Python 3 pour l'apprenti sorcier

Arnaud Calmettes (nohar)

Table des matières

Avant-propos

Vous désirez vous initier à l'art de la programmation, et je vous en félicite.

Vous ne vous en doutez peut-être pas, mais vous êtes sur le seuil d'un nouveau monde, vaste et passionnant. Un monde qui regorge de formules, de recettes et d'objets magiques. Un monde dans lequel le périple que nous nous apprêtons à accomplir changera à jamais la vision que vous avez de l'univers et ouvrira votre esprit à de nouvelles façons de penser, de créer et de vous amuser.

Oh, c'est cela, riez. Riez donc, innocent que vous êtes ! Je vois bien que vous ne me croyez pas, mais je vous l'affirme : la programmation tient autant de la magie que de la science, et si vous êtes assez brave pour me suivre, *je vais vous le montrer*.

...

Bon, OK, j'exagère peut-être un *tout petit* peu.

En fait, nous n'allons pas partir en voyage ni braver le danger. Promis. Vous allez simplement suivre un cours en toute sécurité et pas à pas, sans bouger de votre fauteuil. Vous êtes rassuré ?

À part ça, **tout ce que je vous ai dit d'autre est vrai !**

1 Introduction

TODO

2 Rencontre du troisième type

> Le renard se tut et regarda longtemps le petit prince :
> – S'il te plaît... apprivoise-moi ! dit-il.
> – Je veux bien, répondit le petit prince, mais je n'ai pas beaucoup de temps. J'ai des amis à découvrir et beaucoup de choses à connaître.
> – On ne connaît que les choses que l'on apprivoise, dit le renard.
>
> *Antoine de Saint-Exupéry – Le Petit Prince*

Maintenant qu'un Python habite dans votre ordinateur, il est temps de faire sa connaissance. Dans ce chapitre, nous allons visiter son habitat naturel : la console interactive, que vous utiliserez tout le temps pour tester vos programmes. Ce sera votre lieu de rencontre privilégié, puisque c'est ici que vous pourrez *dialoguer* avec lui.

Suivez moi ! Il nous attend déjà.

Contact

Nous voici devant l'antre de la bête.

Commençons par l'appeler, si vous le voulez bien. Pour cela, lancez l'interpréteur de commandes Python comme je vous l'ai montré dans le chapitre précédent (en tapant "`python3`" ou bien "`py -3`" dans votre console, suivant votre système d'exploitation).

Sa queue se déroule dans la console, et le voilà qui vous fixe de ses grands yeux verts :

```
Python 3.4.0 (default, Apr 11 2014, 13:05:11)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Ces trois chevrons (`>>>`) constituent **l'invite de commande**. C'est la façon dont Python vous signale qu'il vous écoute et qu'il attend vos instructions. Montrez-vous bien élevé, saluez-le et validez en appuyant sur la touche `<Entrée>`.

```
>>> Bonjour
Traceback:
  File "<stdin>", line 1, in <module>
NameError: name 'Bonjour' is not defined
```

Ouh là, attention, reculez ! Vous l'avez contrarié.

Ce que Python vient de vous cracher au visage, c'est un message qui signifie qu'il n'a pas compris ce que vous lui dites. En effet, pour créer un programme, il ne suffit pas de donner

des ordres en français à votre interpréteur. Il faut que vous lui parliez **en Python**, et c'est justement ce que vous êtes venu apprendre.

Pour le moment je vais vous aider. Entourez donc votre phrase avec des guillemets (`"`), comme ceci :

```
>>> "Bonjour"
'Bonjour'
```

Le voilà plus coopératif. Python vous a renvoyé la politesse. Poursuivons :

