

Python 3 pour l'apprenti sorcier

Arnaud Calmettes (nohar)

1^{er} mars 2015

Table des matières

Avant-propos	3
1 Introduction	4
1.1 Pourquoi « pour l'apprenti sorcier » ?	4
1.2 Le langage Python	4
1.2.1 Comment ça fonctionne ?	5
1.2.2 Pourquoi apprendre Python ?	5
1.3 Installer Python	6
2 Rencontre du troisième type	7
2.1 Contact	7
2.2 Une bête de calcul	8
2.2.1 Saisir un nombre	8
2.2.2 Opérations courantes	9
2.3 Ce qu'il faut retenir	11
3 Des noms et des objets	12
3.1 Les variables	12
3.1.1 Des étiquettes pour s'y retrouver	12
3.1.2 Un peu de syntaxe	13
3.1.3 Comment choisir un nom	15
3.2 C'est l'histoire d'un type...	16
3.2.1 Plusieurs types de données	16
3.2.2 Un autre type d'objet : les fonctions	19
3.3 Exercices	22
3.4 Ce qu'il faut retenir	22
4 Branchements conditionnels	24
4.1 Notre programme d'exemple	24
4.1.1 Écrire un programme dans un fichier	25
4.1.2 Jouons !	26
4.2 Vrai ou faux ?	26
4.2.1 Les booléens	26
4.2.2 Opérateurs de comparaison	27
4.2.3 Encadrements	28

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 voilà rassuré ?

Tant mieux, parce que **tout le reste est vrai !**

1 Introduction

La magie est un pont. Un pont qui te permet d'aller du monde visible vers l'invisible. Et d'apprendre les leçons des deux mondes.

Paulo Coelho

1.1 Pourquoi « pour l'apprenti sorcier » ?

Qu'y a-t'il de magique à programmer ? La question est légitime. Après tout, il s'agit d'une activité banale de nos jours. Depuis quelques années, la programmation en Python est même enseignée dans les lycées français, donc reconnue par l'Éducation Nationale. Comment diable peut-on considérer qu'une *discipline scolaire* relève de la magie ? C'est insensé !

Et pourtant.

N'attendez pas de moi que je vous fournisse une réponse exhaustive dans cette introduction ; je vais plutôt m'efforcer de vous le montrer tout au long du cours, en essayant de vous le faire voir au travers de mes propres yeux.

Permettez-moi quand même de vous faire remarquer que l'école n'a pas pour vocation d'enseigner le *beau*, mais d'abord de transmettre, sinon le *vrai*, du moins les rudiments de la pensée moderne. Ainsi, étudier Balzac ou Hugo au collège n'a malheureusement rien de passionnant, apprendre la philosophie d'Épicure pour le Bac ne suscite que rarement l'éveil, et on pouffe à peine à la lecture *imposée de la scène trois pour jeudi prochain* d'une comédie de Molière. Ce n'est pas le métier du professeur que de transmettre une passion.

Bien sûr, ça n'empêche pas cette trempe de professeurs d'exister, encore heureux ! On en a tous eu un ou deux qui nous ont marqué à vie parce qu'ils ont su transcender le discours de prof à élève, éveiller notre curiosité et notre désir d'en savoir plus, mais ils ne sont que l'exception qui confirme la règle : l'école, ce n'est pas le *cercle des poètes disparus*, c'est le moule de la société de demain.

Revenons-en à la programmation : ce n'est pas parce que l'école lui ôte toute sa magie que celle-ci n'est pas réelle ni palpable. Il suffit d'être attentif pour la remarquer. Programmer un jeu vidéo ne consiste-t'il pas à *façonner un univers* qui n'obéit qu'aux règles de notre imagination ? Comment passer à côté du caractère divin d'une telle réalisation ? Et même sans aller jusque là, si je vous demande quel est l'art qui consiste à *invoquer* les entités d'un monde abstrait pour bénéficier de leur puissance et accomplir notre volonté, ne pensez-vous pas d'abord à de la sorcellerie ? C'est pourtant précisément ce que nous nous apprêtons à faire du bout de nos claviers. Ça, et bien plus encore !

1.2 Le langage Python

Python est un langage de programmation moderne créé au tout début des années 1990 par un ingénieur hollandais du nom de Guido van Rossum. Le langage doit son nom à la passion de son créateur pour

les célèbres Monty Python, auxquels on peut trouver d'innombrables références dans la documentation. Aujourd'hui, après plus de deux décennies d'existence, Python est dans sa troisième version majeure. À au moment où sont écrites ces lignes, la dernière version stable du langage est la 3.4.

