VINCENT LE GOFF

APPRENEZ À PROGRAMMER EN PYTHON

DÉVELOPPER EN PYTHON N'A JAMAIS ÉTÉ AUSSI FACILE !

le Livre du Zéro

www.siteduzero.com

Simple IT 2011 - ISBN : 979-10-90085-03-9

***ATTENTION** : Ce document ne contient qu'une partie des chapitres du livre « Apprenez à programmer en Python», de V. Le Goff. Il s'agit d'une sélection de moins de 100 pages, n'abordant que les aspects de python liée au cours de Modélisation et Robotique. L'intégralité du texte peut être trouvé en ligne.*

Yann Chevaleyre

Chapitre 1

Qu'est-ce que Python ?

Python est un langage puissant, à la fois facile à apprendre et riche en possibilités. Dès l'instant où vous l'installez sur votre ordinateur, vous disposez de nombreuses fonctionnalités intégrées au langage que nous allons découvrir tout au long de ce livre.

Il est, en outre, très facile d'étendre les fonctionnalités existantes, comme nous allons le voir. Ainsi, il existe ce qu'on appelle des **bibliothèques** qui aident le développeur à travailler sur des projets particuliers. Plusieurs bibliothèques peuvent ainsi être installées pour, par exemple, développer des interfaces graphiques en Python.

Concrètement, voilà ce qu'on peut faire avec Python :

- de petits programmes très simples, appelés **scripts**, chargés d'une mission très précise sur votre ordinateur ;
- des programmes complets, comme des jeux, des suites bureautiques, des logiciels multimédias, des clients de messagerie...
- des projets très complexes, comme des progiciels (ensemble de plusieurs logiciels pouvant fonctionner ensemble, principalement utilisés dans le monde professionnel).

Voici quelques-unes des fonctionnalités offertes par Python et ses bibliothèques :

- créer des interfaces graphiques ;
- faire circuler des informations au travers d'un réseau ;
- dialoguer d'une façon avancée avec votre système d'exploitation ;
- ... et j'en passe...

Bien entendu, vous n'allez pas apprendre à faire tout cela en quelques minutes. Mais ce cours vous donnera des bases suffisamment larges pour développer des projets qui pourront devenir, par la suite, assez importants.

Pour utiliser Python chez vous, vous avez trois options (de la plus simple à la plus complexe)

- Vous jouer avec Python en ligne, sur l'un des sites suivants :
  https://repl.it/
  https://www.brython.info/tests/console.html?lang=fr

- Vous pouvez installer Python sur votre ordinateur, je vous conseille pour cela d'aller sur le site suivant : https://www.continuum.io/downloads
  et de cliquer sur « Graphical Installer » sous « Python 3.5 version »

- Vous pouvez enfin installer l'environnement que nous utilisons en TP. Pour cela, il faut installer virtualbox (disponible ici : https://www.virtualbox.org/), puis télécharger l'image Linux contenant une installation propre de Python à cette adresse : http://lipn.univ-paris13.fr/~chevaleyre/pmwiki/files/Xubuntu32.ova
  ensuite, il faudra importer cette image dans virtualbox.

Chapitre 2

Premiers pas avec l'interpréteur de commandes Python

Difficulté :

Après les premières notions théoriques et l'installation de Python, il est temps de découvrir un peu l'interpréteur de commandes de ce langage. Même si ces petits tests vous semblent anodins, vous découvrirez dans ce chapitre les premiers rudiments de la syntaxe du langage et je vous conseille fortement de me suivre pas à pas, surtout si vous êtes face à votre premier langage de programmation.

Comme tout langage de programmation, Python a une syntaxe claire : on ne peut pas lui envoyer n'importe quelle information dans n'importe quel ordre. Nous allons voir ici ce que Python mange… et ce qu'il ne mange pas.

Où est-ce qu'on est, là ?

Pour commencer, je vais vous demander de retourner dans l'interpréteur de commandes Python (je vous ai montré, à la fin du chapitre précédent, comment y accéder en fonction de votre système d'exploitation).

Je vous rappelle les informations qui figurent dans cette fenêtre, même si elles peuvent être différentes chez vous en fonction de votre version et de votre système d'exploitation.

```
1 Python 3.2.1 (default, Jul 10 2011, 21:51:15) [MSC v.1500 32
      bit (Intel)] on win 32
2 Type "help", "copyright", "credits" or "license" for more
      information.
3 >>>
```

À sa façon, Python vous souhaite la bienvenue dans son interpréteur de commandes.

Attends, attends. C'est quoi cet interpréteur ?

Souvenez-vous, au chapitre précédent, je vous ai donné une brève explication sur la différence entre langages compilés et langages interprétés. Eh bien, cet interpréteur de commandes va nous permettre de tester directement du code. Je saisis une ligne d'instructions, j'appuie sur la touche Entrée de mon clavier, je regarde ce que me répond Python (s'il me dit quelque chose), puis j'en saisis une deuxième, une troisième… Cet interpréteur est particulièrement utile pour comprendre les bases de Python et réaliser nos premiers petits programmes. Le principal inconvénient, c'est que le code que vous saisissez n'est pas sauvegardé (sauf si vous l'enregistrez manuellement, mais chaque chose en son temps).

Dans la fenêtre que vous avez sous les yeux, l'information qui ne change pas d'un système d'exploitation à l'autre est la série de trois chevrons qui se trouve en bas à gauche des informations : >>>. Ces trois signes signifient : « je suis prêt à recevoir tes instructions ».

Comme je l'ai dit, les langages de programmation respectent une syntaxe claire. Vous ne pouvez pas espérer que l'ordinateur comprenne si, dans cette fenêtre, vous commencez par lui demander : « j'aimerais que tu me codes un jeu vidéo génial ». Et autant que vous le sachiez tout de suite (bien qu'à mon avis, vous vous en doutiez), on est très loin d'obtenir des résultats aussi spectaculaires à notre niveau.

Tout cela pour dire que, si vous saisissez n'importe quoi dans cette fenêtre, la probabilité est grande que Python vous indique, clairement et fermement, qu'il n'a rien compris.

Si, par exemple, vous saisissez « premier test avec Python », vous obtenez le résultat suivant :

```
1 >>> premier test avec Python
2 File "<stdin>", line 1
```

```
premier test avec Python
^
SyntaxError: invalid syntax
>>>
```

Eh oui, l'interpréteur parle en anglais et les instructions que vous saisirez, comme pour l'écrasante majorité des langages de programmation, seront également en anglais. Mais pour l'instant, rien de bien compliqué : l'interpréteur vous indique qu'il a trouvé un problème dans votre ligne d'instruction. Il vous indique le numéro de la ligne (en l'occurence la première), qu'il vous répète obligeamment (ceci est très utile quand on travaille sur un programme de plusieurs centaines de lignes). Puis il vous dit ce qui l'arrête, ici : `SyntaxError: invalid syntax`. Limpide n'est-ce pas ? Ce que vous avez saisi est incompréhensible pour Python. Enfin, la preuve qu'il n'est pas rancunier, c'est qu'il vous affiche à nouveau une série de trois chevrons, montrant bien qu'il est prêt à retenter l'aventure.

Bon, c'est bien joli de recevoir un message d'erreur au premier test mais je me doute que vous aimeriez bien voir des trucs qui fonctionnent, maintenant. C'est parti donc.

Vos premières instructions : un peu de calcul mental pour l'ordinateur

C'est assez trivial, quand on y pense, mais je trouve qu'il s'agit d'une excellente manière d'aborder pas à pas la syntaxe de Python. Nous allons donc essayer d'obtenir les résultats de calculs plus ou moins compliqués. Je vous rappelle encore une fois qu'exécuter les tests en même temps que moi sur votre machine est une très bonne façon de vous rendre compte de la syntaxe et surtout, de la retenir.

Saisir un nombre

Vous avez pu voir sur notre premier (et à ce jour notre dernier) test que Python n'aimait pas particulièrement les suites de lettres qu'il ne comprend pas. Par contre, l'interpréteur adore les nombres. D'ailleurs, il les accepte sans sourciller, sans une seule erreur :

```
>>> 7
7
>>>
```

D'accord, ce n'est pas extraordinaire. On saisit un nombre et l'interpréteur le renvoie. Mais dans bien des cas, ce simple retour indique que l'interpréteur a bien compris et que votre saisie est en accord avec sa syntaxe. De même, vous pouvez saisir des nombres à virgule.

```
>>> 9.5
```



```
9.5
>>>
```

Attention : on utilise ici la notation anglo-saxonne, c'est-à-dire que le point remplace la virgule. La virgule a un tout autre sens pour Python, prenez donc cette habitude dès maintenant.

Il va de soi que l'on peut tout aussi bien saisir des nombres négatifs (vous pouvez d'ailleurs faire l'essai).

Opérations courantes

Bon, il est temps d'apprendre à utiliser les principaux opérateurs de Python, qui vont vous servir pour la grande majorité de vos programmes.

Addition, soustraction, multiplication, division

Pour effectuer ces opérations, on utilise respectivement les symboles +, -, * et /.

```
>>> 3 + 4
7
>>> -2 + 93
91
>>> 9.5 + 2
11.5
>>> 3.11 + 2.08
5.1899999999999995
>>>
```

Pourquoi ce dernier résultat approximatif ?

Python n'y est pas pour grand chose. En fait, le problème vient en grande partie de la façon dont les nombres à virgule sont écrits dans la mémoire de votre ordinateur. C'est pourquoi, en programmation, on préfère travailler autant que possible avec des nombres entiers. Cependant, vous remarquerez que l'erreur est infime et qu'elle n'aura pas de réel impact sur les calculs. Les applications qui ont besoin d'une précision mathématique à toute épreuve essayent de pallier ces défauts par d'autres moyens mais ici, ce ne sera pas nécessaire.

Faites également des tests pour la soustraction, la multiplication et la division : il n'y a rien de difficile.

Division entière et modulo

Si vous avez pris le temps de tester la division, vous vous êtes rendu compte que le résultat est donné avec une virgule flottante.

```
>>> 10 / 5
2.0
>>> 10 / 3
3.3333333333333335
>>>
```

Il existe deux autres opérateurs qui permettent de connaître le résultat d'une division entière et le reste de cette division.

Le premier opérateur utilise le symbole « // ». Il permet d'obtenir la partie entière d'une division.

```
>>> 10 // 3
3
>>>
```

L'opérateur « % », que l'on appelle le « modulo », permet de connaître le reste de la division.

```
>>> 10%3
1
>>>
```

Ces notions de *partie entière* et de *reste de division* ne sont pas bien difficiles à comprendre et vous serviront très probablement par la suite.

Si vous avez du mal à en saisir le sens, sachez donc que :

- La partie entière de la division de 10 par 3 est le résultat de cette division, sans tenir compte des chiffres au-delà de la virgule (en l'occurence, 3).
- Pour obtenir le modulo d'une division, on « récupère » son reste. Dans notre exemple, $10/3 = 3$ et il reste 1. Une fois que l'on a compris cela, ce n'est pas bien compliqué.

Souvenez-vous bien de ces deux opérateurs, et surtout du modulo « % », dont vous aurez besoin dans vos programmes futurs.

En résumé

- L'interpréteur de commandes Python permet de tester du code au fur et à mesure qu'on l'écrit.
- L'interpréteur Python accepte des nombres et est capable d'effectuer des calculs.
- Un nombre décimal s'écrit avec un point et non une virgule.
- Les calculs impliquant des nombres décimaux donnent parfois des résultats approximatifs, c'est pourquoi on préfèrera, dans la mesure du possible, travailler avec des nombres entiers.

Chapitre 3

Le monde merveilleux des variables

Difficulté :

Au chapitre précédent, vous avez saisi vos premières instructions en langage Python, bien que vous ne vous en soyez peut-être pas rendu compte. Il est également vrai que les instructions saisies auraient fonctionné dans la plupart des langages. Ici, cependant, nous allons commencer à approfondir un petit peu la syntaxe du langage, tout en découvrant un concept important de la programmation : les variables.

Ce concept est essentiel et vous ne pouvez absolument pas faire l'impasse dessus. Mais je vous rassure, il n'y a rien de compliqué, que de l'utile et de l'agréable.

C'est quoi, une variable ? Et à quoi cela sert-il ?

Les variables sont l'un des concepts qui se retrouvent dans la majorité (et même, en l'occurrence, la totalité) des langages de programmation. Autant dire que sans variable, on ne peut pas programmer, et ce n'est pas une exagération.

C'est quoi, une variable ?

Une variable est une donnée de votre programme, stockée dans votre ordinateur. C'est un code alpha-numérique que vous allez lier à une donnée de votre programme, afin de pouvoir l'utiliser à plusieurs reprises et faire des calculs un peu plus intéressants avec. C'est bien joli de savoir faire des opérations mais, si on ne peut pas stocker le résultat quelque part, cela devient très vite ennuyeux.

Voyez la mémoire de votre ordinateur comme une grosse armoire avec plein de tiroirs. Chaque tiroir peut contenir une donnée ; certaines de ces données seront des variables de votre programme.

Comment cela fonctionne-t-il ?

Le plus simplement du monde. Vous allez dire à Python : « je veux que, dans une variable que je nomme `age`, tu stockes mon âge, pour que je puisse le retenir (si j'ai la mémoire très courte), l'augmenter (à mon anniversaire) et l'afficher si besoin est ».

Comme je vous l'ai dit, on ne peut pas passer à côté des variables. Vous ne voyez peut-être pas encore tout l'intérêt de stocker des informations de votre programme et pourtant, si vous ne stockez rien, vous ne pouvez pratiquement rien faire.

En Python, pour donner une valeur à une variable, il suffit d'écrire `nom_de_la_variable = valeur`.

Une variable doit respecter quelques règles de syntaxe incontournables :

1. Le nom de la variable ne peut être composé que de lettres, majuscules ou minuscules, de chiffres et du symbole souligné « _ » (appelé *underscore* en anglais).
2. Le nom de la variable ne peut pas commencer par un chiffre.
3. Le langage Python est sensible à la casse, ce qui signifie que des lettres majuscules et minuscules ne constituent pas la même variable (la variable `AGE` est différente de `aGe`, elle-même différente de `age`).

Au-delà de ces règles de syntaxe incontournables, il existe des conventions définies par les programmeurs eux-mêmes. L'une d'elle, que j'ai tendance à utiliser assez souvent, consiste à écrire la variable en minuscules et à remplacer les espaces éventuels par un espace souligné « _ ». Si je dois créer une variable contenant mon âge, elle se nommera donc `mon_age`. Une autre convention utilisée consiste à passer en majuscule le premier caractère de chaque mot, à l'exception du premier mot constituant la variable. La variable contenant mon âge se nommerait alors `monAge`.

Vous pouvez utiliser la convention qui vous plaît, ou même en créer une bien à vous, mais essayez de rester cohérent et de n'utiliser qu'une seule convention d'écriture. En effet, il est essentiel de pouvoir vous repérer dans vos variables dès que vous commencez à travailler sur des programmes volumineux.

Ainsi, si je veux associer mon âge à une variable, la syntaxe sera :

```
1 mon_age = 21
```

L'interpréteur vous affiche aussitôt trois chevrons sans aucun message. Cela signifie qu'il a bien compris et qu'il n'y a eu aucune erreur.

Sachez qu'on appelle cette étape *l'affectation de valeur à une variable* (parfois raccourci en « affectation de variable »). On dit en effet qu'on a affecté la valeur 21 à la variable `mon_age`.

On peut afficher la valeur de cette variable en la saisissant simplement dans l'interpréteur de commandes.

```
1 >>> mon_age
2 21
3 >>>
```

Les espaces séparant « = » du nom et de la valeur de la variable sont facultatifs. Je les mets pour des raisons de lisibilité.

Bon, c'est bien joli tout cela, mais qu'est-ce qu'on fait avec cette variable ?

Eh bien, tout ce que vous avez déjà fait au chapitre précédent, mais cette fois en utilisant la variable comme un nombre à part entière. Vous pouvez même affecter à d'autres variables des valeurs obtenues en effectuant des calculs sur la première et c'est là toute la puissance de ce mécanisme.

Essayons par exemple d'augmenter de 2 la variable `mon_age`.

```
1 >>> mon_age = mon_age + 2
2 >>> mon_age
3 23
4 >>>
```

Encore une fois, lors de l'affectation de la valeur, rien ne s'affiche, ce qui est parfaitement normal.

Maintenant, essayons d'affecter une valeur à une autre variable d'après la valeur de `mon_age`.

```
1 >>> mon_age_x2 = mon_age * 2
```

```
>>> mon_age_x2
46
>>>
```

Encore une fois, je vous invite à tester en long, en large et en travers cette possibilité. Le concept n'est pas compliqué mais extrêmement puissant. De plus, comparé à certains langages, affecter une valeur à une variable est extrêmement simple. Si la variable n'est pas créée, Python s'en charge automatiquement. Si la variable existe déjà, l'ancienne valeur est supprimée et remplacée par la nouvelle. Quoi de plus simple ?

Certains mots-clés de Python sont **réservés**, c'est-à-dire que vous ne pouvez pas créer des variables portant ce nom.

En voici la liste pour Python 3 :

and	del	from	none	true
as	elif	global	nonlocal	try
assert	else	if	not	while
break	except	import	or	with
class	false	in	pass	yield
continue	finally	is	raise	
def	for	lambda	return	

Ces mots-clés sont utilisés par Python, vous ne pouvez pas construire de variables portant ces noms. Vous allez découvrir dans la suite de ce cours la majorité de ces mots-clés et comment ils s'utilisent.

Les types de données en Python

Là se trouve un concept très important, que l'on retrouve dans beaucoup de langages de programmation. Ouvrez grand vos oreilles, ou plutôt vos yeux, car vous devrez être parfaitement à l'aise avec ce concept pour continuer la lecture de ce livre. Rassurez-vous toutefois, du moment que vous êtes attentifs, il n'y a rien de compliqué à comprendre.

Qu'entend-on par « type de donnée » ?

Jusqu'ici, vous n'avez travaillé qu'avec des nombres. Et, s'il faut bien avouer qu'on ne fera que très rarement un programme sans aucun nombre, c'est loin d'être la seule donnée que l'on peut utiliser en Python. À terme, vous serez même capables de créer vos propres types de données, mais n'anticipons pas.

Python a besoin de connaître quels types de données sont utilisés pour savoir quelles opérations il peut effectuer avec. Dans ce chapitre, vous allez apprendre à travailler avec

des chaînes de caractères, et multiplier une chaîne de caractères ne se fait pas du tout comme la multiplication d'un nombre. Pour certains types de données, la multiplication n'a d'ailleurs aucun sens. Python associe donc à chaque donnée un type, qui va définir les opérations autorisées sur cette donnée en particulier.

Les différents types de données

Nous n'allons voir ici que les incontournables et les plus faciles à manier. Des chapitres entiers seront consacrés aux types plus complexes.

Les nombres entiers

Et oui, Python différencie les entiers des nombres à virgule flottante !

Pourquoi cela ?

Initialement, c'est surtout pour une question de place en mémoire mais, pour un ordinateur, les opérations que l'on effectue sur des nombres à virgule ne sont pas les mêmes que celles sur les entiers, et cette distinction reste encore d'actualité de nos jours.

Le type entier se nomme `int` en Python (qui correspond à l'anglais « integer », c'est-à-dire entier). La forme d'un entier est un nombre sans virgule.

```
3
```

Nous avons vu au chapitre précédent les opérations que l'on pouvait effectuer sur ce type de données et, même si vous ne vous en souvenez pas, les deviner est assez élémentaire.

Les nombres flottants

Les flottants sont les nombres à virgule. Ils se nomment `float` en Python (ce qui signifie « flottant » en anglais). La syntaxe d'un nombre flottant est celle d'un nombre à virgule (n'oubliez pas de remplacer la virgule par un point). Si ce nombre n'a pas de partie flottante mais que vous voulez qu'il soit considéré par le système comme un flottant, vous pouvez lui ajouter une partie flottante de 0 (exemple **52.0**).

```
3.152
```

Les nombres après la virgule ne sont pas infinis, puisque rien n'est infini en informatique. Mais la précision est assez importante pour travailler sur des données très fines.

Les chaînes de caractères

Heureusement, les types de données disponibles en Python ne sont pas limités aux seuls nombres, bien loin de là. Le dernier type « simple » que nous verrons dans ce chapitre

est la chaîne de caractères. Ce type de donnée permet de stocker une série de lettres, pourquoi pas une phrase.

On peut écrire une chaîne de caractères de différentes façons :

- entre guillemets (`"ceci est une chaîne de caractères"`) ;
- entre apostrophes (`'ceci est une chaîne de caractères'`) ;
- entre triples guillemets (`"""ceci est une chaîne de caractères"""`).

On peut, à l'instar des nombres (et de tous les types de données) stocker une chaîne de caractères dans une variable (`ma_chaine = "Bonjour, la foule !"`)

Si vous utilisez les délimiteurs simples (le guillemet ou l'apostrophe) pour encadrer une chaîne de caractères, il se pose le problème des guillemets ou apostrophes que peut contenir ladite chaîne. Par exemple, si vous tapez `chaine = 'J'aime le Python!'`, vous obtenez le message suivant :

```
File "<stdin>", line 1
chaine = 'J'aime le Python!'
         ^
SyntaxError: invalid syntax
```

Ceci est dû au fait que l'apostrophe de « J'aime » est considérée par Python comme la fin de la chaîne et qu'il ne sait pas quoi faire de tout ce qui se trouve au-delà. Pour pallier ce problème, il faut **échapper** les apostrophes se trouvant au cœur de la chaîne. On insère ainsi un caractère anti-slash « \ » avant les apostrophes contenues dans le message.

```
chaine = 'J\'aime le Python!'
```

On doit également échapper les guillemets si on utilise les guillemets comme délimiteurs.

```
chaine2 = "\"Le seul individu formé, c'est celui qui a appris
    comment apprendre (...)\" (Karl Rogers, 1976)"
```

Le caractère d'échappement « \ » est utilisé pour créer d'autres signes très utiles. Ainsi, « \n » symbolise un saut de ligne (`"essai\nsur\nplusieurs\nlignes"`). Pour écrire un véritable anti-slash dans une chaîne, il faut l'échapper lui-même (et donc écrire « \\ »).

L'interpréteur affiche les sauts de lignes comme on les saisit, c'est-à-dire sous forme de « \n ». Nous verrons dans la partie suivante comment afficher réellement ces chaînes de caractères et pourquoi l'interpréteur ne les affiche pas comme il le devrait.