```
>>> "Je m'appelle Arnaud. Et toi ?"
"Je m'appelle Arnaud. Et toi ?"
>>> "Mais, je te l'ai déjà dit..."
"Mais, je te l'ai déjà dit..."
>>> "Hé ! Tu arrêtes de répéter tout ce que je dis ?!"
'Hé ! Tu arrêtes de répéter tout ce que je dis ?!'
```

Rassurez-vous : aussi taquin soit-il, Python n'est pas vraiment en train de se moquer de vous. Laissez-moi vous expliquer.

Le rôle de cette console interactive est d'**interpréter** les instructions que vous lui soumettez. Chaque fois que vous appuyez sur `<Entrée>`, vous envoyez l'expression que vous venez de taper à Python, celui-ci l'évalue et vous retourne *sa valeur*. Jusqu'ici, vous lui avez envoyé des expressions qui ont toutes la même forme : du texte entouré de guillemets. Il s'agit de données de base que peut manipuler l'interpréteur et que l'on appelle des *chaînes de caractères*. Python s'est contenté d'évaluer ces données, et de vous retourner leurs valeurs telles qu'il les a comprises, c'est-à-dire qu'il vous a renvoyé les chaînes telles quelles.

Nous reparlerons très bientôt des chaînes de caractères. Pour l'heure, si nous essayions de lui faire évaluer des choses plus utiles ?

Une bête de calcul

Je vous ai dit que la console interactive sert à *évaluer des expressions*, mais que sont ces fameuses « expressions » ? Nous allons commencer à le découvrir dans la suite de ce chapitre.

Saisir un nombre

Vous avez pu voir juste au-dessus que Python n'aime pas du tout les suites de lettres qu'il ne comprend pas. Par contre, non seulement il comprend très bien les nombres, mais en plus, il les adore !

```
>>> 42
42
```

Vous pouvez même saisir des nombres à virgule, regardez :

```
>>> 13.37
13.37
```

> **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, que nous découvrirons plus loin.**

En programmation, ces nombres à virgules sont appelé des **flottants**. Ils ne sont pas représentés de la même façon dans la mémoire de l'ordinateur que les nombres entiers. Ainsi, même si leurs valeurs sont égales, sachez que `5` n'est pas tout à fait la même chose que `5.0`.

Il va de soi que l'on peut tout aussi bien saisir des nombres négatifs. Essayez. ;)

D'accord, ce n'est pas extraordinaire. On saisit un nombre et Python le répète : la belle affaire ! Néanmoins, ce n'est pas aussi inutile que vous pourriez le penser. Le fait qu'il nous renvoie une valeur signifie que Python a bien compris **la forme** et **le sens** de ce que vous avez saisi. Autrement dit, cela signifie que *les expressions* que vous avez tapées sont **valides** dans le langage Python, donc que vous pouvez utiliser cette console pour vérifier la validité d'une expression en cas de doute.

> **Un peu de vocabulaire :**
> L'ensemble de règles qui déterminent *la forme* des expressions acceptées par un langage de programmation constitue la **syntaxe** de ce langage. *Le sens* de ces expressions, quant à lui, est déterminé par sa **sémantique**.

Nous avons maintenant un première observation à retenir : les nombres, tout comme les chaînes de caractères (`"Bonjour"`), sont des expressions valides en Python. Ces expressions désignent **des données** que l'on peut manipuler dans ce langage.

Opérations courantes

Maintenant que nous savons que Python comprend les nombres, voyons un peu ce qu'il est capable de faire avec.

Addition, soustraction, multiplication, division

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