TODO

1.2.1 Comment ça fonctionne ?

Qu'il s'agisse d'un jeu vidéo, d'un navigateur internet, ou même d'un système d'exploitation comme Windows ou Linux, un **programme**, ce n'est au fond rien d'autre qu'une série d'instructions que l'ordinateur doit exécuter. Suivant le langage (ou plutôt la technologie) que l'on utilise, cela peut se présenter sous plusieurs formes.

1.2.2 Pourquoi apprendre Python ?

Il serait extrêmement lourd de détailler ici toutes les qualités qui font de Python un langage idéal pour apprendre à programmer. Néanmoins, si vous êtes à la recherche du bon langage pour vous lancer, il faut bien que vous puissiez le comparer aux autres pour faire votre choix. Voici donc quelques raisons pour lesquelles vous devriez choisir Python comme premier langage de programmation.

Commençons par ses qualités intrinsèques. Le langage a été pensé pour être **le plus lisible possible**. Contrairement à presque tous les autres langages de programmation, Python impose aux développeurs certaines règles de mise en forme et incite donc les débutants à prendre *les bonnes habitudes* dès le début de leur apprentissage.

De plus, Python est un langage à **très haut niveau** d'abstraction. Cela signifie que c'est le langage qui se charge de gérer pour vous la mémoire utilisée par votre programme. Cela permet au développeur de se concentrer sur ce qu'il veut que son programme fasse réellement, plutôt que d'encombrer le code d'instructions rébarbatives et bien souvent propices aux erreurs.

Ensuite, Python est **portable**. À quelques exceptions près, un programme écrit en Python peut être exécuté sur n'importe quel système actuel (Windows, GNU/Linux, Mac OS, ou encore les smartphones et tablettes sous Android ou iOS).

Une autre qualité fondamentale de Python est que « **les piles sont incluses dans la boîte** ». Cela signifie que la *bibliothèque standard* du langage (qui est livrée avec lorsque vous installez Python) est extrêmement riche et contient du code qui permet au développeur de réaliser beaucoup de choses courantes sans avoir à réinventer la roue. Par exemple, elle contient des modules permettant de lire ou enregistrer des fichiers XML ou CSV, d'échanger des informations sur un réseau, de télécharger des pages web, d'interagir avec votre système d'exploitation et même de créer de petites interfaces graphiques (des programmes avec des fenêtres).

Enfin, Python est une technologie **libre** et **très populaire**. Cela signifie :

- qu'il est développé, maintenu et documenté par une communauté de passionnés du monde entier ;
- qu'il est et restera toujours **gratuit** ;
- que vous avez le droit de l'utiliser pour n'importe quel programme, que vous comptiez le vendre ou non ;

- qu’il est **extrêmement bien documenté**, donc qu’à condition de travailler votre niveau d’anglais, vous pouvez vous reposer sur sa documentation plutôt que d’apprendre son utilisation par cœur ;
- que vous ne vous retrouverez jamais seul face à un problème puisqu’il existe de nombreuses communautés, y compris francophones, prêtes à aider les débutants.

1.3 Installer Python

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.

2.1 Contact

Nous voici devant l’autre 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 ?

2.2 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.

2.2.1 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 virgule sont appelés 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 une 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.

2.2.2 Opérations courantes

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

2.2.2.1 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.

2.2.2.2 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
```

2.2.2.3 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.

2.3 Ce qu'il faut retenir

Récapitulons ce que nous avons appris dans ce 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*.

3.1 Les variables

3.1.1 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 *référéncé*. 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 !

3.1.2 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 `=`.

1. Annonce en gare de la SNCF