Utiliser les triples guillemets pour encadrer une chaîne de caractères dispense d'échapper les guillemets et apostrophes, et permet d'écrire plusieurs lignes sans symboliser les retours à la ligne au moyen de « \n ».


```
>>> chaine3 = """Ceci est un nouvel
... essai sur plusieurs
... lignes"""
>>>
```

Notez que les trois chevrons sont remplacés par trois points : cela signifie que l'interpréteur considère que vous n'avez pas fini d'écrire cette instruction. En effet, celle-ci ne s'achève qu'une fois la chaîne refermée avec trois nouveaux guillemets. Les sauts de lignes seront automatiquement remplacés, dans la chaîne, par des « \n ».

Vous pouvez utiliser, à la place des trois guillemets, trois apostrophes qui jouent exactement le même rôle. Je n'utilise personnellement pas ces délimiteurs, mais sachez qu'ils existent et ne soyez pas surpris si vous les voyez un jour dans un code source.

Voilà, nous avons bouclé le rapide tour d'horizon des types simples. Qualifier les chaînes de caractères de type simple n'est pas strictement vrai mais nous n'allons pas, dans ce chapitre, entrer dans le détail des opérations que l'on peut effectuer sur ces chaînes. C'est inutile pour l'instant et ce serait hors sujet. Cependant, rien ne vous empêche de tester vous mêmes quelques opérations comme l'addition et la multiplication (dans le pire des cas, Python vous dira qu'il ne peut pas faire ce que vous lui demandez et, comme nous l'avons vu, il est peu rancunier).

Un petit bonus

Au chapitre précédent, nous avons vu les opérateurs « classiques » pour manipuler des nombres mais aussi, comme on le verra plus tard, d'autres types de données. D'autres opérateurs ont été créés afin de simplifier la manipulation des variables.

Vous serez amenés par la suite, et assez régulièrement, à incrémenter des variables. L'incrémentation désigne l'augmentation de la valeur d'une variable d'un certain nombre. Jusqu'ici, j'ai procédé comme ci-dessous pour augmenter une variable de 1 :

```
variable = variable + 1
```

Cette syntaxe est claire et intuitive mais assez longue, et les programmeurs, tout le monde le sait, sont des fainéants nés. On a donc trouvé plus court.

```
variable += 1
```

L'opérateur `+=` revient à ajouter à la variable la valeur qui suit l'opérateur. Les opérateurs `-=`, `*=` et `/=` existent également, bien qu'ils soient moins utilisés.

Quelques trucs et astuces pour vous faciliter la vie

Python propose un moyen simple de permuter deux variables (échanger leur valeur). Dans d'autres langages, il est nécessaire de passer par une troisième variable qui retient l'une des deux valeurs… ici c'est bien plus simple :

```
>>> a = 5
```

```
>>> b = 32
>>> a,b = b,a # permutation
>>> a
32
>>> b
5
>>>
```

Comme vous le voyez, après l'exécution de la ligne 3, les variables `a` et `b` ont échangé leurs valeurs. On retrouvera cette distribution d'affectation bien plus loin.

On peut aussi affecter assez simplement une même valeur à plusieurs variables :

```
>>> x = y = 3
>>> x
3
>>> y
3
>>>
```

Enfin, ce n'est pas encore d'actualité pour vous mais sachez qu'on peut couper une instruction Python, pour l'écrire sur deux lignes ou plus.

```
>>> 1 + 4 - 3 * 19 + 33 - 45 * 2 + (8 - 3) \
... -6 + 23.5
-86.5
>>>
```

Comme vous le voyez, le symbole « \ » permet, avant un saut de ligne, d'indiquer à Python que « cette instruction se poursuit à la ligne suivante ». Vous pouvez ainsi morceler votre instruction sur plusieurs lignes.

Première utilisation des fonctions

Eh bien, tout cela avance gentiment. Je me permets donc d'introduire ici, dans ce chapitre sur les variables, l'utilisation des fonctions. Il s'agit finalement bien davantage d'une application concrète de ce que vous avez appris à l'instant. Un chapitre entier sera consacré aux fonctions, mais utiliser celles que je vais vous montrer n'est pas sorcier et pourra vous être utile.

Utiliser une fonction

À quoi servent les fonctions ?

Une fonction exécute un certain nombre d'instructions déjà enregistrées. En gros, c'est comme si vous enregistriez un groupe d'instructions pour faire une action précise et que vous lui donniez un nom. Vous n'avez plus ensuite qu'à appeler cette fonction par son nom autant de fois que nécessaire (cela évite bon nombre de répétitions). Mais nous verrons tout cela plus en détail par la suite.

La plupart des fonctions ont besoin d'au moins un paramètre pour travailler sur une donnée ; ces paramètres sont des informations que vous passez à la fonction afin qu'elle travaille dessus. Les fonctions que je vais vous montrer ne font pas exception. Ce concept vous semble peut-être un peu difficile à saisir dans son ensemble mais rassurez-vous, les exemples devraient tout rendre limpide.

Les fonctions s'utilisent en respectant la syntaxe suivante :
`nom_de_la_fonction(parametre_1,parametre_2,...,parametre_n)`.

- Vous commencez par écrire le nom de la fonction.
- Vous placez entre parenthèses les paramètres de la fonction. Si la fonction n'attend aucun paramètre, vous devrez quand même mettre les parenthèses, sans rien entre elles.

La fonction « type »

Dans la partie précédente, je vous ai présenté les types de données simples, du moins une partie d'entre eux. Une des grandes puissances de Python est qu'il comprend automatiquement de quel type est une variable et cela lors de son affectation. Mais il est pratique de pouvoir savoir de quel type est une variable.

La syntaxe de cette fonction est simple :

```
type(nom_de_la_variable)
```

La fonction renvoie le type de la variable passée en paramètre. Vu que nous sommes dans l'interpréteur de commandes, cette valeur sera affichée. Si vous saisissez dans l'interpréteur les lignes suivantes :

```
>>> a = 3
>>> type(a)
```

Vous obtenez :

```
<class 'int'>
```

Python vous indique donc que la variable `a` appartient à la classe des entiers. Cette notion de classe ne sera pas approfondie avant bien des chapitres mais sachez qu'on peut la rapprocher d'un type de donnée.

Vous pouvez faire le test sans passer par des variables :

```
>>> type(3.4)
<class 'float'>
```

```
>>> type("un essai")
<class 'str'>
>>>
```

str est l'abréviation de « string » qui signifie chaîne (sous-entendu, de caractères) en anglais.

La fonction `print`

La fonction `print` permet d'afficher la valeur d'une ou plusieurs variables.

Mais... on ne fait pas exactement la même chose en saisissant juste le nom de la variable dans l'interpréteur ?

Oui et non. L'interpréteur affiche bien la valeur de la variable car il affiche automatiquement tout ce qu'il peut, pour pouvoir suivre les étapes d'un programme. Cependant, quand vous ne travaillerez plus avec l'interpréteur, taper simplement le nom de la variable n'aura aucun effet. De plus, et vous l'aurez sans doute remarqué, l'interpréteur entoure les chaînes de caractères de délimiteurs et affiche les caractères d'échappement, tout ceci encore pour des raisons de clarté.

La fonction `print` est dédiée à l'affichage uniquement. Le nombre de ses paramètres est variable, c'est-à-dire que vous pouvez lui demander d'afficher une ou plusieurs variables. Considérez cet exemple :

```
>>> a = 3
>>> print(a)
>>> a = a + 3
>>> b = a - 2
>>> print("a =", a, "et b =", b)
```

Le premier *appel* à `print` se contente d'afficher la valeur de la variable `a`, c'est-à-dire « 3 ». Le second appel à `print` affiche :

```
a = 6 et b = 4
```

Ce deuxième appel à `print` est peut-être un peu plus dur à comprendre. En fait, on passe quatre paramètres à `print`, deux chaînes de caractères et les variables `a` et `b`. Quand Python interprète cet appel de fonction, il va afficher les paramètres dans l'ordre de passage, en les séparant par un espace.

Relisez bien cet exemple, il montre tout l'intérêt des fonctions. Si vous avez du mal à le comprendre dans son ensemble, décortiquez-le en prenant indépendamment chaque paramètre.

Testez l'utilisation de `print` avec d'autres types de données et en insérant des chaînes avec des sauts de lignes et des caractères échappés, pour bien vous rendre compte de la différence.

Un petit « Hello World ! » ?

Quand on fait un cours sur un langage, quel qu'il soit, il est d'usage de présenter le programme « Hello World ! », qui illustre assez rapidement la syntaxe superficielle d'un langage.

Le but du jeu est très simple : écrire un programme qui affiche « Hello World ! » à l'écran. Dans certains langages, notamment les langages compilés, vous pourrez nécessiter jusqu'à une dizaine de lignes pour obtenir ce résultat. En Python, comme nous venons de le voir, il suffit d'une seule ligne :

```
>>> print("Hello World !")
```

Pour plus d'informations, n'hésitez pas à consulter la page Wikipédia consacrée à « Hello World ! » ; vous avez même des codes rédigés en différents langages de programmation, cela peut être intéressant.

▷ Wikipédia - « Hello World ! »
Code web : 696192

En résumé

- Les variables permettent de conserver dans le temps des données de votre programme.
- Vous pouvez vous servir de ces variables pour différentes choses : les afficher, faire des calculs avec, etc.
- Pour affecter une valeur à une variable, on utilise la syntaxe `nom_de_variable = valeur`.
- Il existe différents types de variables, en fonction de l'information que vous désirez conserver : `int`, `float`, chaîne de caractères etc.
- Pour afficher une donnée, comme la valeur d'une variable par exemple, on utilise la fonction `print`.

Chapitre 4

Les structures conditionnelles

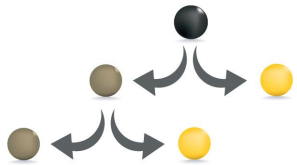
Difficulté :

Jusqu'à présent, nous avons testé des instructions d'une façon linéaire : l'interpréteur exécutait au fur et à mesure le code que vous saisissiez dans la console. Mais nos programmes seraient bien pauvres si nous ne pouvions, de temps à autre, demander à exécuter certaines instructions dans un cas, et d'autres instructions dans un autre cas.

Dans ce chapitre, je vais vous parler des structures conditionnelles, qui vont vous permettre de faire des tests et d'aller plus loin dans la programmation.

Les conditions permettent d'exécuter une ou plusieurs instructions dans un cas, d'autres instructions dans un autre cas.

Vous finirez ce chapitre en créant votre premier « vrai » programme : même si vous ne pensez pas encore pouvoir faire quelque chose de très consistant, à la fin de ce chapitre vous aurez assez de matière pour coder un petit programme dans un but très précis.



Vos premières conditions et blocs d'instructions

Forme minimale en `if`

Les conditions sont un concept essentiel en programmation (oui oui, je me répète à force mais il faut avouer que des concepts essentiels, on n'a pas fini d'en voir). Elles vont vous permettre de faire une action précise si, par exemple, une variable est positive, une autre action si cette variable est négative, ou une troisième action si la variable est nulle. Comme un bon exemple vaut mieux que plusieurs lignes d'explications, voici un exemple clair d'une condition prise sous sa forme la plus simple.

Dès à présent dans mes exemples, j'utiliserai des commentaires. Les commentaires sont des messages qui sont ignorés par l'interpréteur et qui permettent de donner des indications sur le code (car, vous vous en rendrez compte, relire ses programmes après plusieurs semaines d'abandon, sans commentaire, ce peut être parfois plus qu'ardu). En Python, un commentaire débute par un dièse (« # ») et se termine par un saut de ligne. Tout ce qui est compris entre ce # et ce saut de ligne est ignoré. Un commentaire peut donc occuper la totalité d'une ligne (on place le # en début de ligne) ou une partie seulement, après une instruction (on place le # après la ligne de code pour la commenter plus spécifiquement).

Cela étant posé, revenons à nos conditions :

```
1 >>> # Premier exemple de condition
2 >>> a = 5
3 >>> if a > 0: # Si a est supérieur à 0
4 ...     print("a est supérieur à 0.")
5 ...
6 a est supérieur à 0.
7 >>>
```

Détaillons ce code, ligne par ligne :

1. La première ligne est un commentaire décrivant qu'il s'agit du premier test de condition. Elle est ignorée par l'interpréteur et sert juste à vous renseigner sur le code qui va suivre.
2. Cette ligne, vous devriez la comprendre sans aucune aide. On se contente d'affecter la valeur 5 à la variable `a`.
3. Ici se trouve notre test conditionnel. Il se compose, dans l'ordre :
   - du mot clé `if` qui signifie « si » en anglais ;
   - de la condition proprement dite, `a > 0`, qu'il est facile de lire (une liste des opérateurs autorisés pour la comparaison sera présentée plus bas) ;
   - du signe deux points, « : », qui termine la condition et est indispensable : Python affichera une erreur de syntaxe si vous l'omettez.

4. Ici se trouve l'instruction à exécuter dans le cas où `a` est supérieur à `0`. Après que vous ayez appuyé sur Entrée à la fin de la ligne précédente, l'interpréteur vous présente la série de trois points qui signifie qu'il attend la saisie du **bloc d'instructions** concerné avant de l'interpréter. Cette instruction (et les autres instructions à exécuter s'il y en a) est indentée, c'est-à-dire décalée vers la droite. Des explications supplémentaires seront données un peu plus bas sur les indentations.
5. L'interpréteur vous affiche à nouveau la série de trois points et vous pouvez en profiter pour saisir une nouvelle instruction dans ce bloc d'instructions. Ce n'est pas le cas pour l'instant. Vous appuyez donc sur Entrée sans avoir rien écrit et l'interpréteur vous affiche le message « a est supérieur à 0 », ce qui est assez logique vu que `a` est effectivement supérieur à `0`.

Il y a deux notions importantes sur lesquelles je dois à présent revenir, elles sont complémentaires ne vous en faites pas.

La première est celle de bloc d'instructions. On entend par bloc d'instructions une série d'instructions qui s'exécutent dans un cas précis (par condition, comme on vient de le voir, par répétition, comme on le verra plus tard...). Ici, notre bloc n'est constitué que d'une seule instruction (la ligne 4 qui fait appel à `print`). Mais rien ne vous empêche de mettre plusieurs instructions dans ce bloc.

```
1 a = 5
2 b = 8
3 if a > 0:
4     # On incrémente la valeur de b
5     b += 1
6     # On affiche les valeurs des variables
7     print("a =",a,"et b =",b)
```

La seconde notion importante est celle d'indentation. On entend par indentation un certain décalage vers la droite, obtenu par un (ou plusieurs) espaces ou tabulations.

Les indentations sont essentielles pour Python. Il ne s'agit pas, comme dans d'autres langages tels que le C++ ou le Java, d'un confort de lecture mais bien d'un moyen pour l'interpréteur de savoir où se trouvent le début et la fin d'un bloc.

Forme complète (`if`, `elif` et `else`)

Les limites de la condition simple en `if`

La première forme de condition que l'on vient de voir est pratique mais assez incomplète.

Considérons, par exemple, une variable `a` de type entier. On souhaite faire une action si cette variable est positive et une action différente si elle est négative. Il est possible d'obtenir ce résultat avec la forme simple d'une condition :


```
1 >>> a = 5
2 >>> if a > 0: # Si a est positif
3 ...     print("a est positif.")
4 ... if a < 0: # a est négatif
5 ...     print("a est négatif.")
```

Amusez-vous à changer la valeur de `a` et exécutez à chaque fois les conditions ; vous obtiendrez des messages différents, sauf si `a` est égal à 0. En effet, aucune action n'a été prévue si a vaut 0.

Cette méthode n'est pas optimale, tout d'abord parce qu'elle nous oblige à écrire deux conditions séparées pour tester une même variable. De plus, et même si c'est dur à concevoir par cet exemple, dans le cas où la variable remplirait les deux conditions (ici c'est impossible bien entendu), les deux portions de code s'exécuteraient.

La condition `if` est donc bien pratique mais insuffisante.

L'instruction `else` :

Le mot-clé `else`, qui signifie « sinon » en anglais, permet de définir une première forme de complément à notre instruction `if`.

```
1 >>> age = 21
2 >>> if age >= 18: # Si age est supérieur ou égal à 18
3 ...     print("Vous êtes majeur.")
4 ... else: # Sinon (age inférieur à 18)
5 ... print("Vous êtes mineur.")
```

Je pense que cet exemple suffit amplement à exposer l'utilisation de `else`. La seule subtilité est de bien se rendre compte que Python exécute soit l'un, soit l'autre, et jamais les deux. Notez que cette instruction `else` doit se trouver au même niveau d'indentation que l'instruction `if` qu'elle complète. De plus, elle se termine également par deux points puisqu'il s'agit d'une condition, même si elle est sous-entendue.

L'exemple de tout à l'heure pourrait donc se présenter comme suit, avec l'utilisation de `else` :

```
1 >>> a = 5
2 >>> if a > 0:
3 ...     print("a est supérieur à 0.")
4 ... else:
5 ...     print("a est inférieur ou égal à 0.")
```

Mais... le résultat n'est pas tout à fait le même, si ?

Non, en effet. Vous vous rendrez compte que, cette fois, le cas où `a` vaut `0` est bien pris en compte. En effet, la condition initiale prévoit d'exécuter le premier bloc d'instructions si `a` est strictement supérieur à `0`. Sinon, on exécute le second bloc d'instructions.

Si l'on veut faire la différence entre les nombres positifs, négatifs et nuls, il va falloir utiliser une condition intermédiaire.

L'instruction `elif` :

Le mot clé `elif` est une contraction de « else if », que l'on peut traduire très littéralement par « sinon si ». Dans l'exemple que nous venons juste de voir, l'idéal serait d'écrire :

- si `a` est strictement supérieur à `0`, on dit qu'il est positif ;
- sinon si `a` est strictement inférieur à `0`, on dit qu'il est négatif ;
- sinon, (`a` ne peut qu'être égal à `0`), on dit alors que `a` est nul.

Traduit en langage Python, cela donne :

```
>>> if a > 0: # Positif
...     print("a est positif.")
... elif a < 0: # Négatif
...     print("a est négatif.")
... else: # Nul
    print("a est nul.")
```

De même que le `else`, le `elif` est sur le même niveau d'indentation que le `if` initial. Il se termine aussi par deux points. Cependant, entre le `elif` et les deux points se trouve une nouvelle condition. Linéairement, le schéma d'exécution se traduit comme suit :

1. On regarde si `a` est strictement supérieur à `0`. Si c'est le cas, on affiche « a est positif » et on s'arrête là.
2. Sinon, on regarde si `a` est strictement inférieur à `0`. Si c'est le cas, on affiche « a est négatif » et on s'arrête.
3. Sinon, on affiche « a est nul ».

Attention : quand je dis « on s'arrête », il va de soi que c'est uniquement pour cette condition. S'il y a du code après les trois blocs d'instructions, il sera exécuté dans tous les cas.

Vous pouvez mettre autant de `elif` que vous voulez après une condition en `if`. Tout comme le `else`, cette instruction est facultative et, quand bien même vous construiriez une instruction en `if`, `elif`, vous n'êtes pas du tout obligé de prévoir un `else` après. En revanche, l'instruction `else` ne peut figurer qu'une fois, clôturant le bloc de la condition. Deux instructions `else` dans une même condition ne sont pas envisageables et n'auraient de toute façon aucun sens.

Sachez qu'il est heureusement possible d'imbriquer des conditions et, dans ce cas, l'indentation permet de comprendre clairement le schéma d'exécution du programme. Je vous laisse essayer cette possibilité, je ne vais pas tout faire à votre place non plus :)

De nouveaux opérateurs

Les opérateurs de comparaison

Les conditions doivent nécessairement introduire de nouveaux opérateurs, dits **opérateurs de comparaison.** Je vais les présenter très brièvement, vous laissant l'initiative de faire des tests car ils ne sont réellement pas difficiles à comprendre.

Opérateur	Signification littérale
<	Strictement inférieur à
>	Strictement supérieur à
<=	Inférieur ou égal à
>=	Supérieur ou égal à
==	Égal à
!=	Différent de

Attention : l'égalité de deux valeurs est comparée avec l'opérateur « == » et non « = ». Ce dernier est en effet l'opérateur d'affectation et ne doit pas être utilisé dans une condition.

Prédicats et booléens

Avant d'aller plus loin, sachez que les conditions qui se trouvent, par exemple, entre `if` et les deux points sont appelés des **prédicats.** Vous pouvez tester ces prédicats directement dans l'interpréteur pour comprendre les explications qui vont suivre.

```
>>> a = 0
>>> a == 5
False
>>> a > -8
True
>>> a != 33.19
True
>>>
```

L'interpréteur renvoie tantôt `True` (c'est-à-dire « vrai »), tantôt `False` (c'est-à-dire « faux »).

`True` et `False` sont les deux valeurs possibles d'un type que nous n'avons pas vu jusqu'ici : le type booléen (bool).

N'oubliez pas que True et False sont des valeurs ayant leur première lettre en majuscule. Si vous commencez à écrire True sans un 'T' majuscule, Python ne va pas comprendre.

Les variables de ce type ne peuvent prendre comme valeur que vrai ou faux et peuvent être pratiques, justement, pour stocker des prédicats, de la façon que nous avons vue ou d'une façon plus détournée.

```
>>> age = 21
>>> majeur = False
>>> if age >= 18:
>>>     majeur = True
>>>
```

À la fin de cet exemple, majeur vaut `True`, c'est-à-dire « vrai », si l'âge est supérieur ou égal à 18. Sinon, il continue de valoir `False`. Les booléens ne vous semblent peut-être pas très utiles pour l'instant mais vous verrez qu'ils rendent de grands services !

Les mots-clés `and`, `or` et `not`

Il arrive souvent que nos conditions doivent tester plusieurs prédicats, par exemple quand l'on cherche à vérifier si une variable quelconque, de type entier, se trouve dans un intervalle précis (c'est-à-dire comprise entre deux nombres). Avec nos méthodes actuelles, le plus simple serait d'écrire :

```
# On fait un test pour savoir si a est comprise dans l'
    intervalle allant de 2 à 8 inclus
a = 5
if a >= 2:
    if a <= 8:
        print("a est dans l'intervalle.")
    else:
        print("a n'est pas dans l'intervalle.")
else:
    print("a n'est pas dans l'intervalle.")
```

Cela marche mais c'est assez lourd, d'autant que, pour être sûr qu'un message soit affiché à chaque fois, il faut fermer chacune des deux conditions à l'aide d'un `else` (la seconde étant imbriquée dans la première). Si vous avez du mal à comprendre cet exemple, prenez le temps de le décortiquer, ligne par ligne, il n'y a rien que de très simple.

Il existe cependant le mot clé `and` (qui signifie « et » en anglais) qui va nous rendre ici un fier service. En effet, on cherche à tester à la fois si `a` est supérieur ou égal à 2 et inférieur ou égal à 8. On peut donc réduire ainsi les conditions imbriquées :

```
if a>=2 and a<=8:
    print("a est dans l'intervalle.")
```



```
else:
    print("a n'est pas dans l'intervalle.")
```

Simple et bien plus compréhensible, avouez-le.

