: TraceImmeuble(3)
  ______
 _UUUUUU_
_UUUUUUUU_
UUUUUUUUUU
|        |
| OO  OO |
|________|
|        |
| OO  OO |
|________|
|        |
| OO  OO |
|________|
```

En fait, lorsque la fonction est appelée, l'interpréteur associe les valeurs passées en paramètres et les noms présents dans la définition. Tout se passe comme si, avant même les premières instructions de la fonction, on avait l'affectation `etages=3`.

Si l'on passe en paramètre des noms ou des expressions plutôt que directement des valeurs, alors celles-ci sont évaluées avant que l'affectation n'ait lieu.

2.4 Notion de portée

L'affectation `etages=3`, qui a eu lieu lors de l'appel précédent, n'a duré que le temps de la fonction. Dès que l'on sort de la fonction, elle est oubliée :

```
In [4]: etages
NameError: name 'etages' is not defined
```

Il en est de même pour le nom utilisé par la boucle **for** :

```
In [5]: i
NameError: name 'i' is not defined
```

Par ailleurs, si un nom `etages` existait avant l'appel à la fonction, il existera toujours après l'appel, et sa valeur n'aura pas été perturbée par l'appel, comme ci-dessous :

```
In [6]: etages=5

In [7]: TraceImmeuble(3)
(snip)

In [8]: etages
Out[8]: 5
```

Le nom `etages` désigne donc la valeur 3 à l'*intérieur* de la fonction, mais *uniquement à l'intérieur de la fonction*. En fait, ce n'est pas le même nom pour Python : le `etages` de la fonction est un nom *local* à la fonction. Même si la fonction le réaffecte, cela n'aura pas de conséquences à l'extérieur de la fonction. Plus généralement, tous les noms qui font l'objet d'une affectation à l'intérieur d'une fonction sont des noms *locaux* qui cessent d'exister lorsque l'on en sort[3] :

```
In [9]: def foo() :
   ...:     x = 2
   ...:     print(x, x+3)

In [10]: foo()
2 5

In [11]: x
NameError: name 'x' is not defined
```

2.5 Noms globaux

En fait, à l'intérieur d'une fonction, s'il n'y a pas un nom *local* identique, on peut accéder aux noms définis à l'extérieur de la fonction, comme dans l'exemple ci-dessous :

```
In [12]: def bar() :
    ...:     print(z)

In [13]: z = 3

In [14]: bar()
Out[14]: 3
```

3. Pour la petite histoire, il est fréquent en informatique d'utiliser comme nom de fonction, faute d'un meilleur nom, `foo` et `bar` ; une habitude dont l'histoire est compliquée, mais qui serait notamment passée par un club du MIT qui, possiblement, se serait inspiré du terme d'argot militaire FUBAR signifiant « bousillé au point d'être méconnaissable » (Fucked-Up Beyond All Recognition).

Toutefois, **c'est une très mauvaise habitude en général**, et il est très vivement recommandé que **toutes les grandeurs dont la fonction peut avoir besoin soient passées via ses arguments**. Cela aidera beaucoup la personne qui vous relira (laquelle peut être vous-même dans deux ans) à savoir ce dont la fonction a besoin. Sans cela, il n'est pas évident que vous vous souveniez qu'il est nécessaire qu'une variable z existe lorsque l'on appelle ladite fonction.

Un nom défini en-dehors de la fonction est appelé nom *global* par opposition aux noms locaux définis dans la fonction. En règle générale, il n'est pas possible de réaffecter un nom global à l'intérieur d'une fonction, puisque la présence d'une affectation pour ce nom dans la fonction crée un nom local [4].

2.6 Retour sur les appels et paramètres

Si l'on souhaite avoir plus d'un paramètre pour une fonction, on peut indiquer plusieurs noms séparés par des virgules :

```
def foo(x, y) :
    print(x, "^", y, "=", x**y)
```

Lors de l'appel, on indique autant de valeurs ou d'expressions qu'il y a de paramètres :

```
In [15]: foo(1+1, 6//2)
2 ^ 3 = 8
```

L'interpréteur Python commence par déterminer la valeur de chaque expression à l'intérieur des parenthèses (2 et 3 ici), puis appelle la fonction proprement dite, en associant aux noms locaux x et y les valeurs 2 et 3 dans le cas présent.

Considérons à présent l'exemple suivant :