```
>>> 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 juste 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
```

3.1.3 Comment choisir un nom

Nommer les choses n'est pas facile, surtout quand on n'en a pas l'habitude. Au début de votre apprentissage, vous serez tentés de choisir des noms très génériques pour vos variables, comme `x`, `y`, `a`, `b` ou encore `plop`, `choucroute`, ou encore pire : `variable`, `variable2`, `variable3`.

Voici quelques conseils pour nommer correctement vos variables.

- Un nom de variable doit **obligatoirement** commencer par une lettre, ou bien par le symbole *underscore* (le “`_`” sous la touche `<8>` des claviers français).
- Donnez à vos variables un nom qui décrit leur rôle plutôt que leur nature : `somme`, `produit`, `prix`, `pt_de_vie` sont mieux que `nombre` ou `entier`.
- Même si Python 3 les accepte, **interdisez-vous les caractères accentués**. Contentez-vous de symboles simples. Il vaut mieux nommer une variable `prenom` plutôt que `prénom`, ou `coeur` plutôt que `cœur`.
- À quelques exceptions courantes près (du style `x` et `y` pour des coordonnées), il est préférable de choisir des noms qui font **au moins 3 caractères**. `nom` est préférable à `nm`, `option` est préférable à `op`. Et dans tous les cas, n'appellez **jamais** une variable `l` (L minuscule). Suivant la police de caractères utilisée, il est beaucoup trop facile de confondre le L minuscule avec le chiffre `1` (un) ou le `i` majuscule.
- À une seule exception près (que nous verrons dans un autre chapitre) une variable devrait tout le temps être écrite *en lettres minuscules* quitte à séparer les mots d'un *underscore*. Ainsi, préférez `pos_joueur` à `PosJoueur`.
- 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 :

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

Vous vous direz sûrement que certaines de ces règles relèvent « des goûts et des couleurs ». Sachez qu'en Python, il existe une norme qui décrit les *conventions de nommage* et que la grande majorité des développeurs Python la respectent.

Cette norme est la [PEP-08](#). La respecter n'est pas obligatoire, mais vous pouvez être sûr que la remarque vous sera faite si quelqu'un relit votre code, alors autant prendre de bonnes habitudes dès le départ. ;)

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

3.2 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 ».

3.2.1 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.

3.2.1.1 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 !

3.2.1.2 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,

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*...

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 !

3.2.1.3 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 ?

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 coolypmique !'
"Python, c'est coolypmique !"
```

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 signifie que la saisie n'est pas terminée, et que Python attend que je la termine sur une nouvelle ligne. Ici, j'ai simplement 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.

3.2.2 Un autre type d'objet : les fonctions

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'*invoker* en accolant 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 Python les a affichés l'un à la suite de l'autre.

Vous l'aurez deviné ; la fonction `print(...)` sert à afficher des objets 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 æternam*. 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, le résultat 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.

Voyons maintenant une autre fonction utile : `type()`. Cette fonction retourne le type de l'objet qu'on lui passe en argument :

```
>>> type(42)
```

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

```
>>> type(42.5)
```

```
<class 'float'>
>>> type("Bonjour !")
<class 'str'>
```

Comme vous pouvez le constater, je ne vous ai pas menti tout à l'heure : les entiers sont bien des `int`, les nombres à virgule des `float` et les chaînes de caractères des `str`.

Que veut dire ce `class` ?

En Python, une *classe* est simplement un synonyme de *type*. Nous verrons beaucoup plus loin dans ce cours (quand nous créerons nos propres types d'objets) ce qu'est précisément une classe. Pour l'instant, reprenez simplement qu'un type et une classe sont exactement la même chose en Python.

Il peut être intéressant de remarquer qu'à chacun des trois types de base que nous connaissons correspond une fonction du même nom. Essayez de deviner, grâce à l'exemple suivant, à quoi servent ces fonctions :

```
>>> entier = 42
>>> flottant = float(entier)
>>> flottant
42.0
>>> chaine = str(entier)
>>> chaine
'42'
>>> int(chaine)
42
```

Alors ?

Euh, ça transforme un objet d'un type vers un autre ?

Exactement ! On peut utiliser ces fonctions pour réaliser des conversions entre types. En fait, les fonctions `int()`, `float()` et `str()` s'appellent des *constructeurs*. Elles servent à construire des entiers, des flottants et des chaînes de caractères à partir d'autres objets.

Dernière remarque. Quelle autre différence peut-on observer entre ces trois fonctions et la fonction `print()` ?

Regardez cet exemple :

```
>>> a = str(42.512)
>>> a
'42.512'
>>> b = print("Coucou")
Coucou
>>> b
>>>
```

Dans cet exemple, j'ai transformé le nombre 42.512 en une chaîne de caractères que j'ai affecté à la variable `a`. En somme, j'ai affecté le *résultat* de `str(42.512)` à une variable.