Sur le même mode, il existe le mot clé `or` qui signifie cette fois « ou ». Nous allons prendre le même exemple, sauf que nous allons évaluer notre condition différemment.

Nous allons chercher à savoir si `a` n'est pas dans l'intervalle. La variable ne se trouve pas dans l'intervalle si elle est inférieure à 2 ou supérieure à 8. Voici donc le code :

```
if a<2 or a>8:
    print("a n'est pas dans l'intervalle.")
else:
    print("a est dans l'intervalle.")
```

Enfin, il existe le mot clé `not` qui « inverse » un prédicat. Le prédicat `not a==5` équivaut donc à `a!=5`.

`not` rend la syntaxe plus claire. Pour cet exemple, j'ajoute à la liste un nouveau mot clé, `is`, qui teste l'égalité non pas des valeurs de deux variables, mais de leurs références. Je ne vais pas rentrer dans le détail de ce mécanisme avant longtemps. Il vous suffit de savoir que pour les entiers, les flottants et les booléens, c'est strictement la même chose. Mais pour tester une égalité entre variables dont le type est plus complexe, préférez l'opérateur « == ». Revenons à cette démonstration :

```
>>> majeur = False
>>> if majeur is not True:
...     print("Vous n'êtes pas encore majeur.")
...
Vous n'êtes pas encore majeur.
>>>
```

Si vous parlez un minimum l'anglais, ce prédicat est limpide et d'une simplicité sans égale.

Vous pouvez tester des prédicats plus complexes de la même façon que les précédents, en les saisissant directement, sans le `if` ni les deux points, dans l'interpréteur de commande. Vous pouvez utiliser les parenthèses ouvrantes et fermantes pour encadrer des prédicats et les comparer suivant des priorités bien précises (nous verrons ce point plus loin, si vous n'en comprenez pas l'utilité).

Votre premier programme !

À quoi on joue ?

L'heure du premier TP est venue. Comme il s'agit du tout premier, et parce qu'il y a quelques indications que je dois vous donner pour que vous parveniez jusqu'au bout, je vous accompagnerai pas à pas dans sa réalisation.

Avant de commencer

Vous allez dans cette section écrire votre premier programme. Vous allez sûrement tester les syntaxes directement dans l'interpréteur de commandes.

Vous pourriez préférer écrire votre code directement dans un fichier que vous pouvez ensuite exécuter. Si c'est le cas, je vous renvoie au chapitre traitant de ce point, que vous trouverez à la page 349 de ce livre.

Sujet

Le but de notre programme est de déterminer si une année saisie par l'utilisateur est bissextile. Il s'agit d'un sujet très prisé des enseignants en informatique quand il s'agit d'expliquer les conditions. Mille pardons, donc, à ceux qui ont déjà fait cet exercice dans un autre langage mais je trouve que ce petit programme reprend assez de thèmes abordés dans ce chapitre pour être réellement intéressant.

Je vous rappelle les règles qui déterminent si une année est bissextile ou non (vous allez peut-être même apprendre des choses que le commun des mortels ignore).

Une année est dite bissextile si c'est un multiple de 4, sauf si c'est un multiple de 100. Toutefois, elle est considérée comme bissextile si c'est un multiple de 400. Je développe :

- Si une année n'est pas multiple de 4, on s'arrête là, elle n'est pas bissextile.
- Si elle est multiple de 4, on regarde si elle est multiple de 100.
  - Si c'est le cas, on regarde si elle est multiple de 400.
    - Si c'est le cas, l'année est bissextile.
    - Sinon, elle n'est pas bissextile.
  - Sinon, elle est bissextile.

Solution ou résolution

Voilà. Le problème est posé clairement (sinon relisez attentivement l'énoncé autant de fois que nécessaire), il faut maintenant réfléchir à sa résolution en termes de programmation. C'est une phase de transition assez délicate de prime abord et je vous conseille de schématiser le problème, de prendre des notes sur les différentes étapes, sans pour l'instant penser au code. C'est une phase purement algorithmique, autrement dit, on réfléchit au programme sans réfléchir au code proprement dit.

Vous aurez besoin, pour réaliser ce petit programme, de quelques indications qui sont réellement spécifiques à Python. Ne lisez donc ceci qu'après avoir cerné et clairement écrit le problème d'une façon plus algorithmique. Cela étant dit, si vous peinez à trouver

une solution, ne vous y attardez pas. Cette phase de réflexion est assez difficile au début et, parfois il suffit d'un peu de pratique et d'explications pour comprendre l'essentiel.

La fonction `input()`

Tout d'abord, j'ai mentionné une année saisie par l'utilisateur. En effet, depuis tout à l'heure, nous testons des variables que nous déclarons nous-mêmes, avec une valeur précise. La condition est donc assez ridicule.

`input()` est une fonction qui va, pour nous, caractériser nos premières interactions avec l'utilisateur : le programme réagira différemment en fonction du nombre saisi par l'utilisateur.

`input()` accepte un paramètre facultatif : le message à afficher à l'utilisateur. Cette instruction interrompt le programme et attend que l'utilisateur saisisse ce qu'il veut puis appuie sur Entrée. À cet instant, la fonction renvoie ce que l'utilisateur a saisi. Il faut donc piéger cette valeur dans une variable.

```
>>> # Test de la fonction input
>>> annee = input("Saisissez une année : ")
Saisissez une année : 2009
>>> print(annee)
'2009'
>>>
```

Il subsiste un problème : le type de la variable `annee` après l'appel à `input()` est... une chaîne de caractères. Vous pouvez vous en rendre compte grâce aux apostrophes qui encadrent la valeur de la variable quand vous l'affichez directement dans l'interpréteur.

C'est bien ennuyeux : nous qui voulions travailler sur un entier, nous allons devoir convertir cette variable. Pour convertir une variable vers un autre type, il faut utiliser le nom du type comme une fonction (c'est d'ailleurs exactement ce que c'est).

```
>>> type(annee)
<type 'str'>
>>> # On veut convertir la variable en un entier, on utilise
>>> # donc la fonction int qui prend en paramètre la variable
>>> # d'origine
>>> annee = int(annee)
>>> type(annee)
<type 'int'>
>>> print(annee)
2009
>>>
```

Bon, parfait ! On a donc maintenant l'année sous sa forme entière. Notez que, si vous saisissez des lettres lors de l'appel à `input()`, la conversion renverra une erreur.

L'appel à la fonction `int()` en a peut-être déconcerté certains. On passe en paramètre de cette fonction la variable contenant la chaîne de caractères issue de `input()`, pour tenter de la convertir. La fonction `int()` renvoie la valeur convertie en entier et on la récupère donc dans la même variable. On évite ainsi de travailler sur plusieurs variables, sachant que la première n'a plus aucune utilité à présent qu'on l'a convertie.

Test de multiples

Certains pourraient également se demander comment tester si un nombre a est multiple d'un nombre b. Il suffit, en fait, de tester le reste de la division entière de b par a. Si ce reste est nul, alors a est un multiple de b.

```
>>> 5 % 2 # 5 n'est pas un multiple de 2
1
>>> 8 % 2 # 8 est un multiple de 2
0
>>>
```

À vous de jouer

Je pense vous avoir donné tous les éléments nécessaires pour réussir. À mon avis, le plus difficile est la phase de réflexion qui précède la composition du programme. Si vous avez du mal à réaliser cette opération, passez à la correction et étudiez-la soigneusement. Sinon, on se retrouve à la section suivante.

Bonne chance !

Correction

C'est l'heure de comparer nos méthodes et, avant de vous divulguer le code de ma solution, je vous précise qu'elle est loin d'être la seule possible. Vous pouvez très bien avoir trouvé quelque chose de différent mais qui fonctionne tout aussi bien.

Attention... la voiiiciiiiiii...

```
# Programme testant si une année, saisie par l'utilisateur,
# est bissextile ou non

annee = input("Saisissez une année : ") # On attend que l'
    utilisateur saisisse l'année qu'il désire tester
annee = int(annee) # Risque d'erreur si l'utilisateur n'a pas
    saisi un nombre
bissextile = False # On crée un booléen qui vaut vrai ou faux
                   # selon que l'année est bissextile ou non

```



```
if annee % 400 == 0:
    bissextile = True
elif annee % 100 == 0:
    bissextile = False
elif annee % 4 == 0:
    bissextile = True
else:
    bissextile = False

if bissextile: # Si l'année est bissextile
    print("L'année saisie est bissextile.")
else:
    print("L'année saisie n'est pas bissextile.")
```

Je vous rappelle que vous pouvez enregistrer vos codes dans des fichiers afin de les exécuter. Je vous renvoie à la page 349 pour plus d'informations.

Je pense que le code est assez clair, reste à expliciter l'enchaînement des conditions. Vous remarquerez qu'on a inversé le problème. On teste en effet d'abord si l'année est un multiple de 400, ensuite si c'est un multiple de 100, et enfin si c'est un multiple de 4. En effet, le `elif` garantit que, si `annee` est un multiple de 100, ce n'est pas un multiple de 400 (car le cas a été traité au-dessus). De cette façon, on s'assure que tous les cas sont gérés. Vous pouvez faire des essais avec plusieurs années et vous rendre compte si le programme a raison ou pas.

L'utilisation de `bissextile` comme d'un prédicat à part entière vous a peut-être déconcertés. C'est en fait tout à fait possible et logique, puisque `bissextile` est un booléen. Il est de ce fait vrai ou faux et donc on peut le tester simplement. On peut bien entendu aussi écrire `if bissextile==True:`, cela revient au même.

Un peu d'optimisation

Ce qu'on a fait était bien mais on peut l'améliorer. D'ailleurs, vous vous rendrez compte que c'est presque toujours le cas. Ici, il s'agit bien entendu de notre condition, que je vais passer au crible afin d'en construire une plus courte et plus logique, si possible. On peut parler d'optimisation dans ce cas, même si l'optimisation intègre aussi et surtout les ressources consommées par votre application, en vue de diminuer ces ressources et d'améliorer la rapidité de l'application. Mais, pour une petite application comme celle-ci, je ne pense pas qu'on perdra du temps sur l'optimisation du temps d'exécution.

Le premier détail que vous auriez pu remarquer, c'est que le `else` de fin est inutile. En effet, la variable `bissextile` vaut par défaut `False` et conserve donc cette valeur si le cas n'est pas traité (ici, quand l'année n'est ni un multiple de 400, ni un multiple de 100, ni un multiple de 4).

Ensuite, il apparaît que nous pouvons faire un grand ménage dans notre condition car les deux seuls cas correspondant à une année bissextile sont « si l'année est un multiple

de 400 »ou « si l'année est un multiple de 4 mais pas de 100 ».

Le prédicat correspondant est un peu délicat, il fait appel aux priorités des parenthèses. Je ne m'attendais pas que vous le trouviez tout seuls mais je souhaite que vous le compreniez bien à présent.

```
1 # Programme testant si une année, saisie par l'utilisateur, est
      bissextile ou non
2
3 annee = input("Saisissez une année : ") # On attend que l'
    utilisateur saisisse l'année qu'il désire tester
4 annee = int(annee) # Risque d'erreur si l'utilisateur n'a pas
    saisi un nombre
5
6 if annee % 400 == 0 or (annee % 4 == 0 and annee % 100 != 0):
7     print("L'année saisie est bissextile.")
8 else:
9     print("L'année saisie n'est pas bissextile.")
```

▷ Copier ce code
Code web : 886842

Du coup, on n'a plus besoin de la variable `bissextile`, c'est déjà cela de gagné. Nous sommes passés de 16 lignes de code à seulement 7 (sans compter les commentaires et les sauts de ligne) ce qui n'est pas rien.

En résumé

- Les conditions permettent d'exécuter certaines instructions dans certains cas, d'autres instructions dans un autre cas.
- Les conditions sont marquées par les mot-clés `if` (« si »), `elif` (« sinon si ») et `else` (« sinon »).
- Les mot-clés `if` et `elif` doivent être suivis d'un test (appelé aussi prédicat).
- Les booléens sont des données soit vraies (`True`) soit fausses (`False`).

Chapitre 5

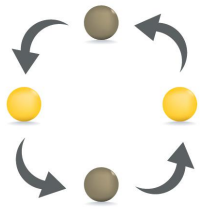
Les boucles

Difficulté :

Les boucles sont un concept nouveau pour vous. Elles vont vous permettre de répéter une certaine opération autant de fois que nécessaire. Le concept risque de vous sembler un peu théorique car les applications pratiques présentées dans ce chapitre ne vous paraîtront probablement pas très intéressantes. Toutefois, il est impératif que cette notion soit comprise avant que vous ne passiez à la suite. Viendra vite le moment où vous aurez du mal à écrire une application sans boucle.

En outre, les boucles peuvent permettre de parcourir certaines séquences comme les chaînes de caractères pour, par exemple, en extraire chaque caractère.

Alors, on commence ?



En quoi cela consiste-t-il ?

Comme je l'ai dit juste au-dessus, les boucles constituent un moyen de répéter un certain nombre de fois des instructions de votre programme. Prenons un exemple simple, même s'il est assez peu réjouissant en lui-même : écrire un programme affichant la table de multiplication par 7, de 1 * 7 à 10 * 7.

... bah quoi ?

Bon, ce n'est qu'un exemple, ne faites pas cette tête, et puis je suis sûr que ce sera utile pour certains. Dans un premier temps, vous devriez arriver au programme suivant :

```
1  print(" 1 * 7 =", 1 * 7)
2  print(" 2 * 7 =", 2 * 7)
3  print(" 3 * 7 =", 3 * 7)
4  print(" 4 * 7 =", 4 * 7)
5  print(" 5 * 7 =", 5 * 7)
6  print(" 6 * 7 =", 6 * 7)
7  print(" 7 * 7 =", 7 * 7)
8  print(" 8 * 7 =", 8 * 7)
9  print(" 9 * 7 =", 9 * 7)
10 print("10 * 7 =", 10 * 7)
```

... et le résultat :

```
1   1 * 7 = 7
2   2 * 7 = 14
3   3 * 7 = 21
4   4 * 7 = 28
5   5 * 7 = 35
6   6 * 7 = 42
7   7 * 7 = 49
8   8 * 7 = 56
9   9 * 7 = 63
10 10 * 7 = 70
```

Je vous rappelle que vous pouvez enregistrer vos codes dans des fichiers. Vous trouverez la marche à suivre à la page 349 de ce livre.

Bon, c'est sûrement la première idée qui vous est venue et cela fonctionne, très bien même. Seulement, vous reconnaîtrez qu'un programme comme cela n'est pas bien utile. Essayons donc le même programme mais, cette fois-ci, en utilisant une variable ; ainsi, si on décide d'afficher la table de multiplication de 6, on n'aura qu'à changer la valeur de la variable ! Pour cet exemple, on utilise une variable `nb` qui contiendra 7. Les instructions seront légèrement différentes mais vous devriez toujours pouvoir écrire ce programme :

```
1  nb = 7
2  print(" 1 *", nb, "=", 1 * nb)
```



```
3  print(" 2 *", nb, "=", 2 * nb)
4  print(" 3 *", nb, "=", 3 * nb)
5  print(" 4 *", nb, "=", 4 * nb)
6  print(" 5 *", nb, "=", 5 * nb)
7  print(" 6 *", nb, "=", 6 * nb)
8  print(" 7 *", nb, "=", 7 * nb)
9  print(" 8 *", nb, "=", 8 * nb)
10 print(" 9 *", nb, "=", 9 * nb)
11 print("10 *", nb, "=", 10 * nb)
```

Le résultat est le même, vous pouvez vérifier. Mais le code est quand-même un peu plus intéressant : on peut changer la table de multiplication à afficher en changeant la valeur de la variable `nb`.

Mais ce programme reste assez peu pratique et il accomplit une tâche bien répétitive. Les programmeurs étant très paresseux, ils préfèrent utiliser les boucles.

La boucle `while`

La boucle que je vais présenter se retrouve dans la plupart des autres langages de programmation et porte le même nom. Elle permet de répéter un **bloc d'instructions** tant qu'une condition est vraie (`while` signifie « tant que » en anglais). J'espère que le concept de **bloc d'instructions** est clair pour vous, sinon je vous renvoie au chapitre précédent.

La syntaxe de `while` est :

```
while condition:
    # instruction 1
    # instruction 2
    # ...
    # instruction N
```

Vous devriez reconnaître la forme d'un bloc d'instructions, du moins je l'espère.

Quelle condition va-t-on utiliser ?

Eh bien, c'est là le point important. Dans cet exemple, on va créer une variable qui sera incrémentée dans le bloc d'instructions. Tant que cette variable sera inférieure à 10, le bloc s'exécutera pour afficher la table.

Si ce n'est pas clair, regardez ce code, quelques commentaires suffiront pour le comprendre :

```
1  nb = 7 # On garde la variable contenant le nombre dont on veut
       la table de multiplication
2  i = 0 # C'est notre variable compteur que nous allons incré
       menter dans la boucle
```

```
3
4  while i < 10: # Tant que i est strictement inférieure à 10
5      print(i + 1, "*", nb, "=", (i + 1) * nb)
6      i += 1 # On incrémente i de 1 à chaque tour de boucle
```

Analysons ce code ligne par ligne :

1. On instancie la variable `nb` qui accueille le nombre sur lequel nous allons travailler (en l'occurence, 7). Vous pouvez bien entendu faire saisir ce nombre par l'utilisateur, vous savez le faire à présent.
2. On instancie la variable `i` qui sera notre compteur durant la boucle. `i` est un standard utilisé quand il est question de boucles et de variables s'incrémentant mais il va de soi que vous auriez pu lui donner un autre nom. On l'initialise à 0.
3. Un saut de ligne ne fait jamais de mal !
4. On trouve ici l'instruction `while` qui se décode, comme je l'ai indiqué en commentaire, en « **tant que i est strictement inférieure à 10** ». N'oubliez pas les deux points à la fin de la ligne.
5. La ligne du `print`, vous devez la reconnaître. Maintenant, la plus grande partie de la ligne affichée est constituée de variables, à part les signes mathématiques. Vous remarquez qu'à chaque fois qu'on utilise `i` dans cette ligne, pour l'affichage ou le calcul, on lui ajoute 1 : cela est dû au fait qu'en programmation, on a l'habitude (habitude que vous devrez prendre) de commencer à compter à partir de 0. Seulement ce n'est pas le cas de la table de multiplication, qui va de 1 à 10 et non de 0 à 9, comme c'est le cas pour les valeurs de `i`. Certes, j'aurais pu changer la condition et la valeur initiale de `i`, ou même placer l'incrémentation de `i` avant l'affichage, mais j'ai voulu prendre le cas le plus courant, le format de boucle que vous retrouverez le plus souvent. Rien ne vous empêche de faire les tests et je vous y encourage même.
6. Ici, on incrémente la variable `i` de 1. Si on est dans le premier tour de boucle, `i` passe donc de 0 à 1. Et alors, puisqu'il s'agit de la fin du bloc d'instructions, on revient à l'instruction `while`. `while` vérifie que la valeur de `i` est toujours inférieure à 10. Si c'est le cas (et ça l'est pour l'instant), on exécute à nouveau le bloc d'instructions. En tout, on exécute ce bloc 10 fois, jusqu'à ce que `i` passe de 9 à 10. Alors, l'instruction `while` vérifie la condition, se rend compte qu'elle est à présent fausse (la valeur de `i` n'est pas inférieure à 10 puisqu'elle est maintenant égale à 10) et s'arrête. S'il y avait du code après le bloc, il serait à présent exécuté.

N'oubliez pas d'incrémenter `i` ! Sinon, vous créez ce qu'on appelle une boucle infinie, puisque la valeur de `i` n'est jamais supérieure à 10 et la condition du `while`, par conséquent, toujours vraie... La boucle s'exécute donc à l'infini, du moins en théorie. Si votre ordinateur se lance dans une boucle infinie à cause de votre programme, pour interrompre la boucle, vous devrez taper CTRL + C dans la fenêtre de l'interpréteur (sous Windows ou Linux). Python ne le fera pas tout seul car, pour lui, il se passe bel et bien quelque chose. De toute façon, il est incapable de différencier une boucle infinie d'une boucle finie : c'est au programmeur de le faire.

La boucle `for`

Comme je l'ai dit précédemment, on retrouve l'instruction `while` dans la plupart des autres langages. Dans le C++ ou le Java, on retrouve également des instructions `for` mais qui n'ont pas le même sens. C'est assez particulier et c'est le point sur lequel je risque de manquer d'exemples dans l'immédiat, toute son utilité se révélant au chapitre sur les listes. Notez que, si vous avez fait du Perl ou du PHP, vous pouvez retrouver les boucles `for` sous un mot-clé assez proche : `foreach`.

L'instruction `for` travaille sur des séquences. Elle est en fait spécialisée dans le parcours d'une séquence de plusieurs données. Nous n'avons pas vu (et nous ne verrons pas tout de suite) ces séquences assez particulières mais très répandues, même si elles peuvent se révéler complexes. Toutefois, il en existe un type que nous avons rencontré depuis quelque temps déjà : les chaînes de caractères.

Les chaînes de caractères sont des séquences... de caractères ! Vous pouvez parcourir une chaîne de caractères (ce qui est également possible avec `while` mais nous verrons plus tard comment). Pour l'instant, intéressons-nous à `for`.

L'instruction `for` se construit ainsi :

```
1  for element in sequence:
```

`element` est une variable créée par le `for`, ce n'est pas à vous de l'instancier. Elle prend successivement chacune des valeurs figurant dans la séquence parcourue.

Ce n'est pas très clair ? Alors, comme d'habitude, tout s'éclaire avec le code !

```
1  chaine = "Bonjour les ZEROS"
2  for lettre in chaine:
3      print(lettre)
```

Ce qui nous donne le résultat suivant :

```
1  B
2  o
3  n
4  j
5  o
6  u
7  r
8
9  l
10 e
11 s
12
13 Z
14 E
15 R
16 0
17 S
```

Est-ce plus clair ? En fait, la variable `lettre` prend successivement la valeur de chaque lettre contenue dans la chaîne de caractères (d'abord B, puis o, puis n...). On affiche ces valeurs avec `print` et cette fonction revient à la ligne après chaque message, ce qui fait que toutes les lettres sont sur une seule colonne. Littéralement, la ligne 2 signifie « **pour lettre dans chaine** ». Arrivé à cette ligne, l'interpréteur va créer une variable `lettre` qui contiendra le premier élément de la chaîne (autrement dit, la première lettre). Après l'exécution du bloc, la variable `lettre` contient la seconde lettre, et ainsi de suite tant qu'il y a une lettre dans la chaîne.

Notez bien que, du coup, il est inutile d'incrémenter la variable `lettre` (ce qui serait d'ailleurs assez ridicule vu que ce n'est pas un nombre). Python se charge de l'incrémentation, c'est l'un des grands avantages de l'instruction `for`.