```
In [16]: x = 3

In [17]: z = 5

In [18]: foo(z, x)
5 ^ 3 = 125
```

Examinons ce qui se passe en détail lors de l'appel :

- l'interpréteur détermine la valeur de chaque expression passée en paramètres, donc `foo(z, x)` est traduite en `foo(5, 3)` ;
- PUIS l'interpréteur cherche ce que désigne le nom `foo`, qui se trouve bien être une fonction dont la définition est « `def foo(x, y)` » ; il crée des noms *locaux* x et y et leur affecte respectivement les valeurs 5 et 3 ;
- enfin, l'interpréteur exécute les instructions constituant la fonction que désigne le nom `foo`.

Il est important de comprendre que les noms utilisés à l'intérieur de la fonction n'ont absolument pas besoin d'être identiques aux noms utilisés à l'extérieur de la fonctions. Python détermine la *valeur* de chacun des paramètres, et les associe aux noms locaux dans l'ordre dans lequel ils apparaissent... sauf exception...

2.7 Paramètres nommés, noms locaux

Lorsque l'on appelle une fonction, l'ordre dans lequel sont passés les paramètres a évidemment de l'importance, puisque les affectations des noms locaux sont faites normalement dans l'ordre. Ce n'est pas toujours pratique, car lors de l'appel, on peut avoir oublié dans quel ordre se trouvent les paramètres.

Python fournit un mécanisme intéressant pour simplifier la tâche aux programmeurs : on peut placer les arguments de la fonction dans n'importe quel ordre si l'on indique le nom local de l'argument avant la valeur. Pour être plus clair, on peut écrire :

```
In [19]: foo(x=2, y=3)
2 ^ 3 = 8
```

mais également :

```
In [20]: foo(y=3, x=2)
2 ^ 3 = 8
```

Dans le second cas, les paramètres ne sont pas passés dans l'ordre, mais puisque l'on précise à quel nom local est associé chacune des valeurs, l'interpréteur Python saura s'y retrouver !

Considérons enfin un dernier exemple :

```
In [21]: x = 3

In [22]: z = 5

In [23]: foo(y=x, x=z)
5 ^ 3 = 125
```

Il faut lire sur cet exemple : dans la fonction, il faut affecter au nom local x la valeur désignée par z, soit 5, et au nom local y la valeur désignée par x, soit 3. On s'efforcera d'éviter de tels cas, quelque peu pathologiques !

Comme utiliser des paramètres nommés permet de mélanger l'ordre des arguments, *à partir du moment où l'on utilise un paramètre nommé, tous les paramètres qui apparaissent à sa droite doivent l'être* !

4. Il existe une manière de contourner ce problème, mais puisque ce genre de pratique est très déconseillée, nous ne la détaillerons pas ici.

Beaucoup de fonctions standard de Python n'ont pas de nom affectés aux différents arguments, aussi n'est-il pas possible, pour celles-ci, de fournir les arguments de la sorte.

À l'inverse, certains arguments doivent *impérativement* être des arguments nommés. Vous en connaissez déjà : les arguments nommés sep et end de la fonction `print`, par exemple (la raison ici est que la fonction `print` acceptant un nombre quelconque d'arguments, Python aurait eu du mal à savoir les différencier). Nous avons déjà vu que l'on pouvait les mettre dans n'importe quel ordre, mais que l'on ne pouvait rien ajouter à leur suite. En voilà donc la raison !

2.8 Types des paramètres

Les arguments d'une fonction peuvent être de n'importe quel type utilisé par l'interpréteur Python : des valeurs numériques, comme on l'a vu pour l'instant, mais également des listes, des chaînes de caractères, ou même des fonctions ou des modules !

Par exemple, la fonction `Trace` définie par :

```
from matplotlib.pyplot import plot, show

def Trace(f, intervalle, n) :
    pas = (intervalle[1]-intervalle[0]) / (n-1)
    x = [ intervalle[0] + pas*i for i in range(n) ]
    y = [ f(xi) for xi in x ]

    plot(x, y)
    show()
```

attend comme paramètres une fonction, une liste et un entier. Elle trace alors le graphe de la fonction sur l'intervalle en question. Par exemple, pour afficher le graphe de $\sin(x)$ sur $[0,5]$, on peut appeler la fonction de la façon suivante :

```
In [24]: from math import sin

In [25]: Trace(sin, [0, 5], 100)
```

3 Des procédures aux fonctions

3.1 Retourner un résultat

Pour l'instant, les fonctions que l'on a écrites effectuent quelque chose (principalement un affichage), ce que l'on appelle un *effet de bord de la fonction*, mais ne retournent aucun résultat, contrairement aux fonctions en mathématiques. C'est ce que l'on appelle parfois en informatique des *procédures*.

Pour retourner un résultat, il suffit de placer dans la fonction le mot-clé **`return`** suivi de la valeur à renvoyer (qui peut se présenter sous la forme d'une expression).

Nous avons déjà étudié une fonction permettant de calculer et d'afficher x^y :