Puis j'ai essayé d'affecter le résultat de `print("Coucou")` à la variable `b`. Comme prévu, l'appel à `print()` a affiché la chaîne `Coucou` à l'écran mais la variable `b`... ne contient rien du tout, même si elle existe !

Cela montre que **certaines fonctions retournent un résultat** alors que d'autres non. En informatique, comme on aime bien donner des noms différents à des trucs différents, on appelle parfois une fonction qui ne retourne aucun résultat une *procédure* (ou encore une *routine*) plutôt qu'une fonction.

Mais assez parlé des fonctions. Nous aurons l'occasion d'y revenir plus tard.

3.3 Exercices

Parce que la programmation ne s'apprend pas seulement dans un grimoire, tous les chapitres de ce cours contiendront dorénavant des exercices.

Faites-les !

Non, sérieusement, *faites-les*. Le seul moyen pour vous de retenir ce que vous venez de lire et que votre apprentissage soit utile, c'est de réveiller votre cerveau et d'être actif.

Soient les variables suivantes :

```
>>> x = 38
>>> y = 54
```

Je vous demande simplement de créer les quatres variables qui suivent :

- `somme` devra référencer la somme de `x` et `y`,
- `produit` désignera le résultat de la multiplication de `x` par `y`,
- `diff` sera la différence entre `x` et `y`,
- et `modulo` le reste de la division euclidienne de `y` par `x`.

En guise de second exercice, je vous demande de trouver un moyen de **permuter** ces deux variables. C'est-à-dire écrire du code tel que vous obteniez dans la console le résultat suivant :

```
>>> x
54
>>> y
38
```

Ne trichez pas! `x, y = 54, 38` n'est pas une solution valide. ;)

Votre code doit permuter ces deux variables quel que soit leurs valeurs. À titre d'indication, il existe deux solutions : la première en trois instructions, la seconde en une seule ligne.

Bonne chance !

3.4 Ce qu'il faut retenir

Nous avons vu beaucoup de choses dans ce chapitre. Voici ce que vous devriez en retenir :

- Une variable, c'est une *étiquette* que l'on colle sur un objet pour lui donner un *nom*,
- Tant qu'un objet est référencé par une variable, on peut le manipuler : Python se souviendra de lui,
- Le *type* d'un objet, c'est ce qui décrit sa nature : à quoi il ressemble, comment on l'utilise, etc.
- Vous connaissez quatre types d'objets différents :

- Les `int` sont les nombres entiers.
- Les `float` sont les nombres à virgule.
- Les `str` sont les chaînes de caractères ; elles contiennent du *texte*.
- Les fonctions sont des *actions* que l'on peut invoquer pour les faire exécuter par Python.

Soufflez ! Ce chapitre était très théorique. Prenez un peu de temps pour le digérer. À partir du chapitre suivant, nous allons commencer à réaliser de véritables programmes.

4 Branchements conditionnels

- Quarante-deux ! cria Loonquawl. Et c'est tout ce que t'as à nous montrer au bout de sept millions et demi d'années de boulot ?
- J'ai vérifié très soigneusement, dit l'ordinateur, et c'est incontestablement la réponse exacte. Je crois que le problème, pour être tout à fait franc avec vous, est que vous n'avez jamais vraiment bien saisi la question.

Douglas Adams – Le Guide du Voyageur Galactique

Vous ne vous en doutez sûrement pas, mais dans la totalité des programmes qu'il exécute, l'ordinateur passe son temps à **se poser des questions**. Oh, bien sûr, il ne se demande pas ce qu'il va manger ce soir ni quel est le sens de sa vie, mais il se pose des questions d'ordinateur, qui lui permettent de décider *ce qu'il va faire ensuite*.

Par exemple, imaginons un programme qui demande à l'utilisateur d'entrer un mot de passe. À un moment donné de ce programme, l'ordinateur va devoir décider si le mot de passe entré par l'utilisateur est le bon, et il exécutera une action différente suivant que ce soit le cas ou non. Ce genre de carrefour dans l'exécution d'un programme s'appelle un **branchement conditionnel**, et c'est ce que nous allons étudier tout au long de ce chapitre.

Dans ce chapitre, nous allons apprendre :

- ce qu'est une valeur **booléenne**,
- la notion de **prédicat**,
- à effectuer différentes actions dans différentes conditions.