```
>>> 3 + 4
7
>>> -2 + 93
91
>>> 9.5 - 2
7.5
>>> 3.11 + 2.08
5.1899999999999995
```

> *Pourquoi le dernier résultat est-il approximatif ?*

Python n'y est pas pour grand chose. En fait, cela vient de la façon dont les nombres flottants sont représentés dans la mémoire de votre ordinateur. Pour que tous les nombres

à virgule aient une taille fixe en mémoire, ceux-ci sont représentés avec un nombre limité de chiffres significatifs, ce qui impose à l'ordinateur de faire des approximations parfois un peu déroutantes.

Cependant, vous remarquerez que l'erreur (à 0.000000000000001 près) est infime et qu'elle n'aura pas de gros impact sur les calculs. Les applications qui ont besoin d'une précision mathématique à toute épreuve peuvent pallier ce défaut par d'autres moyens. Ici, ce ne sera pas nécessaire.

Faites également des tests pour 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.

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

En d'autres termes, quelles que soient ses opérandes, la division de Python donnera toujours son résultat sous la forme d'un flottant. Il existe deux autres opérateurs qui permettent de connaître respectivement le résultat d'une *division euclidienne* et le reste de cette division.

La division euclidienne, c'est celle que vous apprenez à poser à l'école. Rappelons rapidement sa définition : la division euclidienne de a par b est l'opération qui consiste à trouver le *quotient* q et le *reste* r tel que $0 \leq r < b$, vérifiant :

$$a = q \times b + r$$

L'exemple ci-dessous montre que la division euclidienne de 10 par 3 a pour quotient 3 et pour reste 1. Soit $10 = 3 \times 3 + 1$.

```
10 | 3
   +---
 1 | 3
   |
```

L'opérateur permettant d'obtenir le quotient d'une division euclidienne est appelé « division entière ». On le note avec le symbole `//`.

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

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

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

Puissance

Si vous êtes curieux, vous avez probablement déjà essayé de calculer une *puissance* avec Python, de la même façon que vous auriez utilisé une calculatrice. Sur cette dernière, vous vous souvenez certainement que l'opérateur de puissance correspond à la touche "^". Essayons !

```
>>> 3 ^ 2
1
```

En voilà, un résultat bizarre. Tout le monde sait bien que $3^2 = 9$! o_O

Rassurez-vous. Python le sait également. C'est simplement que l'opérateur `^` ne désigne pas une puissance pour lui, mais une toute autre opération qui porte le nom barbare de « XOR bit-à-bit » et que vous pouvez vous empresser d'oublier pour le moment.

L'opérateur de puissance, quant à lui, se note `**`. Regardez :

```
>>> 3 ** 2
9
```

Vous pouvez même vous en servir pour calculer une puissance qui n'est pas entière. Par exemple, une racine carrée (rappelez-vous que $\sqrt{x} = x^{\frac{1}{2}}$) :

```
>>> 25 ** (1/2)
5.0
```

Notez que le résultat nous est donné sous la forme d'un nombre à virgule, puisque l'un des deux opérandes, `1/2`, n'est pas une valeur entière.

Tout cela nous amène à deux nouvelles observations. La première, c'est qu'en plus des nombres, les expressions mathématiques sont également des expressions valides en Python.

La seconde, c'est que votre nouvel ami serpent est un génie en calcul mental. Cela n'a rien d'étonnant ; souvenez-vous que la principale fonction d'un programme (donc d'un langage de programmation) est de nous permettre à nous, humains, de faire faire des calculs à un ordinateur. En fait, pour lui, *toute instruction est un calcul à effectuer*, y compris, évidemment, les petites opérations de calcul mental que nous venons de lui soumettre.

En résumé

Récapitulons ce que nous avons appris dans ce premier chapitre.

- L'invite de commandes Python permet de tester du code au fur et à mesure qu'on l'écrit.
- Python est capable d'effectuer des calculs, exactement comme une calculatrice.
- Un nombre décimal s'écrit avec un point et non une virgule. Les calculs impliquant des nombres décimaux donnent parfois des résultats approximatifs.
- Python différencie la *division réelle* (`/`) de la *division euclidienne* (`//` et `%`).

Plus important encore, en termes de vocabulaire :

- Les chaînes de caractères (comme `"Salut !"`) et les nombres sont des **expressions valides** en Python.

— La forme des expressions acceptées par un langage de programmation est sa **syntaxe**.
— Le sens que donne le langage aux expressions qu'il accepte est sa **sémantique**.

Bien entendu, Python est bien plus qu'une simple calculatrice. Vous allez très vite vous en rendre compte dans les prochains chapitres.

3 Des noms et des objets

> *Brisingr* signifie *feu*. C'*est* le feu. La chose est le mot. Connais le mot, tu domineras la chose.
>
> *Brom – Eragon (Twentieth Century Fox, 2006)*

C'est l'un des principes fondamentaux de la magie comme de la programmation : nommer quelque chose, c'est le contrôler. Quand on écrit un programme, on manipule des **objets** (des *choses* abstraites) en leur donnant des **noms**. C'est la base, la première notion essentielle que vous devez maîtriser pour ~~jeter des sorts~~ programmer.

Dans ce chapitre, nous allons apprendre :

— à nommer les objets pour les manipuler,
— que tous les objets ne sont pas de la même nature,
— que certains objets représentent des *données*,
— que d'autres sont des *actions*.

Les variables

Des étiquettes pour s'y retrouver

En Python, une variable n'est rien d'autre qu'un **nom** que l'on donne à un objet pour se souvenir de lui et le manipuler. Regardez :