```
def Foo(x, y) :
    print(x**y)
```

Si l'on veut *retourner* la valeur x^y, on écrira [5]

```
def Bar(x, y) :
    return x**y
```

Pour comprendre la différence entre ces deux fonctions, essayons-les dans l'interpréteur :

```
In [26]: Foo(2, 3)
8

In [27]: Bar(2, 3)
Out[27]: 8
```

La fonction `Foo` *affiche* la valeur 8 et ne retourne rien.

La fonction `Bar`, elle, *retourne* la valeur 8 sans rien afficher.

La différence est encore plus évidente lorsque l'on essaie de faire quelque chose de l'éventuelle valeur retournée :

```
In [28]: x = Foo(2, 3)
8

In [29]: x
Out[29]: None

In [30]: x = Bar(2, 3)

In [31]: x
Out[31]: 8
```

Dans le premier cas, `x` reçoit le résultat de l'appel à `foo`, ou pour être plus précis l'absence de résultat (que Python représente par la valeur particulière « **`None`** » [6]). Le 8 apparaît car c'est un affichage.

Dans le second cas, `x` reçoit le résultat de l'appel à `bar`, soit 8. Il n'apparaît rien lors de l'appel car il n'y a pas d'affichage.

5. **`return`** étant un mot-clé et non une fonction, il n'est pas besoin de parenthèses !
6. En fait, toutes les fonctions en Python doivent retourner un résultat ; la valeur particulière **`None`** est automatiquement retournée si l'on termine la fonction sans trouver d'instruction **`return`**.

Même chose si le résultat est utilisé dans un calcul :

```
In [32]: sin(Foo(2, 3))
8
TypeError: a float is required

In [33]: sin(Bar(2, 3))
Out[33]: 0.9893582466233818
```

En général, un **return** pour retourner un résultat à l'utilisateur est très préférable à un `print` qui se contente de faire un affichage. Si le but est d'afficher au résultat, c'est de préférence le morceau de code qui effectue l'appel qui devrait se charger d'afficher le retour de la fonction, pas la fonction elle-même.

Il est à noter que l'exécution de la fonction cesse *aussitôt* que l'on rencontre un **return**. Ainsi :

```
In [34]: def Test() :
    ...:     print("foo")
    ...:     return 42
    ...:     print("bar")

In [35]: Test()
foo
Out[35]: 42
```

Seule la première instruction `print` est effectuée puisque l'on sort de la fonction lorsque l'on parvient au **return**.

Une même fonction peut contenir plusieurs instructions **return**, comme dans cette fonction « valeur absolue » :

```
def Absolue(x) :
    if x>=0 :
        return x
    else :
        return -x
```

Si l'on parvient à la fin d'une fonction sans avoir rencontré de **return**, la fonction renverra « **None** », puisqu'elle doit toujours renvoyer quelque chose, dans l'hypothèse où un utilisateur essaierait d'associer un nom au résultat de la fonction.

Contrairement à ce qui se passe dans certains langages, il est possible pour une fonction de retourner des résultats de types différents, selon par exemple quels arguments ont été fournis à la fonction.

Il est dès lors très important d'avoir une idée claire de ce que la fonction attend comme arguments, de ce qu'elle fait, et des résultats qu'elle retourne !

N'importe quel objet que l'interpréteur Python peut manipuler peut ainsi être retourné par une fonction. Par exemple une liste :

```
def Foo(n) :
    return [n+1] * n
```

Auquel cas, le résultat d'un appel de la fonction, par exemple « `Foo(3)` » correspond à une liste, et peut s'utiliser comme une liste :

```
In [36]: Foo(3)
Out[36]: [ 4, 4, 4 ]

In [37]: Foo(3)[2]+2
Out[37]: 6
```

Ou bien une fonction :

```
def Bar(x) :
    if x>0 :
        return sin
    return cos
```

Ce qui donne :

```
In [38]: Bar(-1)
Out[38]: <function math.cos>

In [39]: Bar(-1)(0.0)
Out[39]: 1.0
```

3.2 Résultats multiples

Une conséquence de la sortie immédiate lorsque l'on rencontre un **return** est que l'on n'a qu'une unique opportunité pour renvoyer un résultat. Si l'on veut retourner deux résultats, on peut par exemple retourner un *couple* de valeurs [7] (ou bien une liste) :