À l'instar des conditions que nous avons vues jusqu'ici, `in` peut être utilisée ailleurs que dans une boucle `for`.

```
1 chaine = "Bonjour les ZER0S"
2 for lettre in chaine:
3     if lettre in "AEIOUYaeiouy": # lettre est une voyelle
4         print(lettre)
5     else: # lettre est une consonne... ou plus exactement,
          lettre n'est pas une voyelle
6         print("*")
```

... ce qui donne :

```
1  *
2  o
3  *
4  *
5  o
6  u
7  *
8  *
9  *
10 e
11 *
12 *
13 *
14 E
15 *
16 *
17 *
```

Voilà ! L'interpréteur affiche les lettres si ce sont des voyelles et, sinon, il affiche des « * ». Notez bien que le 0 n'est pas affiché à la fin, Python ne se doute nullement qu'il s'agit d'un « o » stylisé.

Retenez bien cette utilisation de `in` dans une condition. On cherche à savoir si un élément quelconque est contenu dans un ensemble donné (ici, si la lettre est contenue dans « AEIOUYaeiouy », c'est-à-dire si `lettre` est une voyelle). On retrouvera plus

loin cette fonctionnalité.

Un petit bonus : les mots-clés `break` et `continue`

Je vais ici vous montrer deux nouveaux mots-clés, `break` et `continue`. Vous ne les utiliserez peut-être pas beaucoup mais vous devez au moins savoir qu'ils existent... et à quoi ils servent.

Le mot-clé `break`

Le mot-clé `break` permet tout simplement d'interrompre une boucle. Il est souvent utilisé dans une forme de boucle que je n'approuve pas trop :

```
1 while 1: # 1 est toujours vrai -> boucle infinie
2     lettre = input("Tapez 'Q' pour quitter : ")
3     if lettre == "Q":
4         print("Fin de la boucle")
5         break
```

La boucle `while` a pour condition `1`, c'est-à-dire une condition qui sera *toujours* vraie. Autrement dit, en regardant la ligne du `while`, on pense à une boucle infinie. En pratique, on demande à l'utilisateur de taper une lettre (un 'Q' pour quitter). Tant que l'utilisateur ne saisit pas cette lettre, le programme lui redemande de taper une lettre. Quand il tape 'Q', le programme affiche `Fin de la boucle` et la boucle s'arrête grâce au mot-clé `break`.

Ce mot-clé permet d'arrêter une boucle quelle que soit la condition de la boucle. Python sort immédiatement de la boucle et exécute le code qui suit la boucle, s'il y en a.

C'est un exemple un peu simpliste mais vous pouvez voir l'idée d'ensemble. Dans ce cas-là et, à mon sens, dans la plupart des cas où `break` est utilisé, on pourrait s'en sortir en précisant une véritable condition à la ligne du `while`. Par exemple, pourquoi ne pas créer un booléen qui sera **vrai** tout au long de la boucle et **faux** quand la boucle doit s'arrêter ? Ou bien tester directement si `lettre != « Q »` dans le `while` ?

Parfois, `break` est véritablement utile et fait gagner du temps. Mais ne l'utilisez pas à outrance, préférez une boucle avec une condition claire plutôt qu'un bloc d'instructions avec un `break`, qui sera plus dur à appréhender d'un seul coup d'œil.

Le mot-clé `continue`

Le mot-clé `continue` permet de... continuer une boucle, en repartant directement à la ligne du `while` ou `for`. Un petit exemple s'impose, je pense :

```
1 i = 1
2 while i < 20: # Tant que i est inférieure à 20
3     if i % 3 == 0:
4         i += 4 # On ajoute 4 à i
```

```
5           print("On incrémente i de 4. i est maintenant égale à",
                i)
6           continue # On retourne au while sans exécuter les
                autres lignes
7       print("La variable i =", i)
8       i += 1 # Dans le cas classique on ajoute juste 1 à i
```

Voici le résultat :

```
1  La variable i = 1
2  La variable i = 2
3  On incrémente i de 4. i est maintenant égale à 7
4  La variable i = 7
5  La variable i = 8
6  On incrémente i de 4. i est maintenant égale à 13
7  La variable i = 13
8  La variable i = 14
9  On incrémente i de 4. i est maintenant égale à 19
10 La variable i = 19
```

Comme vous le voyez, tous les trois tours de boucle, `i` s'incrémente de 4. Arrivé au mot-clé `continue`, Python n'exécute pas la fin du bloc mais revient au début de la boucle en testant à nouveau la condition du `while`. Autrement dit, quand Python arrive à la ligne 6, il saute à la ligne 2 sans exécuter les lignes 7 et 8. Au nouveau tour de boucle, Python reprend l'exécution normale de la boucle (`continue` n'ignore la fin du bloc que pour le tour de boucle courant).

Mon exemple ne démontre pas de manière éclatante l'utilité de `continue`. Les rares fois où j'utilise ce mot-clé, c'est par exemple pour supprimer des éléments d'une liste, mais nous n'avons pas encore vu les listes. L'essentiel, pour l'instant, c'est que vous vous souveniez de ces deux mots-clés et que vous sachiez ce qu'ils font, si vous les rencontrez au détour d'une instruction. Personnellement, je n'utilise pas très souvent ces mots-clés mais c'est aussi une question de goût.

En résumé

- Une boucle sert à répéter une portion de code en fonction d'un prédicat.
- On peut créer une boucle grâce au mot-clé `while` suivi d'un prédicat.
- On peut parcourir une séquence grâce à la syntaxe `for element in sequence:`.

Chapitre 6

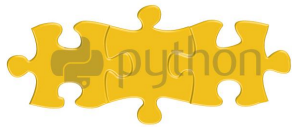
Pas à pas vers la modularité (1/2)

Difficulté :

En programmation, on est souvent amené à utiliser plusieurs fois des groupes d'instructions dans un but très précis. Attention, je ne parle pas ici de boucles. Simplement, vous pourrez vous rendre compte que la plupart de nos tests pourront être regroupés dans des blocs plus vastes, fonctions ou modules. Je vais détailler tranquillement ces deux concepts.

Les fonctions permettent de regrouper plusieurs instructions dans un bloc qui sera appelé grâce à un nom. D'ailleurs, vous avez déjà vu des fonctions : `print` et `input` en font partie par exemple.

Les modules permettent de regrouper plusieurs fonctions selon le même principe. Toutes les fonctions mathématiques, par exemple, peuvent être placées dans un module dédié aux mathématiques.



Les fonctions : à vous de jouer

Nous avons utilisé pas mal de fonctions depuis le début de ce tutoriel. On citera pour mémoire `print`, `type` et `input`, sans compter quelques autres. Mais vous devez bien vous rendre compte qu'il existe un nombre incalculable de fonctions déjà construites en Python. Toutefois, vous vous apercevrez aussi que, très souvent, un programmeur crée ses propres fonctions. C'est le premier pas que vous ferez, dans ce chapitre, vers la **modularité**. Ce terme un peu barbare signifie que nous allons nous habituer à regrouper dans des fonctions des parties de notre code que nous serons amenés à réutiliser. Au prochain chapitre, nous apprendrons à regrouper nos fonctions ayant un rapport entre elles dans un fichier, pour constituer un module, mais n'anticipons pas.

La création de fonctions

Nous allons, pour illustrer cet exemple, reprendre le code de la table de multiplication, que nous avons vu au chapitre précédent et qui, décidément, n'en finit pas de vous poursuivre.

Nous allons emprisonner notre code calculant la table de multiplication par 7 dans une fonction que nous appellerons `table_par_7`.

On crée une fonction selon le schéma suivant :

```
def nom_de_la_fonction(parametre1, parametre2, parametre3,
    parametreN):
    # Bloc d'instructions
```

Les blocs d'instructions nous courent après aussi, quel enfer. Si l'on décortique la ligne de définition de la fonction, on trouve dans l'ordre :

- `def`, mot-clé qui est l'abréviation de « define » (définir, en anglais) et qui constitue le prélude à toute construction de fonction.
- Le nom de la fonction, qui se nomme exactement comme une variable (nous verrons par la suite que ce n'est pas par hasard). N'utilisez pas un nom de variable déjà instanciée pour nommer une fonction.
- La liste des paramètres qui seront fournis lors d'un appel à la fonction. Les paramètres sont séparés par des virgules et la liste est encadrée par des parenthèses ouvrante et fermante (là encore, les espaces sont optionnels mais améliorent la lisibilité).
- Les deux points, encore et toujours, qui clôturent la ligne.

Les parenthèses sont obligatoires, quand bien même votre fonction n'attendrait aucun paramètre.

Le code pour mettre notre table de multiplication par 7 dans une fonction serait donc :

```
def table_par_7():
    nb = 7
    i = 0 # Notre compteur ! L'auriez-vous oublié ?
    while i < 10: # Tant que i est strictement inférieure à 10,
        print(i + 1, "*", nb, "=", (i + 1) * nb)
        i += 1 # On incrémente i de 1 à chaque tour de boucle.
```


Quand vous exécutez ce code à l'écran, il ne se passe rien. Une fois que vous avez retrouvé les trois chevrons, essayez d'appeler la fonction :

```
>>> table_par_7()
1 * 7 = 7
2 * 7 = 14
3 * 7 = 21
4 * 7 = 28
5 * 7 = 35
6 * 7 = 42
7 * 7 = 49
8 * 7 = 56
9 * 7 = 63
10 * 7 = 70
>>>
```

Bien, c'est, euh, exactement ce qu'on avait réussi à faire au chapitre précédent et l'intérêt ne saute pas encore aux yeux. L'avantage est que l'on peut appeler facilement la fonction et réafficher toute la table sans avoir besoin de tout réécrire !

Mais, si on saisit des paramètres pour pouvoir afficher la table de 5 ou de 8...?

Oui, ce serait déjà bien plus utile. Je ne pense pas que vous ayez trop de mal à trouver le code de la fonction :

```
def table(nb):
    i = 0
    while i < 10: # Tant que i est strictement inférieure à 10,
        print(i + 1, "*", nb, "=", (i + 1) * nb)
        i += 1 # On incrémente i de 1 à chaque tour de boucle.
```

Et là, vous pouvez passer en argument différents nombres, `table(8)` pour afficher la table de multiplication par 8 par exemple.

On peut aussi envisager de passer en paramètre le nombre de valeurs à afficher dans la table.

```
def table(nb, max):
    i = 0
    while i < max: # Tant que i est strictement inférieure à la
        variable max,
        print(i + 1, "*", nb, "=", (i + 1) * nb)
        i += 1
```

Si vous tapez à présent `table(11, 20)`, l'interpréteur vous affichera la table de 11, de 1*11 à 20*11. Magique non ?

Dans le cas où l'on utilise plusieurs paramètres sans les nommer, comme ici, il faut respecter l'ordre d'appel des paramètres, cela va de soi. Si vous commencez à mettre le nombre d'affichages en premier paramètre alors que, dans la définition, c'était le second, vous risquez d'avoir quelques surprises. Il est possible d'appeler les paramètres dans le désordre mais il faut, dans ce cas, préciser leur nom : nous verrons cela plus loin.

Si vous fournissez en second paramètre un nombre négatif, vous avez toutes les chances de créer une magnifique boucle infinie... vous pouvez l'empêcher en rajoutant des vérifications avant la boucle : par exemple, si le nombre est négatif ou nul, je le mets à 10. En Python, on préférera mettre un commentaire en tête de fonction ou une `docstring`, comme on le verra ultérieurement, pour indiquer que `max` doit être positif, plutôt que de faire des vérifications qui au final feront perdre du temps. Une des phrases reflétant la philosophie du langage et qui peut s'appliquer à ce type de situation est « *we're all consenting adults here* » [1] (sous-entendu, quelques avertissements en commentaires sont plus efficaces qu'une restriction au niveau du code). On aura l'occasion de retrouver cette phrase plus loin, surtout quand on parlera des objets.

Valeurs par défaut des paramètres

On peut également préciser une valeur par défaut pour les paramètres de la fonction. Vous pouvez par exemple indiquer que le nombre maximum d'affichages doit être de 10 par défaut (c'est-à-dire si l'utilisateur de votre fonction ne le précise pas). Cela se fait le plus simplement du monde :

```
1 def table(nb, max=10):
2     """Fonction affichant la table de multiplication par nb
3     de 1*nb à max*nb
4
5     (max >= 0)"""
6     i = 0
7     while i < max:
8         print(i + 1, "*", nb, "=", (i + 1) * nb)
9         i += 1
```

Il suffit de rajouter `=10` après `max`. À présent, vous pouvez appeler la fonction de deux façons : soit en précisant le numéro de la table et le nombre maximum d'affichages, soit en ne précisant que le numéro de la table (`table(7)`). Dans ce dernier cas, `max` vaudra 10 par défaut.

J'en ai profité pour ajouter quelques lignes d'explications que vous aurez sans doute remarquées. Nous avons placé une chaîne de caractères, sans la capturer dans une variable, juste en-dessous de la définition de la fonction. Cette chaîne est ce qu'on

1. « Nous sommes entre adultes consentants ».

appelle une `docstring` que l'on pourrait traduire par une chaîne d'aide. Si vous tapez `help(table)`, c'est ce message que vous verrez apparaître. Documenter vos fonctions est également une bonne habitude à prendre. Comme vous le voyez, on indente cette chaîne et on la met entre triple guillemets. Si la chaîne figure sur une seule ligne, on pourra mettre les trois guillemets fermants sur la même ligne ; sinon, on préférera sauter une ligne avant de fermer cette chaîne, pour des raisons de lisibilité. Tout le texte d'aide est indenté au même niveau que le code de la fonction.

Enfin, sachez que l'on peut appeler des paramètres par leur nom. Cela est utile pour une fonction comptant un certain nombre de paramètres qui ont tous une valeur par défaut. Vous pouvez aussi utiliser cette méthode sur une fonction sans paramètre par défaut, mais c'est moins courant.

Prenons un exemple de définition de fonction :

```
1 def fonc(a=1, b=2, c=3, d=4, e=5):
2     print("a =", a, "b =", b, "c =", c, "d =", d, "e =", e)
```

Simple, n'est-ce pas ? Eh bien, vous avez de nombreuses façons d'appeler cette fonction. En voici quelques exemples :

Instruction	Résultat
`fonc()`	a = 1 b = 2 c = 3 d = 4 e = 5
`fonc(4)`	a = 4 b = 2 c = 3 d = 4 e = 5
`fonc(b=8, d=5)`	a = 1 b = 8 c = 3 d = 5 e = 5
`fonc(b=35, c=48, a=4, e=9)`	a = 4 b = 35 c = 48 d = 4 e = 9

Je ne pense pas que des explications supplémentaires s'imposent. Si vous voulez changer la valeur d'un paramètre, vous tapez son nom, suivi d'un signe égal puis d'une valeur (qui peut être une variable bien entendu). Peu importent les paramètres que vous précisez (comme vous le voyez dans cet exemple où tous les paramètres ont une valeur par défaut, vous pouvez appeler la fonction sans paramètre), peu importe l'ordre d'appel des paramètres.

Signature d'une fonction

On entend par « signature de fonction » les éléments qui permettent au langage d'identifier ladite fonction. En C++, par exemple, la signature d'une fonction est constituée de son nom et du type de chacun de ses paramètres. Cela veut dire que l'on peut trouver plusieurs fonctions portant le même nom mais dont les paramètres diffèrent. Au moment de l'appel de fonction, le compilateur recherche la fonction qui s'applique à cette signature.

En Python comme vous avez pu le voir, on ne précise pas les types des paramètres. Dans ce langage, la signature d'une fonction est tout simplement son nom. Cela signifie que vous ne pouvez définir deux fonctions du même nom (si vous le faites, l'ancienne définition est écrasée par la nouvelle).

```
1 def exemple():
```

```
2       print("Un exemple d'une fonction sans paramètre")
3
4   exemple()
5
6   def exemple(): # On redéfinit la fonction exemple
7       print("Un autre exemple de fonction sans paramètre")
8
9   exemple()
```

A la ligne 1 on définit la fonction `exemple`. On l'appelle une première fois à la ligne 4. On redéfinit à la ligne 6 la fonction `exemple`. L'ancienne définition est écrasée et l'ancienne fonction ne pourra plus être appelée.

Retenez simplement que, comme pour les variables, un nom de fonction ne renvoie que vers une fonction unique, on ne peut surcharger de fonctions en Python.

L'instruction `return`

Ce que nous avons fait était intéressant, mais nous n'avons pas encore fait le tour des possibilités de la fonction. Et d'ailleurs, même à la fin de ce chapitre, il nous restera quelques petites fonctionnalités à voir. Si vous vous souvenez bien, il existe des fonctions comme `print` qui ne renvoient rien (attention, « renvoyer » et « afficher » sont deux choses différentes) et des fonctions telles que `input` ou `type` qui renvoient une valeur. Vous pouvez capturer cette valeur en plaçant une variable devant (exemple `variable2 = type(variable1)`). En effet, les fonctions travaillent en général sur des données et renvoient le résultat obtenu, suite à un calcul par exemple.

Prenons un exemple simple : une fonction chargée de mettre au carré une valeur passée en argument. Je vous signale au passage que Python en est parfaitement capable sans avoir à coder une nouvelle fonction, mais c'est pour l'exemple.

```
1   def carre(valeur):
2       return valeur * valeur
```

L'instruction `return` signifie qu'on va **renvoyer**[2] la valeur, pour pouvoir la récupérer ensuite et la stocker dans une variable par exemple. Cette instruction arrête le déroulement de la fonction, le code situé après le `return` ne s'exécutera pas.

```
1   variable = carre(5)
```

La variable `variable` contiendra, après exécution de cette instruction, 5 au carré, c'est-à-dire 25.

Sachez que l'on peut renvoyer plusieurs valeurs que l'on sépare par des virgules, et que l'on peut les capturer dans des variables également séparées par des virgules, mais je m'attarderai plus loin sur cette particularité. Retenez simplement la définition d'une fonction, les paramètres, les valeurs par défaut, l'instruction `return` et ce sera déjà bien.

2. Certains d'entre vous ont peut-être l'habitude d'employer le mot « retourner » ; il s'agit d'un anglicisme et je lui préfère l'expression « renvoyer ».

Les fonctions `lambda`

Nous venons de voir comment créer une fonction grâce au mot-clé `def`. Python nous propose un autre moyen de créer des fonctions, des fonctions extrêmement courtes car limitées à une seule instruction.

Pourquoi une autre façon de créer des fonctions ? La première suffit, non ?

Disons que ce n'est pas tout à fait la même chose, comme vous allez le voir. Les fonctions lambda sont en général utilisées dans un certain contexte, pour lequel définir une fonction à l'aide de `def` serait plus long et moins pratique.

Syntaxe

Avant tout, voyons la syntaxe d'une définition de fonction `lambda`. Nous allons utiliser le mot-clé `lambda` comme ceci : `lambda arg1, arg2,... : instruction de retour`.

Je pense qu'un exemple vous semblera plus clair. On veut créer une fonction qui prend un paramètre et renvoie ce paramètre au carré.

```
1   >>> lambda x: x * x
2   <function <lambda> at 0x00BA1B70>
3   >>>
```

D'abord, on a le mot-clé `lambda` suivi de la liste des arguments, séparés par des virgules. Ici, il n'y a qu'un seul argument, c'est `x`. Ensuite figure un nouveau signe deux points « : » et l'instruction de la `lambda`. C'est le résultat de l'instruction que vous placez ici qui sera renvoyé par la fonction. Dans notre exemple, on renvoie donc `x * x`.

Comment fait-on pour appeler notre `lambda` ?

On a bien créé une fonction `lambda` mais on ne dispose ici d'aucun moyen pour l'appeler. Vous pouvez tout simplement stocker votre fonction `lambda` nouvellement définie dans une variable, par une simple affectation :

```
1   >>> f = lambda x: x * x
2   >>> f(5)
3   25
4   >>> f(-18)
5   324
6   >>>
```

Un autre exemple : si vous voulez créer une fonction `lambda` prenant deux paramètres et renvoyant la somme de ces deux paramètres, la syntaxe sera la suivante :

```
lambda x, y: x + y
```

Utilisation

À notre niveau, les fonctions `lambda` sont plus une curiosité que véritablement utiles. Je vous les présente maintenant parce que le contexte s'y prête et que vous pourriez en rencontrer certaines sans comprendre ce que c'est.

Il vous faudra cependant attendre un peu pour que je vous montre une réelle application des `lambda`. En attendant, n'oubliez pas ce mot-clé et la syntaxe qui va avec... on passe à la suite !

À la découverte des modules

Jusqu'ici, nous avons travaillé avec les fonctions de Python chargées au lancement de l'interpréteur. Il y en a déjà un certain nombre et nous pourrions continuer et finir cette première partie sans utiliser de module Python... ou presque. Mais il faut bien qu'à un moment, je vous montre cette possibilité des plus intéressantes !

Les modules, qu'est-ce que c'est ?

Un module est grossièrement un bout de code que l'on a enfermé dans un fichier. On emprisonne ainsi des fonctions et des variables ayant toutes un rapport entre elles. Ainsi, si l'on veut travailler avec les fonctionnalités prévues par le module (celles qui ont été enfermées dans le module), il n'y a qu'à **importer** le module et utiliser ensuite toutes les fonctions et variables prévues.

Il existe un grand nombre de modules disponibles avec Python sans qu'il soit nécessaire d'installer des bibliothèques supplémentaires. Pour cette partie, nous prendrons l'exemple du module `math` qui contient, comme son nom l'indique, des fonctions mathématiques. Inutile de vous inquiéter, nous n'allons pas nous attarder sur le module lui-même pour coder une calculatrice scientifique, nous verrons surtout les différentes méthodes d'importation.

La méthode `import`

Lorsque vous ouvrez l'interpréteur Python, les fonctionnalités du module `math` ne sont pas incluses. Il s'agit en effet d'un module, il vous appartient de l'importer si vous vous dites « tiens, mon programme risque d'avoir besoin de fonctions mathématiques ». Nous allons voir une première syntaxe d'importation.

```
>>> import math
>>>
```


La syntaxe est facile à retenir : le mot-clé `import`, qui signifie « importer » en anglais, suivi du nom du module, ici `math`.

Après l'exécution de cette instruction, rien ne se passe... en apparence. En réalité, Python vient d'importer le module `math`. Toutes les fonctions mathématiques contenues dans ce module sont maintenant accessibles. Pour appeler une fonction du module, il faut taper le nom du module suivi d'un point « . » puis du nom de la fonction. C'est la même syntaxe pour appeler des variables du module. Voyons un exemple :

```
>>> math.sqrt(16)
4
>>>
```

Comme vous le voyez, la fonction `sqrt` du module `math` renvoie la racine carrée du nombre passé en paramètre.