```
>>> x = 40
>>> x
40
```

Dans cet exemple, j'ai créé une variable `x` à laquelle j'ai *affecté* la valeur `40`, puis j'ai demandé à Python de me donner la valeur de `x`, et celui-ci m'a retourné `40`. À partir de maintenant, je peux utiliser le nom `x` dans toutes les expressions où j'aurai besoin de ce nombre :

```
>>> x + 2
42
>>> x * 3
120
```

Mais pourquoi est-ce qu'on appelle ça une variable ? me demanderez-vous à juste titre.

Eh bien parce que les données qu'elles désignent ne sont pas *fixes*. Contrairement à la chaîne `"Bonjour"` ou au nombre `40` (que l'on appelle, par opposition, des *constantes*) la valeur d'une variable peut changer au cours de l'exécution du programme : elle est… variable,

quoi. Autrement dit, `"Paris"` sera toujours `"Paris"`, mais `rien` n'est jamais figé dans le marbre. ;)

Par exemple, rien ne m'empêche de décider que `40`, c'est nul, et que `x` ferait mieux de désigner le nombre `25` ou la chaîne `"Salut"`. Python se contentera de changer docilement la valeur associée à ce nom.

```
>>> x = 25
>>> x
25
>>> x = "Salut"
>>> x
'Salut'
```

> **Il faut voir une variable en Python comme une simple étiquette, un *post-it* sur lequel on écrit un nom et que l'on peut coller sur un objet, comme ça nous chante, quitte à le décoller plus tard pour le coller sur autre chose.**

De nombreux cours de programmation choisissent de présenter les variables comme des *boîtes* dans lesquelles on peut ranger des valeurs : si c'est l'image que vous avez en tête, **chassez-la immédiatement de votre esprit** ! C'est une bonne analogie pour certains langages de programmation, mais pas ici. En Python, une variable n'est vraiment qu'*un nom* attaché à quelque chose. Contrairement à une boîte qui a nécessairement un volume (qu'elle soit vide ou non), vous pouvez considérer qu'une variable de Python ne prend (presque) pas de place dans la mémoire de votre ordinateur : que vous la colliez sur une souris ou sur un éléphant, une étiquette aura toujours la même taille négligeable.

Nous aurons l'occasion de revenir sur cette histoire d'étiquettes (ou *références*) dans un autre chapitre. En attendant, gardez bien cette métaphore en mémoire *et ne me parlez plus jamais de boîtes !*

> *D'accord, une variable est une étiquette que l'on colle sur les choses pour leur donner un nom, mais... c'est tout ? Ça ne sert* qu'à ça *?*

En effet, ça ne sert qu'à « ça », mais ce « ça » est beaucoup plus important que ce que vous imaginez. En donnant un nom à un objet, vous demandez à Python de *se souvenir de lui*. Littéralement. Vous lui ordonnez de le garder bien au chaud dans un coin de sa mémoire. Ce n'est pas anodin du tout !

Tant qu'un objet est affecté à une variable, celui-ci reste accessible à Python : on dit qu'il est *référencé*. Et l'inverse est vrai également : si un objet n'est référencé par aucune variable, alors Python se donne le droit de l'oublier en l'effaçant de sa mémoire. C'est exactement comme quand vous prenez le train :

> *Nous vous demandons d'étiqueter convenablement tous vos bagages. Tout colis ou objet abandonné sera systématiquement détruit.* [1]

En résumé : si vous comptez utiliser un objet, nommez-le !

1. Annonce en gare de la SNCF

Un peu de syntaxe

Ce simple mécanisme de *nommage* est tellement utilisé en Python que celui-ci propose un certain nombre de raccourcis pour définir, modifier ou supprimer des variables.

Comme nous venons de le voir, on affecte un objet à une variable en utilisant le symbole `=`.