```
def Foo(x) :
    return x+1, 2*x
```

Cette fonction peut être utilisée de plusieurs façons. Interactivement :

```
In [40]: Foo(5)
Out[40]: 6, 10
```

7. En Python, un tel couple est en fait appelé *tuple*.

En assignant le résultat à un nom :

```
In [41]: res = Foo(5)

In [42]: res
Out[42]: 6, 10

In [43]: res[0]
Out[43]: 6
```

Ici, x reçoit directement le *couple* des deux valeurs retournées. On peut ensuite accéder aux valeurs individuellements comme avec une liste, en écrivant `x[0]` et `x[1]`.

On peut aussi assigner chacun des deux éléments du couples à deux noms distincts :

```
In [44]: x, y = Foo(5)

In [45]: x
Out[45]: 6

In [46]: y
Out[46]: 10
```

3.3 Message d'aide

On peut inclure un message d'aide dans la définition d'une fonction, accessible via la fonction `help`. **C'est une excellente habitude à prendre !** Il suffit de le placer immédiatement après la déclaration de la fonction, encadré par des triples guillemets [8] :

```
def Foo() :
    """Ne prend pas d'argument
       et retourne la valeur 42"""

    return 42
```

Ce qui permet ensuite de demander de l'aide concernant la fonction :

```
In [47]: help(Foo)
Help on function foo in module __main__:

Foo()
    Ne prend pas d'argument
    et retourne la valeur 42
```

8. Ce qui permet d'aller librement à la ligne et donc d'inclure des messages d'aide sur plusieurs lignes.

4 Quelques exemples

4.1 PGCD et PPCM

Reprenons l'exemple du calcul du PGCD du second chapitre. On peut écrire la fonction correspondante de la façon suivante :

```
def PGCD(a, b) :
    """Retourne le PGCD des entiers positifs a et b"""
```

On peut écrire une seconde fonction retournant son pendant, le PPCM (plus petit commum multiple), très simplement en utilisant la fonction précédente :

```
def PPCM(a, b) :
    """Retourne le PPCM des entiers positifs a et b"""
```

4.2 Factorielle

La factorielle d'un entier n strictement positif, notée $n!$, est définie par :

$$n! = \prod_{k=1}^{n} k = 1 \times 2 \times 3 \times \ldots \times (n-1) \times n$$

On peut aisément écrire cette fonction en quelques lignes :

```
def Fact(n) :
    """Retourne la factorielle de l'entier positif n"""
```

La plupart de ces fonctions existent en fait déjà dans le module `math`. Le PGCD est la fonction `math.gcd`, la factorielle la fonction `math.factorial`.

4.3 Une autre version de `sum`, `max` et `list.count`

Il est également aisé d'écrire nos propres versions de fonctions retournant la somme des éléments d'une liste, le plus grand élément d'une liste, ou le nombre d'occurrences d'une valeur dans une liste :

```
def Somme(liste) :
    """Retourne la somme des éléments de la liste"""
```

```
def Maximum(liste) :
    """Retourne le plus grand élément de la liste"""
```

```
def CompteOccurrence(liste, valeur) :
    """Retourne le nombre d'occurrences de valeur dans liste"""
```

4.4 Théorème de Pythagore

Supposons que l'on connaisse les longueur a et b de deux des côtés d'un triangle rectangle (non aplati). On souhaite écrire une fonction retournant une liste de toutes les valeurs possibles pour la longueur du troisième côté :

```
def Pythagore(a, b) :
    """Retourne la liste des longueurs possibles
    pour le troisième côté d'un triangle non dégénéré
    ayant un angle droit et deux côtés de longueurs a et b"""
```

5 Recherches dans une liste

5.1 Présentation du problème

Nous allons aborder à présent chercher à déterminer si un élément est présent dans une liste. On souhaite ainsi écrire une fonction `Contient` qui accepterait deux arguments, `elem` et `liste`, et qui retournerait **`True`** si `elem` est présent dans `liste`, et **`False`** dans le cas contraire.

Évidemment, on pourrait écrire la fonction de la sorte [9] :

```
def Contient(elem, liste) :
    """Retourne True si elem est dans liste et False sinon"""

    return elem in liste
```

Mais nous allons présentement essayer de l'implémenter sans utiliser le mot-clé « **`in`** », afin de mieux comprendre comment les choses se passent.

5.2 Recherche linéaire

Si l'on n'a aucune information particulière concernant la liste, la seule solution consiste à examiner les éléments de la liste un par un, afin de voir si l'élément est celui recherché.

9. Ce qui est parfaitement correct et même souhaitable, en l'absence d'informations sur le contenu de la liste.

On peut ainsi écrire les choses de la sorte :