4.1 Notre programme d'exemple

Voici un programme qui illustre ce que nous allons apprendre dans ce chapitre :

```
saisie = input("Saisissez votre mot de passe : ")

if saisie == 'abracadabra':
    print("Accès autorisé")
else:
    print("Accès refusé")
```

Nous allons découvrir dans la suite comment celui-ci fonctionne en détail. Contentons nous de jouer avec dans un premier temps.

4.1.1 Écrire un programme dans un fichier

À partir de ce chapitre, nous allons travailler sur des programmes d'exemple. Il existe bien des façons d'exécuter un programme en Python suivant votre système d'exploitation. Pour ne pas surcharger le propos, je me contenterai de vous montrer ici la méthode qui marchera chez tout le monde.

Ouvrez le programme IDLE livré avec Python. Vous devriez voir apparaître une fenêtre dans le style de la figure 1.

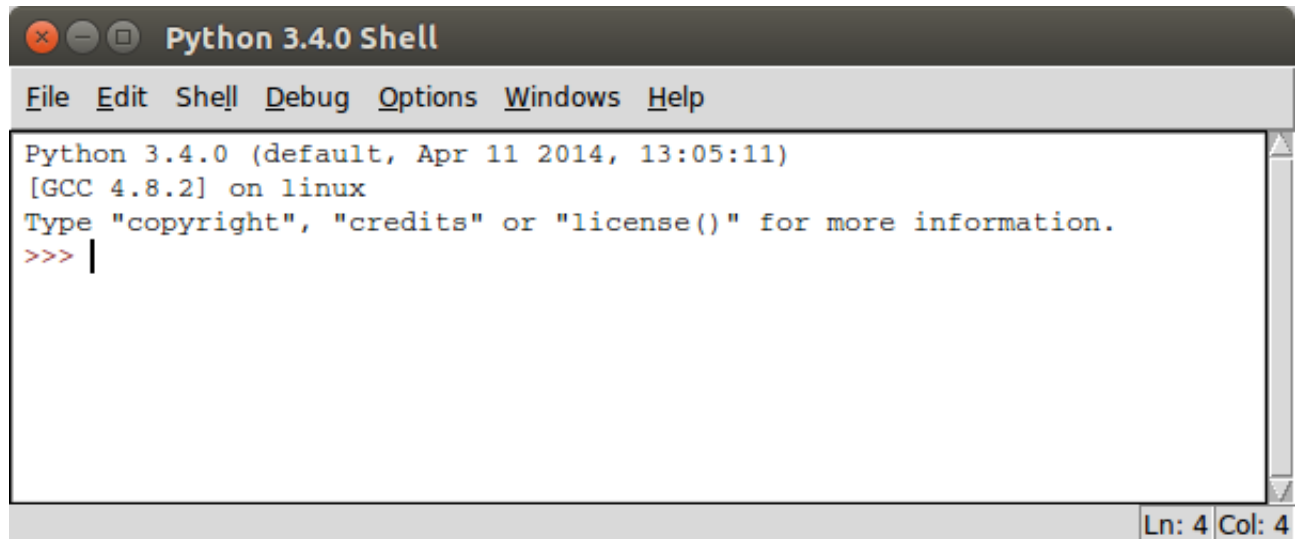


FIGURE 4.1 – La console IDLE

Puis cliquez sur **File** -> **New File** ou bien appuyez simultanément sur les touches **<Ctrl>** et **<N>**.

Ceci devrait faire apparaître une fenêtre vide dans laquelle vous pouvez écrire un programme en Python. Recopiez dedans notre programme d'exemple, comme sur la figure 2.