Mais comment suis-je censé savoir quelles fonctions existent et ce que fait `math.sqrt` dans ce cas précis ?

J'aurais dû vous montrer cette fonction bien plus tôt car, oui, c'est une fonction qui va nous donner la solution. Il s'agit de `help`, qui prend en argument la fonction ou le module sur lequel vous demandez de l'aide. L'aide est fournie en anglais mais c'est de l'anglais technique, c'est-à-dire une forme de l'anglais que vous devrez maîtriser pour programmer, si ce n'est pas déjà le cas. Une grande majorité de la documentation est en anglais, bien que vous puissiez maintenant en trouver une bonne part en français.

```
>>> help("math")
Help on built-in module math:

NAME
    math

FILE
    (built-in)

DESCRIPTION
    This module is always available. It provides access to the
    mathematical functions defined by the C standard.

FUNCTIONS
    acos(...)
        acos(x)

        Return the arc cosine (measured in radians) of x.

    acosh(...)
```

```
21          acosh(x)
22
23          Return the hyperbolic arc cosine (measured in radians)
                of x.
24
25      asin(...)
26  -- Suite  --
```

Si vous parlez un minimum l'anglais, vous avez accès à une description exhaustive des fonctions du module `math`. Vous voyez en haut de la page le nom du module, le fichier qui l'héberge, puis la description du module. Ensuite se trouve une liste des fonctions, chacune étant accompagnée d'une courte description.

Tapez `Q` pour revenir à la fenêtre d'interpréteur, `Espace` pour avancer d'une page, `Entrée` pour avancer d'une ligne. Vous pouvez également passer un nom de fonction en paramètre de la fonction `help`.

```
>>> help("math.sqrt")
Help on built-in function sqrt in module math:

sqrt(...)
    sqrt(x)

    Return the square root of x.

>>>
```

Ne mettez pas les parenthèses habituelles après le nom de la fonction. C'est en réalité la référence de la fonction que vous envoyez à `help`. Si vous rajoutez les parenthèses ouvrantes et fermantes après le nom de la fonction, vous devrez préciser une valeur. Dans ce cas, c'est la valeur renvoyée par `math.sqrt` qui sera analysée, soit un nombre (entier ou flottant).

Nous reviendrons plus tard sur le concept des références des fonctions. Si vous avez compris pourquoi il ne fallait pas mettre de parenthèses après le nom de la fonction dans `help`, tant mieux. Sinon, ce n'est pas grave, nous y reviendrons en temps voulu.

Utiliser un espace de noms spécifique

En vérité, quand vous tapez `import math`, cela crée un espace de noms dénommé « math », contenant les variables et fonctions du module `math`. Quand vous tapez `math.sqrt(25)`, vous précisez à Python que vous souhaitez exécuter la fonction `sqrt` contenue dans l'espace de noms `math`. Cela signifie que vous pouvez avoir, dans l'espace de noms principal, une autre fonction `sqrt` que vous avez définie vous-mêmes. Il n'y aura pas de conflit entre, d'une part, la fonction que vous avez créée et que vous appellerez grâce à l'instruction `sqrt` et, d'autre part, la fonction `sqrt` du module `math`

que vous appellerez grâce à l'instruction `math.sqrt`.

Mais, concrètement, un espace de noms, c'est quoi ?

Il s'agit de regrouper certaines fonctions et variables sous un préfixe spécifique. Prenons un exemple concret :

```
import math
a = 5
b = 33.2
```

Dans l'espace de noms principal, celui qui ne nécessite pas de préfixe et que vous utilisez depuis le début de ce tutoriel, on trouve :

- La variable `a`.
- La variable `b`.
- Le module `math`, qui se trouve dans un espace de noms s'appelant `math` également. Dans cet espace de noms, on trouve :
  - la fonction `sqrt` ;
  - la variable `pi` ;
  - et bien d'autres fonctions et variables...

C'est aussi l'intérêt des modules : des variables et fonctions sont stockées à part, bien à l'abri dans un espace de noms, sans risque de conflit avec vos propres variables et fonctions. Mais dans certains cas, vous pourrez vouloir changer le nom de l'espace de noms dans lequel sera stocké le module importé.

```
import math as mathematiques
mathematiques.sqrt(25)
```

Qu'est-ce qu'on a fait là ?

On a simplement importé le module `math` en spécifiant à Python de l'héberger dans l'espace de noms dénommé « mathematiques » au lieu de `math`. Cela permet de mieux contrôler les espaces de noms des modules que vous importerez. Dans la plupart des cas, vous n'utiliserez pas cette fonctionnalité mais, au moins, vous savez qu'elle existe. Quand on se penchera sur les packages, vous vous souviendrez probablement de cette possibilité.

Une autre méthode d'importation : `from ... import ...`

Il existe une autre méthode d'importation qui ne fonctionne pas tout à fait de la même façon. En fonction du résultat attendu, j'utilise indifféremment l'une ou l'autre de ces méthodes. Reprenons notre exemple du module `math`. Admettons que nous ayons

uniquement besoin, dans notre programme, de la fonction renvoyant la valeur absolue d'une variable. Dans ce cas, nous n'allons importer que la fonction, au lieu d'importer tout le module.

```
1 >>> from math import fabs
2 >>> fabs(-5)
3 5
4 >>> fabs(2)
5 2
6 >>>
```

Pour ceux qui n'ont pas encore étudié les valeurs absolues, il s'agit tout simplement de l'opposé de la variable si elle est négative, et de la variable elle-même si elle est positive. Une valeur absolue est ainsi toujours positive.

Vous aurez remarqué qu'on ne met plus le préfixe `math.` devant le nom de la fonction. En effet, nous l'avons importée avec la méthode `from` : celle-ci charge la fonction depuis le module indiqué et la place dans l'interpréteur au même plan que les fonctions existantes, comme `print` par exemple. Si vous avez compris les explications sur les espaces de noms, vous voyez que `print` et `fabs` sont dans le même espace de noms (principal).

Vous pouvez appeler toutes les variables et fonctions d'un module en tapant « * » à la place du nom de la fonction à importer.

```
1 >>> from math import *
2 >>> sqrt(4)
3 2
4 >>> fabs(5)
5 5
```

À la ligne 1 de notre programme, l'interpréteur a parcouru toutes les fonctions et variables du module `math` et les a importées directement dans l'espace de noms principal sans les emprisonner dans l'espace de noms `math`.

Bilan

Quelle méthode faut-il utiliser ?

Vaste question ! Je dirais que c'est à vous de voir. La seconde méthode a l'avantage inestimable d'économiser la saisie systématique du nom du module en préfixe de chaque fonction. L'inconvénient de cette méthode apparaît si l'on utilise plusieurs modules de cette manière : si par hasard il existe dans deux modules différents deux fonctions portant le même nom, l'interpréteur ne conservera que la dernière fonction appelée[3]. Conclusion... c'est à vous de voir en fonction de vos besoins !

3. Je vous rappelle qu'il ne peut y avoir deux variables ou fonctions portant le même nom.

En résumé

- Une fonction est une portion de code contenant des instructions, que l'on va pouvoir réutiliser facilement.
- Découper son programme en fonctions permet une meilleure organisation.
- Les fonctions peuvent recevoir des informations en entrée et renvoyer une information grâce au mot-clé `return`.
- Les fonctions se définissent de la façon suivante : `def nom_fonction(parametre1, parametre2, parametreN):`

Chapitre 10

Notre premier objet : les chaînes de caractères

Difficulté :

Les objets... vaste sujet ! Avant d'en créer, nous allons d'abord voir de quoi il s'agit. Nous allons commencer avec les chaînes de caractères, un type que vous pensez bien connaître.

Dans ce chapitre, vous allez découvrir petit à petit le mécanisme qui se cache derrière la notion d'objet. Ces derniers font partie des notions incontournables en Python, étant donné que tout ce que nous avons utilisé jusqu'ici... est un objet !

Vous avez dit objet ?

La première question qui risque de vous empêcher de dormir si je n'y réponds pas tout de suite, c'est :

Mais c'est quoi un objet ?

Eh bien, j'ai lu beaucoup de définitions très différentes et je n'ai pas trouvé de point commun à toutes ces définitions. Nous allons donc partir d'une définition incomplète mais qui suffira pour l'instant : **un objet est une structure de données, comme les variables, qui peut contenir elle-même d'autres variables et fonctions.** On étoffera plus loin cette définition, elle suffit bien pour le moment.

Je ne comprends rien. Passe encore qu'une variable en contienne d'autres, après tout les chaînes de caractères contiennent bien des caractères, mais qu'une variable contienne des fonctions... Cela rime à quoi ?

Je pourrais passer des heures à expliquer la théorie du concept que vous n'en seriez pas beaucoup plus avancés. J'ai choisi de vous montrer les objets par l'exemple et donc, vous allez très rapidement voir ce que tout cela signifie. Mais vous allez devoir me faire confiance, au début, sur l'utilité de la méthode objet.

Avant d'attaquer, une petite précision. J'ai dit qu'un objet était un peu comme une variable... en fait, pour être exact, il faut dire qu'une variable est un objet. Toutes les variables avec lesquelles nous avons travaillé jusqu'ici sont des objets. Les fonctions que nous avons vues sont également des objets. *En Python, tout est objet* : gardez cela à l'esprit.

Les méthodes de la classe `str`

Oh la la, j'en vois qui grimacent rien qu'en lisant le titre. Vous n'avez pourtant aucune raison de vous inquiéter ! On va y aller tout doucement.

Posons un problème : comment peut-on passer une chaîne de caractères en minuscules ? Si vous n'avez lu jusqu'à présent que les premiers chapitres, vous ne pourrez pas faire cet exercice ; j'ai volontairement évité de trop aborder les chaînes de caractères jusqu'ici. Mais admettons que vous arriviez à coder une fonction prenant en paramètre la chaîne en question. Vous aurez un code qui ressemble à ceci :

```
>>> chaine = "NE CRIE PAS SI FORT !"
>>> mettre_en_minuscule(chaine)
'ne crie pas si fort !'
```

Sachez que, dans les anciennes versions de Python, il y avait un module spécialisé dans

le traitement des chaînes de caractères. On importait ce module et on pouvait appeler la fonction passant une chaîne en minuscules. Ce module existe d'ailleurs encore et reste utilisé pour certains traitements spécifiques. Mais on va découvrir ici une autre façon de faire. Regardez attentivement :

```
>>> chaine = "NE CRIE PAS SI FORT !"
>>> chaine.lower() # Mettre la chaîne en minuscule
'ne crie pas si fort !'
```

La fonction `lower` est une nouveauté pour vous. Vous devez reconnaître le point « . » qui symbolisait déjà, dans le chapitre sur les modules, une relation d'appartenance (`a.b` signifiait que `b` était contenu dans `a`). Ici, il possède la même signification : la fonction `lower` est une fonction de la variable `chaine`.

La fonction `lower` est propre aux chaînes de caractères. Toutes les chaînes peuvent y faire appel. Si vous tapez `type(chaine)` dans l'interpréteur, vous obtenez `<class 'str'>`. Nous avons dit qu'une variable est issue d'un type de donnée. Je vais à présent reformuler : un **objet** est issu d'une **classe**. La **classe** est une forme de type de donnée, sauf qu'elle permet de définir des fonctions et variables propres au type. C'est pour cela que, dans toutes les chaînes de caractères, on peut appeler la fonction `lower`. C'est tout simplement parce que la fonction `lower` a été définie dans la classe `str`. Les fonctions définies dans une classe sont appelées des **méthodes**.

Récapitulons. Nous avons découvert :

- Les **objets**, que j'ai présentés comme des variables, pouvant contenir d'autres variables ou fonctions (que l'on appelle **méthodes**). On appelle une méthode d'un objet grâce à `objet.methode()`.
- Les **classes**, que j'ai présentées comme des types de données. Une classe est un modèle qui servira à construire un objet ; c'est dans la classe qu'on va définir les méthodes propres à l'objet.

Voici le mécanisme qui vous permet d'appeler la méthode `lower` d'une chaîne :

1. Les développeurs de Python ont créé la classe `str` qui est utilisée pour créer des chaînes de caractères. Dans cette classe, ils ont défini plusieurs méthodes, comme `lower`, qui pourront être utilisées par n'importe quel objet construit sur cette classe.
2. Quand vous écrivez `chaine = "NE CRIE PAS SI FORT !"`, Python reconnaît qu'il doit créer une chaîne de caractères. Il va donc créer un objet d'après la classe (le modèle) qui a été définie à l'étape précédente.
3. Vous pouvez ensuite appeler toutes les méthodes de la classe `str` depuis l'objet `chaine` que vous venez de créer.

Ouf ! Cela fait beaucoup de choses nouvelles, du vocabulaire et des concepts un peu particuliers.

Vous ne voyez peut-être pas encore tout l'intérêt d'avoir des méthodes définies dans une certaine classe. Cela permet d'abord de bien séparer les diverses fonctionnalités

(on ne peut pas passer en minuscules un nombre entier, cela n'a aucun sens). Ensuite, c'est plus intuitif, une fois passé le choc de la première rencontre.

Bon, on parle, on parle, mais on ne code pas beaucoup !

Mettre en forme une chaîne

Non, vous n'allez pas apprendre à mettre une chaîne en gras, souligné, avec une police Verdana de 15px... Nous ne sommes encore que dans une console. Nous venons de présenter `lower`, il existe d'autres méthodes. Avant tout, voyons un contexte d'utilisation.

Certains d'entre vous se demandent peut-être l'intérêt de passer des chaînes en minuscules... alors voici un petit exemple.

```
chaine = str() # Crée une chaîne vide
# On aurait obtenu le même résultat en tapant chaine = ""

while chaine.lower() != "q":
    print("Tapez 'Q' pour quitter...")
    chaine = input()

print("Merci !")
```

Vous devez comprendre rapidement ce programme. Dans une boucle, on demande à l'utilisateur de taper la lettre « q » pour quitter. Tant que l'utilisateur saisit une autre lettre, la boucle continue de s'exécuter. Dès que l'utilisateur appuie sur la touche Q de son clavier, la boucle s'arrête et le programme affiche « Merci ! ». Cela devrait vous rappeler quelque chose... direction le TP de la partie 1 pour ceux qui ont la mémoire courte.

La petite nouveauté réside dans le test de la boucle : `chaine.lower() != "q"`. On prend la chaîne saisie par l'utilisateur, on la passe en minuscules et on regarde si elle est différente de « q ». Cela veut dire que l'utilisateur peut taper « q » en majuscule ou en minuscule, dans les deux cas la boucle s'arrêtera.

Notez que `chaine.lower()` renvoie la chaîne en minuscules mais ne modifie pas la chaîne. C'est très important, nous verrons pourquoi dans le prochain chapitre.

Notez aussi que nous avons appelé la fonction `str` pour créer une chaîne vide. Je ne vais pas trop compliquer les choses mais sachez qu'appeler ainsi un type en tant que fonction permet de créer un objet de la classe. Ici, `str()` crée un objet *chaîne de caractères*. Nous avons vu dans la première partie le mot-clé `int()`, qui crée aussi un entier[1].

Bon, voyons d'autres méthodes. Je vous invite à tester mes exemples (ils sont commentés, mais on retient mieux en essayant par soi-même).

```
>>> minuscules = "une chaine en minuscules"
>>> minuscules.upper() # Mettre en majuscules
'UNE CHAINE EN MINUSCULES'
```

1. Si nécessaire depuis un autre type, ce qui permet de convertir une chaîne en entier.

```
>>> minuscule.capitalize() # La première lettre en majuscule
'Une chaine en minuscules'
>>> espaces = "   une  chaine avec  des espaces   "
>>> espaces.strip() # On retire les espaces au début et à la
    fin de la chaîne
'une  chaine avec  des espaces'
>>> titre = "introduction"
>>> titre.upper().center(20)
'    INTRODUCTION    '
>>>
```

La dernière instruction mérite quelques explications.

On appelle d'abord la méthode `upper` de l'objet `titre`. Cette méthode, comme vous l'avez vu plus haut, renvoie en majuscules la chaîne de caractères contenue dans l'objet.

On appelle ensuite la méthode `center`, méthode que nous n'avons pas encore vue et qui permet de centrer une chaîne. On lui passe en paramètre la taille de la chaîne que l'on souhaite obtenir. La méthode va ajouter alternativement un espace au début et à la fin de la chaîne, jusqu'à obtenir la longueur demandée. Dans cet exemple, `titre` contient la chaîne `'introduction'`, chaîne qui (en minuscules ou en majuscules) mesure 12 caractères. On demande à `center` de centrer cette chaîne dans un espace de 20 caractères. La méthode `center` va donc placer 4 espaces avant le titre et 4 espaces après, pour faire 20 caractères en tout.

Bon, mais maintenant, sur quel objet travaille `center` ? Sur `titre` ? Non. Sur la chaîne renvoyée par `titre.upper()`, c'est-à-dire le titre en majuscules. C'est pourquoi on peut « chaîner » ces deux méthodes : `upper`, comme la plupart des méthodes de chaînes, travaille sur une chaîne et renvoie une chaîne… qui elle aussi va posséder les méthodes propres à une chaîne de caractères. Si ce n'est pas très clair, faites quelques tests, avec `titre.upper()` et `titre.center(20)`, en passant par une seconde variable si nécessaire, pour vous rendre compte du mécanisme ; ce n'est pas bien compliqué.

Je n'ai mis ici que quelques méthodes, il y en a bien d'autres. Vous pouvez en voir la liste dans l'aide, en tapant, dans l'interpréteur : `help(str)`.

Formater et afficher une chaîne

Attends, on a appris à faire cela depuis cinq bons chapitres ! On ne va pas tout réapprendre quand même ?

Heureusement que non ! Mais nous allons apprendre à considérer ce que nous savons à travers le modèle objet. Et vous allez vous rendre compte que, la plupart du temps, nous n'avons fait qu'effleurer les fonctionnalités du langage.

Je ne vais pas revenir sur ce que j'ai dit, pour afficher une chaîne, on passe par la fonction `print`.

```
chaine = "Bonjour tout le monde !"
```



```
print(chaine)
```

Rien de nouveau ici. En revanche, nous allons un peu changer nos habitudes en ce qui concerne l'affichage de plusieurs variables.

Jusqu'ici, nous avons utilisé `print` en lui imputant plusieurs paramètres. Cela fonctionne mais nous allons voir une méthode légèrement plus souple, qui d'ailleurs n'est pas seulement utile pour l'affichage.

```
>>> prenom = "Paul"
>>> nom = "Dupont"
>>> age = 21
>>> print("Je m'appelle {0} {1} et j'ai {2} ans.".format(prenom
    , nom, age))
Je m'appelle Paul Dupont et j'ai 21 ans.
```

Mais ! C'est quoi cela ?

Question légitime. Voyons un peu.

Première syntaxe de la méthode `format`

Nous avons utilisé une méthode de la classe `str` pour formater notre chaîne. De gauche à droite, nous avons :

- une chaîne de caractères qui ne présente rien de particulier, sauf ces accolades entourant des nombres, d'abord `0`, puis `1`, puis `2` ;
- nous appelons la méthode `format` de cette chaîne en lui passant en paramètres les variables à afficher, dans un ordre bien précis ;
- quand Python exécute cette méthode, il remplace dans notre chaîne `{0}` par la première variable passée à la méthode `format` (soit le prénom), `{1}` par la deuxième variable… et ainsi de suite.

Souvenez-vous qu'en programmation, on commence à compter à partir de 0.

Bien, mais on aurait pu faire exactement la même chose en passant plusieurs valeurs à `print`, non ?

Absolument. Mais rappelez-vous que cette fonctionnalité est bien plus puissante qu'un simple affichage, vous pouvez formater des chaînes de cette façon. Ici, nous avons directement affiché la chaîne formatée, mais nous aurions pu la stocker :

```
>>> nouvelle_chaine = "Je m'appelle {0} {1} et j'ai {2} ans.".
    format(prenom, nom, age)
>>>
```

Pour faire la même chose sans utiliser `format`, on aurait dû concaténer des chaînes, c'est-à-dire les mettre bout à bout en respectant une certaine syntaxe. Nous allons voir cela un peu plus loin mais cette solution reste plus élégante.

Dans cet exemple, nous avons appelé les variables dans l'ordre où nous les placions dans `format`, mais ce n'est pas une obligation. Considérez cet exemple :

```
>>> prenom = "Paul"
>>> nom = "Dupont"
>>> age = 21
>>> print( \
...     "Je m'appelle {0} {1} ({3} {0} pour l'administration) et
    j'ai {2} " \
...     "ans.".format(prenom, nom, age, nom.upper()))
Je m'appelle Paul Dupont (DUPONT Paul pour l'administration) et
     j'ai 21 ans.
```

J'ai coupé notre instruction, plutôt longue, à l'aide du signe « \ » placé avant un saut de ligne, pour indiquer à Python que l'instruction se prolongeait au-dessous.

Si vous avez du mal à comprendre l'exemple, relisez l'instruction en remplaçant vous-mêmes les nombres entre accolades par les variables.

Dans la plupart des cas, on ne précise pas le numéro de la variable entre accolades.

```
>>> date = "Dimanche 24 juillet 2011"
>>> heure = "17:00"
>>> print("Cela s'est produit le {}, à {}.".format(date, heure)
    )
Cela s'est produit le Dimanche 24 juillet 2011, à 17:00.
>>>
```

Naturellement, cela ne fonctionne que si vous donnez les variables dans le bon ordre dans `format`.

Cette syntaxe suffit la plupart du temps mais elle n'est pas forcément intuitive quand on insère beaucoup de variables : on doit retenir leur position dans l'appel à `format` pour comprendre laquelle est affichée à tel endroit. Mais il existe une autre syntaxe.

Seconde syntaxe de la méthode `format`

On peut également nommer les variables que l'on va afficher, c'est souvent plus intuitif que d'utiliser leur indice. Voici un nouvel exemple :

```
# formatage d'une adresse
adresse = """
  {no_rue}, {nom_rue}
  {code_postal} {nom_ville} ({pays})"""
  .format(no_rue=5, nom_rue="rue des Postes", code_postal=75003
      , nom_ville="Paris", pays="France")
print(adresse)
```

... affichera :

```
5, rue des Postes
  75003 Paris (France)
```

Je pense que vous voyez assez précisément en quoi consiste cette deuxième syntaxe de `format`. Au lieu de donner des nombres entre accolades, on spécifie des noms de variables qui doivent correspondre à ceux fournis comme mots-clés dans la méthode `format`. Je ne m'attarderai pas davantage sur ce point, je pense qu'il est assez clair comme cela.