```
def Contient(elem, liste) :
    """Retourne True si elem est dans liste et False sinon"""

    present = False

    for e in liste :
        if e == elem :
            present = True

    return present
```

Cette première solution n'est pas très bonne, car si l'élément est présent dès le début de la liste, on parcourt quand même inutilement le reste de la liste. Une meilleure solution serait :

```
def Contient(elem, liste) :
    """Retourne True si elem est dans liste et False sinon"""

    present = False

    i = 0
    while not present and i<len(liste) :
        if liste[i] == elem :
            present = True

    return present
```

Toutefois, il existe une autre solution, potentiellement plus simple. Dès que l'exécution de la fonction tombe sur une instruction **return**, l'exécution de la fonction cesse immédiatement, et la valeur fournie est retournée. En combinant cette propriété et le fait que l'on puisse avoir plusieurs **return** dans une même fonction, on peut préférer écrire :

```
def Contient(elem, liste) :
    """Retourne True si elem est dans liste et False sinon"""

    # On examine chaque élément de la liste
    for e in liste :
        if e == elem :
            # L'élément vient d'être trouvé, on retourne True
            return True

    # L'élément n'a toujours pas été trouvé à l'issue de la boucle
    return False
```

Précisons quand même que la présence de nombreux **return** dans une même fonction, surtout s'ils sont disséminés et que la fonction est longue, peut rendre la lecture et la compréhension de la fonction plus difficile, donc pensez aux commentaires.

5.3 Cas d'une liste triée – recherche dichotomique

Présentation de l'algorithme

La fonction précédente nécessite de vérifier tous les éléments un par un si l'élément n'est pas présent dans la liste, et potentiellement un très grand nombre d'éléments, surtout s'il se situe vers la fin de la liste.

Lorsque l'on cherche un mot dans le dictionnaire, ou un nom dans un annuaire, heureusement, on n'a pas besoin de tout lire à chaque fois. En effet, ces ouvrages contiennent des données *triées*, et donc un élément donné ne peut pas se trouver n'importe où.

Nous allons à présent essayer de vérifier, plus efficacement que précédemment, si un élément est présent dans une liste *triée* (on supposera un ordre croissant).

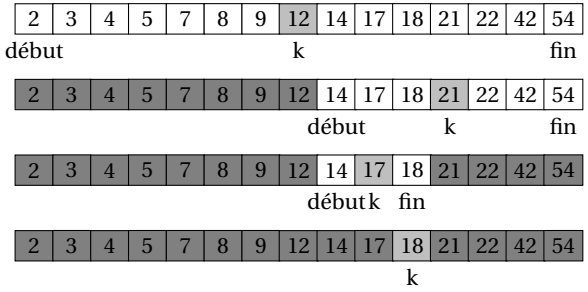
Comment fait-on pour rechercher un mot dans le dictionnaire ? On l'ouvre à peu près vers l'endroit où l'on pense que le mot se trouve ; puis, si les mots que l'on trouve se situent avant le mot recherché, on regarde quelques pages après, et ainsi de suite.

L'idée ici est de faire pratiquement la même chose : on regarde l'élément d'indice k, au milieu de la liste.

- si, par chance, `L[k]==elem`, on en a terminé ;
- si `L[k]>elem`, il faudra chercher `elem` parmi les éléments d'indice `0` à `k-1`.
- si `L[k]<elem`, il faudra chercher `elem` parmi les éléments d'indice `k+1` à `N-1`.

Dans les deux derniers cas, il nous faut chercher l'élément dans une des deux moitiés de la liste. On prend donc l'élément au milieu de la moitié correspondante, et on recommence.

Illustrons cet algorithme sur un exemple : est-ce que 18 figure dans la liste suivante ?



À chaque étape, on élimine au moins une moitié des éléments de la liste. Il suffit donc ici de quatre tests pour trouver le 18 dans la liste. Une recherche sur 19 conduirait aux quatre mêmes tests, pour conclure cette fois que 19 n'est pas dans la liste. Soit bien moins des quinze tests nécessaires à une recherche linéaire.

Implémentation

Comment implémenter cet algorithme ? Il faut savoir, à chaque étape, dans quelle partie de la liste il nous reste à chercher notre élément. On définira donc deux noms, `debut` et `fin`, qui indiqueront parmi quels indices `i` vérifiant `debut`⩽`i`⩽`fin` l'élément recherché est susceptible de se trouver.

Initialement, on prendra donc `debut=0` et `fin=len(L)-1`.

Le calcul de l'indice k, correspondant au milieu d'une plage d'indices donnée, est également simple : il suffit de prendre $k = \left\lfloor \frac{\text{debut+fin}}{2} \right\rfloor$, soit en langage Python `k=(debut+fin)//2`.

Le reste s'écrit tout naturellement. On s'arrête lorsqu'il n'y a plus *aucun* élément entre `debut` et `fin`, c'est-à-dire lorsque `debut>fin` (en effet, si `debut=fin`, il reste encore un élément à traiter !) :