```
>>> langage = "Python"
```

Ce symbole, dans le jargon des programmeurs, on l'appelle *l'opérateur d'affectation*. On peut aussi s'en servir pour affecter simultanément plusieurs valeurs à plusieurs variables, pour peu que celles-ci soient séparées par des virgules :

```
>>> langage, version = "Python", 3.4
>>> langage
'Python'
>>> version
3.4
```

Si l'on souhaite affecter une seule valeur à plusieurs variables en même temps, on peut même chaîner les affectations, comme ceci :

```
>>> x = y = 0
>>> x
0
>>> y
0
```

Notez que dans ce cas, nous avons simplement collé deux étiquettes sur le même objet. Si on en déplace une, l'autre ne sera pas impactée.

Enfin, si vous souhaitez supprimer purement et simplement une variable, vous pouvez utiliser le mot-clé `del` pour demander à Python de l'oublier.

```
>>> prenom = "Clem"
>>> prenom
'Clem'
>>> del prenom
>>> prenom
Traceback:
    ...
NameError: name 'prenom' is not defined
```

> **Notez bien que `del`, dans ce contexte, sert à supprimer une *variable*, mais pas forcément l'objet qu'elle référence.**

La différence c'est que si plusieurs variables pointent sur le même objet, la suppression de l'une d'entre elles n'affectera pas les autres :