La concaténation de chaînes

Nous allons glisser très rapidement sur le concept de concaténation, assez intuitif d'ailleurs. On cherche à regrouper deux chaînes en une, en mettant la seconde à la suite de la première. Cela se fait le plus simplement du monde :

```
>>> prenom = "Paul"
>>> message = "Bonjour"
>>> chaine_complete = message + prenom # On utilise le symbole
    '+' pour concaténer deux chaînes
... print(chaine_complete) # Résultat :
BonjourPaul
>>> # Pas encore parfait, il manque un espace
... # Qu'à cela ne tienne !
... chaine_complete = message + " " + prenom
>>> print(chaine_complete) # Résultat :
Bonjour Paul
>>>
```

C'est assez clair je pense. Le signe « + » utilisé pour ajouter des nombres est ici utilisé pour **concaténer** deux chaînes. Essayons à présent de concaténer des chaînes et des nombres :

```
>>> age = 21
>>> message = "J'ai " + age + " ans."
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
>>>
```

Python se fâche tout rouge ! Certains langages auraient accepté cette syntaxe sans sourciller mais Python n'aime pas cela du tout.

Au début de la première partie, nous avons dit que Python était un langage à **typage dynamique**, ce qui signifie qu'il identifie lui-même les types de données et que les variables peuvent changer de type au cours du programme. Mais Python est aussi un langage **fortement typé**, et cela veut dire que les types de données ne sont pas là juste pour faire joli, on ne peut pas les ignorer. Ainsi, on veut ici ajouter une chaîne à un entier et à une autre chaîne. Python ne comprend pas : est-ce que les chaînes contiennent des nombres qu'il doit convertir pour les ajouter à l'entier ou est-ce que l'entier doit être converti en chaîne puis concaténé avec les autres chaînes ? Python ne sait pas. Il ne le fera pas tout seul. Mais il se révèle être de bonne volonté puisqu'il suffit de lui demander de convertir l'entier pour pouvoir le concaténer aux autres chaînes.

```
>>> age = 21
>>> message = "J'ai " + str(age) + " ans."
>>> print(message)
J'ai 21 ans.
>>>
```

On appelle `str` pour convertir un objet en une chaîne de caractères, comme nous avons appelé `int` pour convertir un objet en entier. C'est le même mécanisme, sauf que convertir un entier en chaîne de caractères ne lèvera vraissemblablement aucune exception.

Le typage fort de Python est important, il est un fondement de sa philosophie. J'ai tendance à considérer qu'un langage faiblement typé crée des erreurs qui sont plus difficiles à repérer alors qu'ici, il nous suffit de convertir explicitement le type pour que Python sache ce qu'il doit faire.

Parcours et sélection de chaînes

Nous avons vu très rapidement dans la première partie un moyen de parcourir des chaînes. Nous allons en voir ici un second qui fonctionne par indice.

Parcours par indice

Vous devez vous en souvenir : j'ai dit qu'une chaîne de caractères était une séquence constituée… de caractères. En fait, une chaîne de caractères est elle-même constituée de chaînes de caractères, chacune d'elles n'étant composée que d'un seul caractère.

Accéder aux caractères d'une chaîne

Nous allons apprendre à accéder aux lettres constituant une chaîne. Par exemple, nous souhaitons sélectionner la première lettre d'une chaîne.

```
>>> chaine = "Salut les ZEROS !"
>>> chaine[0] # Première lettre de la chaîne
'S'
>>> chaine[2] # Troisième lettre de la chaîne
'l'
>>> chaine[-1] # Dernière lettre de la chaîne
'!'
>>>
```

On précise entre crochets □ l'indice (la position du caractère auquel on souhaite accéder).

Rappelez-vous, on commence à compter à partir de 0. La première lettre est donc à l'indice 0, la deuxième à l'indice 1, la troisième à l'indice 2… On peut accéder aux lettres en partant de la fin à l'aide d'un indice négatif. Quand vous tapez `chaine[-1]`, vous accédez ainsi à la dernière lettre de la chaîne (enfin, au dernier caractère, qui n'est pas une lettre ici).

On peut obtenir la longueur de la chaîne (le nombre de caractères qu'elle contient) grâce à la fonction `len`.

```
>>> chaine = "Salut"
>>> len(chaine)
5
>>>
```

Pourquoi ne pas avoir défini cette fonction comme une méthode de la classe `str` ? Pourquoi ne pourrait-on pas faire `chaine.len()` ?

En fait c'est un peu le cas, mais nous le verrons bien plus loin. `str` n'est qu'un exemple parmi d'autres de séquences (on en découvrira d'autres dans les prochains chapitres) et donc les développeurs de Python ont préféré créer une fonction qui travaillerait sur les séquences au sens large, plutôt qu'une méthode pour chacune de ces classes.

Méthode de parcours par `while`

Vous en savez assez pour parcourir une chaîne grâce à la boucle `while`. Notez que, dans la plupart des cas, on préférera parcourir une séquence avec `for`. Il est néanmoins bon de savoir procéder de différentes manières, cela vous sera utile parfois.

Voici le code auquel vous pourriez arriver :

```
chaine = "Salut"
```

```
i = 0 # On appelle l'indice 'i' par convention
while i < len(chaine):
    print(chaine[i]) # On affiche le caractère à chaque tour de
        boucle
    i += 1
```

N'oubliez pas d'incrémenter `i`, sinon vous allez avoir quelques surprises.

Si vous essayez d'accéder à un indice qui n'existe pas (par exemple 25 alors que votre chaîne ne fait que 20 caractères de longueur), Python lèvera une exception de type `IndexError`.

Enfin, une dernière petite chose : vous ne pouvez changer les lettres de la chaîne en utilisant les indices.

```
>>> mot = "lac"
>>> mot[0] = "b" # On veut remplacer 'l' par 'b'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>>
```

Python n'est pas content. Il ne veut pas que vous utilisiez les indices pour modifier des caractères de la chaîne. Pour ce faire, il va falloir utiliser la sélection.

Sélection de chaînes

Nous allons voir comment sélectionner une partie de la chaîne. Si je souhaite, par exemple, sélectionner les deux premières lettres de la chaîne, je procéderai comme dans l'exemple ci-dessous.

```
>>> presentation = "salut"
>>> presentation[0:2] # On sélectionne les deux premières
    lettres
'sa'
>>> presentation[2:len(presentation)] # On sélectionne la chaî
    ne sauf les deux premières lettres
'lut'
>>>
```

La sélection consiste donc à extraire une partie de la chaîne. Cette opération renvoie le morceau de la chaîne sélectionné, sans modifier la chaîne d'origine.

Sachez que l'on peut sélectionner du début de la chaîne jusqu'à un indice, ou d'un indice jusqu'à la fin de la chaîne, sans préciser autant d'informations que dans nos exemples. Python comprend très bien si on sous-entend certaines informations.

```
>>> presentation[:2] # Du début jusqu'à la troisième lettre non
    comprise
'sa'
```



```
>>> presentation[2:] # De la troisième lettre (comprise) à la
    fin
'lut'
>>>
```

Maintenant, nous pouvons reprendre notre exemple de tout à l'heure pour constituer une nouvelle chaîne, en remplaçant une lettre par une autre :

```
>>> mot = "lac"
>>> mot = "b" + mot[1:]
>>> print(mot)
bac
>>>
```

Voilà !

Cela reste assez peu intuitif, non ?

Pour remplacer des lettres, cela paraît un peu lourd en effet. Et d'ailleurs on s'en sert assez rarement pour cela. Pour rechercher/remplacer, nous avons à notre disposition les méthodes `count`, `find` et `replace`, à savoir « compter », « rechercher » et « remplacer ».

En résumé

- Les variables utilisées jusqu'ici sont en réalité des objets.
- Les types de données utilisés jusqu'ici sont en fait des classes. Chaque objet est modelé sur une classe.
- Chaque classe définit certaines fonctions, appelées méthodes, qui seront accessibles depuis l'objet grâce à `objet.methode(arguments)`.
- On peut directement accéder à un caractère d'une chaîne grâce au code suivant : `chaine[position_dans_la_chaine]`.
- Il est tout à fait possible de sélectionner une partie de la chaîne grâce au code suivant : `chaine[indice_debut:indice_fin]`.

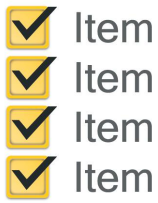
Chapitre 11

Les listes et tuples (1/2)

Difficulté :

J'aurai réussi à vous faire connaître et, j'espère, aimer le langage Python sans vous apprendre les listes. Mais allons ! Cette époque est révolue. Maintenant que nous commençons à étudier l'objet sous toutes ses formes, je ne vais pas pouvoir garder le secret plus longtemps : il existe des listes en Python. Pour ceux qui ne voient même pas de quoi je parle, vous allez vite vous rendre compte qu'avec les dictionnaires (que nous verrons plus loin), c'est un type, ou plutôt une classe, dont on aura du mal à se passer.

Les listes sont des séquences. En fait, leur nom est plutôt explicite, puisque ce sont des objets capables de contenir d'autres objets de n'importe quel type. On peut avoir une liste contenant plusieurs nombres entiers (1, 2, 50, 2000 ou plus, peu importe), une liste contenant des flottants, une liste contenant des chaînes de caractères... et une liste mélangeant ces objets de différents types.



Créons et éditons nos premières listes

D'abord c'est quoi, une liste ?

En Python, les listes sont des objets qui peuvent en contenir d'autres. Ce sont donc des séquences, comme les chaînes de caractères, mais au lieu de contenir des caractères, elles peuvent contenir n'importe quel objet. Comme d'habitude, on va s'occuper du concept des listes avant de voir tout son intérêt.

Création de listes

On a deux moyens de créer des listes. Si je vous dis que la classe d'une liste s'appelle, assez logiquement, `list`, vous devriez déjà voir une manière de créer une liste.

Non ? ...

Vous allez vous habituer à cette syntaxe :

```
>>> ma_liste = list() # On crée une liste vide
>>> type(ma_liste)
<class 'list'>
>>> ma_liste
[]
>>>
```

Là encore, on utilise le nom de la classe comme une fonction pour **instancier** un objet de cette classe.

Quand vous affichez la liste, vous pouvez constater qu'elle est vide. Entre les crochets (qui sont les délimiteurs des listes en Python), il n'y a rien. On peut également utiliser ces crochets pour créer une liste.

```
>>> ma_liste = [] # On crée une liste vide
>>>
```

Cela revient au même, vous pouvez vérifier. Toutefois, on peut également créer une liste non vide, en lui indiquant directement à la création les objets qu'elle doit contenir.

```
>>> ma_liste = [1, 2, 3, 4, 5] # Une liste avec cinq objets
>>> print(ma_liste)
[1, 2, 3, 4, 5]
>>>
```

La liste que nous venons de créer compte cinq objets de type `int`. Ils sont classés par ordre croissant. Mais rien de tout cela n'est obligatoire.

- Vous pouvez faire des listes de toute longueur.
- Les listes peuvent contenir n'importe quel type d'objet.

– Les objets dans une liste peuvent être mis dans un ordre quelconque. Toutefois, la structure d'une liste fait que chaque objet *a sa place* et que l'ordre compte.

```
>>> ma_liste = [1, 3.5, "une chaine", []]
>>>
```

Nous avons créé ici une liste contenant quatre objets de types différents : un entier, un flottant, une chaîne de caractères et... une autre liste.

Voyons à présent comment accéder aux éléments d'une liste :

```
>>> ma_liste = ['c', 'f', 'm']
>>> ma_liste[0] # On accède au premier élément de la liste
'c'
>>> ma_liste[2] # Troisième élément
'm'
>>> ma_liste[1] = 'Z' # On remplace 'f' par 'Z'
>>> ma_liste
['c', 'Z', 'm']
>>>
```

Comme vous pouvez le voir, on accède aux éléments d'une liste de la même façon qu'on accède aux caractères d'une chaîne de caractères : on indique entre crochets l'indice de l'élément qui nous intéresse.

Contrairement à la classe `str`, la classe `list` vous permet de remplacer un élément par un autre. Les listes sont en effet des types dits **mutables.**

Insérer des objets dans une liste

On dispose de plusieurs méthodes, définies dans la classe `list`, pour ajouter des éléments dans une liste.

Ajouter un élément à la fin de la liste

On utilise la méthode `append` pour ajouter un élément à la fin d'une liste.

```
>>> ma_liste = [1, 2, 3]
>>> ma_liste.append(56) # On ajoute 56 à la fin de la liste
>>> ma_liste
[1, 2, 3, 56]
>>>
```

C'est assez simple non ? On passe en paramètre de la méthode `append` l'objet que l'on souhaite ajouter à la fin de la liste.

La méthode append, comme beaucoup de méthodes de listes, travaille directement sur l'objet et ne renvoie donc rien !

Ceci est extrêmement important. Dans le chapitre précédent, nous avons vu qu'aucune des méthodes de chaînes ne modifie l'objet d'origine mais qu'elles renvoient toutes un nouvel objet, qui est la chaîne modifiée. Ici c'est le contraire : les méthodes de listes ne renvoient rien mais modifient l'objet d'origine. Regardez ce code si ce n'est pas bien clair :

```
>>> chaine1 = "une petite phrase"
>>> chaine2 = chaine1.upper() # On met en majuscules chaine1
>>> chaine1                   # On affiche la chaîne d'origine
'une petite phrase'
>>> # Elle n'a pas été modifiée par la méthode upper
... chaine2                   # On affiche chaine2
'UNE PETITE PHRASE'
>>> # C'est chaine2 qui contient la chaîne en majuscules
... # Voyons pour les listes à présent
... liste1 = [1, 5.5, 18]
>>> liste2 = liste1.append(-15) # On ajoute -15 à liste1
>>> liste1                      # On affiche liste1
[1, 5.5, 18, -15]
>>> # Cette fois, l'appel de la méthode a modifié l'objet d'
    origine (liste1)
... # Voyons ce que contient liste2
... liste2
>>> # Rien ? Vérifions avec print
... print(liste2)
None
>>>
```

Je vais expliquer les dernières lignes. Mais d'abord, il faut que vous fassiez bien la différence entre les méthodes de chaînes, où l'objet d'origine n'est jamais modifié et qui renvoient un nouvel objet, et les méthodes de listes, qui ne renvoient rien mais modifient l'objet d'origine.

J'ai dit que les méthodes de listes ne renvoient rien. On va pourtant essayer de « capturer » la valeur de retour dans `liste2`. Quand on essaye d'afficher la valeur de `liste2` par saisie directe, on n'obtient rien. Il faut l'afficher avec `print` pour savoir ce qu'elle contient : `None`. C'est l'objet vide de Python. En réalité, quand une fonction ne renvoie rien, elle renvoie `None`. Vous retrouverez peut-être cette valeur de temps à autre, ne soyez donc pas surpris.

Insérer un élément dans la liste

Nous allons passer assez rapidement sur cette seconde méthode. On peut, très simplement, insérer un objet dans une liste, à l'endroit voulu. On utilise pour cela la méthode `insert`.

```
>>> ma_liste = ['a', 'b', 'd', 'e']
>>> ma_liste.insert(2, 'c') # On insère 'c' à l'indice 2
>>> print(ma_liste)
```

```
['a', 'b', 'c', 'd', 'e']
```

Quand on demande d'insérer `c` à l'indice 2, la méthode va décaler les objets d'indice supérieur ou égal à 2. `c` va donc s'intercaler entre `b` et `d`.

Concaténation de listes

On peut également agrandir des listes en les concaténant avec d'autres.

```
>>> ma_liste1 = [3, 4, 5]
>>> ma_liste2 = [8, 9, 10]
>>> ma_liste1.extend(ma_liste2) # On insère ma_liste2 à la fin
    de ma_liste1
>>> print(ma_liste1)
[3, 4, 5, 8, 9, 10]
>>> ma_liste1 = [3, 4, 5]
>>> ma_liste1 + ma_liste2
[3, 4, 5, 8, 9, 10]
>>> ma_liste1 += ma_liste2 # Identique à extend
>>> print(ma_liste1)
[3, 4, 5, 8, 9, 10]
>>>
```

Voici les différentes façons de concaténer des listes. Vous pouvez remarquer l'opérateur + qui concatène deux listes entre elles et renvoie le résultat. On peut utiliser += assez logiquement pour étendre une liste. Cette façon de faire revient au même qu'utiliser la méthode `extend`.

Suppression d'éléments d'une liste

Nous allons voir rapidement comment supprimer des éléments d'une liste, avant d'apprendre à les parcourir. Vous allez vite pouvoir constater que cela se fait assez simplement. Nous allons voir deux méthodes pour supprimer des éléments d'une liste :

- le mot-clé `del` ;
- la méthode `remove`.

Le mot-clé `del`

C'est un des mots-clés de Python, que j'aurais pu vous montrer plus tôt. Mais les applications de `del` me semblaient assez peu pratiques avant d'aborder les listes.

`del` (abréviation de *delete*) signifie « supprimer » en anglais. Son utilisation est des plus simple : `del variable_a_supprimer`. Voyons un exemple.

```
>>> variable = 34
>>> variable
```



```
34
>>> del variable
>>> variable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'variable' is not defined
>>>
```

Comme vous le voyez, après l'utilisation de `del`, la variable n'existe plus. Python l'efface tout simplement. Mais on peut également utiliser `del` pour supprimer des éléments d'une séquence, comme une liste, et c'est ce qui nous intéresse ici.

```
>>> ma_liste = [-5, -2, 1, 4, 7, 10]
>>> del ma_liste[0] # On supprime le premier élément de la
    liste
>>> ma_liste
[-2, 1, 4, 7, 10]
>>> del ma_liste[2] # On supprime le troisième élément de la
    liste
>>> ma_liste
[-2, 1, 7, 10]
>>>
```

La méthode `remove`

On peut aussi supprimer des éléments de la liste grâce à la méthode `remove` qui prend en paramètre non pas l'indice de l'élément à supprimer, mais l'élément lui-même.

```
>>> ma_liste = [31, 32, 33, 34, 35]
>>> ma_liste.remove(32)
>>> ma_liste
[31, 33, 34, 35]
>>>
```

La méthode `remove` parcourt la liste et en retire l'élément que vous lui passez en paramètre. C'est une façon de faire un peu différente et vous appliquerez `del` ou `remove` en fonction de la situation.

La méthode `remove` ne retire que la première occurrence de la valeur trouvée dans la liste !

Notez au passage que le mot-clé `del` n'est pas une méthode de liste. Il s'agit d'une fonctionnalité de Python qu'on retrouve dans la plupart des objets conteneurs, tels que les listes que nous venons de voir, ou les dictionnaires que nous verrons plus tard. D'ailleurs, `del` sert plus généralement à supprimer non seulement des éléments d'une séquence mais aussi, comme nous l'avons vu, des variables.

Nous allons à présent voir comment parcourir une liste, même si vous devez déjà avoir votre petite idée sur la question.

Le parcours de listes

Vous avez déjà dû vous faire une idée des méthodes pour parcourir une liste. Je vais passer brièvement dessus, vous ne verrez rien de nouveau ni, je l'espère, de très surprenant.

```
>>> ma_liste = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
>>> i = 0 # Notre indice pour la boucle while
>>> while i < len(ma_liste):
...     print(ma_liste[i])
...     i += 1 # On incrémente i, ne pas oublier !
...
a
b
c
d
e
f
g
h
>>> # Cette méthode est cependant préférable
... for elt in ma_liste: # elt va prendre les valeurs successives des éléments de ma_liste
...     print(elt)
...
a
b
c
d
e
f
g
h
>>>
```

Il s'agit des mêmes méthodes de parcours que nous avons vues pour les chaînes de caractères, au chapitre précédent. Nous allons cependant aller un peu plus loin.

La fonction `enumerate`

Les deux méthodes que nous venons de voir possèdent toutes deux des inconvénients :

- la méthode utilisant `while` est plus longue à écrire, moins intuitive et elle est perméable aux boucles infinies, si l'on oublie d'incrémenter la variable servant de compteur ;
- la méthode par `for` se contente de parcourir la liste en capturant les éléments dans une variable, sans qu'on puisse savoir où ils sont dans la liste.

C'est vrai dans le cas que nous venons de voir. Certains codeurs vont combiner les deux méthodes pour plus de flexibilité mais, très souvent, le code obtenu est moins lisible. Heureusement, les développeurs de Python ont pensé à nous.

```
>>> ma_liste = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
>>> for i, elt in enumerate(ma_liste):
...     print("À l'indice {} se trouve {}.".format(i, elt))
...
À l'indice 0 se trouve a.
À l'indice 1 se trouve b.
À l'indice 2 se trouve c.
À l'indice 3 se trouve d.
À l'indice 4 se trouve e.
À l'indice 5 se trouve f.
À l'indice 6 se trouve g.
À l'indice 7 se trouve h.
>>>
```

Pas de panique !

Nous avons ici une boucle `for` un peu surprenante. Entre `for` et `in`, nous avons deux variables, séparées par une virgule.

En fait, `enumerate` prend en paramètre une liste et renvoie un objet qui peut être associé à une liste contenant deux valeurs par élément : l'indice et l'élément de la liste parcouru.

Ce n'est sans doute pas encore très clair. Essayons d'afficher cela un peu mieux :

```
>>> for elt in enumerate(ma_liste):
...     print(elt)
...
(0, 'a')
(1, 'b')
(2, 'c')
(3, 'd')
(4, 'e')
(5, 'f')
(6, 'g')
(7, 'h')
>>>
```

Quand on parcourt chaque élément de l'objet renvoyé par `enumerate`, on voit des **tuples** qui contiennent deux éléments : d'abord l'indice, puis l'objet se trouvant à cet indice dans la liste passée en argument à la fonction `enumerate`.

Les tuples sont des séquences, assez semblables aux listes, sauf qu'on ne peut modifier un tuple après qu'il ait été créé. Cela signifie qu'on définit le contenu d'un tuple (les objets qu'il doit contenir) lors de sa création, mais qu'on ne peut en ajouter ou en retirer par la suite.

Si les parenthèses vous déconcertent trop, vous pouvez imaginer, à la place, des crochets : dans cet exemple, cela revient au même.

Quand on utilise `enumerate`, on capture l'indice et l'élément dans deux variables distinctes. Voyons un autre exemple pour comprendre ce mécanisme :

```
>>> autre_liste = [
...     [1, 'a'],
...     [4, 'd'],
...     [7, 'g'],
...     [26, 'z'],
... ] # J'ai étalé la liste sur plusieurs lignes
>>> for nb, lettre in autre_liste:
...     print("La lettre {} est la {}e de l'alphabet.".format(
    lettre, nb))
...
La lettre a est la 1e de l'alphabet.
La lettre d est la 4e de l'alphabet.
La lettre g est la 7e de l'alphabet.
La lettre z est la 26e de l'alphabet.
>>>
```

J'espère que c'est assez clair dans votre esprit. Dans le cas contraire, décomposez ces exemples, le déclic devrait se faire.