```
def ContientDicho(elem, liste) :
    """Retourne True si elem est dans liste et False sinon
       liste doit être triée par valeurs croissantes"""

    # Initialement, tous les indices sont à considérer
    debut, fin = 0, len(liste)-1

    # Tant qu'il reste des emplacements possibles
    while debut <= fin :
        # On détermine le milieu de la plage d'indices
        k = (debut + fin) // 2

        # Et on compare l'élément qui s'y trouve à elem
        if liste[k] == elem :
            return True
        elif liste[k] > elem :
            fin = k-1
        else :
            debut = k+1

    # S'il ne reste plus de possibilités, elem n'est pas dans liste
    return False
```

L'algorithme de la recherche dichotomique est explicitement cité au programme, et il est important que le compreniez parfaitement et que vous sachiez le réécrire. Méfiez-vous de son apparente simplicite, le diable se cache dans les détails et on a tôt fait de commettre une petite erreur (comparaison stricte, dans l'expression booléenne du while, mauvaise mise à jour des indices, division flottante et non pas entière, etc.)

Pour la petite histoire, une expérience menée chez Bell (une entreprise pionnière de l'informatique, qui a créé entre autres choses un langage qui est l'ancêtre du langage C), dans laquelle on a laissé plusieurs heures à des programmeurs professionnels pour programmer une recherche dichotomique. Seulement 10% d'entre eux ont réussi à proposer un code sans aucune erreur (certes dans un langage un peu moins accessible que Python).

Donald Knuth signale d'ailleurs que si l'algorithme a été proposé en 1946, il a fallu attendre 1962 pour en voir la première implémentation correcte.

Évaluation du nombre maximal de tests

Si la liste contient 1 élément, il faudra évidemment un seul test pour conclure. Si la liste contient 3 éléments ou moins, la méthode dichotomique nécessitera au plus deux tests.

Si avec n tests on peut traiter une liste de longueur au plus l_n, alors avec $n+1$ tests, on pourra traiter une liste de longueur au plus $l_{n+1} = 2l_n + 1$. On peut aisément montrer, puique $l_1 = 1$ et $l_2 = 3$, que cela correspond à $l_n = 2^n - 1$.

Ainsi, pour une liste de longueur l, il suffira de k tests où k vérifie $l \leq 2^k - 1$, soit $\ln(l+1) \leq k\ln(2)$, donc $k \geq \ln_2(l+1)$.

Le nombre minimal de tests à effectuer pour déterminer si une valeur est présente dans une liste *triée* de longueur l est donc $\lceil \ln_2(l+1) \rceil$.

Pour une recherche dans un dictionnaire français courant (qui contient de l'ordre de 60000 mots), il suffirait donc, en principe, de vérifier dix-huit mots pour pouvoir conclure quant à la présence d'un mot donné dans ledit dictionnaire avec l'algorithme de la recherche dichotomique.

Pourquoi couper au milieu ?

On peut se demander pourquoi couper au milieu plutôt qu'ailleurs. Supposons que la liste contient initialement N éléments, et examinons la première coupure.

La chance de tomber « par hasard » sur la bonne valeur ne dépend pas de l'élément choisi, donc on ne s'intéressera qu'à ce qui se passe si l'élément choisi n'est pas celui recherché.

Les indices vont de 0 à N − 1. Choisissons l'élément d'indice k pour couper la liste en deux, pas nécessairement au milieu. On suppose donc que l'élément d'indice k n'est pas l'élément recherché. On se retrouve avec deux listes, l'une contenant k éléments, l'autre contenant $N - k - 1$ éléments.

L'élément recherché pouvant être n'importe où, la probabilité de le trouver dans chacune de ces deux listes est directement proportionnelle au nombre d'éléments dans chacune des deux listes.

L'espérance du nombre d'éléments qui restent à examiner ensuite peut donc s'écrire

$$E = k \times \frac{k}{N-1} + (N-k-1) \times \frac{N-k-1}{N-1} = \frac{k^2 + (N-k-1)^2}{N-1}$$

On souhaite évidemment que la liste qui reste à examiner soit la plus courte possible, donc on veut minimiser cette espérance, donc choisir le k qui minimise $k^2 + (N-k-1)^2$.

La dérivée par rapport à k de cette expression est $2 \times k - 2 \times (N-k-1)$, ce qui donne $4k - 2(N-1)$. Cette expression s'annule pour $k = \frac{N-1}{2}$. Puisque la dérivée seconde est positive, cette valeur correspond bien à un minimum de l'espérance.

k devant être entier, on effectuera au besoin un arrondi à un des deux entiers les plus proches (ce que fait la division entière dans l'algorithme).

6 Quelques remarques pour clore

6.1 Arguments mutables

Lorsque l'on travaille avec des paramètres dans une fonction Python, nous l'avons vu, l'interpréteur associe un nom local à l'objet passé en paramètre. Si ce nom est réaffecté, cela n'influe donc pas l'objet lui-même. En général, ce qui se passe dans la fonction n'a pas d'influence sur les objets définis à l'extérieur de la fonction.

Toutefois, si l'on effectue, à l'intérieur d'une fonction, une *mutation* d'un objet qui a été passé en argument de la fonction, cela a naturellement un effet hors de la fonction :