```
>>> a = b = 1
>>> b
1
>>> del b
```

```
>>> a
1
```

Voilà qui devrait suffir en ce qui concerne les affectations. Et si nous nous intéressions maintenant à ces fameux objets que vous venez d'apprendre à nommer ?

`TODO : passer ce qui suit dans un paragraphe sur les conventions de nommage.`

> **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.

C'est l'histoire d'un type...

Jusqu'à maintenant, nous avons joué rapidement avec des données toutes simples (`42`, `"Bonjour"`), et depuis le début de ce chapitre, je vous parle de ces données comme étant des *objets* sur lesquels on peut coller des étiquettes.

La notion d'*objet* en programmation est assez vaste, et c'est la raison pour laquelle vous en découvrirez les détails de façon progressive, à mesure que nous avancerons dans ce cours. Vous vous apercevrez bientôt qu'en Python, *tout* est considéré comme un objet, mais l'idée que je voudrais que vous gardiez en tête pour le moment c'est que **les objets sont les choses que l'on manipule dans un programme**.

Et la première caractéristique d'un objet, c'est son **type**. On parle également de « type de données ».

Plusieurs types de données

Vous serez d'accord avec moi si je vous dis qu'une éponge et un tournevis ne sont absolument pas la même chose. Pourtant, ce sont deux objets, non ? Eh bien en programmation, on dira que ce sont deux objets *de types différents*.

Comme le dit le dicton, *on ne peut pas additionner des choux et des carottes* et *on ne mélange pas les torchons avec les serviettes*. Chaque objet porte un *type* qui caractérise la façon dont on peut l'utiliser : vous imaginez bien qu'on ne fera pas les mêmes choses avec le nombre `42` qu'avec la chaîne `"Je mange des ours"`.

> **Le type d'un objet, c'est ce qui définit ses caractéristiques : à quoi il ressemble, comment on l'utilise, à quoi il sert... bref, ce qu'il *est*.**

Jusqu'ici, vous avez croisé des objets de trois types différents. Prenons le temps de les examiner de plus près, si vous le voulez bien.

Les nombres entiers

Nous l'avons vu au chapitre précédent, Python fait la différence entre les nombres entiers et les nombres à virgule. Un nombre entier, c'est donc un nombre *sans virgule*, comme `3`, `920000` ou encore `-12`. Ces nombres, Python les appelle des `int`. Il s'agit de l'abréviation de *integer* (le mot anglais pour *nombre entier*). En français, nous nous contenterons de les appeler « des entiers ».

Étant donné que nous avons déjà passé en revue les opérations que l'on peut appliquer sur ces nombres, nous n'allons pas les répéter ici.

Par contre, il peut être intéressant de savoir que Python accepte plusieurs notations pour désigner des entiers. Si les notions de *notation binaire* ou *hexadécimale* ne vous disent rien, vous pouvez sauter le paragraphe suivant.

> Dans beaucoup de langages de programmation, les nombres peuvent être représentés selon différents systèmes de numération.
> Par exemple, on peut les noter sous leur forme binaire en préfixant la valeur de `0b` :
>
> ```
> >>> 0b10
> 2
> >>> 0b11
> 3
> >>> 0b101010
> 42
> ```
>
> Ou bien selon la notation hexadécimale, avec le préfixe `0x` :
>
> ```
> >>> 0x10
> 16
> >>> 0xff
> 255
> ```
>
> Ou encore en octal en utilisant le préfixe `0o` :
>
> ```
> >>> 0o10
> 8
> >>> 0o755
> 493
> ```

> ‘ Python comprendra tous ces nombres comme des entiers. Seule la notation change, pas leur type.

Enfin, si vous savez déjà programmer dans un autre langage, il est possible que vous vous demandiez quelles sont *les bornes* (les valeurs minimale et maximale) des entiers. Sachez qu'en Python, les entiers ne sont pas bornés : tant que vous disposez de suffisamment de mémoire pour les faire tenir dedans, Python est capable de représenter des entiers indéfiniment grands ou petits.

Vous ne me croyez pas ? Eh bien, demandez-lui donc de calculer `123456789 ** 500` ; vous verrez si je vous raconte des bobards !

Les nombres flottants

Les « flottants » sont des nombres à virgule, comme `3.1416` ou `-169.8`. Python les appelle des `float`. D'emblée, on peut se demander quel peut bien être le rapport entre un nombre et le fait de *flotter*. Eh bien ce qui flotte, c'est justement la virgule : la véritable appellation de ces nombres est « nombres à virgule flottante ».

Je vous ai dit plus tôt que ces nombres sont limités à une taille fixe en mémoire (64 bits sur la plupart des ordinateurs). Imaginons que cela corresponde à 16 chiffres significatifs, même s'il s'agit d'une très grossière approximation de la réalité. Comment faire, dans ce cas, pour représenter un nombre positif supérieur à 10^{16} ou inférieur à 10^{-16} ? Eh bien l'idée, c'est qu'une partie de la donnée sert à décrire où se trouve la virgule (la *puissance de dix* du nombre, son *exposant*) alors que le reste nous donne les 16 chiffres significatifs (qui forment la *mantisse*[2] du nombre).

En somme, pour représenter les nombres `123.0`, `1230000.0` et `0.00000123`, votre ordinateur se contente de déplacer la virgule vers la droite ou la gauche en faisant varier l'exposant, tout en gardant une mantisse identique. C'est parce que cette virgule se déplace “toute seule” en fonction de l'ordre de grandeur du nombre qu'on la qualifie de *flottante*.

Cette façon de représenter les nombres décimaux n'est pas propre à Python : elle est encodée dans le microprocesseur de votre ordinateur. Elle lui permet d'effectuer des calculs extrêmement rapidement, avec une précision imparfaite, certes, mais plus que correcte sur des nombres arbitrairement grands ou petits. Cela dit, pour votre culture, sachez que Python propose également dans sa bibliothèque standard un type de données qui permet de représenter les nombres décimaux avec une précision réglable et extrêmement fine. Nous n'en parlerons pas dans ce cours, mais sachez que ça existe !

Les chaînes de caractères

Il n'y a pas que les nombres dans la vie, il y a le texte aussi ! Et ce texte, on le représente en Python dans des objets qu'il appelle des `str`.

> *Des* streuh *? C'est quoi un* streuh *?*

2. Si vous trouvez que ce mot n'est pas très joli, figurez-vous qu'on peut faire encore bien pire que ça ! En effet, une norme internationale préconise l'utilisation du terme *significande* à la place de *mantisse*...

C'est l'abréviation du mot *string* (ne rigolez pas !), que l'on peut traduire par *chaîne* dans ce contexte, sous-entendu *chaîne de caractères*.

Comme nous l'avons déjà vu, on peut créer une chaîne de caractères en plaçant le texte entre guillemets doubles, comme ceci : `"Salut, Python !"`. Toutefois, c'est loin d'être la seule façon de les écrire. Imaginez que vous ayez besoin que votre chaîne *contienne* des guillemets, comment s'y prendre ?

Pour cela, vous avez deux solutions. La première et... la seconde.

La première, c'est d'entourer votre texte de *guillemets simples* (l'*apostrophe* ' sur votre clavier), comme ceci :