On écrit ici la définition de la liste sur plusieurs lignes pour des raisons de lisibilité. On n'est pas obligé de mettre des anti-slashs « \ » en fin de ligne car, tant que Python ne trouve pas de crochet fermant la liste, il continue d'attendre sans interpréter la ligne. Vous pouvez d'ailleurs le constater avec les points qui remplacent les chevrons au début de la ligne, tant que la liste n'a pas été refermée.

Quand on travaille sur une liste que l'on parcourt en même temps, on peut se retrouver face à des erreurs assez étranges, qui paraissent souvent incompréhensibles au début.

Par exemple, on peut être confronté à des exceptions `IndexError` si on tente de supprimer certains éléments d'une liste en la parcourant.

Nous verrons au prochain chapitre comment faire cela proprement, pour l'heure il vous suffit de vous méfier d'un parcours qui modifie une liste, surtout sa structure. D'une façon générale, évitez de parcourir une liste dont la taille évolue en même temps.

Allez ! On va jeter un coup d'œil aux tuples, pour conclure ce chapitre !

Un petit coup d'œil aux tuples

Nous avons brièvement vu les **tuples** un peu plus haut, grâce à la fonction `enumerate`. Les tuples sont des listes immuables, qu'on ne peut modifier. En fait, vous allez vous rendre compte que nous utilisons depuis longtemps des tuples sans nous en rendre compte.

Un tuple se définit comme une liste, sauf qu'on utilise comme délimiteur des parenthèses au lieu des crochets.

```
tuple_vide = ()
tuple_non_vide = (1,)
tuple_non_vide = (1, 3, 5)
```

À la différence des listes, les tuples, une fois créés, ne peuvent être modifiés : on ne peut plus y ajouter d'objet ou en retirer.

Une petite subtilité ici : si on veut créer un tuple contenant un unique élément, on doit quand même mettre une virgule après celui-ci. Sinon, Python va automatiquement supprimer les parenthèses et on se retrouvera avec une variable lambda et non un tuple contenant cette variable.

Mais à quoi cela sert-il ?

Il est assez rare que l'on travaille directement sur des tuples. C'est, après tout, un type que l'on ne peut pas modifier. On ne peut supprimer d'éléments d'un tuple, ni en ajouter. Cela vous paraît peut-être encore assez abstrait mais il peut être utile de travailler sur des données sans pouvoir les modifier.

En attendant, voyons plutôt les cas où nous avons utilisé des tuples sans le savoir.

Affectation multiple

Tous les cas que nous allons voir sont des cas d'affectation multiple. Vous vous souvenez ?

```
>>> a, b = 3, 4
>>> a
3
>>> b
4
>>>
```

On a également utilisé cette syntaxe pour permuter deux variables. Eh bien, cette syntaxe passe par des tuples qui ne sont pas déclarés explicitement. Vous pourriez écrire :

```
>>> (a, b) = (3, 4)
>>>
```

Quand Python trouve plusieurs variables ou valeurs séparées par des virgules et sans délimiteur, il va les mettre dans des tuples. Dans le premier exemple, les parenthèses sont sous-entendues et Python comprend ce qu'il doit faire.

Une fonction renvoyant plusieurs valeurs

Nous ne l'avons pas vu jusqu'ici mais une fonction peut renvoyer deux valeurs ou même plus :

```
def decomposer(entier, divise_par):
    """Cette fonction retourne la partie entière et le reste de
    entier / divise_par"""

    p_e = entier // divise_par
    reste = entier % divise_par
    return p_e, reste
```

Et on peut ensuite capturer la partie entière et le reste dans deux variables, au retour de la fonction :

```
>>> partie_entiere, reste = decomposer(20, 3)
>>> partie_entiere
6
>>> reste
2
>>>
```

Là encore, on passe par des tuples sans que ce soit indiqué explicitement à Python. Si vous essayez de faire `retour = decomposer(20, 3)`, vous allez capturer un tuple contenant deux éléments : la partie entière et le reste de 20 divisé par 3.

Nous verrons plus loin d'autres exemples de tuples et d'autres utilisations. Je pense que cela suffit pour cette fois.

En résumé

- Une liste est une séquence mutable pouvant contenir plusieurs autres objets.
- Une liste se construit ainsi : `liste = [element1, element2, elementN]`.
- On peut insérer des éléments dans une liste à l'aide des méthodes `append`, `insert` et `extends`.

- On peut supprimer des éléments d'une liste grâce au mot-clé `del` ou à la méthode `remove`.
- Un tuple est une séquence pouvant contenir des objets. À la différence de la liste, le tuple ne peut être modifié une fois créé.

Chapitre 12

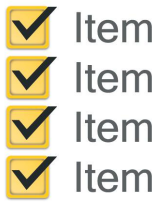
Les listes et tuples (2/2)

Difficulté :

Les listes sont très utilisées en Python. Elles sont liées à pas mal de fonctionnalités, dont certaines plutôt complexes. Aussi ai-je préféré scinder l'approche des listes en deux chapitres. Vous allez voir dans celui-ci quelques fonctionnalités qui ne s'appliquent qu'aux listes et aux tuples, et qui pourront vous être extrêmement utiles. Je vous conseille donc, avant tout, d'être bien à l'aise avec les listes et leur création, parcours, édition, suppression...

D'autre part, comme pour la plupart des sujets abordés, je ne peux faire un tour d'horizon exhaustif de toutes les fonctionnalités de chaque objet présenté. Je vous invite donc à lire la documentation, en tapant `help(list)`, pour accéder à une liste exhaustive des méthodes.

C'est parti !



Entre chaînes et listes

Nous allons voir un moyen de transformer des chaînes en listes et réciproquement.

Il est assez surprenant, de prime abord, qu'une conversion soit possible entre ces deux types qui sont tout de même assez différents. Mais comme on va le voir, il ne s'agit pas réellement d'une conversion. Il va être difficile de démontrer l'utilité de cette fonctionnalité tout de suite, mieux valent quelques exemples.

Des chaînes aux listes

Pour « convertir » une chaîne en liste, on va utiliser une méthode de chaîne nommée `split` (« éclater » en anglais). Cette méthode prend un paramètre qui est une autre chaîne, souvent d'un seul caractère, définissant comment on va découper notre chaîne initiale.

C'est un peu compliqué et cela paraît très tordu... mais regardez plutôt :

```
>>> ma_chaine = "Bonjour à tous"
>>> ma_chaine.split(" ")
['Bonjour', 'à', 'tous']
>>>
```

On passe en paramètre de la méthode `split` une chaîne contenant un unique espace. La méthode renvoie une liste contenant les trois mots de notre petite phrase. Chaque mot se trouve dans une case de la liste.

C'est assez simple en fait : quand on appelle la méthode `split`, celle-ci découpe la chaîne en fonction du paramètre donné. Ici la première case de la liste va donc du début de la chaîne au premier espace (non inclus), la deuxième case va du premier espace au second, et ainsi de suite jusqu'à la fin de la chaîne.

Sachez que `split` possède un paramètre par défaut, un code qui représente les espaces, les tabulations et les sauts de ligne. Donc vous pouvez très bien faire `ma_chaine.split()`, cela revient ici au même.

Des listes aux chaînes

Voyons l'inverse à présent, c'est-à-dire si on a une liste contenant plusieurs chaînes de caractères que l'on souhaite fusionner en une seule. On utilise la méthode de chaîne `join` (« joindre » en anglais). Sa syntaxe est un peu surprenante :

```
>>> ma_liste = ['Bonjour', 'à', 'tous']
>>> " ".join(ma_liste)
'Bonjour à tous'
>>>
```

En paramètre de la méthode `join`, on passe la liste des chaînes que l'on souhaite « ressouder ». La méthode va travailler sur l'objet qui l'appelle, ici une chaîne de caractères contenant un unique espace. Elle va insérer cette chaîne entre chaque paire de chaînes de la liste, ce qui au final nous donne la chaîne de départ, « Bonjour à tous ».

N'aurait-il pas été plus simple ou plus logique de faire une méthode de liste, prenant en paramètre la chaîne faisant la jonction ?

Ce choix est en effet contesté mais, pour ma part, je ne trancherai pas. Le fait est que c'est cette méthode qui a été choisie et, avec un peu d'habitude, on arrive à bien lire le résultat obtenu. D'ailleurs, nous allons voir comment appliquer concrètement ces deux méthodes.

Une application pratique

Admettons que nous ayons un nombre flottant dont nous souhaitons afficher la partie entière et les trois premières décimales uniquement de la partie flottante. Autrement dit, si on a un nombre flottant tel que « 3.999999999999998 », on souhaite obtenir comme résultat « 3.999 ». D'ailleurs, ce serait plus joli si on remplaçait le point décimal par la virgule, à laquelle nous sommes plus habitués.

Là encore, je vous invite à essayer de faire ce petit exercice par vous-même. On part du principe que la valeur de retour de la fonction chargée de la pseudo-conversion est une chaîne de caractères. Voici quelques exemples d'utilisation de la fonction que vous devriez coder :

```
>>> afficher_flottant(3.99999999999998)
'3,999'
>>> afficher_flottant(1.5)
'1,5'
>>>
```

Voici la correction que je vous propose :

```
def afficher_flottant(flottant):
    """Fonction prenant en paramètre un flottant et renvoyant
       une chaîne de caractères représentant la troncature de
       ce nombre. La partie flottante doit avoir une longueur
       maximum de 3 caractères.

    De plus, on va remplacer le point décimal par la virgule"""

    if type(flottant) is not float:
        raise TypeError("Le paramètre attendu doit être un
            flottant")
    flottant = str(flottant)
    partie_entiere, partie_flottante = flottant.split(".")
```



```
    # La partie entière n'est pas à modifier
    # Seule la partie flottante doit être tronquée
    return ",".join([partie_entiere, partie_flottante[:3]])
```

En s'assurant que le type passé en paramètre est bien un flottant, on garantit qu'il n'y aura pas d'erreur lors du fractionnement de la chaîne. On est sûr qu'il y aura forcément une partie entière et une partie flottante séparées par un point, même si la partie flottante n'est constituée que d'un 0. Si vous n'y êtes pas arrivés par vous-même, étudiez bien cette solution, elle n'est pas forcément évidente au premier coup d'œil. On fait intervenir un certain nombre de mécanismes que vous avez vus il y a peu, tâchez de bien les comprendre.

Les listes et paramètres de fonctions

Nous allons droit vers une fonctionnalité des plus intéressantes, qui fait une partie de la puissance de Python. Nous allons étudier un cas assez particulier avant de généraliser : les fonctions dont le nombre de paramètres est inconnu.

Notez malgré tout que ce point est assez délicat. Si vous n'arrivez pas bien à le comprendre, laissez cette section de côté, cela ne vous pénalisera pas.

Les fonctions dont on ne connaît pas à l'avance le nombre de paramètres

Vous devriez tout de suite penser à la fonction `print` : on lui passe une liste de paramètres qu'elle va afficher, dans l'ordre où ils sont placés, séparés par un espace (ou tout autre délimiteur choisi).

Vous n'allez peut-être pas trouver d'applications de cette fonctionnalité dans l'immédiat mais, tôt ou tard, cela arrivera. La syntaxe est tellement simple que c'en est déconcertant :

```
def fonction(*parametres):
```

On place une étoile * devant le nom du paramètre qui accueillera la liste des arguments. Voyons plus précisément comment cela se présente :

```
>>> def fonction_inconnue(*parametres):
...     """Test d'une fonction pouvant être appelée avec un
    nombre variable de paramètres"""
...
...     print("J'ai reçu : {}.".format(parametres))
...
>>> fonction_inconnue() # On appelle la fonction sans paramètre
J'ai reçu : ().
>>> fonction_inconnue(33)
J'ai reçu : (33,).
>>> fonction_inconnue('a', 'e', 'f')
```

```
J'ai reçu : ('a', 'e', 'f').
>>> var = 3.5
>>> fonction_inconnue(var, [4], "...")
J'ai reçu : (3.5, [4], '...').
>>>
```

Je pense que cela suffit. Comme vous le voyez, on peut appeler la `fonction_inconnue` avec un nombre indéterminé de paramètres, allant de 0 à l'infini (enfin, théoriquement). Le fait de préciser une étoile `*` devant le nom du paramètre fait que Python va placer tous les paramètres de la fonction dans un **tuple**, que l'on peut ensuite traiter comme on le souhaite.

Et les paramètres nommés dans l'histoire ? Comment sont-ils insérés dans le tuple ?

Ils ne le sont pas. Si vous tapez `fonction_inconnue(couleur="rouge")`, vous allez avoir une erreur : `fonction_inconnue() got an unexpected keyword argument 'couleur'`. Nous verrons au prochain chapitre comment capturer ces paramètres nommés.

Vous pouvez bien entendu définir une fonction avec plusieurs paramètres qui doivent être fournis quoi qu'il arrive, suivis d'une liste de paramètres variables :

```
def fonction_inconnue(nom, prenom, *commentaires):
```

Dans cet exemple de définition de fonction, vous devez impérativement préciser un nom et un prénom, et ensuite vous mettez ce que vous voulez en commentaire, aucun paramètre, un, deux... ce que vous voulez.

Si on définit une liste variable de paramètres, elle doit se trouver après la liste des paramètres standard.

Au fond, cela est évident. Vous ne pouvez avoir une définition de fonction comme `def fonction_inconnue(*parametres, nom, prenom)`. En revanche, si vous souhaitez avoir des paramètres nommés, il faut les mettre après cette liste. Les paramètres nommés sont un peu une exception puisqu'ils ne figureront de toute façon pas dans le tuple obtenu. Voyons par exemple la définition de la fonction `print` :

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

Ne nous occupons pas du dernier paramètre. Il définit le descripteur vers lequel `print` envoie ses données ; par défaut, c'est l'écran.

D'où viennent ces points de suspension dans les paramètres ?

En fait, il s'agit d'un affichage un peu plus agréable. Si on veut réellement avoir la définition en code Python, on retombera plutôt sur :

```
def print(*values, sep=' ', end='\n', file=sys.stdout):
```

Petit exercice : faire une fonction `afficher` identique à `print`, c'est-à-dire prenant un nombre indéterminé de paramètres, les affichant en les séparant à l'aide du paramètre nommé `sep` et terminant l'affichage par la variable `fin`. Notre fonction `afficher` ne comptera pas de paramètre `file`. En outre, elle devra passer par `print` pour afficher (on ne connaît pas encore d'autres façons de faire). La seule contrainte est que l'appel à `print` ne doit compter qu'un seul paramètre non nommé. Autrement dit, avant l'appel à `print`, la chaîne devra avoir été déjà formatée, prête à l'affichage.

Pour que ce soit plus clair, je vous mets la définition de la fonction, ainsi que la `docstring` que j'ai écrite :

```
def afficher(*parametres, sep=' ', fin='\n'):
    """Fonction chargée de reproduire le comportement de print.

    Elle doit finir par faire appel à print pour afficher le ré
        sultat.
    Mais les paramètres devront déjà avoir été formatés.
    On doit passer à print une unique chaîne, en lui spécifiant
         de ne rien mettre à la fin :

    print(chaine, end='')"""

    # Les paramètres sont sous la forme d'un tuple
    # Or on a besoin de les convertir
    # Mais on ne peut pas modifier un tuple
    # On a plusieurs possibilités, ici je choisis de convertir
        le tuple en liste
    parametres = list(parametres)
    # On va commencer par convertir toutes les valeurs en chaî
        ne
    # Sinon on va avoir quelques problèmes lors du join
    for i, parametre in enumerate(parametres):
        parametres[i] = str(parametre)
    # La liste des paramètres ne contient plus que des chaînes
        de caractères
    # À présent on va constituer la chaîne finale
    chaine = sep.join(parametres)
    # On ajoute le paramètre fin à la fin de la chaîne
    chaine += fin
    # On affiche l'ensemble
    print(chaine, end='')
```

J'espère que ce n'était pas trop difficile et que, si vous avez fait des erreurs, vous avez pu les comprendre.

Ce n'est pas du tout grave si vous avez réussi à coder cette fonction d'une manière différente. *En programmation, il n'y a pas qu'une solution, il y a des solutions.*

Transformer une liste en paramètres de fonction

C'est peut-être un peu moins fréquent mais vous devez connaître ce mécanisme puisqu'il complète parfaitement le premier. Si vous avez un tuple ou une liste contenant des paramètres qui doivent être passés à une fonction, vous pouvez très simplement les transformer en paramètres lors de l'appel. Le seul problème c'est que, côté démonstration, je me vois un peu limité.

```
>>> liste_des_parametres = [1, 4, 9, 16, 25, 36]
>>> print(*liste_des_parametres)
1 4 9 16 25 36
>>>
```

Ce n'est pas bien spectaculaire et pourtant c'est une fonctionnalité très puissante du langage. Là, on a une liste contenant des paramètres et on la transforme en une liste de paramètres de la fonction `print`. Donc, au lieu d'afficher la liste proprement dite, on affiche tous les nombres, séparés par des espaces. C'est exactement comme si vous aviez fait `print(1, 4, 9, 16, 25, 36)`.

Mais quel intérêt ? Cela ne change pas grand-chose et il est rare que l'on capture les paramètres d'une fonction dans une liste, non ?

Oui je vous l'accorde. Ici l'intérêt ne saute pas aux yeux. Mais un peu plus tard, vous pourrez tomber sur des applications où les fonctions sont utilisées sans savoir quels paramètres elles attendent réellement. Si on ne connaît pas la fonction que l'on appelle, c'est très pratique. Là encore, vous découvrirez cela dans les chapitres suivants ou dans certains projets. Essayez de garder à l'esprit ce mécanisme de transformation.

On utilise une étoile * dans les deux cas. Si c'est dans une définition de fonction, cela signifie que les paramètres fournis non attendus lors de l'appel seront capturés dans la variable, sous la forme d'un tuple. Si c'est dans un appel de fonction, au contraire, cela signifie que la variable sera décomposée en plusieurs paramètres envoyés à la fonction.

J'espère que vous êtes encore en forme, on attaque le point que je considère comme le plus dur de ce chapitre, mais aussi le plus intéressant. Gardez les yeux ouverts !

Les compréhensions de liste

Les compréhensions de liste (« *list comprehensions* » en anglais) sont un moyen de filtrer ou modifier une liste très simplement. La syntaxe est déconcertante au début mais vous allez voir que c'est très puissant.

Parcours simple

Les **compréhensions de liste** permettent de parcourir une liste en en renvoyant une seconde, modifiée ou filtrée. Pour l'instant, nous allons voir une simple modification.

```
>>> liste_origine = [0, 1, 2, 3, 4, 5]
>>> [nb * nb for nb in liste_origine]
[0, 1, 4, 9, 16, 25]
>>>
```

Étudions un peu la ligne 2 de ce code. Comme vous avez pu le deviner, elle signifie en langage plus conventionnel « Mettre au carré tous les nombres contenus dans la liste d'origine ». Nous trouvons dans l'ordre, entre les crochets qui sont les délimiteurs d'une instruction de compréhension de liste :

- `nb * nb` : la valeur de retour. Pour l'instant, on ne sait pas ce qu'est la variable `nb`, on sait juste qu'il faut la mettre au carré. Notez qu'on aurait pu écrire `nb**2`, cela revient au même.
- `for nb in liste_origine` : voilà d'où vient notre variable `nb`. On reconnaît la syntaxe d'une boucle `for`, sauf qu'on n'est pas habitué à la voir sous cette forme.

Quand Python interprète cette ligne, il va parcourir la liste d'origine et mettre chaque élément de la liste au carré. Il renvoie ensuite le résultat obtenu, sous la forme d'une liste qui est de la même longueur que celle d'origine. On peut naturellement capturer cette nouvelle liste dans une variable.

Filtrage avec un branchement conditionnel

On peut aussi filtrer une liste de cette façon :

```
>>> liste_origine = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> [nb for nb in liste_origine if nb%2==0]
[2, 4, 6, 8, 10]
>>>
```

On rajoute à la fin de l'instruction une condition qui va déterminer quelles valeurs seront transférées dans la nouvelle liste. Ici, on ne transfère que les valeurs paires. Au final, on se retrouve donc avec une liste deux fois plus petite que celle d'origine.

Mélangeons un peu tout cela

Il est possible de filtrer et modifier une liste assez simplement. Par exemple, on a une liste contenant les quantités de fruits stockées pour un magasin (je ne suis pas sectaire, vous pouvez prendre des hamburgers si vous préférez). Chaque semaine, le magasin va prendre dans le stock une certaine quantité de chaque fruit, pour la mettre en vente. À ce moment, le stock de chaque fruit diminue naturellement. Inutile, en conséquence, de garder les fruits qu'on n'a plus en stock.

Je vais un peu reformuler. On va avoir une liste simple, qui contiendra des entiers, précisant la quantité de chaque fruit (c'est abstrait, les fruits ne sont pas précisés). On va faire une compréhension de liste pour diminuer d'une quantité donnée toutes les valeurs de cette liste, et on en profite pour retirer celles qui sont inférieures ou égales à 0.

```
>>> qtt_a_retirer = 7 # On retire chaque semaine 7 fruits de chaque sorte
>>> fruits_stockes = [15, 3, 18, 21] # Par exemple 15 pommes, 3 melons...
>>> [nb_fruits-qtt_a_retirer for nb_fruits in fruits_stockes if nb_fruits>qtt_a_retirer]
[8, 11, 14]
>>>
```

Comme vous le voyez, le fruit de quantité 3 n'a pas survécu à cette semaine d'achats. Bien sûr, cet exemple n'est pas complet : on n'a aucun moyen fiable d'associer les nombres restants aux fruits. Mais vous avez un exemple de filtrage et modification d'une liste.

Prenez bien le temps de regarder ces exemples : au début, la syntaxe des compréhensions de liste n'est pas forcément simple. Faites des essais, c'est aussi le meilleur moyen de comprendre.

Nouvelle application concrète

De nouveau, c'est à vous de travailler.

Nous allons en gros reprendre l'exemple précédent, en le modifiant un peu pour qu'il soit plus cohérent. Nous travaillons toujours avec des fruits sauf que, cette fois, nous allons associer un nom de fruit à la quantité restant en magasin. Nous verrons au prochain chapitre comment le faire avec des dictionnaires ; pour l'instant on va se contenter de listes :

```
>>> inventaire = [
...     ("pommes", 22),
...     ("melons", 4),
...     ("poires", 18),
...     ("fraises", 76),
...     ("prunes", 51),
... ]
>>>
```

Recopiez cette liste. Elle contient des tuples, contenant chacun un couple : le nom du fruit et sa quantité en magasin.