```
In [48]: def foo(L) :
    ...:     L[0] = 17
    ...:     L = [ 10, 23 ]
    ...:     L[1] = 42
    ...:     print("L =", L)

In [49]: P = [ 4, 9 ]

In [50]: foo(P)
L = [10, 42]

In [51]: P
Out[51]: [17, 9]
```

Dans cet exemple, le nom local `L` désigne la *même* liste, à l'entrée de la fonction, que celle désignée par le nom `P`. La première mutation sur la liste désignée par `L` a donc un effet sur celle désignée par `P`.

La seconde instruction de la fonction *réaffecte* ensuite le nom local `L`, de sorte qu'ensuite `L` et `P` ne désignent plus les mêmes listes. Les mutations sur la liste désignée par `L` n'ont alors plus d'influence sur celle désignée par `P`.

Attention donc aux objets *mutables* (comme les listes) passés en paramètre ! En général, lorsque l'on appelle une fonction, on ne s'attend pas à ce que la fonction modifie les objets passés en paramètres, excepté lorsque c'est explicitement le but de la fonction. Aussi peut-il être utile d'effectuer une copie des paramètres mutables dès le début de la fonction, avant d'effectuer des opérations sur ces objets.

6.2 Paramètres par défaut

Lorsque l'on définit une fonction, il est possible de spécifier des valeurss « par défaut » pour tout ou partie des paramètres. Si, lorsque l'on appelle la fonction, on ne spécifie pas de valeur pour l'argument en question, la valeur par défaut est utilisée [10]. Par exemple :

```
In [52]: def pow(x, y=2) :
    ...:     print(x, '^', y, '=', x**y)

In [53]: pow(3, 4)
3 ^ 4 = 81

In [54]: pow(3)
3 ^ 2 = 9
```

Dans le second appel, dans l'exemple ci-dessus, le second paramètre étant manquant, l'interpréteur Python a utilisé la valeur `2` pour `y`.

Par exemple, la fonction `print` que l'on a déjà vu définit pour valeur par défaut pour `sep` une espace (`' '`), et pour `end` un retour à la ligne (`'\n'`). Ainsi, l'utilisateur n'a pas à définir une valeur pour ces deux paramètres si une espace pour séparer les objets à afficher et un retour à la ligne lorsque l'on a terminé lui convient.

```
In [55]: help(print)
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

Évidemment, si l'on omet un paramètre, il faut omettre les suivants également (ou utiliser des paramètres nommés) pour que l'interpréteur s'y retrouve ! Il ne peut en effet pas deviner quels paramètres ont été omis.

10. Précisons que la valeur par défaut est construite une seule fois lors de la définition de la fonction, et réutilisée à chaque appel, et n'est pas reconstruite lors de chaque appel, ce qui peut avoir parfois de l'importance.

Exercices

Ex. 1 – Jours, heures, minutes et secondes

Proposer une fonction `JHMS(n)` qui prend en argument un nombre (entier) `n` de secondes et affiche le temps correspondant en jours, heures, minutes et secondes. Par exemple, `JHMS(97250)` devra afficher `1j 3h 0min 50s`.