```
>>> '"Diantre !" dis-je.'
'"Diantre !" dis-je.'
```

Remarquez que c'est exactement sous cette forme que la console Python vous retourne le résultat.

La seconde solution, c'est d'*échapper* les guillemets contenus dans la chaîne, en utilisant le symbole *anti-slash* (\), comme ceci :

```
>>> "\"Diantre !\" dis-je."
'"Diantre !" dis-je.'
```

Ce symbole \ sert à introduire un caractère spécial dans les chaînes. Ici, les guillemets ont ceci de spécial qu'ils *ne servent pas à délimiter une chaîne*. C'est exactement la même chose lorsque vous souhaitez échapper une apostrophe dans une chaîne délimitée par des guillemets simples :

```
>>> 'Python, c\'est coolympique !'
"Python, c'est coolympique !"
```

Regardez la réaction de Python sur le dernier exemple. Pour éviter d'échapper l'apostrophe, il a choisi de délimiter la chaîne avec des guillemets doubles, pour une fois. *Essayez de le piéger en saisissant une chaîne qui contient à la fois des guillemets et des apostrophes, pour voir.*

Notez également que l'anti-slash est lui-même un caractère spécial : pour l'incorporer dans une chaîne, il faut parfois l'échapper, même si Python est assez intelligent pour se rendre compte quand ce n'est pas nécessaire.

```
>>> "ceci \ est un anti-slash"
'ceci \\ est un anti-slash'
>>> "je vous offre un anti-slash: \"
  File "<stdin>", line 1
    "je vous offre un anti-slash: \"
                                    ^
SyntaxError: EOL while scanning string literal

>>> "je vous offre un anti-slash: \\"
'je vous offre un anti-slash: \\'
```

Et si je veux écrire une chaîne qui fasse plusieurs lignes ?

Pour cela, vous avez encore deux alternatives. La première et... d'accord, j'arrête avec cette blague.

La première alternative, c'est d'utiliser le caractère spécial `\n`, qui représente un *retour à la ligne*.