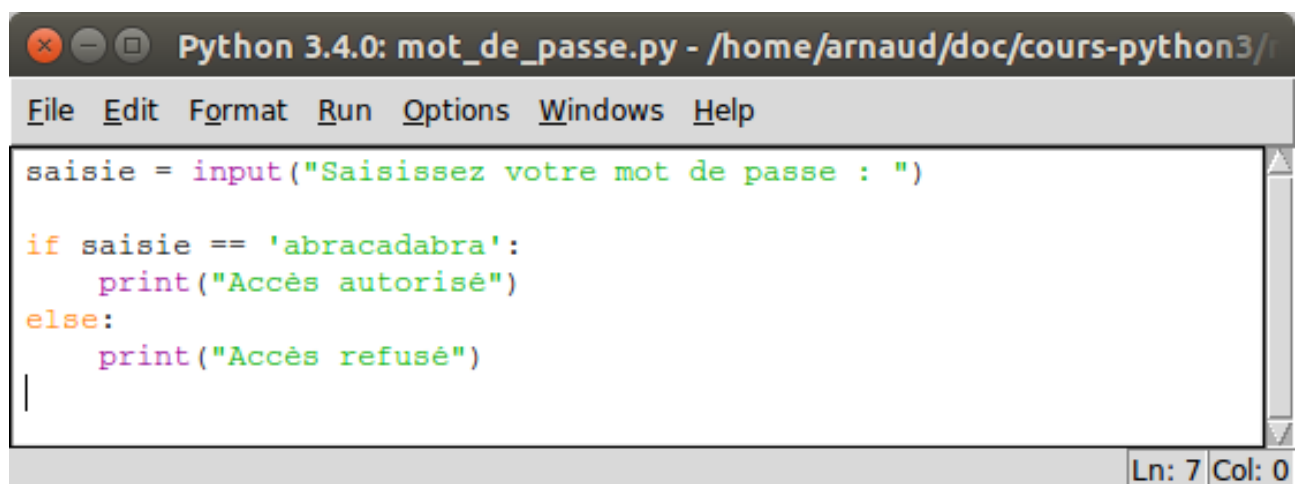
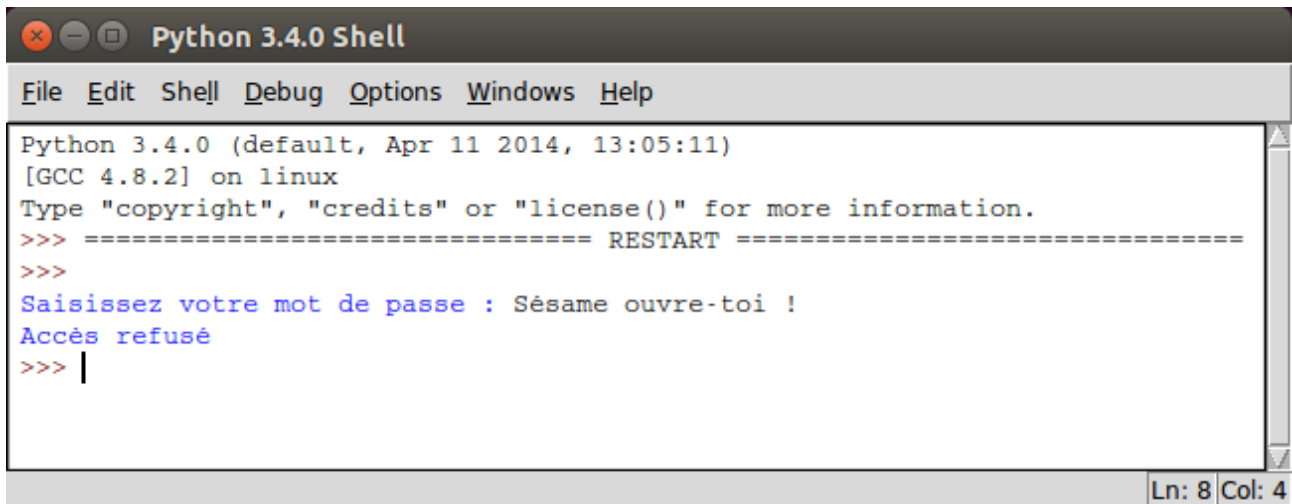


FIGURE 4.2 – Notre programme d'exemple

Enfin, exécutez ce programme en appuyant sur la touche **<F5>**. IDLE va vous demander si vous souhaitez sauvegarder ce fichier au passage, enregistrez-le en lui donnant l'extension **.py**, par exemple

sous le nom `motdepasse.py`. Une fois ceci fait, le programme se lance dans la console interactive, comme sur la figure 3.



```
Python 3.4.0 Shell
File Edit Shell Debug Options Windows Help
Python 3.4.0 (default, Apr 11 2014, 13:05:11)
[GCC 4.8.2] on linux
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
Saisissez votre mot de passe : Sésame ouvre-toi !
Accès refusé
>>> |
```

FIGURE 4.3 – Exécution

Essayez d’exécuter plusieurs fois ce programme avec des mots de passe différents, puis essayez avec le bon : `abracadabra`. Comme vous pouvez le constater, le programme n’affiche “Accès autorisé” que si vous lui entrez le bon mot de passe.