Ex. 2 – Déplacement d'une reine d'échecs

On s'intéresse à un échiquier sur lequel est posé une unique reine sur la case (i_1, j_1). On rappelle qu'une reine aux échecs peut se déplacer d'un nombre quelconque de cases en ligne, en colonne, ou en diagonale.

On souhaite savoir en combien de déplacements la reine peut atteindre une case (i_2, j_2). Proposer une fonction `NbCoups(i1, j1, i2, j2)` retournant un entier correspondant au nombre de déplacements nécessaires pour aller d'une case à l'autre.

Ex. 3 – Moyenne et médianne

1. On dispose de deux entiers x et y. Proposer une fonction `Z(x, y)` retournant le plus petit entier z tel que la moyenne de l'ensemble $\{x, y, z\}$ et la médiane de ce même ensemble (définie comme l'élément qui, parmi les trois, se retrouvait à la position centrale s'ils étaient rangés par ordre croissant) soient égales.

2. On suppose à présent x et y positifs. Modifier la fonction `Z(x, y)` de sorte qu'elle retourne le plus petit entier positif z tel que la moyenne de l'ensemble $\{x, y, z\}$ et la médiane de ce même ensemble soient égales.

3. Enfin, on considère quatre entiers positifs v, w, x et y. Proposer une fonction `Z(v, w, x, y)` qui retourne le plus petit entier positif z tel que la moyenne et la médiane de l'ensemble $\{v, w, x, y, z\}$ soient égales.

Ex. 4 – Insertion dans une liste triée

Approche dichotomique

On suppose que `L` désigne une liste contenant des valeurs triées par ordre croissant. On souhaite déterminer où peut être inséré un élément supplémentaire `x` (si plusieurs positions sont possibles, on choisira la plus à droite possible).

On cherche donc à déterminer le plus petit indice `i` vérifiant `L[i]>x`. Si tous les éléments de `L` sont inférieurs ou égaux à `x`, on retournera la longueur de `L` puisque `x` puisqu'il faudrait alors l'insérer à l'extrémité droite de la liste.

Afin de déterminer cet indice `i` (dans le cas où `x` ne doit pas se retrouver à la fin de la liste), on se propose d'utiliser une méthode dichotomique. On utilise pour ce faire une approche dichotomique, et on définit deux noms `debut` et `fin`, de sorte qu'à tout instant

- l'indice `i` recherché vérifie `debut`$\leqslant$`i<fin`;
- `L[fin-1]>x`.

1. Comment éliminer le cas particulier où `x` doit se retrouver à la fin de la liste ? On suppose dans la suite que l'on ne se trouve pas dans ce cas.

2. Quels doivent être les valeurs initiales de `debut` et `fin` ? On justifiera que les deux conditions précédentes sont bien vérifiées.

3. À quelle condition sur `debut` et `fin` la recherche dichotomique doit-elle s'arrêter ? Comment en déduit-on alors le `i` recherché ?

4. Si cette condition n'est pas vérifiée, proposer une façon de restreindre l'intervale dans lequel on recherche `i`.

5. Rédiger l'algorithme utilisant le principe de la dichotomie retournant l'indice `i` recherché.

L'élément `x` est alors inséré avec l'instruction `L.insert(i, x)`

Approche utilisant des permutations

On suppose que `L` désigne une liste contenant des valeurs triées par ordre croissant. On souhaite ajouter dans cette liste un élément `x`, de sorte que la liste reste triée. On se propose de procéder de la façon suivante :

- on place l'élément `x` à la fin de la liste ;
- tant que ledit élément est précédé, dans la liste, par un élément qui lui est strictement supérieur, on permute les deux éléments en question.

6. Proposer un programme Python implémentant l'algorithme précédent et comparer son efficacité (en terme de temps d'exécution) avec l'approche précédente.

Ex. 5 – Point fixe

Proposer un programme déterminant, étant donné une liste `L` d'entiers distincts triés par ordre croissant, s'il existe au moins un index `i` vérifiant `L[i] == i`.

On cherchera à proposer une solution efficace (effectuant de l'ordre de $\log(n)$ comparaisons si n est le nombre d'éléments dans liste).

Ex. 6 – Grandes puissances

Proposer une fonction `UnitePuissance(p, q)` qui prend en argument deux entiers positifs `p` et `q` et retourne le chiffre des unités de p^q. `p` et `q` pouvant être grands, on souhaite obtenir le résultat *sans calculer explicitement* p^q !

Indication : pour obtenir un programme réellement rapide, même pour de très grands entiers, il peut être intéressant de se s'intéresser dans un premier temps à la valeur de p^9...