```
>>> "voici une chaîne\nsur plusieurs lignes"
'voici une chaîne\nsur plusieurs lignes'
```

Hé, mais ça marche pas !

Si si, ça fonctionne. Regardez :

```
>>> print("voici une chaîne\nsur plusieurs lignes")
voici une chaîne
sur plusieurs lignes
```

Nous verrons juste après à quoi sert ce "`print`". Remarquez simplement que l'affichage correspond bien à ce que nous voulions.

La seconde alternative, c'est d'entourer notre chaîne avec des *triples guillemets* (`"""`) ou des *triples apostrophes*.

Essayons :

```
>>> """voici une chaîne
... sur plusieurs lignes"""
'voici une chaîne\nsur plusieurs lignes'
```

Les trois points (`...`) qui s'affichent dans cet exemple sont un indice visuel de la console Python qui signifient que la saisie n'est pas terminée, et qu'elle attend que je poursuive celle-ci sur une nouvelle ligne. Ici, j'ai tapé `"""voici une chaîne`, `<Entrée>`, `sur plusieurs lignes"""`.

Si vous utilisez la console IDLE, ces trois points n'apparaîtront pas. Ne soyez donc pas dérouté par la différence d'affichage avec les exemples du cours.

D'autres types d'objets

Juste un peu plus haut, je vous ai montré un exemple bizarre où j'ai écrit `print('une chaîne')`. Qu'est-ce que c'est que ce `print` ?

La première observation que l'on peut faire, c'est que `print` est un nom que Python reconnaît :

```
>>> print
<built-in function print>
```

Ce que l'interpréteur vient de répondre, c'est que `print` est une *fonction*. C'est-à-dire un objet qui sert à *faire quelque chose*.

Pour utiliser une fonction, il faut l'*appeler*. Pour cela, il faut accoller des parenthèses après son nom :

```
>>> print()

>>>
```

Mais... il ne s'est rien passé ?!

Si si, regardez plus attentivement : Python a sauté une ligne. En fait, l'appel `print()` affiche une ligne vide. Essayons autre chose :

```
>>> print("Bonjour !")
Bonjour !
>>> x = 2 + 2
>>> print("x =", x)
x = 4
```

Cette fois, nous avons passé *un argument* à la fonction `print` (la chaîne `"Bonjour !"`), et le résultat affiché est le contenu de cette chaîne. Puis nous lui avons passé deux objets (la chaîne `"x ="` et le nombre étiqueté par la variable `x`) et celui-ci les a affichés à la suite.

Vous l'aurez deviné ; la fonction `print(...)` sert à afficher ses arguments à l'écran. C'est d'ailleurs ce que signifie son nom : *imprimer* (sous-entendu *imprimer à l'écran*, soit *afficher*).

Mais depuis le début on affiche des trucs à l'écran, il suffit de les taper dans la console pour ça !

C'est vrai, sauf que nous n'allons pas utiliser le console *ad vitam aeternam*. À un moment donné, nos programmes vont devenir plus compliqués, et nous devrons les écrire dans des fichiers pour les exécuter hors de la console interactive. Dans ce cas, nous utiliserons `print(...)` pour afficher des informations à l'écran.

De plus, son comportement n'est pas tout à fait le même :

```
>>> ma_chaine = "Je vous offre un anti-slash : \\"
>>> ma_chaine
'Je vous offre un anti-slash : \\'
>>> print(ma_chaine)
Je vous offre un anti-slash : \
```

Lorsque j'envoie `ma_chaine` à l'interpréteur, celui-ci affiche la **représentation** de l'objet (c'est-à-dire la chaîne telle qu'on la tape pour la créer), alors que si je l'affiche avec `print(ma_chaine)`, Python affiche **le message** contenu dans la chaîne, en remplaçant les caractères échappés et en supprimant les guillemets.