Votre mission est de trier cette liste en fonction de la quantité de chaque fruit. Autrement dit, on doit obtenir quelque chose de similaire à :

```
[
```



```
    ("fraises", 76),
    ("prunes", 51),
    ("pommes", 22),
    ("poires", 18),
    ("melons", 4),
]
```

Pour ceux qui n'ont pas eu la curiosité de regarder dans la documentation des listes, je signale à votre attention la méthode `sort` qui permet de trier une liste. Vous pouvez également utiliser la fonction `sorted` qui prend en paramètre la liste à trier (ce n'est pas une méthode de liste, faites attention). `sorted` renvoie la liste triée sans modifier la liste d'origine, ce qui peut être utile dans certaines circonstances, précisément celle-ci. À vous de voir, vous pouvez y arriver par les deux méthodes.

Bien entendu, essayez de faire cet exercice en utilisant les compréhensions de liste.

Je vous donne juste un petit indice : vous ne pouvez trier la liste comme cela, il faut l'inverser (autrement dit, placer la quantité avant le nom du fruit) pour pouvoir ensuite la trier par quantité.

Voici la correction que je vous propose :

```
# On change le sens de l'inventaire, la quantité avant le nom
inventaire_inverse = [(qtt, nom_fruit) for nom_fruit,qtt in inventaire]
# On n'a plus qu'à trier dans l'ordre décroissant l'inventaire inversé
# On reconstitue l'inventaire trié
inventaire = [(nom_fruit, qtt) for qtt,nom_fruit in sorted(inventaire_inverse, \
    reverse=True)]
```

Cela marche et le traitement a été fait en deux lignes.

Vous pouvez trier l'inventaire inversé avant la reconstitution, si vous trouvez cela plus compréhensible. Il faut privilégier la lisibilité du code.

```
# On change le sens de l'inventaire, la quantité avant le nom
inventaire_inverse = [(qtt, nom_fruit) for nom_fruit,qtt in inventaire]
# On trie l'inventaire inversé dans l'ordre décroissant
inventaire_inverse.sort(reverse=True)
# Et on reconstitue l'inventaire
inventaire = [(nom_fruit, qtt) for qtt,nom_fruit in inventaire_inverse)]
```

Faites des essais, entraînez-vous, vous en aurez sans doute besoin, la syntaxe n'est pas très simple au début. Et évitez de tomber dans l'extrême aussi : certaines opérations ne sont pas faisables avec les compréhensions de listes ou alors elles sont trop condensées pour être facilement compréhensibles. Dans l'exemple précédent, on aurait très bien pu remplacer nos deux à trois lignes d'instructions par une seule, mais cela aurait été dur à lire. Ne sacrifiez pas la lisibilité pour le simple plaisir de raccourcir votre code.

En résumé

- On peut découper une chaîne en fonction d'un séparateur en utilisant la méthode `split` de la chaîne.
- On peut joindre une liste contenant des chaînes de caractères en utilisant la méthode de chaîne `join`. Cette méthode doit être appelée sur le séparateur.
- On peut créer des fonctions attendant un nombre inconnu de paramètres grâce à la syntaxe `def fonction_inconnue(*parametres):` (les paramètres passés se retrouvent dans le tuple *parametres*).
- Les *compréhensions de listes* permettent de parcourir et filtrer une séquence en en renvoyant une nouvelle.
- La syntaxe pour effectuer un filtrage est la suivante `:nouvelle_squence = [element for element in ancienne_squence if condition]`.

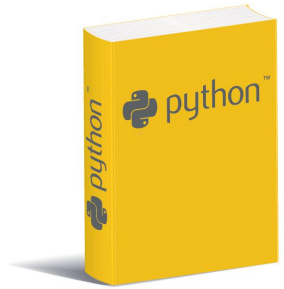
Chapitre 13

Les dictionnaires

Difficulté :

Maintenant que vous commencez à vous familiariser avec la programmation orientée objet, nous allons pouvoir aller un peu plus vite sur les manipulations « classiques » de ce type, pour nous concentrer sur quelques petites spécificités propres aux dictionnaires.

Les dictionnaires sont des objets pouvant en contenir d'autres, à l'instar des listes. Cependant, au lieu d'héberger des informations dans un ordre précis, ils associent chaque objet contenu à une clé (la plupart du temps, une chaîne de caractères). Par exemple, un dictionnaire peut contenir un carnet d'adresses et on accède à chaque contact en précisant son nom.



Création et édition de dictionnaires

Un dictionnaire est un type de données extrêmement puissant et pratique. Il se rapproche des listes sur certains points mais, sur beaucoup d'autres, il en diffère totalement. Python utilise ce type pour représenter diverses fonctionnalités : on peut par exemple retrouver les attributs d'un objet grâce à un dictionnaire particulier.

Mais n'anticipons pas. Dans les deux chapitres précédents, nous avons découvert les listes. Les objets de ce type sont des objets conteneurs, dans lesquels on trouve d'autres objets. Pour accéder à ces objets contenus, il faut connaître leur position dans la liste. Cette position se traduit par des entiers, appelés indices, compris entre 0 (inclus) et la taille de la liste (non incluse). Tout cela, vous devez déjà le savoir.

Le dictionnaire est aussi un objet conteneur. Il n'a quant à lui aucune structure ordonnée, à la différence des listes. De plus, pour accéder aux objets contenus dans le dictionnaire, on n'utilise pas nécessairement des indices mais des **clés** qui peuvent être de bien des types distincts.

Créer un dictionnaire

Là encore, je vous donne le nom de la classe sur laquelle se construit un dictionnaire : `dict`. Vous devriez du même coup trouver la première méthode d'instanciation d'un dictionnaire :

```
>>> mon_dictionnaire = dict()
>>> type(mon_dictionnaire)
<class 'dict'>
>>> mon_dictionnaire
{}
>>> # Du coup, vous devriez trouver la deuxième manière de cré
    er un dictionnaire vide
... mon_dictionnaire = {}
>>> mon_dictionnaire
{}
>>>
```

Les parenthèses délimitent les tuples, les crochets délimitent les listes et les accolades {} délimitent les dictionnaires.

Voyons comment ajouter des clés et valeurs dans notre dictionnaire vide :

```
>>> mon_dictionnaire = {}
>>> mon_dictionnaire["pseudo"] = "Prolixe"
>>> mon_dictionnaire["mot de passe"] = "*"
>>> mon_dictionnaire
{'mot de passe': '*', 'pseudo': 'Prolixe'}
>>>
```

Nous indiquons entre crochets la clé à laquelle nous souhaitons accéder. Si la clé n'existe pas, elle est ajoutée au dictionnaire avec la valeur spécifiée après le signe =. Sinon, l'ancienne valeur à l'emplacement indiqué est remplacée par la nouvelle :

```
1  >>> mon_dictionnaire = {}
2  >>> mon_dictionnaire["pseudo"] = "Prolixe"
3  >>> mon_dictionnaire["mot de passe"] = "*"
4  >>> mon_dictionnaire["pseudo"] = "6pri1"
5  >>> mon_dictionnaire
6  {'mot de passe': '*', 'pseudo': '6pri1'}
7  >>>
```

La valeur `'Prolixe'` pointée par la clé `'pseudo'` a été remplacée, à la ligne 4, par la valeur `'6pri1'`. Cela devrait vous rappeler la création de variables : si la variable n'existe pas, elle est créée, sinon elle est remplacée par la nouvelle valeur.

Pour accéder à la valeur d'une clé précise, c'est très simple :

```
1  >>> mon_dictionnaire["mot de passe"]
2  '*'
3  >>>
```

Si la clé n'existe pas dans le dictionnaire, une exception de type `KeyError` sera levée.

Généralisons un peu tout cela : nous avons des dictionnaires, qui peuvent contenir d'autres objets. On place ces objets et on y accède grâce à des clés. Un dictionnaire ne peut naturellement pas contenir deux clés identiques (comme on l'a vu, la seconde valeur écrase la première). En revanche, rien n'empêche d'avoir deux valeurs identiques dans le dictionnaire.

Nous avons utilisé ici, pour nos clés et nos valeurs, des chaînes de caractères. Ce n'est absolument pas obligatoire. Comme avec les listes, vous pouvez utiliser des entiers comme clés :

```
1  >>> mon_dictionnaire = {}
2  >>> mon_dictionnaire[0] = "a"
3  >>> mon_dictionnaire[1] = "e"
4  >>> mon_dictionnaire[2] = "i"
5  >>> mon_dictionnaire[3] = "o"
6  >>> mon_dictionnaire[4] = "u"
7  >>> mon_dictionnaire[5] = "y"
8  >>> mon_dictionnaire
9  {0: 'a', 1: 'e', 2: 'i', 3: 'o', 4: 'u', 5: 'y'}
10 >>>
```

On a l'impression de recréer le fonctionnement d'une liste mais ce n'est pas le cas : rappelez-vous qu'un dictionnaire n'a pas de structure ordonnée. Si vous supprimez par exemple l'indice 2, le dictionnaire, contrairement aux listes, ne va pas décaler toutes les clés d'indice supérieur à l'indice supprimé. Il n'a pas été conçu pour.

On peut utiliser quasiment tous les types comme clés et on peut utiliser absolument tous les types comme valeurs.

Voici un exemple un peu plus atypique de clés : on souhaite représenter un plateau d'échecs. Traditionnellement, on représente une case de l'échiquier par une lettre (de A à H) suivie d'un chiffre (de 1 à 8). La lettre définit la colonne et le chiffre définit la ligne. Si vous n'êtes pas sûrs de comprendre, regardez la figure 13.1.

FIGURE 13.1 – Échiquier

Pourquoi ne pas faire un dictionnaire dont les clés seront des tuples contenant la lettre et le chiffre identifiant la case, auxquelles on associe comme valeurs le nom des pièces ?

```
1  echiquier = {}
2  echiquier['a', 1] = "tour blanche" # En bas à gauche de l'é
       chiquier
3  echiquier['b', 1] = "cavalier blanc" # À droite de la tour
4  echiquier['c', 1] = "fou blanc" # À droite du cavalier
5  echiquier['d', 1] = "reine blanche" # À droite du fou
6  # ... Première ligne des blancs
7  echiquier['a', 2] = "pion blanc" # Devant la tour
8  echiquier['b', 2] = "pion blanc" # Devant le cavalier, à droite
       du pion
9  # ... Seconde ligne des blancs
```

Dans cet exemple, nos tuples sont sous-entendus. On ne les place pas entre parenthèses. Python comprend qu'on veut créer des tuples, ce qui est bien, mais l'important est que vous le compreniez bien aussi. Certains cours encouragent à toujours placer des parenthèses autour des tuples quand on les utilise. Pour ma part, je pense que, si vous gardez à l'esprit qu'il s'agit de tuples, que vous n'avez aucune peine à l'identifier, cela

suffit. Si vous faites la confusion, mettez des parenthèses autour des tuples en toutes circonstances.

On peut aussi créer des dictionnaires déjà remplis :

```
placard = {"chemise":3, "pantalon":6, "tee-shirt":7}
```

On précise entre accolades la clé, le signe deux points « : » et la valeur correspondante. On sépare les différents couples `clé : valeur` par une virgule. C'est d'ailleurs comme cela que Python vous affiche un dictionnaire quand vous le lui demandez.

Certains ont peut-être essayé de créer des dictionnaires déjà remplis avant que je ne montre comment faire. Une petite précision, si vous avez tapé une instruction similaire à :

```
mon_dictionnaire = {'pseudo', 'mot de passe'}
```

Avec une telle instruction, ce n'est pas un dictionnaire que vous créez, mais un `set`.

Un `set` (ensemble) est un objet conteneur (lui aussi), très semblable aux listes sauf qu'il ne peut contenir deux objets identiques. Vous ne pouvez pas trouver deux fois dans un `set` l'entier 3 par exemple. Je vous laisse vous renseigner sur les `sets` si vous le désirez.

Supprimer des clés d'un dictionnaire

Comme pour les listes, vous avez deux possibilités mais elles reviennent sensiblement au même :

- le mot-clé `del` ;
- la méthode de dictionnaire `pop`.

Je ne vais pas m'attarder sur le mot-clé `del`, il fonctionne de la même façon que pour les listes :

```
placard = {"chemise":3, "pantalon":6, "tee shirt":7}
del placard["chemise"]
```

La méthode `pop` supprime également la clé précisée mais elle renvoie la valeur supprimée :

```
>>> placard = {"chemise":3, "pantalon":6, "tee shirt":7}
>>> placard.pop("chemise")
3
>>>
```

La méthode `pop` renvoie la valeur qui a été supprimée en même temps que la clé. Cela peut être utile parfois.

Voilà pour le tour d'horizon. Ce fut bref et vous n'avez pas vu toutes les méthodes, bien entendu. Je vous laisse consulter l'aide pour une liste détaillée.

Un peu plus loin

On se sert parfois des dictionnaires pour stocker des fonctions.

Je vais juste vous montrer rapidement le mécanisme sans trop m'y attarder. Là, je compte sur vous pour faire des tests si vous êtes intéressés. C'est encore un petit quelque chose que vous n'utiliserez peut-être pas tous les jours mais qu'il peut être utile de connaître.

Les fonctions sont manipulables comme des variables. Ce sont des objets, un peu particuliers mais des objets tout de même. Donc on peut les prendre pour valeur d'affectation ou les ranger dans des listes ou dictionnaires. C'est pourquoi je présente cette fonctionnalité à présent, auparavant j'aurais manqué d'exemples pratiques.

```
>>> print_2 = print # L'objet print_2 pointera sur la fonction print
>>> print_2("Affichons un message")
Affichons un message
>>>
```

On copie la fonction `print` dans une autre variable `print_2`. On peut ensuite appeler `print_2` et la fonction va afficher le texte saisi, tout comme `print` l'aurait fait.

En pratique, on affecte rarement des fonctions de cette manière. C'est peu utile. Par contre, on met parfois des fonctions dans des dictionnaires :

```
>>> def fete():
...     print("C'est la fête.")
...
>>> def oiseau():
...     print("Fais comme l'oiseau...")
...
>>> fonctions = {}
>>> fonctions["fete"] = fete # on ne met pas les parenthèses
>>> fonctions["oiseau"] = oiseau
>>> fonctions["oiseau"]
<function oiseau at 0x00BA5198>
>>> fonctions["oiseau"]() # on essaye de l'appeler
Fais comme l'oiseau...
>>>
```

Prenons dans l'ordre si vous le voulez bien :

- On commence par définir deux fonctions, `fete` et `oiseau` (pardonnez l'exemple).
- On crée un dictionnaire nommé `fonctions`.
- On met dans ce dictionnaire les fonctions `fete` et `oiseau`. La clé pointant vers la fonction est le nom de la fonction, tout bêtement, mais on aurait pu lui donner un nom plus original.
- On essaye d'accéder à la fonction `oiseau` en tapant `fonctions[« oiseau »]`. Python nous renvoie un truc assez moche, `<function oiseau at 0x00BA5198>`, mais vous

comprenez l'idée : c'est bel et bien notre fonction `oiseau`. Toutefois, pour l'appeler, il faut des parenthèses, comme pour toute fonction qui se respecte.
– En tapant `fonctions["oiseau"]()`, on accède à la fonction `oiseau` et on l'appelle dans la foulée.

On peut stocker les références des fonctions dans n'importe quel objet conteneur, des listes, des dictionnaires... et d'autres classes, quand nous apprendrons à en faire. Je ne vous demande pas de comprendre absolument la manipulation des références des fonctions, essayez simplement de retenir cet exemple. Dans tous les cas, nous aurons l'occasion d'y revenir.

Les méthodes de parcours

Comme vous pouvez le penser, le parcours d'un dictionnaire ne s'effectue pas tout à fait comme celui d'une liste. La différence n'est pas si énorme que cela mais, la plupart du temps, on passe par des méthodes de dictionnaire.

Parcours des clés

Peut-être avez-vous déjà essayé par vous-mêmes de parcourir un dictionnaire comme on l'a fait pour les listes :

```
>>> fruits = {"pommes":21, "melons":3, "poires":31}
>>> for cle in fruits:
...     print(cle)
...
melons
poires
pommes
>>>
```

Comme vous le voyez, si on essaye de parcourir un dictionnaire « simplement », on parcourt en réalité la liste des clés contenues dans le dictionnaire.

Mais... les clés ne s'affichent pas dans l'ordre dans lequel on les a entrées... c'est normal ?

Les dictionnaires n'ont pas de structure ordonnée, gardez-le à l'esprit. Donc en ce sens oui, c'est tout à fait normal.

Une méthode de la classe `dict` permet d'obtenir ce même résultat. Personnellement, je l'utilise plus fréquemment car on est sûr, en lisant l'instruction, que c'est la liste des clés que l'on parcourt :

```
>>> fruits = {"pommes":21, "melons":3, "poires":31}
>>> for cle in fruits.keys():
...     print(cle)
...
melons
poires
pommes
>>>
```


La méthode `keys` (« clés » en anglais) renvoie la liste des clés contenues dans le dictionnaire. En vérité, ce n'est pas tout à fait une liste (essayez de taper `fruits.keys()` dans votre interpréteur) mais c'est une séquence qui se parcourt comme une liste.

Parcours des valeurs

On peut aussi parcourir les valeurs contenues dans un dictionnaire. Pour ce faire, on utilise la méthode `values` (« valeurs » en anglais).

```
>>> fruits = {"pommes":21, "melons":3, "poires":31}
>>> for valeur in fruits.values():
...     print(valeur)
...
3
31
21
>>>
```

Cette méthode est peu utilisée pour un parcours car il est plus pratique de parcourir la liste des clés, cela suffit pour avoir les valeurs correspondantes. Mais on peut aussi, bien entendu, l'utiliser dans une condition :

```
>>> if 21 in fruits.values():
...     print("Un des fruits se trouve dans la quantité 21.")
...
Un des fruits se trouve dans la quantité 21.
>>>
```

Parcours des clés et valeurs simultanément

Pour avoir en même temps les indices et les objets d'une liste, on utilise la fonction `enumerate`, j'espère que vous vous en souvenez. Pour faire de même avec les dictionnaires, on utilise la méthode `items`. Elle renvoie une liste, contenant les couples `clé : valeur`, sous la forme d'un `tuple`. Voyons comment l'utiliser :

```
>>> fruits = {"pommes":21, "melons":3, "poires":31}
>>> for cle, valeur in fruits.items():
...     print("La clé {} contient la valeur {}.".format(cle,
    valeur))
```

```
 ...
La clé melons contient la valeur 3.
La clé poires contient la valeur 31.
La clé pommes contient la valeur 21.
 >>>
```

Il est parfois très pratique de parcourir un dictionnaire avec ses clés et les valeurs associées.

Entraînez-vous, il n'y a que cela de vrai. Pourquoi pas reprendre l'exercice du chapitre précédent, avec notre inventaire de fruits ? Sauf que le type de l'inventaire ne serait pas une liste mais un dictionnaire associant les noms des fruits aux quantités ?

Il nous reste une petite fonctionnalité supplémentaire à voir et on en aura fini avec les dictionnaires.

Les dictionnaires et paramètres de fonction

Cela ne vous rappelle pas quelque chose ? J'espère bien que si, on a vu quelque chose de similaire au chapitre précédent.

Si vous vous souvenez, on avait réussi à intercepter tous les paramètres de la fonction… sauf les paramètres nommés.

Récupérer les paramètres nommés dans un dictionnaire

Il existe aussi une façon de capturer les paramètres nommés d'une fonction. Dans ce cas, toutefois, ils sont placés dans un dictionnaire. Si, par exemple, vous appelez la fonction ainsi : `fonction(parametre='a')`, vous aurez, dans le dictionnaire capturant les paramètres nommés, une clé `'parametre'` liée à la valeur `'a'`. Voyez plutôt :

```
>>> def fonction_inconnue(**parametres_nommes):
...     """Fonction permettant de voir comment récupérer les
    paramètres nommés
...     dans un dictionnaire"""
...
...
...     print("J'ai reçu en paramètres nommés : {}.".format(
    parametres_nommes))
...
>>> fonction_inconnue() # Aucun paramètre
J'ai reçu en paramètres nommés : {}
>>> fonction_inconnue(p=4, j=8)
J'ai reçu en paramètres nommés : {'p': 4, 'j': 8}
>>>
```

Pour capturer tous les paramètres nommés non précisés dans un dictionnaire, il faut mettre deux étoiles ** avant le nom du paramètre.

Si vous passez des paramètres non nommés à cette fonction, Python lèvera une exception.

Ainsi, pour avoir une fonction qui accepte n'importe quel type de paramètres, nommés ou non, dans n'importe quel ordre, dans n'importe quelle quantité, il faut la déclarer de cette manière :

```
def fonction_inconnue(*en_liste, **en_dictionnaire):
```

Tous les paramètres non nommés se retrouveront dans la variable `en_liste` et les paramètres nommés dans la variable `en_dictionnaire`.

Mais à quoi cela peut-il bien servir d'avoir une fonction qui accepte n'importe quel paramètre ?

Pour l'instant à pas grand chose mais cela viendra. Quand on abordera le chapitre sur les décorateurs, vous vous en souviendrez et vous pourrez vous féliciter de connaître cette fonctionnalité.

Transformer un dictionnaire en paramètres nommés d'une fonction

Là encore, on peut faire exactement l'inverse : transformer un dictionnaire en paramètres nommés d'une fonction. Voyons un exemple tout simple :

```
>>> parametres = {"sep":" >> ", "end":" -\n"}
>>> print("Voici", "un", "exemple", "d'appel", **parametres)
Voici >> un >> exemple >> d'appel -
>>>
```

Les paramètres nommés sont transmis à la fonction par un dictionnaire. Pour indiquer à Python que le dictionnaire doit être transmis comme des paramètres nommés, on place deux étoiles avant son nom ** dans l'appel de la fonction.

Comme vous pouvez le voir, c'est comme si nous avions écrit :

```
>>> print("Voici", "un", "exemple", "d'appel", sep=" >> ", end=
    -\n")
Voici >> un >> exemple >> d'appel -
>>>
```

Pour l'instant, vous devez trouver que c'est bien se compliquer la vie pour si peu. Nous verrons dans la suite de ce cours qu'il n'en est rien, en fait, même si nous n'utilisons pas cette fonctionnalité tous les jours.

En résumé

- Un dictionnaire est un objet conteneur associant des clés à des valeurs.
- Pour créer un dictionnaire, on utilise la syntaxe `dictionnaire = {cle1:valeur1, cle2=valeur2, cleN=valeurN}`.
- On peut ajouter ou remplacer un élément dans un dictionnaire : `dictionnaire[cle] = valeur`.
- On peut supprimer une clé (et sa valeur correspondante) d'un dictionnaire en utilisant, au choix, le mot-clé `del` ou la méthode `pop`.
- On peut parcourir un dictionnaire grâce aux méthodes `keys` (parcourt les clés), `values` (parcourt les valeurs) ou `items` (parcourt les couples clé-valeur).
- On peut capturer les paramètres nommés passés à une fonction en utilisant cette syntaxe : `def fonction_inconnue(**parametres_nommes` : (les paramètres nommés se retrouvent dans le dictionnaire `parametres_nommes`).