4.1.2 Jouons !

Avant même de chercher à comprendre comment ce script marche, essayez de le modifier pour changer son comportement. Par exemple :

- Faites en sorte que le mot de passe soit autre chose que « `abracadabra` ».
- Traitez l’utilisateur d’idiot lorsqu’il entre le mauvais mot de passe.

C’est une bonne habitude à prendre lorsque vous êtes confronté à un code de ce cours : réussir à bidouiller un programme pour lui faire faire ce que l’on veut, c’est beaucoup plus efficace qu’un long discours théorique !

4.2 Vrai ou faux ?

4.2.1 Les booléens

Je vous ai dit plus haut que même si l’ordinateur se pose des questions à longueur de temps, il ne peut pas se poser n’importe lesquelles. En fait, les questions qu’il se pose sont des questions *logiques*, c’est-à-dire qu’elles n’acceptent que deux réponses possibles : oui ou non.

Par exemple, nous pouvons demander à Python si 2 est inférieur à 38.

Pour cela, nous allons formuler notre question d’une façon qui va peut-être vous sembler bizarre : nous allons *affirmer* que 2 est inférieur à 38, et Python nous répondra que si c’est vrai ou faux :

```
>>> 2 < 38
True
```

À l'inverse, si on lui dit que 7 est inférieur à 4, Python nous répondra que c'est faux :

```
>>> 7 < 4
False
```

En programmation, une expression qui peut être vraie (**True**) ou fausse (**False**) s'appelle **un prédicat**, et les deux valeurs de vérité possibles d'un prédicat sont un type d'objet à part entière, c'est le type **booléen**. Python appelle ces valeurs des **bool** :

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

4.2.2 Opérateurs de comparaison

La plupart des prédicats que nous allons écrire au départ utiliseront des opérateurs un peu spéciaux que nous appelons *opérateurs de comparaison*.

Ces opérateurs sont relativement simples à comprendre, ils fonctionnent comme en maths :

Opérateur	Signification
$x < y$	x est strictement inférieur à y
$x \leq y$	x est inférieur ou égal à y
$x == y$	x est égal à y
$x \neq y$	x est différent de y
$x \geq y$	x est supérieur ou égal à y
$x > y$	x est strictement supérieur à y

TABLE 4.1 – Opérateurs de comparaison usuels

Remarquez que l'opérateur d'égalité se note `==`, contrairement à l'opérateur d'affectation (`=`) que nous utilisons depuis le début de ce cours.

Ces opérateurs s'appliquent sur tous les types de données de base que nous avons vus jusqu'à présent. Si, pour les nombres leur signification est évidente, il y a de quoi être perplexe sur leur utilisation avec des chaînes de caractères :

```
>>> 'bacon' > 'oeufs'
False
>>> 'bacon' != 'oeufs'
True
>>> 'bacon' <= 'oeufs'
True
```

```
>>> 'spam' == 'spam'
True
```

En fait, il existe une relation d'ordre sur les chaînes de caractères, que l'on appelle pompeusement l'**ordre lexicographique**. Dans les faits, il s'agit plus ou moins de l'ordre alphabétique, à ceci près que l'on différencie les lettres majuscules et minuscules.

Ne nous attardons pas trop sur l'ordre lexicographique pour le moment. Vous pouvez faire des tests dans la console pour comprendre comment celui-ci fonctionne, mais en réalité, il y a peu de chances que vous ayez besoin de comparer les chaînes de caractères autrement qu'avec `==` et `!=` avant un bon moment. Nous reviendrons sur ce sujet lorsque nous étudierons la façon dont les chaînes sont encodées par Python.

4.2.3 Encadrements

Une particularité des opérateurs de comparaison, à laquelle on pense trop rarement, est que ceux-ci peuvent être chaînés pour exprimer des *encadrements*. Par exemple, on peut vérifier qu'un nombre est strictement compris entre 0 et 10 :

```
>>> n = 4
>>> 0 < n < 10
True
```

Ou bien qu'il est strictement positif mais inférieur ou égal à 10 :

```
>>> n = 10
>>> 0 < n <= 10
True
```

On peut même laisser libre cours à notre fantaisie, en écrivant des tests un peu plus alambiqués :

```
>>> a, b = 5, 3
>>> 0 < a < b < 10
False
>>> a, b = 3, 5
>>> 0 < a < b < 10
True
```

Gardez bien ces encadrements en tête : ils sont particulièrement efficaces pour expliquer des conditions parfois complexes de façon très lisible.